Protocol Specification, Testing, and Verification VIII
S. Aggarwal and K. Sabnani (Editors)
Elsevier Science Publishers B.V. (North-Holland)
© IFIP, 1988

A Relational Notation for State Transition Systems*

Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

A. Udaya Shankar
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742

The basic building blocks for specifying systems in the relational notation are state variables and events. Each event is a binary relation on the system state space, and is specified by a formula in a predicate logic notation. We strived for a notation that is easy to learn so that it will be widely accessible to protocol engineers. Our proof method, based on a fragment of linear-time temporal logic, was also designed to be easy to learn and understand.

Using the relational notation, we present a theory of refinement and projection of systems. *Refinement* and *projection* are relations defined between state transition systems. These relations have been applied to simplify the verification of multifunction protocols, to derive specifications and proofs of systems hierarchically, and to address the protocol conversion problem. Several relations between systems defined by other authors, *A simulates B*, *A implements B*, and *A is a superposition of B*, where *A* and *B* denote systems, are compared with refinement and projection.

1. Introduction

The concepts of *state* and *state transition* are fundamental to many formalisms for the specification and analysis of systems. The situation in a physical system can usually be described by the values of a set of variables. For example, a state description of a spacecraft would include its spatial coordinates, its mass, and its velocity, among others. The state of a physical system changes in the course of time. Such changes are called state transitions. In modeling such a system, the state description is assumed to contain sufficient information such that the future behavior of the system is determined only by its present state, and not its past history. (Such models are said to be Markovian.)

The most general state transition systems would allow states described by continuous variables. We shall not be that general. Let us consider state transition systems specified by a pair (S, T), where S is a finite or countably infinite set of states and T is a binary relation on S. Each element of T defines a state transition. The system is *deterministic* if T is a function or a partial function; that is, for each state s_i in S there is at

*The work of Simon S. Lam was supported by National Science Foundation under grant no. NCR-8613338. The work of A. Udaya Shankar was supported by National Science Foundation under grant no. ECS-8502113.

most one state s_j in S such that (s_i,s_j) is in T. Otherwise, the system is *nondeterministic.* Given (S,T) and an initial condition on the system state, a sequence of states $<s_0,s_1, \cdots>$ is said to be a *path* if s_0 satisfies the initial condition and, for $i \geq 0$, (s_i,s_{i+1}) is in T.

The sets S and T are adequate for the specification and analysis of some systems, e.g., connection management protocols for CCITT Recommendations X.21 and X.25. For most systems, however, their specification, and possibly their analysis also, would benefit from having some *structure* in S and in T. Towards this goal, many models and notations have been developed using state transition systems as their foundation. We name just a few: difference equations, Markov chains, programming languages, communicating finite state machines (CFSMs), Petri nets, and various temporal logics. Each was designed for a special-purpose application. Some are designed primarily for specification, such as many programming languages [CCITT 85, ISO 85]. Some are designed for the numerical generation of paths, such as difference equations and CFSMs. Some are designed for the calculation of path probabilities and state probabilities from a given set of state transition probabilities, such as Markov chains and stochastic Petri nets. Some are designed for formal reasoning about system behaviors, such as temporal logics and some programming languages [Chandy & Misra 88, Clarke & Emerson 81, Lamport 83, Owicki & Lamport 82, Pnueli 86].

The CFSM model is widely used for protocol specification in practice [IBM 80, Sabnani 86, West 78]. Because CFSMs are analyzed by a numerical generation of all paths, the CFSM model is applicable to the analysis of just a small number of protocols. Why is it widely used by practitioners despite this shortcoming? The answer may be that most engineers (and programmers) are familiar with the concepts of states and state transitions. Thus they are accustomed to operational reasoning based upon state transitions [Piatkowski 86]. The same is not true for reasoning methods of other formalisms. Even for temporal logics, whose underlying models are state transition systems, the need to master a new notation presents a barrier to the majority of engineers. For the formalisms of CSP and CCS, the additional requirement of reasoning with event traces will further restrict the accessibility of these formalisms despite their mathematical elegance [Milner 80, Hoare 85].

We hasten to say that while operational reasoning based upon state transitions allows engineers to grasp intuitively and quickly what a protocol does, it is often unreliable and should be avoided by the protocol's designer in verification.

During the course of our research on modeling time-dependent distributed systems and methods for protocol construction, we developed a notation for specifying state transition systems [Shankar & Lam 84, 87a, 87b]. We believe that our notation is easy to learn because it represents states and state transitions explicitly. Our notation for specifying and proving protocol properties, based upon a fragment of linear-time temporal logic, was also designed to be easy to learn and understand. Proofs of properties presented by a protocol's designer or verifier can be checked in a mechanical fashion by applying a small set of inference rules. We hope that our notation will be widely accessible to protocol engineers.

Note that the task of constructing a protocol to satisfy a given specification is still nontrivial and still requires ingenuity and insight from the protocol's designer. To facilitate this task, we present a theory of refinement and projection of systems. *Refinement* and *projection* are relations between systems. These relations have been applied to simplify the verification of multifunction protocols, to derive specifications and proofs of systems hierarchically, and to reason about the correctness of protocol converters. (See

Section 6 for references.)

2. Notation

A state transition system is specified by a finite set of *state variables*, $\mathbf{v}=(v_1,v_2,\cdots)$, a finite set of *events*, $e_1,e_2,\cdots$, and an initial condition on the system state.

For every state variable, there is a specified domain of allowed values. The system state is represented by the set of values assumed by the state variables. The state space S of the transition system is the cartesian product of the state variable domains.

An event can occur only when the values in $\mathbf{v}$ satisfy the event's enabling condition. The occurrence of the event updates the values of the state variables in one *atomic* action. Let $\mathbf{v}'$ represent the value of the state vector after the event occurrence. We specify each event by stating the relationship between $\mathbf{v}$ and $\mathbf{v}'$. Note that each event defines a relation on the state space S of the transition system. The union of these relations over all events defines T of the transition system.

Instead of using a programming language, we employ a predicate logic notation to specify events. Let p denote a formula and $\mathbf{x}$ a set of variables.[1] We say that p is a formula in $\mathbf{x}$ if the free variables of p are in $\mathbf{x}$.

In our notation, an event e is specified by a formula in $\mathbf{v}\cup\mathbf{v}'$. For convenience, we use e to refer to both the event's name and the formula that specifies it. We adopt the convention that any variable v' in $\mathbf{v}'$ that does not appear in e is not changed by the occurrence of e; that is, the conjunct $v'=v$ is implicit in the event formula. Some examples of event specifications are shown below:

$$e_1 \equiv v_1>2 \wedge v_2' \in \{1,2,5\}$$

$$e_2 \equiv v_1>v_2 \wedge v_2'=5-v_1$$

$$e_3 \equiv v_1>v_2 \wedge v_1+v_2'=5$$

Note that e_2 and e_3 are equivalent; they define the same relation between $\mathbf{v}$ and $\mathbf{v}'$.

The enabling condition of an event e, denoted by $enabled(e)$, is a formula in $\mathbf{v}$. $enabled(e)$ is logically equivalent to $\exists \mathbf{v}'[e]$. Most events encountered in our examples have the following special form:

$$e \equiv enabled(e) \wedge action(e)$$

where $enabled(e)$ is a formula in $\mathbf{v}$ and $action(e)$ is a formula in $\mathbf{v}\cup\mathbf{v}'$. These formulas in the special form must satisfy the following condition: $enabled(e) \Rightarrow \exists \mathbf{v}'[action(e)]$. (Otherwise, part of the enabling condition is specified by $action(e)$.)

Parameters can be included in the definition of events. Let $\mathbf{w}$ be a set of parameters, each with a specified domain of values. Given $\mathbf{w}$, $enabled(e)$ is a formula in $\mathbf{v}\cup\mathbf{w}$, and $action(e)$ is a formula in $\mathbf{v}\cup\mathbf{w}\cup\mathbf{v}'$. For example, event e_3 above can be modified to

$$e_4(N) \equiv v_1>v_2 \wedge v_1+v_2'=N$$

where N is a parameter. Note that for $N=5$, $e_4(5)$ is the same as e_3 above.

The granularity of events in our notation is a design decision. Aside from determining what constitutes atomic actions in the system, it also affects the progress properties of the system. Even for two systems with the same S and the same T, their progress properties may be different because their events are defined differently.

[1]We use *formula* to refer to *well-formed formula*.

In addition to state variables needed to model a system, the state vector **v** may include *auxiliary variables* which are needed for verification only [Owicki & Gries 76]. For example, an auxiliary variable may be needed to record the history of certain event occurrences. Variables in **v** are auxiliary if they satisfy the following: (1) The enabling conditions of events do not depend on values of auxiliary variables. (2) Updates of state variables that are not auxiliary are not affected by values of auxiliary variables.

We use the following example to illustrate two classes of properties of state transition systems that are of interest to us, namely, safety properties and progress properties. Rules for proving these properties are presented in the next section.

Consider the following model of an object moving in two-dimensional space. Imagine that the object is an airplane flying from Austin to Dallas. There are two state variables: x is the horizontal distance from Austin along a straight line between the two cities, and y is the altitude of the airplane. The domain of x is $\{0,1, \cdots ,N\}$ such that $x=0$ indicates that the airplane is at Austin airport and $x=N$ indicates that the airplane is at Dallas airport. The domain of y is the set of natural numbers such that $y=0$ indicates that the airplane is on the ground. The initial condition is $x=0 \wedge y=0$, indicating that the airplane is on the ground at Austin airport. The behavior of the airplane is specified by the following events:

$TakeOff \equiv$	$x=0 \wedge y=0 \wedge x'=1 \wedge 10 \leq y' \leq 20$
$Landing \equiv$	$x=N-1 \wedge 10 \leq y \leq 20 \wedge x'=N \wedge y'=0$
$Fly \equiv$	$1 \leq x \leq N-2 \wedge 10 \leq y \leq 20 \wedge x'=x+1$
$FlyHigher \equiv$	$1 \leq x \leq N-1 \wedge 10 \leq y < 20 \wedge y'=y+1$
$FlyLower \equiv$	$1 \leq x \leq N-1 \wedge 10 < y \leq 20 \wedge y'=y-1$

A safety property of the airplane example may be stated as follows: The airplane is in a specified portion of the air space if it is not at one of the two airports. This can be formalized by the assertion

$$x \neq 0 \wedge x \neq N \Rightarrow 10 \leq y \leq 20 \quad \text{is invariant}$$

Informally, we say that a system state s is *reachable* if there exists a path from an initial state to s in the transition system. The assertion that P is invariant is satisfied if and only if it is satisfied by every reachable state of the system.

Generally, the letters P, Q, R, and I are used in this report to denote state formulas. A *state formula* is a formula in the state variables that can be evaluated over an individual system state to yield a truth value. We say that state s satisfies P if P evaluates to *true* over s.

A progress property of the airplane example may be stated as follows: The airplane, initially in Austin, eventually arrives at Dallas. This can be formalized by the assertion[2]

$$x=0 \wedge y=0 \textit{ leads-to } x=N \wedge y=0$$

Before we give meaning to the temporal operator *leads-to*, we need to introduce the concept of fairness. In any system state, several events may be enabled. For a state transition system with no fairness assumption, the choice of the next event to occur from the set of enabled events is nondeterministic. Such nondeterminism allows the possibility that some events never occur even though they are enabled continuously (or infinitely often). In order for a system to have progress properties, it is often the case that *some* of its events must be scheduled in such a way that certain fairness criteria are satisfied for

[2]In writing assertions containing *leads-to*, we adopt the convention that its binding power is weaker than any of the other logical connectives.

these events. These fairness assumptions should be stated explicity as part of the system specification; they must be satisfied by any implementaton of the specification.

In the relational notation, fairness assumptions are stated for individual events. For example, in order for the airplane example to have the progress property stated above, the events *TakeOff*, *Fly* and *Landing* must be scheduled in such a way that they have *weak fairness*. (Note that for this particular progress property, the other events of the airplane example do not have to be fairly scheduled.) Informally, the meaning of an event having weak fairness is the following: If the event is continuously enabled, it eventually occurs. Before we can give a rigorous definition of weak fairness, we need to define what is a *maximal* path.

The set of maximal paths of a state transition system includes all of its infinite paths. A finite path is maximal only if it ends in a terminal state s_k such that for all $s \neq s_k$, (s_k, s) is not in T.

The semantics of a system without any fairness assumption is given by its set of maximal paths. If fairness criteria are specified for a subset of events of a system, then only a subset of maximal paths are realizable for the system, namely, those maximal paths that satisfy the fairness criteria. We shall refer to this subset as the set of *fair paths*.

Weak fairness: We say that a particular *event e has weak fairness* if and only if each maximal path $\sigma = <s_0, s_1, \cdots>$ that is realizable for the system satisfies the following requirement: If for some s_k in σ and for all $i \geq k$, s_i satisfies $enabled(e)$ then there is some state s_j in σ $(j \geq k)$ such that the transition from s_j to s_{j+1} is due to the occurrence of e.

We are now in a position to give meaning to *leads-to*: *P leads-to Q* in a system if and only if on every fair path $<s_0, s_1, \cdots>$ of the system, if some state s_i in the path satisfies P then there is a state s_j in the path $(j \geq i)$ that satisfies Q.

Another way to state that a system is making progress or doing useful work is by assertions of *real-time behavior*. To do so, some state variables can be used to represent values of clocks and timers in the system. The requirement that an event *must occur* within a certain duration of time is stated as a safety property. Also, clocks are not allowed to reach certain values without the event having occurred; namely, clock ticks are events whose occurrences must preserve the real-time requirements of the system. In addition to proving that the real-time requirements are invariant, it is also necessary to prove that clock values are unbounded [Shankar & Lam 87a]. (This approach was also suggested by Lamport [1986].)

3. Proof Method

Assertions of safety properties are limited to the form: P is invariant, where P is a state formula. Consider an arbitrary state formula R. We use R' to denote the formula obtained by replacing every free variable v in R by v'. Let *Initial* denote a state formula specifying the initial condition on the system state, and e an event. A safety property is proved by the following rule:

Safety rule: P is invariant if, for some R,

(i) $Initial \Rightarrow R$,

(ii) for all e, $R \wedge e \Rightarrow R'$, and

(iii) $R \Rightarrow P$

Note that $P \equiv R \equiv true$ satisfies the above rule for every system. If I is invariant, we can replace R with $I \wedge I' \wedge R$ in parts (ii) and (iii) of the above rule.

We say that P is a *safety property* of a system if and only if P satisfies the safety rule for the system. By convention, for p and q being state formulas containing free variables, $p \Rightarrow q$ is logically valid if and only if $p \Rightarrow q$ is logically valid for all values of the free variables. This convention of universal quantification is also followed for properties and inference rules containing free variables.

Assertions of progress properties are limited to the form P *leads-to* Q, where P and Q are state formulas. Let e_i denote a particular event. We first define a relation between state formulas called *leads-to-via*.

Definition 1: P *leads-to* Q *via* e_i if and only if

(i) $P \wedge e_i \Rightarrow Q'$,

(ii) for all e, $P \wedge e \Rightarrow P' \vee Q'$,

(iii) $P \Rightarrow enabled(e_i)$ is invariant, and

(iv) event e_i has weak fairness.

If I is invariant, it can be used to strengthen the antecedent of each logical implication in the definition; that is, replace P by $I \wedge I' \wedge P$ in parts (i)-(iii) of the definition.

The *leads-to* relation is defined to be the strongest relation that satisfies the following rules. (This definition follows the one by Chandy and Misra [1988].)

Leads-to rules:

[Implication]

P *leads-to* Q if $P \Rightarrow Q$ is invariant

[Event]

P *leads-to* Q if for some e, P *leads-to* Q *via* e

[Transitivity]

P *leads-to* Q if for some R, P *leads-to* R and R *leads-to* Q

[Disjunction]

$P_1 \vee P_2$ *leads-to* Q if P_1 *leads-to* Q and P_2 *leads-to* Q

For a given system, if I is invariant and $P \wedge I$ *leads-to* R, we infer that P *leads-to* R.

Consider an event e that has weak fairness. Let $count(e)$ be an auxiliary variable that counts the number of occurrences of e from the beginning of system execution. Specifically, let the value of $count(e)$ be zero initially. Add $count(e)' = count(e)+1$ as a conjunct to the formula that specifies e. Add $count(e)' = count(e)$ as a conjunct to the formulas that specify other events. The following progress property can be easily proved using the Event leads-to rule and Definition 1. It states a consequence of the weakness fairness assumption for an event.

$$enabled(e) \wedge count(e)=k \ \ leads\text{-}to \ \ count(e)=k+1 \vee \neg enabled(e)$$

We next prove a lemma that will be used in Section 5.

Lemma 1: P_0 *leads-to* $(Q \vee P_2)$ if

(i) P_0 *leads-to* $(Q \vee P_1)$, and

(ii) P_1 *leads-to* $(Q \vee P_2)$

Proof:

(iii) $Q \Rightarrow Q \vee P_2$	(from $Q \Rightarrow Q$)
(iv) Q *leads–to* $(Q \vee P_2)$	(by Implication rule on (iii))
(v) $(Q \vee P_1)$ *leads–to* $(Q \vee P_2)$	(by Disjunction rule on (iv) and (ii))
(vi) P_0 *leads–to* $(Q \vee P_2)$	(by Transitivity rule on (v) and (i))

Q.E.D.

4. Distributed Systems

The basic building blocks of our model consist of state variables and events. Note that the relational notation and proof method introduced above are not dependent on whether the system being specified is distributed or centralized. The assumption that events are atomic means that concurrent actions in different modules of a distributed system are modeled by interleaving them in any order.

Some distributed systems use *shared variables* for interprocess communication. In the relational notation, a state variable can be shared and accessed by any number of events. Thus, no specialization in the notation is necessary for this class of distributed systems.

Some distributed systems use *message-passing* for interprocess communication. For such a system, a state variable is declared to represent the state of each communication channel in the system. These channel state variables are typed. The variable types we use for channels are *sequence* and *set* of message values that are sent along the channels.

Channel state variables can only be accessed by send and receive primitives that are specifically defined for the channels. For example, let $\mathbf{z}_i$ be a state variable of type *sequence* representing channel i that is an unbounded communication channel. Let m be a message value. Define

$$Send_i(m) \equiv \mathbf{z}_i' = \mathbf{z}_i @ m$$

$$Rec_i(m) \equiv \mathbf{z}_i = m @ \mathbf{z}_i'$$

where @ denotes the concatenation operator. Note that $Rec_i(m)$ is *false* if $\mathbf{z}_i$ is empty. (Primitives for channels with a finite capacity and for channels of variable type *set* can be similarly defined.)

An event containing the primitive $Send_i(m)$ is called a send event of m. An event containing the primitive $Rec_i(m)$ is called a receive event of m. Note that each primitive is a formula in state variables, just like other formulas that make up the specification of an event. The names $Send_i(m)$ and $Rec_i(m)$ are introduced primarily for notational convenience and to improve readability of event definitions.

Let M denote a set of messages that are sent along a channel. (Several sets of messages may be specified for a channel. For convenience, *message* and *message value* will be used synonymously in this report.) Let $e_r(m)$ denote an event for receiving m, $m \in M$, from the channel. The specification of events is required to satisfy the following condition: For every message m that can be sent along a channel, say channel i, the enabling conditions of receive events satisfy the following requirement:

$$m \in \mathbf{z}_i \Rightarrow \text{for some } e_r(m), enabled(e_r(m)) \qquad \text{if } \mathbf{z}_i \text{ is a set}$$

$$m = Head(\mathbf{z}_i) \Rightarrow \text{for some } e_r(m), enabled(e_r(m)) \qquad \text{if } \mathbf{z}_i \text{ is a sequence}$$

where *Head* is a function whose value is the first element of $\mathbf{z}_i$ if $\mathbf{z}_i$ is not empty; otherwise, *Head* returns a null value.

Consider message-passing systems with unreliable channels that can lose or reorder messages. To prove progress properties for such systems, it is necessary to assume that these unreliable channels have some progress property. The assumption should be as weak as possible so that it can be satisfied by most communication channels. An informal statement of the *channel progress assumption* is the following: If messages in a set M are sent repeatedly along a channel, one of them is eventually received [Hailpern & Owicki 83].

We next define another *leads−to−via* relation between state formulas. Let e_M denote an event for sending messages in M along a channel, and $count(M)$ an auxiliary variable whose value indicates the number of times messages in M have been sent along the channel by any send event of M since the beginning of system execution.

Definition 2: P *leads−to* Q *via* M if and only if

(i) for all e_r, $\forall m \in M\ [P \wedge e_r(m) \Rightarrow Q']$,

(ii) for all e, $P \wedge e \Rightarrow P' \vee Q'$, and

(iii) for some e_M, $P \wedge count(M) \geq k$ *leads−to* $Q \vee count(M) \geq k+1$ *via* e_M

Given the channel progress assumption, we have the following leads-to rule which is added to the definition of the leads-to relation.

Leads-to rule [Message]:

P *leads−to* Q if for some M, P *leads−to* Q *via* M

We give a few general observations about the specification and verification of distributed systems using our notation and proof method.

While it is not necessary to explicitly define modules (or processes) and associate state variables and events with these modules, it is useful to know the state variables that are accessed by each event. This information is available from the syntax of an event's definition. In a distributed system, each event accesses just a small subset of the system's state variables. This information can be used to substantially simplify applications of the inference rules. In applying the safety rule, for example, if none of the free variables in R is updated by event e then $R \wedge e \Rightarrow R'$ is trivially satisfied.

In a state transition system, the occurrence of a send or a receive event is asynchronous. Each event occurrence corresponds to a transition in the system state space. This is unlike the CSP model in which modules in a message-passing network are explicitly defined and pairs of send and receive events in different module occur synchronously. This observation about state transition systems does not depend on having unbounded communication channels. But it does depend upon having channels with nonzero capacity.

Lastly, in the relational notation, it is also possible to specify synchronous updates of state variables; this is done by placing the updates within the action of the same event.

5. Refinement and Projection of Systems

Consider two state transition systems A and B. Let V_A denote a set of variables $\{v_1, v_2, ..., v_n\}$ and V_B denote the subset $\{v_1, v_2, ..., v_m\}$, where $m \leq n$. Let V_A and V_B be the state variable sets of A and B respectively. Note that since V_B is a subset of V_A there is a projection mapping from the states of A to the states of B; states in A having the same values for $\{v_1, v_2, \cdots, v_m\}$ are mapped to the same state in B. Also, any state formula of system B is a state formula of system A.

Let $\{a_i\}$ denote the set of events of system A, and $\{b_j\}$ the set of events of system B.

We next define a relation between two systems, called *refinement*, by first introducing a relation between events of the two systems. Event a_i in system A is a refinement of events in system B if, for some invariant R_A of system A, for some subset $\{b_1, b_2, ..., b_k\}$ of events in B,

$$R_A \wedge a_i \Rightarrow b_1 \vee b_2 \vee \cdots \vee b_k \qquad \text{(refinement condition)}$$

Very often, a_i is the refinement of a single event. In this case, to check if a_i satisfies the refinement condition, it is sufficient to show either $a_i \Rightarrow b_j$ or $R_A \wedge a_i \Rightarrow b_j$ for some b_j.

Let us explain informally the meaning of an event being a refinement of events in another system. If a_i can take system A from state s_1 to s_2 then there is some event b_j that can take system B from state t_1 to t_2, where t_1 and t_2 are the images of s_1 and s_2 respectively under the projection mapping. This condition can be relaxed by introducing a safety property R_A of system A, in which case the condition has to hold only for each (s_1, s_2) pair such that s_1 and s_2 satisfy R_A. We will have to prove separately that such safety properties introduced are in fact properties of system A. Note that an event a_i satisfies the refinement condition if it has a null image under the projection mapping, namely, $t_1 = t_2$ for all s_1 and s_2 reachable in system A.

Let $Initial_A$ and $Initial_B$ be the initial conditions of A and B respectively.

Definition 3: System A is a refinement of system B if and only if every event in A is a refinement of some events in B and $Initial_A \Rightarrow Initial_B$.

The above condition is also used for defining B to be an image of A under the projection mapping; that is, the relation *image* is the inverse of the relation *refinement* by definition.

The safety properties needed to guarantee events of system A to be refinements of events of system B arise naturally in the following manner. In the process of deriving system A from system B, suppose we want to implement a state variable x in B by two state variables y and z in A. In system A, x is still a state variable but is made auxiliary. To prove $a_i \Rightarrow b_j$, where b_j may contain references to x, an assertion of the relation between x, y, and z in system A must be included as a conjunct of R_A. Note that this relation is akin to the possibilities mapping of Lynch and Tuttle [1987]. For the special case of the relation being a function, the function is just like the state functions used by Lamport [1983].

We provide two examples to illustrate the above observation:

Example 1. Suppose **x** is a channel state variable of type *sequence* of $\{m_1, m_2\}$ in system B. In system A, the channel is represented by a state variable **y** of type *sequence* of $\{m_1, m_2, m_3\}$. Then the function $\mathbf{x} = \mathbf{y} \mid \{m_1, m_2\}$, where the right hand side of the equality denotes the restriction of sequence **y** to $\{m_1, m_2\}$, must be a safety property of system A in order for it to be a refinement of system B.

Example 2. Let x be a state variable of system B. Its domain is the set of natural numbers. The following event is defined in system B:

$$b_1 \equiv x \textit{ is even} \wedge x' = x + 1$$

In deriving system A from system B, suppose we introduce a variable y whose domain is $\{0,1\}$ to replace x, and the following event is defined:

$$a_1 \equiv y = 0 \wedge y' = 1 \wedge x' = x + 1$$

Event a_1 is a refinement of b_1 given that $y{=}x \ mod \ 2$ is a safety property of system A. Note that x can be made an auxiliary variable of system A and it does not have to be implemented.

Theorem 1: If A is a refinement of B then every safety property of B is a safety property of A.

Proof: Let P be a safety property of B. Let R_B denote a safety property of B that makes P satisfy the safety rule. Let R_A be a safety property of A that makes events of A satisfy the refinement condition. We show that P satisfies the safety rule for system A. First, we have $Initial_A \Rightarrow Initial_B \Rightarrow R_B$. Second, we have R_A is invariant in system A, and for any event a_i of A, there exists a subset $\{b_1, b_2, \cdots, b_k\}$ of events in B, such that

$$\begin{aligned} R_B \wedge R_A \wedge a_i &\Rightarrow R_B \wedge (b_1 \vee b_2 \vee \cdots \vee b_k) \\ &\Rightarrow (R_B \wedge b_1) \vee (R_B \wedge b_2) \vee \cdots \vee (R_B \wedge b_k) \\ &\Rightarrow R_B' \end{aligned}$$

Since we already know that $R_B \Rightarrow P$, the proof is complete.

Q.E.D.

The following lemma will be used in the proof of Theorem 2 below.

Lemma 2: If system A is a refinement of system B and

for all event b, $P \wedge b \Rightarrow P' \vee Q'$ in system B

then the following is logically valid

for all event a, $P \wedge a \Rightarrow P' \vee Q'$ in system A.

The proof is immediate by applying the refinement condition.[3]

We next consider the conditions under which progress properties of system B are also progress properties of system A. We distinguish two cases, depending on whether we want to preserve some specific progress properties of B or every progress property of B.

A **hierarchical proof** method can be used to establish specific progress properties in connection with the hierarchical specification of a complex system. Suppose we specify system A in two steps. First, specify system B, a simpler system than A. Second, derive system A to be a refinement of system B. The proof of a progress property such as P *leads−to* Q for system A can be carried out hierarchically as follows. The progress property is first proved for system B by proving a set L of *leads−to−via* relations and showing that the relation P *leads−to* Q is in the closure of the relations in L. Subsequently, it suffices to show that for each relation p *leads−to* q via event b or via message set M in the set L, the relation p *leads−to* q holds for system A.

For some applications, however, it is desirable that *every* progress property of system B be a progress property of system A. This can be achieved by imposing some requirements on the event formulas of A and B.

Let $\{b_j, j \in J\}$ be the set of events of B, where J is an index set. Let $\{a_i\}$ be the set of events of A with an index set that is a superset of J. The conditions for system A to have *well-formed events* with respect to events of system B are given below. (These conditions are adapted from conditions in [Lam & Shankar 84] bearing the same name.)

[3]The relation between P and Q in Lemma 2 is almost the same as the P *unless* Q relation of Chandy and Misra [1988].

WF1 [*refinement of observable events*]

For all $j \in J$, event a_j is a refinement of b_j.

WF2 [*refinement of unobservable events*]

For all $j \notin J$, event a_j is a formula in a set of parameters and the restricted set of state variables $\{v_{m+1}', v_{m+2}', \cdots, v_n'\} \cup \{v_1, v_2, \cdots, v_n\}$.

Events with this restriction in system A are referred to as *null-image events* because their actions do not update state variables that are also in system B; their updates are thus not observable in system B.

The following two conditions are specified for every j in J such that b_j has weak fairness in system B. (The set J in this requirement may be relaxed to a subset of J if only some, but not all, progress properties of system B are to be properties of system A. We will elaborate on this observation later.) In the following, R_A denotes a safety property of system A.

WF3 [*unobservable leads-to*]

The null-image events of system A satisfy the following:

$R_A \wedge \mathit{enabled}(b_j)\ \mathit{leads\text{-}to}\ \mathit{enabled}(a_j)$ via a sequence of null-image events.

By Definition 1, each of the null-image event in the sequence must have weak fairness.

WF4 [*noninterference*]

Events of system A, for all $i, i \neq j$, satisfy the following:

$R_A \wedge \mathit{enabled}(a_j) \wedge a_i \Rightarrow \mathit{enabled}(a_j)'$.

As is evident from the proof of Theorem 2 below, the noninterference condition (WF4) is not needed if the implementation of system A guarantees strong fairness for event a_j. (Informally, if an event having strong fairness is enabled repeatedly, it eventually occurs [Pnueli 86].)

If event a_j, $j \in J$, satisfies the WF3′ condition below, it is said to be *strongly well-formed.* Conditions WF3 and WF4 are not needed for a strongly well-formed event.

WF3′ [*strongly well-formed event*]

$R_A \wedge \mathit{enabled}(b_j) \Rightarrow \mathit{enabled}(a_j)$.

Clearly, WF3′ is much easier to apply than WF3 and WF4. It has been our experience in the specification of communication protocols that many events can be refined to satisfy WF3′. But for some events, WF3 and WF4 are needed.

The first two conditions, WF1 and WF2, guarantee that all events of system A satisfy the refinement condition.

Definition 4: System A is a well-formed refinement of system B if and only if A has well-formed events with respect to B and $\mathit{Initial}_A \Rightarrow \mathit{Initial}_B$.

Theorem 2: Every progress property of system B is a progress property of system A if

(i) system A is a well-formed refinement of system B, and

(ii) for all $j \in J$, if event b_j has weak fairness in system B, then event a_j has weak fairness in system A.

A proof of Theorem 2 is given in [Lam & Shankar 88].

If only *some* progress properties of system B are required to be properties of system A, the conditions for well-formed events can be relaxed as follows. Let L be the set of leads-to-via relations that are needed to prove these progress properties for system B. Instead of proving WF3', or WF3 and WF4, for every j in J such that event b_j has weak fairness, the conditions have to be proved only for those events and message sets whose leads-to-via relations are in L.

Let us now consider a refinement of the specification of the airplane example. Let the state variables x and y be augmented by a third state variable z, with domain over all integers, so that we will be reasoning about trajectories of the airplane in 3-dimensional space. Initially, $z=0$. Five events, denoted by *, are defined in terms of events of the 2-variable system as follows:

$$
\begin{array}{lll}
TakeOff^* & \equiv & TakeOff \wedge z=0 \wedge -10 \le z' \le 10 \\
Landing^* & \equiv & Landing \wedge -10 \le z \le 10 \wedge z'=0 \\
Fly^* & \equiv & Fly \wedge -10 \le z \le 10 \\
FlyHigher^* & \equiv & FlyHigher \wedge -10 \le z \le 10 \\
FlyLower^* & \equiv & FlyLower \wedge -10 \le z \le 10
\end{array}
$$

It is easy to see that the above events are refinements of corresponding events in the 2-variable system. Add the following two events:

$$
\begin{array}{lll}
FlyLeft & \equiv & 1 \le x \le N-1 \wedge 10 \le y \le 20 \wedge -10 < z \le 10 \wedge z'=z-1 \\
FlyRight & \equiv & 1 \le x \le N-1 \wedge 10 \le y \le 20 \wedge -10 \le z < 10 \wedge z'=z+1
\end{array}
$$

The two new events are also refinements because they are null-image events whose occurrences are not observable in the 2-variable system. Hence, the new 3-variable system is a refinement of the 2-variable system. Furthermore, events of the 2-variable systems satisfy the well-formed conditions (in fact, they are strongly well-formed) given the following safety requirement:

$$R_A \equiv (x=0 \wedge y=0 \Rightarrow z=0) \wedge (1 \le x \le N-1 \Rightarrow -10 \le z \le 10)$$

R_A is easily shown to be an invariant of the 3-variable system. Thus safety and progress properties of the 2-variable system are also properties of the 3-variable system. Once proved for the 2-variable system, they do not have to be proved again for the 3-variable system.

6. Discussions and Related Work

The basic building blocks for specifying systems in the relational notation are state variables and events. Events are binary relations on the system state space and they are specified by formulas in a predicate logic notation. We believe that our notation is easy to learn because states and state transitions are represented explicitly. For the same reason, event specifications are also easy to read and understand. The idea of specifying an event as a formula in the values of state variables before and after the event's occurrence is not unique to our work. A similar notation was used for program statements by Lamport [1983] and by Hehner [1984].

Prototypes of the relational notation and the proof method presented in this paper were described in [Shankar & Lam 84, Shankar & Lam 87a]. Our proof method is based upon a fragment of linear-time temporal logic [Chandy & Misra 88, Lamport 83, Owicki & Lamport 82, Pnueli 86]. The method was designed to have a minimal amount of notation with the goal that it will be widely accessible to protocol engineers. Proofs provided by the designer or verifier of a system can be checked in a mechanical fashion by applying a small set of inference rules as well as rules in predicate logic.

In order for a system to have progress properties, its events must be scheduled in such a way that certain fairness criteria are satisfied for some events. We advocate the approach of stating fairness assumptions explicitly for individual events as part of the system specification, noting that for some systems not all events have to be fairly scheduled. This contrasts with the approach of a blanket assumption that all events in a system are fairly scheduled according to a particular criterion.

Using the relational notation, we presented a theory of refinement of state transition systems. The theory is adapted from our earlier work on projection of a state transition system to get image systems [Lam & Shankar 84]. The relation A *is a refinement of* B, for two systems A and B, is the inverse of the relation B *is an image of* A by definition.

Other authors have defined similar relations between state transition systems. Lamport [1985] explained what it means for a program to implement its specification, that is, the relation A *implements* B. He offered two definitions. In a pure axiomatic definition, he said, "the implementation is correct if and only if the axioms comprising the semantics of the implementation imply the axioms of the specification, after the latter are translated by F into assertions about the implementation", where F denotes a mapping from the formulas of B to those of A.

Our refinement condition is reminiscent of the above definition. Specifically, we define systems A and B such that every state variable of B is a state variable of A. Thus, assertions about B are valid assertions about A. No translation is needed.

In the second definition of Lamport, the formal semantics of A and B are sets of behaviors, $paths_A$ and $paths_B$, respectively. (A behavior corresponds to a fair path as defined in Section 2.) Systems A and B satisfy the relation A *implements* B if $paths_A \subseteq F(paths_B)$, where F maps behaviors in $paths_B$ into a set of behaviors in A allowable by B. The definition of A *simulates* B by Lynch and Tuttle [1987] is almost the same.

Lam and Shankar [1984] showed that if B *is a well-formed image of* A then $paths_A = F(paths_B)$. Note the relation "$=$" instead of "$\subseteq$". Consequently, if A *is a well-formed refinement of* B or B *is a well-formed image of* A, then A *simulates* B according to Lynch and Tuttle and A *implements* B according to Lamport. The converse may not be true.

Chandy and Misra [1988] recently defined the relation A *is a superposition of* B. In their approach, program A is obtained from transforming program B by repeated applications of two rules. This approach is attractive because the rules are syntactic and are thus very easy to use. However, the class of systems that can be obtained by these rules is smaller than the class that can be obtained by refinement. In particular, it is easy to see that if A *is a superposition of* B then A *is a well-formed refinement of* B. The converse may not be true. Specifically, if A *is a superposition of* B, then conditions WF1 and WF2 are satisfied. In addition, the superposition rules guarantee $enabled(a_j) \equiv enabled(b_j)$, for all $j \in J$, which implies WF3$'$.

While the relational notation and proof method in this paper are applicable to state transition systems in general, their development has been motivated primarily by protocol systems. In particular, we were dissatisfied with the limited capability of the CFSM model in specifying as well as in verifying communication protocols. Without state variables, specification of sequence numbers and timers in a CFSM model is difficult. Without timers, the modeling of unreliable channels in a CFSM model requires the use of special techniques that do not always work, e.g., virtual messages, unbounded retransmissions, etc.

The ideas and methods in this paper have been applied to the specification and verification of some relatively large protocol systems.

The first application was the verification of a version of the High-level Data Link Control (HDLC) protocol standard with functions of connection management and full-duplex data transfer. Instead of verifying such a multifunction protocol in its entirety, smaller image protocols were obtained by projection and verified [Shankar & Lam 83].

The relational notation was applied to specify an implementation of a two-phase locking system to satisfy two interface specifications: (1) an "upper" interface that offers serializable access to concurrent client programs, and (2) a "lower" interface for accessing a physical database. The implementation was obtained as a refinement of the upper interface [Lam & Shankar 87].

A stepwise refinement heuristic was developed based upon the refinement relation. The heuristic was applied to the construction of sliding window protocols for the transport layer where channels can lose, duplicate, and reorder messages, and the protocols use cyclic sequence numbers [Shankar 86, Shankar & Lam 87b]. It was also applied to the specification and verification of connection management protocols for the transport layer [Murphy & Shankar 87, Murphy & Shankar 88].

In the last two references, Murphy and Shankar demonstrated how a complete transport protocol with functions of connection management and full-duplex data transfer can be obtained by combining protocols constructed for the individual functions. Because the multifunction protocol is a refinement of instances of the single-function protocols, safety properties of the single-function protocols are preserved in the multifunction protocol. Proofs of progress properties of the multifunction protocol were obtained hierarchically [Murphy & Shankar 88].

Recently, the image relation between systems was used to reason about semantics of different communication protocols and conversions between them. The two theorems in this paper were applied to define what it means for a protocol converter to achieve interoperability between entities that implement different protocols [Lam 88].

Acknowledgements

We thank Jayadev Misra who discovered several technical errors in an early draft of this paper. We also thank Mohamed Gouda who provided various constructive comments.

REFERENCES

[CCITT 85] CCITT, Recommendations Z.101 to Z.104, Red Book, Geneva, 1985 (Standard Definition Language).

[Chandy & Misra 88] K.M. Chandy and J. Misra, *A Foundation of Parallel Program Design*, Addison-Wesley, Reading, MA, 1988.

[Clarke & Emerson 81] E.M. Clarke and E.A. Emerson, "Synthesis of Synchronization Skeletons for Branching Time Temporal Logic," *Proceedings Workshop on Logic of*

Programs, LNCS 131, Spring Verlag, 1981.

[Hailpern & Owicki 83] B.T. Hailpern and S. Owicki, "Modular Verification of Computer Communication Protocols," *IEEE Transactions on Communications*, Vol. COM-31, No. 1, January 1983.

[Hehner 84] E.C.R Hehner, "Predicative Programming, Part I and Part II," *Communications of the ACM*, Vol. 27, No. 2, February 1984.

[Hoare 85] C.A.R. Hoare, *Communication Sequential Processes*, Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[IBM 80] IBM Corporation, Systems Network Architecture Format and Protocol Reference Manual: Architecture Logic, IBM Form No. SC32-3112-2, 1980.

[ISO 85] ISO/TC97/SC21/WG16-1 N422 Estelle--A Formal Description Technique Based on an Extended State Transition Model, Feb. 1985.

[Lam 88] S.S. Lam, "Protocol Conversion," *IEEE Transactions on Software Engineering*, Vol. 14, No. 3, March 1988.

[Lam & Shankar 84] S.S. Lam and A. U. Shankar, "Protocol Verification via Projections," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, July 1984.

[Lam & Shankar 87] S.S. Lam and A. U. Shankar, "Specifying an Implementation to Satisfy Interface Specifications: A State Transition Approach," presented at the 26th Lake Arrowhead Workshop on *How will we specify concurrent systems in the year 2000?*, September 1987.

[Lam & Shankar 88] S. S. Lam and A. U. Shankar, "A Relational Notation for State Transition Systems," Technical Report TR-88-21, Department of Computer Sciences, University of Texas at Austin, May 1988.

[Lamport 83] L. Lamport, "What Good is Temporal Logic?" *Proceedings Information Processing 83*, IFIP, 1983.

[Lamport 85] L. Lamport, "What it means for a concurrent program to satisfy a specification: Why no one has specified priority," *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, New Orleans, January 1985.

[Lamport 86] L. Lamport, "A Simple Approach to Specifying Concurrent Systems," Technical Report 15, System Research Center, Digital Corporation, Palo Alto, California, December 1986.

[Lynch & Tuttle 87] N.A. Lynch and M.R. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms," *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Vancouver, B.C., August 1987.

[Manna & Pnueli 84] Z. Manna and A. Pnueli, "Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs," *Science of Computer Programming*, Vol. 4, 1984.

[Milner 80] R. Milner, *A Calculus of Communication Systems*, LNCS 92, Springer Verlag, New York, 1980.

[Murphy & Shankar 87] S.L. Murphy and A.U. Shankar, "A Verified Connection Management Protocol for the Transport Layer," *Proceedings ACM SIGCOMM '87 Workshop*, Stowe, Vermont, August 1987.

[Murphy & Shankar 88] S.L. Murphy and A.U. Shankar, "Service Specification and Protocol Construction for the Transport Layer," *Proceedings ACM SIGCOMM '88 Symposium*, Stanford University, August 1988 (to appear).

[Owicki & Gries 76] S. Owicki and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," *Communications of the ACM*, Vol. 19, No. 5, May 1976.

[Owicki & Lamport 82] S. Owicki and L. Lamport, "Proving Liveness Properties of Concurrent Systems," *ACM TOPLAS*, Vol. 4, No. 3, 1982.

[Piatkowski 86] T. F. Piatkowski, "The State of The Art in Protocol Engineering," *Proceedings ACM Sigcomm '86 Symposium*, Stowe, Vermont, 1986.

[Pnueli 86] A. Pnueli, "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends," in *Current Trends in Concurrency: Overviews and Tutorials*, J.W, deBakker et al. (ed.), LNCS 224, Springer Verlag, 1986.

[Sabnani 86] K. Sabnani, "An Algorithmic Procedure for Protocol Verification," AT&T Bell Laboratories Research Report, Murray Hill, N.J., 1986.

[Shankar 86] A.U. Shankar, "Verified Data Transfer Protocols with Variable Flow Control," Technical Report CS-TR-1746, University of Maryland, December 1986, to appear in *ACM Transactions on Computer Systems*; an abbreviated version appeared in *Proceedings ACM SIGCOMM '86*, Stowe, Vermont, August 1986.

[Shankar & Lam 83] A.U. Shankar and S.S. Lam, "An HDLC Protocol Specification and its Verification Using Image Protocols," *ACM TOCS*, Vol. 1, No. 4, November 1983.

[Shankar & Lam 84] A.U. Shankar and S.S. Lam, "Time-dependent communication protocols, " in *Tutorial: Principles of Communication and Networking Protocols*, S.S. Lam (ed.), IEEE Computer Society, 1984.

[Shankar & Lam 87a] A.U. Shankar and S.S. Lam, "Time-dependent distributed systems: proving safety, liveness, and real-time properties," *Distributed Computing*, Vol. 2, 1987.

[Shankar & Lam 87b] A.U. Shankar and S.S. Lam, "A Stepwise Refinement Heuristic for Protocol Construction," Technical Report CS-TR-1812, Department of Computer Science, University of Maryland, March 1987.

[West 78] C.H. West, "General Techniques for Communications Protocol Validation," *IBM Journal of Research and Development*, Vol. 22, July 1978.