Proving Entailment Between Conceptual State Specifications

Eugene W. Stark [*][†]

September 24, 1986

Abstract

The lack of expressive power of temporal logic as a specification language can be compensated to a certain extent by the introduction of powerful, high-level temporal operators, which are difficult to understand and reason about. A more natural way to increase the expressive power of a temporal specification language is by introducing *conceptual state variables*, which are auxiliary (unimplemented) variables whose values serve as an abstract representation of the internal state of the process being specified. The kind of specifications resulting from the latter approach are called *conceptual state specifications.*

This paper considers a central problem in reasoning about conceptual state specifications: the problem of proving *entailment* between specifications. A technique, based on the notion of *simulation* between *machines*, is shown to be sound for proving entailment. A kind of completeness result can also be shown, if specifications are assumed to satisfy certain well-formedness conditions. The role played by entailment in proofs of correctness is illustrated by the problem of proving that the concatenation of two FIFO buffers implements a FIFO buffer.

[*]This research was performed in part while the author was a graduate student at the Massachusetts Institute of Technology, during which period the author was supported in part by ARO grant DAAG29-84-K-0058, NSF grant DCR-83-02391, and DARPA grant N00014-82-K-0125.

[†]Author's present address: Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York 11794-4400 USA.

1 Introduction

A *process* [1] can be characterized in terms of the possible histories of its accesses to variables. A *specification* describes a process by stating properties that are required to hold of all possible histories for that process. As has been shown by a number of authors ([BK83], [BK84], [Lam83], [HO80], [SM81]), such process specifications can be expressed as sentences in *linear-time temporal logic*. One of the difficulties with temporal logic as a specification language is that, at least in the most basic formulations, it is lacking in expressive power. This lack of expressive power can be compensated to a certain extent by the introduction of a number of powerful temporal operators such as *until*, *chop or combine*, and *iterated combine* ([BK84], [Wol81]). However, these operators do not permit ones intuitive understanding of the desired process behavior to be formalized in the most direct and natural way, and also make reasoning about the resulting specifications more difficult.

An alternative to the use of powerful temporal operators is the technique of *conceptual state specification*. In a conceptual state specification, the behavior of a process with respect to a collection of program variables is specified with the help of a collection of *conceptual state* or *auxiliary* variables, whose values serve as an abstract representation of the internal state of the process. Conceptual state variables appearing in a specification are not intended to be implemented; their introduction serves merely to increase the expressive power of the temporal logic. A process satisfies a conceptual state specification if every computation of that process can be augmented or "explained" through the addition of a history of values for the conceptual state variables, in such a way that the temporal sentence comprising the specification is satisfied.

In contrast to specifications involving the use of powerful temporal operators, conceptual state specifications appear to be a rather direct and natural way to formalize an intuitive understanding of the desired process behavior. For example, a conceptual state specification of a FIFO buffer process B directly formalizes an informal description that begins: "Imagine that process B contains an internal variable **queue**, whose value at any instant records the sequence of messages input to the buffer but not yet output ...," and continues with a description of the *initial state* of a buffer process, the kinds of *state transitions* that may be taken by a buffer process, and a collection of *liveness*

[1] In this paper, we use the term "process" to refer both to a sequential process and a system of concurrently executing sequential processes.

properties that must be satisfied. Conceptual state specifications generally do not require the use of temporal operators other than "henceforth," "eventually," and "next," because conceptual state variables, rather than temporal formulas, are used to summarize the past history of module behavior.

The basic idea of conceptual state specifications is not new, having been proposed previously in various forms by a number of authors. Yonezawa ([Yon77]) describes a specification method that uses "conceptual representations," to specify behaviors in the actor model of computation. The history variables of Hailpern and Owicki ([HO80]) can be viewed as a kind of conceptual state variables, whose values represent the sequences of values passed between processes up until a particular instant. Lamport ([Lam83]) describes a specification technique in which a specification is permitted to refer to a collection of indeterminate *state functions*, whose values summarize the state of a process. The style of specification that results is essentially similar to the conceptual state style illustrated in this paper. However, to show that a particular process satisfies a specification, it is necessary to provide definitions of the state functions in terms of the implemented process state. It would therefore appear that Lamport views state functions as playing more than just an auxiliary role.

Although conceptual state specifications seem to be a natural way to describe process behavior, it is not quite as clear how to perform reasoning with them as it is in the case of ordinary temporal specifications. The somewhat nonstandard appearance of the quantifier "there exists a history for the conceptual state variables such that" in the definition of what it means for a process to satisfy a specification causes a certain amount of difficulty. A central problem in reasoning with conceptual state specifications is the problem of proving an *entailment* or *logical implication* between two specifications, which in general involve different sets of conceptual state variables.

This paper introduces the notion of conceptual state specifications, defines what it means for the entailment relation to hold between two conceptual state specifications, and develops a technique for proving that this relation holds. We are able to show a kind of completeness result for our technique, which states that a true entailment relation can always be proved, assuming the specifications involved satisfy certain well-formedness conditions that can be independently checked. The technique is illustrated by considering the problem of proving that the tandem connection of two FIFO buffers again implements a FIFO buffer.

Our entailment proof technique makes use of the concept of a *simulation* between *machines*, and can be viewed as a generalization of the standard *representation function*, *abstraction function*, or *interpretation* techniques for proving an implementation relationship between an abstract data type and its concrete representation ([GHM78], [Hoa72], [Jon81]) If an abstract data type is viewed as a process, whose communications correspond to invocations of operations of the data type, then standard techniques are capable of proving only *safety* or *invariance* properties. In contrast, our technique permits both safety properties and *liveness* or *eventuality* properties to be proved. The technique used by Goree and Lynch ([Gor81], [Lyn83]) in a hierarchical proof of invariance properties of a concurrency control algorithm can also be viewed as a special case of the technique presented here.

The results of this paper are a reformulation of results reported in [Sta84]. In that document, a number of processes are specified using the conceptual state technique, and several correctness proofs are performed using the technique described here. Experience with these examples forms the basis of the author's opinion that conceptual state specifications are a natural specification method that can support the systematic construction of correctness proofs by the techniques described here.

2 Processes

In this section we define a mathematical model of processes, in which the notion of process is identified with that of certain sets of *histories*, where each history records the accesses to variables made during a particular system execution. In the next section, conceptual state specifications will be defined, and it will be shown how a conceptual state specification is satisfied by a process.

Our model is based on the intuitive conception of a system of concurrently executing sequential processes that interact through changes to the values of shared variables. Only one process is permitted to access each particular variable at any given instant of time. Although we find the shared variable assumption convenient for this paper, it is not essential for the results, and in fact easily can be replaced by a message-passing model, or a model in which processes interact by synchronized communication.

We will represent the computation of such a system of processes in terms of the history of values taken on by the variables. In addition, we shall always be describing a computation from the vantage of a distinguished process in the system, and our representation of computations will include information about which variables were accessed, at each instant of time, by the distinguished process, and which were accessed by the environment of that process. The presence of this information in the model allows us to obtain *composable* temporal specifications ([BK84]).

There is a nonstandard feature of our model that requires some prior explanation. Below we shall define a history to be a certain kind of function from the nonnegative real line to a set of *events*. We shall then define the semantics of our temporal logic language in terms of these "continuous" histories, rather than in terms of discrete sequences as is usually done. A consequence of our approach is that the "next state" operator $\bigcirc$ becomes meaningless, and we replace it with the somewhat weaker notions of "before" and "after" states. The reason for making these nonstandard definitions is to obtain a temporal logic whose sentences are incapable of distinguishing between histories that are identical except for occurrences of "null events," in which no changes are made to the values of variables. The formal statement of the property we require is the Projection Lemma (Lemma 3). Ordinary formulations of temporal logic in terms of discrete sequences do not satisfy the Projection Lemma.

To begin our formal treatment, we assume the existence of a universe $\mathcal{V}$ of *program variables*, and a universe $\mathcal{U}$ of *values*. If $V \subseteq \mathcal{V}$ then a V-*state* is a function $q : V \rightarrow \mathcal{U}$. If q is a V-state, q' is a V'-state and $U \subseteq V \cap V'$, then define $q =_U q'$ if $q(u) = q'(u)$ for all $u \in U$. If $q =_{V \cap V'} q'$, then define the *join* $q \sqcup q'$ to be the unique $(V \cup V')$-state r such that $r =_V q$ and $r =_{V'} q'$. If q is a V-state and $U \subseteq V$, then define the *projection* $\pi_U(q)$ to be the unique U-state q' such that $q' =_U q$.

Definition 1 *(Event) A* V-event *is a pair* $e = (\underline{e}, \widehat{e}, \overline{e})$, *where* $\underline{e}$ *and* $\overline{e}$ *are* V*-states, called the* before state *and the* after state*, respectively, and* $\widehat{e}$ *is a subset of* V*, called the* access set *of* e*. The event* e *is a* null event *if* $\widehat{e} = \emptyset$ *and* $\overline{e} = \underline{e}$.

We extend the $=_U$ notation to events by defining $e =_U e'$ iff $\underline{e} =_U \underline{e}'$, $\overline{e} =_U \overline{e}'$, and $\widehat{e} \cap U = \widehat{e}' \cap U$. The notations $\sqcup$ and $\pi_U()$ can then be extended to events in an obvious way.

Intuitively, a V-event records the results of a single step of execution, viewed from the vantage of a particular process, say P, in a system. The before state $\underline{e}$ of a V-event e records the values of the variables V "just before" the step in question, and the after state $\overline{e}$ records the values of the variables "just after" the step in question. The access set $\widehat{e}$ records the set of variables accessed by the process in the step. Changes to the values of variables in $\widehat{e}$ are attributed to the action of process P. Changes to the values of variables not in $\widehat{e}$ are attributed to the action of the environment of process P. Access sets are a refinement of, and serve the same purpose as, the environment and process actions of [BK84].

Definition 2 *(History) Let $\mathcal{R}^+$ be the set of nonnegative real numbers. A V-*history *is a function x from $\mathcal{R}^+$ to V-events, with the following property: For all $t \in \mathcal{R}^+$, there exists $\epsilon > 0$ such that*

1. $\underline{x}(t') = \overline{x}(t') = \underline{x}(t)$ *and* $\widehat{x}(t') = \emptyset$ *for all* $t' \in \mathcal{R}^+$ *with* $t - \epsilon < t' < t$,
2. $\overline{x}(t) = \underline{x}(t') = \overline{x}(t')$ *and* $\widehat{x}(t') = \emptyset$ *for all* $t' \in \mathcal{R}^+$ *with* $t < t' < t + \epsilon$.

We extend the notations $=_U$, $\sqcup$, and $\pi_V()$ to histories in the obvious way.

Intuitively, a V-history is a record of all steps that occur during a single execution of a process, along with their time of occurrence. The two requirements state that each instant of time at which a nonnull event occurs is surrounded by an interval of time during which only null events occur. These requirements intuitively correspond to the idea that processes execute at a finite rate, and formally ensure that histories have a certain local finiteness property, as we now show.

Define a subset $T \subseteq \mathcal{R}^+$ to be *locally finite* if $T \cap I$ is finite whenever I is a bounded interval of $\mathcal{R}^+$. Note that a locally finite set T always has a unique enumeration as an increasing sequence, *viz.* $t_0 < t_1 < \ldots$, and if the set T is infinite, then this sequence is unbounded.

Lemma 1 *Suppose x is a V-history. Then the set of all $t \in \mathcal{R}^+$ for which $x(t)$ is nonnull is locally finite.*

Proof – Suppose not, then there is some bounded interval $I \subset \mathcal{R}^+$, such that $x(t)$ is nonnull for infinitely many $t \in I$. We can assume without loss of generality that I is closed. Since the closed, bounded subsets of $\mathcal{R}^+$ are compact, it follows that $\{t \in I : x(t) \text{ nonnull}\}$ has an accumulation point, say t_0, in I. Then t_0 is also an accumulation point of one of the sets $\{t' < t_0 : x(t') \text{ nonnull}\}$ or $\{t' > t_0 : x(t') \text{ nonnull}\}$. Suppose the former, the argument for the latter case is symmetric. Then for all $\epsilon > 0$ there exists $t' \in (t_0 - \epsilon, t_0)$ with $x(t')$ nonnull. This is in contradiction with the definition of a history. ∎

Lemma 2 *Given an infinite locally finite set* $T = \{t_0 < t_1 < \ldots\}$ *with* $t_0 = 0$*, a sequence* $q_0, q_1, \ldots$ *of* V*-states, and a sequence* $U_0, U_1, \ldots$ *of subsets of* V*, there corresponds a unique* V*-history* x *such that* $x(t_k) = (q_k, U_k, q_{k+1})$ *for all* k*, and* $x(t) = (q_{k+1}, \emptyset, q_{k+1})$ *for all* k *and all* $t \in (t_k, t_{k+1})$*. Conversely, if* x *is a* V*-history, then there exists a set* T*, a sequence* $q_0, q_1, \ldots$ *of* V*-states, and a sequence* $U_0, U_1, \ldots$ *of subsets of* U *with the stated properties.*

Proof – Given T, $q_0, q_1, \ldots$, and $U_0, U_1, \ldots$, the stated properties uniquely define a function x from $\mathcal{R}^+$ to V-events, which is easily seen to be a V-history.

Conversely, suppose x is a V-history. Then the infinite set $T = \{0, 1, 2, \ldots\} \cup \{t \in \mathcal{R}^+ : x(t) \text{ nonnull}\}$ is locally finite by Lemma 1. Suppose $T = \{t_0 < t_1 < \ldots\}$. A straightforward argument, which we omit, making use of the compactness of the closed, bounded subsets of $\mathcal{R}^+$, shows that $\overline{x}(t_k) = \underline{x}(t_{k+1})$ for all k, and $x(t) = (\overline{x}(t_k), \emptyset, \underline{x}(t_{k+1}))$ for all k and all $t \in (t_k, t_{k+1})$. We can therefore obtain the required q_k and U_k by defining $q_k = \underline{x}(t_k)$ and $U_k = \widehat{x}(t_k)$. ∎

Definition 3 *(Process) A* V-process *is a set of* V*-histories.*

It should be noted that, although we have here defined a process to be an *arbitrary* set of V-histories, not every such set should be regarded as computable or realizable in the sense that it is the behavior of a process definition expressed in a particular concurrent programming language. For example, the empty set of histories is evidently not realizable, since any program must have at least one execution. In general, we shall have in mind a particular subset of all processes, which we call the *realizable* processes, and which are the processes denotable in a particular programming language under consideration. The definition of entailment between specifications, and subsequent results based on this definition, should then be relativized to the set of realizable processes. Since this relativization introduces certain complications which are inessential for the purposes of this paper, we suppose here that every process is realizable. The reader is referred to [Sta84] for an attempt to address the general situation.

3 Conceptual State Specifications

As a concrete medium in which to express conceptual state specifications, we define, for each set of variables V, a corresponding first-order temporal logic $\mathcal{T}(V)$, whose sentences are interpreted as properties of V-histories. The language $\mathcal{T}(V)$ is syntactically similar to other linear-time temporal logics ([Lam80], [MP83], [Pnu77]), containing the temporal operators $\Box$ (henceforth) and $\Diamond$ (eventually). However, we do not permit the use of the next state operator $\bigcirc$, since the notion of the "next" state is (by design) meaningless for histories. We draw a distinction between *program variables*, which are those in V and which cannot be bound by quantifiers, and *logical variables*, which are drawn from a set X, disjoint from V, and which are permitted to be bound by quantifiers.

In a temporal formula, we refer to the state portion of a history through the use of program variables in terms. A program variable $v \in V$ can appear in a term only in the form $\underline{v}$, which denotes the value of a program variable just before the current instant, or in the form $\overline{v}$, which denotes the value of v just after the current instant. Through the use of the $\underline{v}$ and $\overline{v}$ constructs, we obtain some, but not all, of the expressive power normally provided by the $\bigcirc$ operator. We refer to the access set portion of a history through the use of special predicates $\widehat{v}$, of which there is one for each program variable $v \in V$. The predicate $\widehat{v}$ is true iff the variable v is accessed by the process under consideration at the current instant.

To avoid issues concerning the possibility of expressing various functions and relations on the underlying universe $\mathcal{U}$, we assume that for each such function or relation there is a corresponding function or relation symbol in the language $\mathcal{T}(V)$.

The precise syntax of $\mathcal{T}(V)$ is as follows:

Terms:

1. If $v \in V$ is a program variable, then $\underline{v}$ and $\overline{v}$ are terms.

2. If $x \in X$ is a logical variable, then x is a term.

3. If f is an n-ary function on $\mathcal{U}$ and $t_1, \ldots, t_n$ are terms, then $\mathbf{f}(t_1 \ldots t_n)$ is a term, where $\mathbf{f}$ is the function symbol corresponding to f.

Formulas:

1. If $v \in V$ is a program variable, then $\widehat{v}$ is a formula.

2. If r is an n-ary relation on $\mathcal{U}$ and $t_1, \ldots, t_n$ are terms, then $\mathbf{r}(t_1 \ldots t_n)$ is a formula, where $\mathbf{r}$ is the relation symbol corresponding to r.

3. If ϕ, ψ are formulas and $v \in X$ is a logical variable, then $\neg\phi$, $\phi \wedge \psi$ and $(\exists v)\phi$ are formulas.

4. If ϕ is a formula, then $\Box\phi$ is a formula.

Additional logical connectives, universal quantification, and $\Diamond$ are treated as defined constructs in the usual way.

To define the semantics of $\mathcal{T}(V)$, we first define the meaning of a term $\mathbf{t}$ to be a function that takes a V-event e and an X-state q to a value $\mathbf{t}(e, q)$.

Terms:

1. $\underline{v}(e, q) = \underline{e}(v)$; $\overline{v}(e, q) = \overline{e}(v)$.

2. $x(e, q) = q(x)$.

3. $\mathbf{f}(t_1 \ldots t_n)(e, q) = f(t_1(e, q), \ldots t_n(e, q))$.

We next define the *satisfaction relation* $\models$ between a V-history x, an X-state q, and a formula ϕ of $\mathcal{T}(V)$.

Formulas:

1. $x, q \models \widehat{v}$ iff $v \in \widehat{x}(0)$.

2. $x, q \models \mathbf{r}(t_1 \ldots t_n)$ iff $r(t_1(x(0), q), \ldots, t_n(x(0), q))$ holds.

3. $x, q \models \neg\phi$ iff $x, q \not\models \phi$; $x, q \models \phi \wedge \psi$ iff $x, q \models \phi$ and $x, q \models \psi$; $x, q \models (\exists v)\phi$ iff there exists q' with $q' =_{X-\{v\}} q$ such that $x, q' \models \phi$.

4. $x, q \models \Box\phi$ iff for all $t \in \mathcal{R}^+$, $x^{(t)}, q \models \phi$, where $x^{(t)}$ is the history x' such that $x'(t') = x(t + t')$ for all $t' \in \mathcal{R}^+$.

As usual, if ϕ is a sentence (a formula with no free logical variables), then whether $x, q \models \phi$ holds is independent of q, and we may write $x \models \phi$ without ambiguity. We say that a sentence ϕ is *valid*, and we write $\models \phi$ if $x \models \phi$ holds for all V-histories x.

By interpreting sentences of $\mathcal{T}(V)$ over "continuous" histories, rather than discrete sequences as is usually done, we obtain easily the following result. It is crucial in what follows.

Lemma 3 *(Projection Lemma) Suppose V and U are sets of variables, with $U \subseteq V$. If ϕ is a sentence of $\mathcal{T}(U)$ and x is a V-history, then*

$$\pi_U(x) \models \phi \quad \text{iff} \quad x \models \phi,$$

where the satisfaction on the left is taken in $\mathcal{T}(U)$ and that on the right is taken in $\mathcal{T}(V)$.

Proof – By induction on formulas – Straightforward. ∎

Definition 4 *(Conceptual State Specification) A* conceptual state specification *is a three-tuple $S = (V, C, \phi)$, where V is a set of* interface variables, *C is a set of* conceptual state variables *disjoint from V, and ϕ is a sentence of the temporal language $\mathcal{T}(V \cup C)$.*

A V-process P satisfies *a conceptual state specification $S = (V, C, \phi)$ (in which case we write $P \models S$) if to each V-history $x \in P$ there corresponds a C-history y such that $(x \sqcup y) \models \phi$.*

Thus, a process satisfies a conceptual state specification iff every history in the process can be augmented or "explained" by the addition of a history for the conceptual state variables, in such a way that the temporal sentence in the specification is satisfied.

Lemma 4 *Suppose $S = (V, C, \phi)$ is a conceptual state specification. Suppose P, P' are V-processes such that $P \subseteq P'$. If $P' \models S$, then $P \models S$.*

Proof – Obvious. ∎

4 Example: A Buffer Specification

As a concrete example of a conceptual state specification, we treat the problem of specifying the behavior of a process that behaves as an unbounded FIFO buffer. Later we shall consider the problem of proving that the tandem connection of two FIFO buffers is again a FIFO buffer. This example, although trivial from a practical point of view, nevertheless exhibits most of the interesting theoretical issues.

4.1 Informal Buffer Specification

A buffer process has two interface variables: **in** and **out**, which we assume take their values in the set $A \cup \{\perp\}$. The set A is the set of values that the buffer process is capable of buffering, and $\perp$ is a special value, denoting undefined, which plays an important role in the protocol by which the buffer process communicates with its environment. The variable **in** is used for receiving values to be buffered from a producer process, and the variable **out** is used for outputting values to a consumer process.

The behavior of a buffer process can be described with the help of a single conceptual state variable **queue**, whose values are finite sequences of elements of A, representing the sequence of values stored in the buffer. The specification is divided into three parts: a part describing the *initial conditions* that hold at the start of execution, a part describing *state-transition* information, which is concerned with the step-by-step evolution of the values of the variables, and a part describing *liveness properties*.

For the buffer process, the initial conditions state merely that the value of **queue** is the empty sequence.

It is convenient to organize the state-transition part of the buffer specification by classifying each state transition that can occur as an instance of a certain kind of *event*. There are three kinds of events that can occur during the execution of a buffer process. The first kind of event is an *input* event in which a value, say a, is read from the external variable **in**, the value of **in** is reset to $\perp$, and the value of the conceptual state variable **queue** is changed by appending a at the end. When an input event occurs, the variable **out** is not accessed by the buffer process.

The second kind of event is an *output* event in which a value, say a, is removed from the conceptual queue, and written into the output variable **out**. It is required that the variable **out** have value $\perp$ before an output event can occur. When an output event occurs, the variable **in** is not accessed by the buffer process.

The third kind of event is an *environment* event, which can occur at any time, in which any change at all to the variables **in** and **out** is permitted, but in which the conceptual state variable **queue** does not change. The buffer process does not access the variables **in** and **out** during an environment event — rather, such an event represents a possible access of these variables by the environment of the buffer process.

There are two liveness conditions that must be satisfied by a buffer process: one associated with the assimilation of input values, and one associated with the production of output values. The first liveness condition states that if the variable **in** assumes a non-$\perp$ value, and retains this value for a sufficiently long time, then eventually an input event will occur. The second liveness condition states that if the internal queue of the buffer process is ever nonempty, and the variable **out** assumes the value $\perp$ and retains this value for a sufficiently long time, then eventually an output event will occur. Together these conditions ensure that the buffer process eventually transmits values from producer to consumer, if possible.

4.2 Formal Buffer Specification

The informal description of the behavior of a buffer process given above can be formalized as a conceptual state specification:

$$S = (\{\mathbf{in}, \mathbf{out}\}, \{\mathbf{queue}\}, \phi_{\text{buf}}(\mathbf{in}, \mathbf{out}, \mathbf{queue})),$$

where

$$\phi_{\text{buf}}(\mathbf{in}, \mathbf{out}, \mathbf{queue}) \equiv \gamma_{\text{buf}}(\mathbf{in}, \mathbf{out}, \mathbf{queue}) \wedge \tau_{\text{buf}}(\mathbf{in}, \mathbf{out}, \mathbf{queue}) \wedge \lambda_{\text{buf}}(\mathbf{in}, \mathbf{out}, \mathbf{queue}),$$

$$\begin{aligned}
\gamma_{\text{buf}}(i, o, q) &\equiv \underline{q} = \langle\rangle \\
\tau_{\text{buf}}(i, o, q) &\equiv \Box(\text{Ievent}(i, o, q) \vee \text{Oevent}(i, o, q) \vee \text{Nevent}(i, o, q)) \\
\lambda_{\text{buf}}(i, o, q) &\equiv \Box(\Box(\underline{i} \neq \perp) \supset \Diamond\text{Ievent}(i, o, q)) \\
&\qquad \wedge \Box(\underline{q} \neq \langle\rangle \wedge \Box(\underline{o} = \perp) \supset \Diamond\text{Oevent}(i, o, q)),
\end{aligned}$$

and

$$\begin{aligned}
\text{Ievent}(i, o, q) &\equiv \widehat{\imath} \wedge \neg\widehat{o} \wedge \underline{i} \neq \perp \wedge \overline{\imath} = \perp \wedge \overline{q} = \underline{q} \cdot \underline{i} \\
\text{Oevent}(i, o, q) &\equiv \widehat{o} \wedge \neg\widehat{\imath} \wedge \underline{q} \neq \langle\rangle \wedge \underline{o} = \perp \\
&\qquad \wedge \overline{q} = \mathbf{tail}(\underline{q}) \wedge \overline{o} = \mathbf{head}(\underline{q}) \\
\text{Nevent}(i, o, q) &\equiv \neg\widehat{\imath} \wedge \neg\widehat{o} \wedge \overline{q} = \underline{q}.
\end{aligned}$$

The functions **head** and **tail** are the obvious functions on sequences.

The sentence $\gamma_{\text{buf}}(\mathbf{in}, \mathbf{out}, \mathbf{queue})$ expresses the condition that the conceptual state variable **queue** has the empty sequence as its value in an initial state.

The sentence $\tau_{\text{buf}}(\mathbf{in}, \mathbf{out}, \mathbf{queue})$ asserts that for all instants of time, an event appearing in a history for a buffer process must either be an input event, an output event, or an environment event, as discussed above.

The sentence $\lambda_{\text{buf}}(\mathbf{in}, \mathbf{out}, \mathbf{queue})$ expresses the liveness properties mentioned above.

The lengthiness of the specification is primarily due to the fact that we have to say explicitly when a variable is unchanged as a result of an event, as well as how the values of variables change. For more complex processes, it is useful to introduce notational conventions to shorten the state-transition part of a specification. Lamport's "Allowed Changes" notation ([Lam83]) is an example of the kind of abbreviations that can be made.

5 Proving Entailment

In this section, we define the notion of entailment between conceptual state specifications, and consider the problem of how to prove that a conceptual state specification $S = (V, C, \phi)$ entails a conceptual state specification $S' = (V, C', \phi')$.

Definition 5 *(Entailment) A conceptual state specification* $S = (V, C, \phi)$ entails *a conceptual state specification* $S' = (V, C', \phi')$ *(and we write* $S \models S'$*) if every process that satisfies* S *also satisfies* S'*.*

Intuitively, one would expect it to be possible to perform a proof that $S \models S'$ by proving a certain implication in temporal logic. The evident implication is $\phi \supset \phi'$. Although the validity of this implication (taken in the language $\mathcal{T}(V \cup C \cup C')$) is sufficient to imply that $S \models S'$, it is not necessary, and in fact is much too strong a condition to be useful in practice. The reason is that knowing ϕ holds of a $(V \cup C \cup C')$-history tells us nothing about the relationship between the values of the C-variables and the values of the C'-variables. This relationship will clearly be important, in general, for proving that ϕ' holds. We would like to find weaker sufficient conditions for $S \models S'$, which if not necessary, are at least of practical utility.

In this section we show that it is in fact sufficient to find temporal sentences μ_M and $\mu_{M'}$ corresponding to "machines" M and M', such that the temporal implications $\phi \supset \mu_M$ and $\mu_{M'} \wedge$

$\phi \wedge \rho \supset \phi'$ are valid, where ρ is a temporal sentence derived from a "simulation relation" from M to M', which expresses the correspondence between the states of M and those of M'. The simulation relation ρ is a generalization of, and serves a purpose similar to, the *abstraction functions* or *representation functions* used in proofs of implementation relationships between abstract data types ([GHM78], [Hoa72], [Jon81]).

Lemma 5 *Suppose $S = (V, C, \phi)$ and $S' = (V, C', \phi')$ are conceptual state specifications. Then $S \models S'$ iff to each V-history x and C-history y such that $(x \sqcup y) \models \phi$, there corresponds a C'-history y' such that $(x \sqcup y') \models \phi'$.*

Proof – Suppose to each V-history x and C-history y such that $(x \sqcup y) \models \phi$ there corresponds a C'-history y' such that $(x \sqcup y') \models \phi'$. Assume $P \models S$, then to each V-history $x \in P$ there corresponds a C-history y such that $(x \sqcup y) \models \phi$. By hypothesis, there exists a C'-history y' such that $(x \sqcup y') \models \phi'$. Since this is true for all $x \in P$, it follows that $P \models S'$.

Conversely, suppose $S \models S'$. If x is a V-history, and y is a C-history such that $(x \sqcup y) \models \phi$, then the singleton V-process $P = \{x\}$ satisfies S. Since $S \models S'$, it must also be the case that $P \models S'$. This implies the existence of a C'-history y' such that $(x \sqcup y') \models \phi'$. ▮

Next, we define the kind of nondeterministic machine that will be used in our entailment proof technique. Such a machine consists of an "initial state relation," which specifies the states in which computation is permitted to start, and a "state transition relation," which specifies the events that are permitted to occur. Condition (1) of the definition below says that a machine must have an initial state corresponding to any given assignment of values to interface variables. Condition (2) says that it is always possible for a machine to execute a null event (*i.e.* do nothing). Condition (3) is a technical condition which ensures that access information for conceptual state variables is essentially irrelevant in a computation. We impose this condition because conceptual state variables are unimplemented auxiliary variables, for which access information is meaningless.

Definition 6 *(Machine) Suppose V and C are disjoint finite sets of variables. A (V, C)-*machine *is a pair $M = (\gamma_M, \tau_M)$, where γ_M is a unary relation on $(V \cup C)$-states, called the* initial state relation *and τ_M is a unary relation on $(V \cup C)$-events, called the* transition relation*, such that the following conditions are satisfied:*

1. *For all V-states q, there exists a C-state r such that $\gamma_M(q \sqcup r)$ holds.*

2. *For all V-states q and all C-states r, $\tau_M(q \sqcup r, \emptyset, q \sqcup r)$ holds.*

3. *For all V-states q_0, q_1, all C-states r_0, r_1, and all subsets U, U' of $V \cup C$, if $U \cap V = U' \cap V$, then $\tau_M(q_0 \sqcup r_0, U, q_1 \sqcup r_1)$ holds iff $\tau_M(q_0 \sqcup r_0, U', q_1 \sqcup r_1)$ holds.*

The temporal sentence corresponding to *a (V, C)-machine $M = (\gamma_M, \tau_M)$ is the sentence μ_M of $\mathcal{T}(V \cup C)$ defined by*

$$\mu_M \equiv \gamma_M(\underline{V} \cup \underline{C}) \wedge \Box \tau_M(\underline{V} \cup \underline{C}, \widehat{V} \cup \widehat{C}, \overline{V} \cup \overline{C}).$$

Here $\gamma_M(\underline{V} \cup \underline{C})$ denotes the formula, involving the terms $\underline{v}$ for each $v \in V \cup C$, corresponding to the initial state relation γ_M, and $\tau_M(\underline{V} \cup \underline{C}, \widehat{V} \cup \widehat{C}, \overline{V} \cup \overline{C})$ denotes the formula, involving the terms $\underline{v}, \overline{v}$ for each $v \in V \cup C$, and the predicates $\widehat{v}$ for each $v \in V \cup C$, corresponding to the transition relation τ_M.

A computation *of a (V, C)-machine M is a $(V \cup C)$-history $x \sqcup y$ such that $(x \sqcup y) \models \mu_M$.*

We next define the notion of a "simulation" from a machine M, with interface variables V and conceptual state variables C, to a machine M' with the same set of interface variables, but a disjoint set C' of conceptual state variables. Intuitively, a simulation relates states of M to corresponding states of M', so that the initial state and state transition relations are preserved in a certain fashion.

Definition 7 *(Simulation) Suppose M is a (V, C)-machine, and M' is a (V, C')-machine, where $C \cap C' = \emptyset$. A* simulation *from M to M' is a relation $\rho \subseteq V$-states $\times C$-states $\times C'$-states, such that the following hold:*

1. *For all V-states p and C-states q, if $\gamma_M(p \sqcup q)$ holds, then there exists a C'-state q' such that $\gamma_{M'}(p \sqcup q')$ and $\rho(p, q, q')$ hold.*

2. *For all V-states p, p', C-states q, q', all $U \subseteq V$, and all C'-states r, if $\rho(p, q, r)$ and $\tau_M(p \sqcup q, U, p' \sqcup q')$ hold, then there exists a C'-state r' such that $\rho(p', q', r')$ and $\tau_{M'}(p \sqcup r, U, p' \sqcup r')$ hold.*

The following is the main technical lemma used in the proof of the Entailment Theorem below. Intuitively, it says that the existence of a simulation from M to M' ensures that for each computation of M we can obtain a computation of M', in such a way that the two computations can be combined into a single "joint computation" for which the simulation relation invariantly holds.

Lemma 6 *Suppose M is a (V, C)-machine and M' is a (V, C')-machine. Suppose ρ is a simulation from M to M'. Then to each computation $x \sqcup y$ of M, there corresponds a computation $x \sqcup y'$ of M', such that*

$$(x \sqcup y \sqcup y') \models \Box\rho(\underline{V}, \underline{C}, \underline{C'}) \wedge \Box\rho(\overline{V}, \overline{C}, \overline{C'}).$$

Proof – Suppose $x \sqcup y$ is a given computation of M. By Lemma 2, there exists an infinite locally finite set $T = \{t_0 < t_1 < \ldots\}$ with $t_0 = 0$, a sequence $p_0, p_1, \ldots$ of V-states, a sequence $q_0, q_1, \ldots$ of C-states, and a sequence $U_0, U_1, \ldots$ of subsets of $(V \cup C)$, such that $x(t_k) = (p_k \sqcup q_k, U_k, p_{k+1} \sqcup q_{k+1})$ for all k, and $x(t) = (p_{k+1} \sqcup q_{k+1}, \emptyset, p_{k+1} \sqcup q_{k+1})$ for all k and all $t \in (t_k, t_{k+1})$. Since $x \sqcup y$ is a computation of M, we know that $\gamma_M(q_0)$ holds and $\tau_M(q_k, U_k, q_{k+1})$ holds for all k.

It is now a simple matter to construct by induction, using the defining properties of a simulation, a sequence $q'_0, q'_1, \ldots$ of C'-states, such that $\gamma_{M'}(p_0 \sqcup q'_0)$ holds, and $\rho(p_k, q_k, q'_k)$ and $\tau_{M'}(p_k \sqcup q'_k, U_k \cap V, p_{k+1} \sqcup q'_{k+1})$ hold for all k. Define $U'_k = \emptyset$ for all k. Then an application of Lemma 2 to the set T and sequences $U'_0, U'_1, \ldots, q'_0, q'_1, \ldots$ yields a C'-history y' such that $x \sqcup y'$ is the desired computation of M'. ∎

The following result gives our technique for proving entailment between conceptual state specifications.

Theorem 1 *(Entailment Theorem) Suppose $S = (V, C, \phi)$ and $S' = (V, C', \phi')$ are conceptual state specifications, with $C \cap C' = \emptyset$. Suppose we can find a (V, C)-machine M, a (V, C')-machine M', and a simulation ρ from M to M', such that the implications*

$$\phi \supset \mu_M$$

$$\mu_{M'} \wedge \phi \wedge \Box\rho(\underline{V}, \underline{C}, \underline{C'}) \wedge \Box\rho(\overline{V}, \overline{C}, \overline{C'}) \supset \phi'$$

are valid. Then $S \models S'$.

Proof – Suppose M, M', and ρ have the stated properties. By Lemma 5 above, we need only show that to each V-history x and C'-history y such that $(x \sqcup y) \models \phi$, there corresponds a C'-history y' such that $x \sqcup y' \models \phi'$. Given x and y such that $x \sqcup y \models \phi$, we know from the first hypothesized implication that $(x \sqcup y) \models \mu_M$. Since ρ is a simulation, Lemma 6 gives us a y' such that $x \sqcup y' \models \mu_{M'}$ and

$$(x \sqcup y \sqcup y') \models \Box\rho(\underline{V}, \underline{C}, \underline{C'}) \wedge \Box\rho(\overline{V}, \overline{C}, \overline{C'}).$$

By the Projection Lemma,

$$(x \sqcup y \sqcup y') \models \mu_{M'} \wedge \phi \wedge \Box\rho(\underline{V}, \underline{C}, \underline{C}') \wedge \Box\rho(\overline{V}, \overline{C}, \overline{C}').$$

By modus ponens and the second implication assumed valid by hypothesis, it follows that $(x \sqcup y \sqcup y') \models \phi'$. Applying the Projection Lemma again gives us $(x \sqcup y') \models \phi'$. ∎

6 Example: Tandem Connection of Two Buffers

This section illustrates the role played by the Entailment Theorem in a proof that the tandem connection of two FIFO buffers correctly implements a FIFO buffer.

Let

$$\phi_0 \equiv \phi_{\mathrm{buf}}(\mathbf{in}, \mathbf{inout}, \mathbf{queue}_0),$$

$$\phi_1 \equiv \phi_{\mathrm{buf}}(\mathbf{inout}, \mathbf{out}, \mathbf{queue}_1),$$

and

$$\phi_{\mathrm{abs}} \equiv \phi_{\mathrm{buf}}(\mathbf{in}, \mathbf{out}, \mathbf{queue}_{\mathrm{abs}}).$$

Let S be the conceptual state specification

$$(\{\mathbf{in}, \mathbf{out}, \mathbf{inout}\}, \{\mathbf{queue}_0, \mathbf{queue}_1\}, \mathrm{Consis}(\{\mathbf{inout}\}) \wedge \phi_0 \wedge \phi_1),$$

which is the specification satisfied by the tandem connection of two buffer processes. Here the formula $\mathrm{Consis}(U)$, for a finite set of variables U, is given by

$$\mathrm{Consis}(U) \equiv \Box \bigwedge_{v \in U} (\neg\widehat{v} \supset \overline{v} = \underline{v}).$$

Intuitively, the formula $\mathrm{Consis}(\{\mathbf{inout}\})$ states that if the variable **inout** is not accessed in an event by one of the two buffer processes, then its value does not change in that event. This corresponds to the idea that the variable **inout** is an internal variable used for communication between the two buffer processes, and is hidden from access by the external environment.

Let S' be the specification

$$(\{\mathbf{in}, \mathbf{out}, \mathbf{inout}\}, \{\mathbf{queue}_{\mathrm{abs}}\}, \phi_{\mathrm{abs}}),$$

which, if satisfied by a $\{\mathbf{in}, \mathbf{out}, \mathbf{inout}\}$-process P, implies that the projection of P to the variable set $\{\mathbf{in}, \mathbf{out}\}$ satisfies the buffer specification.

To prove the correctness of the implementation, we must prove that the entailment $S \models S'$ holds. To apply the Entailment Theorem, we must determine the machines M and M', find a simulation ρ from M to M', and prove the validity of two implications in temporal logic.

The factorization of the buffer specification into initial conditions, state-transition conditions, and liveness conditions obviously suggests an M and M'. Define

$$\begin{aligned} \gamma_M &\equiv \gamma_{\text{buf}}(\mathbf{in}, \mathbf{inout}, \mathbf{queue}_0) \wedge \gamma_{\text{buf}}(\mathbf{inout}, \mathbf{out}, \mathbf{queue}_1) \\ \tau_M &\equiv \tau_{\text{buf}}(\mathbf{in}, \mathbf{inout}, \mathbf{queue}_0) \wedge \tau_{\text{buf}}(\mathbf{inout}, \mathbf{out}, \mathbf{queue}_1) \wedge \text{Consis}(\{\mathbf{inout}\}) \end{aligned}$$

$$\begin{aligned} \gamma_{M'} &\equiv \gamma_{\text{buf}}(\mathbf{in}, \mathbf{out}, \mathbf{queue}_{\text{abs}}) \\ \tau_{M'} &\equiv \tau_{\text{buf}}(\mathbf{in}, \mathbf{out}, \mathbf{queue}_{\text{abs}}). \end{aligned}$$

It is straightforward to check that M and M' are, in fact, machines, and that the implications

$$\text{Consis}(\{\mathbf{inout}\}) \wedge \phi_0 \wedge \phi_1 \supset \mu_M$$

and

$$\phi_{\text{abs}} \supset \mu_{M'}$$

are valid.

Next, we must define a relation ρ, and show that it is a simulation from M to M'. The appropriate ρ is the one that says that the abstract queue is the concatenation of the two component queues, with the value of **inout** in between, if that value is not $\perp$.

$$\begin{aligned} \rho(\mathbf{inout}, \mathbf{queue}_{\text{abs}}, \mathbf{queue}_0, \mathbf{queue}_1) \equiv\ & (\mathbf{inout} = \perp \supset \mathbf{queue}_{\text{abs}} = \mathbf{queue}_0 \cdot \mathbf{queue}_1) \wedge \\ & (\mathbf{inout} \neq \perp \supset \mathbf{queue}_{\text{abs}} = \mathbf{queue}_0 \cdot \mathbf{inout} \cdot \mathbf{queue}_1). \end{aligned}$$

The proof that ρ is a simulation involves a case analysis based on the different possible combinations of input, output, and environment events that are permitted by the transition relations τ_M and $\tau_{M'}$. The details are straightforward but tedious, and are omitted. In general, the construction of the simulation ρ is the part of the proof that requires insight; once this relation has been constructed, the enumeration of the various cases in the conditions required for ρ to be a simulation, and the construction of the proof for each case, are systematic tasks that are within the ability of automatic or semi-automatic theorem proving programs.

To complete the proof of correctness of the buffer implementation, we must prove the implication

$$(\mu_{M'} \wedge \mathrm{Consis}(\{\mathbf{inout}\}) \wedge \phi_0 \wedge \phi_1 \wedge \Box\rho(\underline{\mathbf{inout}}, \underline{\mathbf{queue}}_0, \underline{\mathbf{queue}}_1, \underline{\mathbf{queue}}_{\mathrm{abs}}) \wedge \\ \Box\rho(\overline{\mathbf{inout}}, \overline{\mathbf{queue}}_0, \overline{\mathbf{queue}}_1, \overline{\mathbf{queue}}_{\mathrm{abs}})) \supset (\mu_{M'} \wedge \phi_{\mathrm{abs}}),$$

or equivalently,

$$(\mu_M \wedge \mu_{M'} \wedge \Box\rho(\underline{\mathbf{inout}}, \underline{\mathbf{queue}}_0, \underline{\mathbf{queue}}_1, \underline{\mathbf{queue}}_{\mathrm{abs}}) \wedge \Box\rho(\overline{\mathbf{inout}}, \overline{\mathbf{queue}}_0, \overline{\mathbf{queue}}_1, \overline{\mathbf{queue}}_{\mathrm{abs}})) \\ \supset (\lambda_{\mathrm{buf}}(\mathbf{in}, \mathbf{inout}, \mathbf{queue}_0) \wedge \lambda_{\mathrm{buf}}(\mathbf{inout}, \mathbf{out}, \mathbf{queue}_1) \supset \lambda_{\mathrm{buf}}(\mathbf{in}, \mathbf{out}, \mathbf{queue}_{\mathrm{abs}})).$$

The intuitive content of this implication is that every "joint computation" of M and M', whose "M-part" satisfies the specification for the tandem connection of two FIFO buffers, and whose M-part and M'-part are related by the simulation relation ρ, also has the property that its M'-part satisfies the specification of a FIFO buffer.

This proof can be performed, for example, by the proof lattice techniques of Owicki and Lamport [OL82]. We omit the details.

7 A Completeness Result

The sufficient conditions given by the Entailment Theorem for proving an entailment are not necessary, in general. However, if we assume the specifications involved satisfy certain well-formedness conditions, we can show that the proof technique given by the Entailment Theorem is complete in the sense that a proof can always be found when an entailment holds.

Definition 8 *(Regularity) Suppose M is a (V,C)-machine, and λ is a sentence of $\mathcal{T}(V \cup C)$. We say that λ is* regular *with respect to M if for all computations $x \sqcup y, x \sqcup y'$ of M, $x \sqcup y \models \lambda$ iff $x \sqcup y' \models \lambda$.*

Intuitively, if λ is regular with respect to M, then whether a computation $x \sqcup y$ of M satisfies λ depends only upon x, and not upon the particular choice of history for the conceptual state variables.

Definition 9 *(Quasi-determinacy) Suppose M is a (V, C)-machine. We say that M is* quasi-determinate *when for all computations $x \sqcup y, x' \sqcup y'$ of M, if $x(t) = x'(t)$ for all $t \in [0, n)$, then there exists a computation $x \sqcup y''$ of M, with $y''(t) = y'(t)$ for all $t \in [0, n)$.*

Intuitively, for a quasi-determinate machine, the particular choice of conceptual state history made on an initial segment of a computation does not affect whether or not that computation can be completed to generate a particular history x for the interface variables.

Definition 10 *(Density) Suppose M is a (V, C)-machine, and λ is a sentence of $\mathcal{T}(V \cup C)$. We say that λ is* dense in M *if the following property holds: For all computations $x \sqcup y$ of M and all $n \in \mathcal{R}^+$, there exists a V-history x' and a C-history y' such that $(x' \sqcup y') \models \mu_M \wedge \lambda$ and such that $(x \sqcup y)(t) = (x' \sqcup y')(t)$ for all $t \in [0, n]$.*

Intuitively, λ is dense in M if every computation of M is arbitrarily close (w.r.t. a metric that measures the length of agreement of prefixes) to a computation of M that satisfies λ.

Theorem 2 *(Completeness Theorem) Suppose*

$$S = (V, C, \mu_M \wedge \lambda) \text{ and } S' = (V, C', \mu_{M'} \wedge \lambda')$$

are conceptual state specifications, with C, C' disjoint. Suppose that λ is dense in M, and that λ' is regular with respect to the quasi-determinate machine M'. If $S \models S'$, then there exists a simulation ρ from M to M', and the implication

$$\mu_{M'} \wedge \mu_M \wedge \lambda \supset \lambda'$$

is valid.

Proof – Suppose $M = (\gamma, \tau)$ and $M' = (\gamma', \tau')$. We first show that $\mu_{M'} \wedge \mu_M \wedge \lambda \supset \lambda'$ is valid. To show this, suppose that x is a V-history, y is a C-history, and y' is a C'-history, such that

$$(x \sqcup y \sqcup y') \models \mu_{M'} \wedge \mu_M \wedge \lambda.$$

Then $(x \sqcup y) \models \mu_M \wedge \lambda$ and $(x \sqcup y') \models \mu_{M'}$ by the Projection Lemma. By Lemma 5 and the assumption that $S \models S'$, there exists a C'-history y'' such that $(x \sqcup y'') \models \mu_{M'} \wedge \lambda'$. The regularity of λ' with respect to M' implies that $(x \sqcup y') \models \lambda'$ iff $(x \sqcup y'') \models \lambda'$, so we conclude that $(x \sqcup y') \models \lambda'$. It follows by the Projection Lemma that $(x \sqcup y \sqcup y') \models \lambda'$.

It remains to prove the existence of the required simulation ρ from M to M'. Define a pair $(p \sqcup q, p \sqcup r)$, where p is a V-state, q is a C-state and r is a C'-state, to be *jointly reachable* according to the following inductive definition:

1. If $\gamma_M(p \sqcup q)$ and $\gamma_{M'}(p \sqcup r)$ both hold, $(p \sqcup q, p \sqcup r)$ is jointly reachable.

2. If $(p \sqcup q, p \sqcup r)$ is jointly reachable, and $\tau_M(p \sqcup q, U, p' \sqcup q')$ and $\tau_{M'}(p \sqcup r, U, p' \sqcup r')$ hold for some $U \subseteq V$, then $(p' \sqcup q', p' \sqcup r')$ is jointly reachable.

If p is a V-state, q is a C-state, and q' is a C'-state, then define $\rho(p, q, r)$ to hold iff the pair $(p \sqcup q, p \sqcup r)$ is jointly reachable.

We claim that ρ is a simulation from M to M'. To show this, we must show two things:

1. For all V-states p and C-states q, if $\gamma_M(p \sqcup q)$ holds, then there exists a C'-state q' such that $\gamma_{M'}(p \sqcup q')$ and $\rho(p, q, q')$ hold.

2. For all V-states p, p', all C-states q, q', all $U \subseteq V$, and all C'-states r, if $\rho(p, q, r)$ and $\tau_M(p \sqcup q, U, p' \sqcup q')$ hold, then there exists a C'-state r' such that $\rho(p', q', r')$ and $\tau_{M'}(p \sqcup r, U, p' \sqcup r')$ hold.

To show 1, suppose p is a V-state and q is a C-state such that $\gamma_M(p \sqcup q)$ holds. Then since M' is a machine, there exists a C'-state r such that $\gamma_{M'}(p \sqcup r)$ holds. Since the pair $(p \sqcup q, p \sqcup r)$ is jointly reachable, it follows that $\rho(p, q, r)$ holds.

To show 2, suppose $\rho(p, q, r)$ and $\tau_M(p \sqcup q, U, p' \sqcup q')$ hold. Then by definition of ρ the pair $(p \sqcup q, p \sqcup r)$ is jointly reachable. We can therefore obtain a sequence $U_0, U_1, \ldots, U_{n-1}$ of subsets of V, a sequence $p_0, p_1, \ldots, p_n$ of V-states, a sequence $q_0, q_1, \ldots, q_n$ of C-states, and a sequence $r_0, r_1, \ldots, r_n$ of C'-states such that $p_n = p$, $q_n = q$, $r_n = r$, $\gamma_M(p_0 \sqcup q_0)$ and $\gamma_{M'}(p_0 \sqcup r_0)$ hold, $\tau_M(p_k \sqcup q_k, U_k, p_{k+1} \sqcup q_{k+1})$ and $\tau_{M'}(p_k \sqcup r_k, U_k, p_{k+1} \sqcup r_{k+1})$ hold for all k with $0 \leq k \leq n-1$, and $\rho(p_k, q_k, r_k)$ holds for all k with $0 \leq k \leq n$.

Extend the sequences p_i and q_i to infinity by defining $p_i = p'$ and $q_i = q'$ for all $i > n$. Extend the sequence U_i to infinity by defining $U_n = U$ and $U_i = \emptyset$ for all $i > n$. Extend the sequence r_i to infinity by defining $r_i = r_n$ for all $i > n$. Define the sequences p'_i and U'_i so that $p'_i = p_i$ and

$U_i' = U_i$ for $0 \leq i < n$, and $p_i' = p_n$ and $U_i' = \emptyset$ for $i \geq n$. Then $\tau_M(p_k \sqcup q_k, U_k, p_{k+1} \sqcup q_{k+1})$ and $\tau_{M'}(p_k' \sqcup r_k, U_k', p_{k+1}' \sqcup r_{k+1})$ hold for all k.

Let $T = \{0, 1, 2, \ldots\}$. Then by Lemma 2, T and the sequences U_k, p_k, and q_k uniquely determine a V-history x_0 and a C-history y_0 such that $x_0(k) \sqcup y_0(k) = (p_k \sqcup q_k, U_k, p_{k+1} \sqcup q_{k+1})$ for all $k \in T$, and $x_0(t) \sqcup y_0(t) = (p_{k+1} \sqcup q_{k+1}, \emptyset, p_{k+1} \sqcup q_{k+1})$ for all $k \in T$ and all $t \in (t_k, t_{k+1})$. Similarly, the sequences U_k', p_k', and r_k uniquely determine a V-history x_0' and a C'-history y_0', which have the additional property that $(x_0' \sqcup y_0')(t) = (x_0 \sqcup y_0)(t)$ for all $t \in [0, n)$. By construction, $(x_0 \sqcup y_0) \models \mu_M$ and $(x_0' \sqcup y_0') \models \mu_{M'}$.

Intuitively, the computation $x_0 \sqcup y_0$ of M is a computation that begins in an initial state, reaches the state $p \sqcup q$ before time n, performs the event $(p \sqcup q, U, p' \sqcup q')$ at time n, and then subsequently performs null events. The computation $x_0' \sqcup y_0'$ of M' is a computation that begins in an initial state, reaches the state $p \sqcup r$ before time n, in a way that is related by ρ to the way in which $p \sqcup q$ is reached by M, and performs null steps subsequently. What we wish to show is that M' can perform a transition to the state $p' \sqcup r'$ at time n, corresponding to the transition to the state $p' \sqcup q'$ that M performs at time n in $x_0 \sqcup y_0$.

By the assumption that λ is dense in M, there exists a $(V \cup C)$-history $(x_1 \sqcup y_1)$ such that $x_1(t) \sqcup y_1(t) = x_0(t) \sqcup y_0(t)$ for all $t \in [0, n]$, and such that $(x_1 \sqcup y_1) \models \mu_M \wedge \lambda$. This implies that the singleton process $\{x_1\}$ satisfies S. By the assumption that $S \models S'$, $\{x_1\} \models S'$, and hence there exists a C'-history y_1' such that $(x_1 \sqcup y_1') \models \mu_{M'} \wedge \lambda'$.

We now know that $(x_1 \sqcup y_1') \models \mu_{M'}$, $(x_0' \sqcup y_0') \models \mu_{M'}$, and $x_1(t) = x_0'(t)$ for all $t \in [0, n)$. By the quasi-determinacy of M', there exists a C'-history y_2', such that $(x_1 \sqcup y_2') \models \mu_{M'}$ and $y_2'(t) = y_0'(t)$ for $t \in [0, n)$.

Since $(x_1 \sqcup y_2') \models \mu_{M'}$, it follows that $\tau_{M'}(x_1(n) \sqcup y_2'(n))$ holds. Since $\underline{x}_1(n) = p$, $\overline{x}_1(n) = p'$, $\widehat{x_1}(n) = U$, and $\underline{y}_2'(n) = r$, it follows that $\overline{y}_2'(n)$ has the property that $\tau_M(p \sqcup r, U, p' \sqcup \overline{y}_2'(n))$ holds, and hence is the desired state r'. ∎

The previous result is not the strongest possible completeness result that one might wish for, since it does not assert the existence of the machines M and M'. Rather, it assumes that the specifications have been presented in such a way that M and M' are made explicit. A better completeness result would show that it is possible to choose M to be a "smallest" machine such that $\lambda \supset \mu_M$ is valid, and M' to be a certain "maximal" machine for λ'. Such a result is apparently not true for the general setup considered in this paper. For example, it can be shown that to each temporal sentence λ that satisfies a "strong satisfiability" condition, there corresponds a "smallest" machine M such that $\models \lambda \supset \mu_M$. However, it is not always the case that λ is dense in M. To obtain a "maximal" machine M' corresponding to λ', such that M' is quasi-determinate and λ' is regular with respect to M' seems at least as problematic. Perhaps, though, by imposing suitable restrictions on the temporal specification language, a result along these lines could be obtained. Alpern and Schneider [AS85] have obtained similar completeness results in a setup where temporal properties are specified as "property recognizers," which are similar to Büchi automata.

8 Summary

We have introduced the notion of a conceptual state specification, which is a kind of temporal logic specification in which conceptual state variables are introduced to increase the expressive power of the temporal language. We have defined the notion of entailment between conceptual state specifications, and have obtained a proof technique for establishing the entailment relationship. The proof technique can be viewed as a generalization of standard techniques for proving the correctness of implementations of abstract data types. We showed that, if the specifications involved are assumed to satisfy certain well-formedness conditions, then true entailment relations can always be established by our technique. The use of the technique was illustrated by a simple example.

The combination of the Entailment Theorem and the Completeness Theorem above suggests a disciplined approach to the use of conceptual state specifications. In particular, a conceptual state specification ought to be presented in "factored" form, where one factor (conjunct) is the temporal sentence corresponding to a machine, and the other factor expresses liveness properties (which cannot be expressed in machine form). Furthermore, the machine and liveness properties should be quite tightly related in that the liveness properties are regular with respect to, and dense in, the machine. Finally, the machine part should be quasi-determinate, which means that it does not permit very much freedom in the choice of the conceptual state history corresponding to any given history for the interface variables.

We are therefore led to the following:

Definition 11 *(Well-Formed) A* well-formed *conceptual state specification is of the form S =*

$(V, C, \mu_M \wedge \lambda)$, *where* λ *is regular with respect to, and dense in, the quasi-determinate machine* M.

Note that well-formedness is a property that can be verified about a specification in isolation, and it is possible to show the buffer specifications of the previous section to be well-formed. In general, the behaviors of many processes are naturally specified by well-formed conceptual state specifications. Exceptions are processes like a lossy buffer process, which behaves like a FIFO buffer except that it is permitted to lose values. Such a process is naturally specified by a conceptual state specification much like the lossless buffer specification presented here, except the occurrence of an input event can result either in no change to the internal queue (the value is lost), or to be added to the end of the queue (the value is destined for transmission). The machine derived from this specification is not quasi-determinate.

Acknowledgement

The author wishes to thank Professor Nancy Lynch for her support and guidance during his thesis research, and Prateek Mishra for commenting on a draft of this paper. The author is also grateful to Professor Albert Meyer for suggesting stylistic improvements to [Sta84] that contributed substantially to the presentation in this paper.

References

[AS85] B. Alpern and F. B. Schneider. *Verifying Temporal Properties Without Using Temporal Logic.* Technical Report TR 85-723, Cornell University Computer Science Department, December 1985.

[BK83] H. Barringer and R. Kuiper. A temporal logic specification method supporting hierarchical development. November 1983. Manuscript, University of Manchester Department of Computer Science.

[BK84] H. Barringer and R. Kuiper. Now you may compose temporal specifications. In *Proceedings of the Sixteenth ACM Symposium on Theory of Computing*, pages 51–63, April 1984.

[GHM78] J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. *Communications of the ACM*, 1048–1064, December 1978.

[Gor81] J. A. Goree. *Internal Consistency of a Distributed Transaction System with Orphan Detection.* Technical Report MIT/LCS/TR-286, M.I.T. Laboratory for Computer Science, 1981.

[HO80] B. T. Hailpern and S. S. Owicki. *Verifying Network Protocols Using Temporal Logic.* Technical Report 192, Stanford University Computer Systems Laboratory, June 1980.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[Jon81] C. B. Jones. *Development Methods for Computer Programs Including a Notion of Interference.* PhD thesis, Wolfson College, June 1981.

[Lam80] L. Lamport. “sometime” is sometimes “not never”. In *Seventh ACM Conference on Principles of Programming Languages*, 1980.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[Lyn83] N. A. Lynch. Concurrency control for resilient nested transactions. In *ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, March 1983.

[MP83] Z. Manna and A. Pnueli. *Verification of Concurrent Programs: A Temporal Proof System.* Technical Report STAN-CS-83-967, Stanford University, jun 1983.

[OL82] S. S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[Pnu77] A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, 1977.

[SM81] R. L. Schwartz and P. M. Melliar-Smith. Temporal logic specification of distributed systems. In *Second International Conference on Distributed Systems*, INRIA, France, April 1981.

[Sta84] E. W. Stark. *Foundations of a Theory of Specification for Distributed Systems.* Technical Report MIT/LCS/TR-342, M. I. T. Laboratory for Computer Science, August 1984.

[Wol81] P. Wolper. Temporal logic can be more expressive. In *22nd Annual Symposium on Foundations of Computer Science*, pages 340–347, 1981.

[Yon77] A. Yonezawa. *Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics.* Technical Report MIT/LCS/TR-191, M.I.T. Laboratory for Computer Science, December 1977.