3

Specifying and Verifying Conditional Progress

Ken Calvert
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30032-0280
calvert@cc.gatech.edu

Abstract

In some otherwise attractive formalisms, it is difficult to specify protocol progress in such a way that the protocol can be proved correct independent of its underlying channels. This is because the specification language cannot express the interdependence between the protocol and its underlying channels, especially if the latter are unboundedly lossy. This paper investigates the extent to which such progress properties can be dealt with using predicate calculus and a simple extension of a well-known logic for progress (leads-to) properties. It turns out that many complex progress specifications have equivalents that are provable in the extended theory. Based on the results, an approach to specification of protocol progress is outlined and illustrated with an example.

Keywords: Protocol Verification, Logics of Programs, Specification Techniques.

1 Introduction

Temporal logic is a widely-studied tool for reasoning about distributed algorithms like protocols. Powerful theories have been developed, which allow very general properties to be specified and verified. As a practical matter, however, the more expressive the specification language, the more complex is the verification machinery required to prove that a program satisfies a specification. Theories like UNITY [7], which features a small, elegant proof system, have proven quite useful for reasoning about all kinds of applications, including protocols [9, 10, 12, 13, 17]—notwithstanding the sacrifices they make in terms of expressive power. When it comes to specifying progress of communication protocols, however, these systems are often inadequate.

A protocol's *progress* specification defines what it may be relied upon to do. For example, a protocol implementing a reliable byte stream service over unreliable channels typically has a progress specification requiring that every byte sent eventually be received. However, such a specification cannot be satisfied by the protocol alone: the underlying channels must not constantly lose messages. Thus the requirement that every byte sent be received constrains *both* the protocol and the underlying channels.

S. T. Vuong et al. (eds.), *Protocol Specification, Testing and Verification XIV*

It is desirable to separate the protocol's progress specification from that of the channels. One way to accomplish this is to condition protocol progress on channel progress: if the channels satisfy X, the protocol satisfies Y, where X is the underlying channels' progress specification, and Y says that every byte sent is delivered. (UNITY allows such conditional properties for certain restricted forms of X and Y.) However, in many cases X will itself be in the form of a conditional. For example, a common specification of an unboundedly lossy channel says "any message sent infinitely often will be received infinitely often." In this case, X in the above specification would have the form "If the user (protocol) satisfies P, the channel will satisfy Q," where P says the message is transmitted infinitely often, and Q says it is received infinitely often. Thus the natural form of the protocol's progress specification is a *nested* conditional $(P \Rightarrow Q) \Rightarrow Y$. We would like to have a theory that allows the protocol to be verified with respect to such specifications completely independently of the channels. Ideally, it should also permit $P \Rightarrow Q$ to be proved of a channel, and allow Y to be proved of the composite using only these two specifications.

Nested dependencies of this form cannot be expressed directly in the specification languages of many of the "streamlined" formalisms, including UNITY. Thus it becomes necessary to employ various tricks when specifying protocol progress using such formalisms. A common approach is to introduce a special proof rule dealing with messages and channels [7, 9, 18]; such a rule gives sufficient conditions for a message to be delivered, stated in terms of the theory's regular properties. These ad hoc approaches work, but showing that the extra rule is valid for a *particular* channel must be done outside the system. Moreover, the approach may not generalize to other instances of progress dependency.

Another approach to specification of such progress dependencies is to introduce the full generality of temporal logic, admitting arbitrary combinations of whatever temporal operators are used [14]. This can yield a very powerful specification language, but with a corresponding increase in the size and complexity of the proof system. For practical applications, we would like to limit the effort required (for a trained protocol engineer) to learn the formal mechanism, to be comparable to, say, that required to learn a medium-sized programming language.

This paper considers progress specifications constructed using a single temporal operator, *leads-to*. We show that a large class of progress specifications, including many with nested dependencies as above, can be represented with combinations of leads-to properties of a certain simple form. The advantage of this approach is that the cost of this additional power is very low in terms of additional mechanism beyond that of a "streamlined" formalism such as UNITY. The proof theory is essentially the UNITY proof theory for leads-to, with a different semantic interpretation of conditional properties, plus the predicate calculus. These results support an approach in which a progress specification is first written down in a natural form expressing dependencies, and then transformed into an equivalent specification in a form that is provable within the given system. The approach has been developed for use with a particular state-based compositional theory of module specifications [3, 4], but it can be applied with other, similar theories, as well. We illustrate the approach for an example protocol specification.

Obviously progress properties alone do not suffice to characterize a protocol. Indeed, the part of the specification dealing with *safety*, which defines allowable states and transitions, is typically considered to be the more fundamental part. This paper deals exclusively with progress specifications, however, because they are more complex than safety specifications: the kind of nested dependencies described above simply do not arise in safety specifications [1]. In any case, numerous techniques for specifying and verifying safety properties are well-known [10, 12, 7].

The rest of the paper is organized as follows. In the next section we define the (basically standard) semantic model and a class of progress properties based on the leads-to relation. Section 3 introduces an example showing how such properties can be used to express conditional progress requirements. Section 4 presents the proof theory and some results about transformation of specifications into provable form. Section 5 outlines conditions under which progress specifications can be rewritten in a provable form. Section 6 concludes with a summary and brief discussion.

2 Preliminaries

In what follows, "program" refers to a concurrent program or protocol specified using some (formal) notation, such as UNITY [7] or the relational notation of Lam and Shankar [4, 10]. Such a specification is assumed to define a set of *state variables* and a corresponding *state space*. The *safety* part of such a specification defines an *initial condition* (assumed here to be *true*) and a set of *allowed transitions* in the state space. The program may also include a *fairness* or basic progress condition. The program is assumed to be fixed throughout this paper.

It is assumed throughout this paper that the objects of our discourse are mathematical rather than textual. In particular, we speak of *predicates* (boolean functions on a domain) rather than *formulas* (textual representations of expressions involving state variables). The operations are considered to be applied to the mathematical objects as opposed to formulas. This is not a problem so long as (i) we are dealing with predicates that are (originally) defined by some formulas in some language, and (ii) the mathematical operations are constructive and have clear textual representations. Both conditions are satisfied throughout this paper.

In what follows, the variables p, q, r, x, y and their primed counterparts are of type "state predicate", i.e. boolean functions on the state space. For any predicate p, a *p-state* is one at which p has the value *true*. The syntactic binding precedence of the boolean operators used in this paper is as follows (most binding at left; symbols grouped together have the same precedence): $\neg, (\wedge\ \vee), \Rightarrow, \equiv$. Later in this paper boolean operators will be used with *temporal predicates* as well as state predicates. The type of a boolean expression is the same as the types of its operands; in expressions where both types occur, liberal use of parentheses will prevent ambiguity.

2.1 Semantics

A concurrent system or protocol is modeled as a generator of infinite sequences of *states*; each sequence represents a possible behavior of the program in terms of its state variables. The set of sequences of a given program contains those infinite sequences corresponding to paths through the state space allowed by the transition relation, which also satisfy the fairness criterion. We refer to these sequences as *behaviors*. The variable $\overline{w}$ ranges over arbitrary behaviors.

Specifications are defined using *temporal predicates*, which are boolean functions on infinite sequences of states. For temporal predicate P, we say a program has property P iff each of its behavior sequences satisfies predicate P. Double square brackets denote universal quantification over behaviors: $[\![P]\!]$ means that the program has property P.

2.2 Leads-to Properties

The progress specification of a program expresses what *shall* happen in each of the program's possible behaviors. For state predicates p and q, $p \leadsto q$ is a temporal predicate that is true for a sequence if and only if every p-state in the sequence either is, or is followed by, a q-state. Thus the property $p \leadsto q$ says that if the program ever reaches a state that satisfies $(p \wedge \neg q)$, it will at some later time reach a state satisfying q. In the rest of this paper, variables P, Q, R range over simple leads-to properties.

In order to reason about progress, we often need to consider the *safety* properties of the system, i.e. its allowed transitions. We use *unless* properties for this purpose. A sequence satisfies p *unless* q iff each state satisfying $(p \wedge \neg q)$ is immediately followed by a state satisfying $p \vee q$. It is not difficult to see that $[\![p \text{ unless } q]\!]$ holds if and only if no transition of the system leads from a $(p \wedge \neg q)$-state to a $(\neg p \wedge \neg q)$-state.

We next define a class of *progress properties*, built up inductively from simple leads-to properties using boolean operators $\wedge$, $\vee$, $\Rightarrow$, $\neg$, $\equiv$ and universal and existential quantification.

- Every simple leads-to property is a progress property.
- If X and Y are progress properties, then so are $X \wedge Y$, $X \vee Y$, and $X \Rightarrow Y$, $X \equiv Y$, and $\neg X$.
- If $X.m$ is a progress property for each m, then so are $\langle \forall\, m :: X.m \rangle$ and $\langle \exists\, m :: X.m \rangle$.

Temporal predicates are manipulated in the usual way using the predicate calculus.

3 Conditional Progress Specifications

Let us consider the example of a Transport Layer protocol that uses unreliable channels at the lower level. The protocol layer consists of two peers, a Sender and a Receiver. Its environment consists of a Sending User and Receiving User above, and two lossy channels below (refer to Figure 1). We describe the interfaces in terms of *auxiliary variables*, which are state variables introduced to make the specification more abstract

and easier to state. In an implementation, auxiliary variables need not be explicitly present, but their values must be definable in terms of those of other state variables.

The upper interface of the protocol (that seen by the Users) is defined in terms of two auxiliary variables, *ins* and *outs*. Each is a sequence of bytes, initially empty. At any time, *ins* is the sequence of all bytes sent so far by the Sending User, while *outs* is the sequence of all bytes delivered so far to the Receiving User. To send a group of bytes, the Sending User appends them to *ins*; to deliver a group of bytes to the Receiving User, the protocol appends them to *outs*. The number of bytes in a sequence s is denoted by $|s|$. At any state where $|ins| > |outs|$, a byte has been sent but not yet delivered, and the system is required to make progress.

The lower-level interface used by the protocol comprises a pair of noisy channels, one in each direction, from the Sender to the Receiver and vice versa. These channels are designated *sr* and *rs* respectively. Channel *sr* (*rs*) has an associated set G_{sr} (G_{rs}) of *messages* that may be sent over it. For each message m in G_{sr} (G_{rs}), channel *sr* (*rs*) also has two boolean variables *sr-in.m* and *sr-out.m* (*rs-in.m* and *rs-out.m*), which are true whenever message m is transmitted or received, respectively, on the channel. Note that there may be more than one message m such that *sr-in.m* is true at any one time.

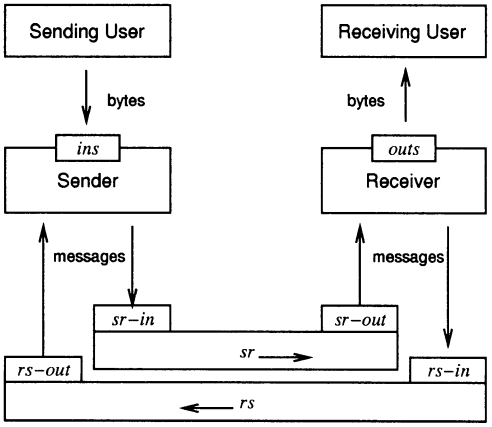

Figure 1: Protocol Configuration for Example

The job of the protocol is to transfer the bytes in *ins* to *outs* by grouping them with control information and sending them over the lower-level channel; the back channel from Receiver to Sender carries acknowledgment and other information.

3.1 Lossy Channel Progress

A lossy channel can lose a message any countable number of times, but if the message is repeatedly transmitted, it *will* eventually be delivered. In other words, for any message

m, if m is transmitted infinitely often, m is received infinitely often. The leads-to property that corresponds to "p holds infinitely often" is $true \leadsto p$. We thus arrive at the following progress specifications for channels sr and rs:

$$\langle \forall\, m : m \in G_{sr} : (true \leadsto sr\text{-}in.m) \;\Rightarrow\; (true \leadsto sr\text{-}out.m) \rangle \tag{1}$$

$$\langle \forall\, m : m \in G_{rs} : (true \leadsto rs\text{-}in.m) \;\Rightarrow\; (true \leadsto rs\text{-}out.m) \rangle \tag{2}$$

It is important to note that each leads-to property constrains *only* the channel, and says that *it need not deliver any message that is transmitted only a finite number of times*. It might seem that this requirement is too weak to be useful, since at any point a protocol will have transmitted a message only a finite number of times. We shall see, however, that it is adequate.

3.2 Protocol Progress

The protocol must ensure that for each byte sent (appended to *ins*), a byte is delivered (appended to *outs*). This is expressed by the property

$$\langle \forall\, k :: |ins| \geq k \leadsto |outs| \geq k \rangle$$

This specification can only be satisfied if the underlying channels are reliable. However, we would like for the protocol progress specification to be written in such a way that it can still be satisfied even if one or both of the underlying channels fails to satisfy *its* specification. Therefore the above property should be conditioned on the progress property of the underlying channels. The resulting property is

$$\begin{array}{rl} & \langle \forall\, m : m \in G_{sr} : (true \leadsto sr\text{-}in.m) \;\Rightarrow\; (true \leadsto sr\text{-}out.m) \rangle \\ \wedge & \langle \forall\, n : n \in G_{rs} : (true \leadsto rs\text{-}in.n) \;\Rightarrow\; (true \leadsto rs\text{-}out.n) \rangle \\ & \quad \Rightarrow \quad (|ins| \geq k \leadsto |outs| \geq k) \end{array} \tag{3}$$

Here we have followed the convention of considering free variables (k, in this case) to be implicitly universally quantified (i.e. over the whole specification). For conciseness, define the following abbreviations:

$$\begin{array}{rcl} IO.k & \stackrel{\text{def}}{=} & |ins| \geq k \leadsto |outs| \geq k \\ ISR.m & \stackrel{\text{def}}{=} & true \leadsto sr\text{-}in.m \\ OSR.m & \stackrel{\text{def}}{=} & true \leadsto sr\text{-}out.m \\ IRS.m & \stackrel{\text{def}}{=} & true \leadsto rs\text{-}in.m \\ ORS.m & \stackrel{\text{def}}{=} & true \leadsto rs\text{-}out.m \end{array}$$

Using these abbreviations and omitting ranges, the specification becomes:

$$\langle \forall\, m :: ISR.m \;\Rightarrow\; OSR.m \rangle \;\wedge\; \langle \forall\, n :: IRS.n \;\Rightarrow\; ORS.n \rangle \;\Rightarrow\; IO.k \tag{4}$$

4 Proving Progress Properties

This section presents a proof system for progress properties. The theory is derived from the UNITY proof system for leads-to, using a different semantic interpretation. The properties that can be proved are of a restricted form, but many other properties have semantic equivalents of this form. After giving the rules for establishing simple leads-to properties of the form $p \leadsto q$, we propose two new rules, which permit implications of the form $\langle \forall m :: x.m \leadsto y.m \rangle \Rightarrow p \leadsto q$ to be proved. Then we define a *disjunction* operation on certain leads-to properties: for properties $p \leadsto p'$ and $q \leadsto q'$ such that $[\![p \textit{ unless } p']\!]$ and $[\![q \textit{ unless } q']\!]$, we define a single leads-to property that is equivalent to $(p \leadsto p') \vee (q \leadsto q')$ with respect to any behavior. Finally, we define a subclass of the class of progress properties defined in Section 2, and show that every property in this subclass is equivalent to a set of properties that can be proved.

4.1 Leads-to Rules

The basis of the theory is a method enabling us to prove *unless* and *ensures* properties for a given program. The results presented here are independent of the particular programming notation and associated rules for establishing p *unless* q and/or p *ensures* q, and therefore we simply assume their existence [4, 7]. We write $\vdash p$ *unless* q to assert existence of a proof that $[\![p \textit{ unless } q]\!]$, and similarly for p *ensures* q.

The inference rules for deriving leads-to properties are given in Figure 2. In the assertion $\theta \vdash P$, θ represents a set of *hypotheses*, each of which is a leads-to property. The assertion means "the property P can be derived from the *ensures* and *unless* properties of the program plus the hypotheses in θ, using the leads-to rules." When $\theta = \emptyset$ (i.e. the proof has no hypotheses), it is omitted from the assertion.

Note that the sets θ have no effect on what is provable using only the rules given in Figure 2: for any sets θ and θ', $\theta \vdash p \leadsto q$ if and only if $\theta' \vdash p \leadsto q$. (This is easily proved by induction on the length of the derivation.)

$$\frac{\vdash p \textit{ ensures } q}{\theta \vdash p \leadsto q} \text{ (Promotion)} \qquad \frac{\theta \vdash p \leadsto r, \quad \theta \vdash r \leadsto q}{\theta \vdash p \leadsto q} \text{ (Transitivity)}$$

$$\frac{\langle \forall m :: \theta \vdash p.m \leadsto q \rangle}{\theta \vdash \langle \exists m :: p.m \rangle \leadsto q} \text{ (Disjunction)} \qquad \frac{\theta \vdash p \leadsto q, \quad \vdash r \textit{ unless } r'}{\theta \vdash p \wedge r \leadsto (q \wedge r) \vee r'} \text{ (PSP)}$$

Figure 2: Basic Proof Rules for Leads-to

The PSP Rule (Progress-Safety-Progress) permits a leads-to property to be derived from an *unless* property and another leads-to property. It is not given as an inference rule in the original UNITY logic [7], but is derived as a metatheorem. However, that proof (by induction on the length of the derivation of $p \leadsto q$) does not go through when leads-to properties can be introduced *by assumption*, as they will be in the next section. Therefore PSP is postulated here as one of the basic rules. It is clear that this does not

change the power of the basic theory, since PSP was derivable as a theorem before. The completeness of the UNITY logic is well-known and has been discussed extensively in the literature [8, 15, 16].

4.2 Conditional Properties

The interpretation of *conditional properties* used here differs from that of UNITY. In UNITY, a conditional property with hypothesis P and conclusion Q —where the hypothesis and conclusion apply to the same program— has the meaning $[\![P]\!] \Rightarrow [\![Q]\!]$. Here, the property $P \Rightarrow Q$ is interpreted as the stronger specification $[\![P \Rightarrow Q]\!]$. However, the method of proving conditional properties is the same as that used in UNITY: we allow assumption of leads-to properties as *hypotheses*. The soundness of the method under the different semantic interpretation follows from the fact that the proof rules given above *are valid for individual behaviors*.

The rules for introducing (sets of) hypotheses and deriving conditional leads-to properties are shown in Figure 3. In these rules, the (bound) variable m in the expressions $\{x.m \leadsto y.m\}$ and $\langle \forall\, m :: \ldots \rangle$ implicitly ranges over some set that is fixed throughout each rule.

$$\frac{\langle \forall\, m :: \vdash x.m \ \textit{unless}\ y.m \rangle}{\theta, \{x.m \leadsto y.m\} \vdash x.m \leadsto y.m} \text{ (Assumption)}$$

$$\frac{\theta, \{x.m \leadsto y.m\} \vdash p \leadsto q}{\theta \vdash \langle \forall\, m :: x.m \leadsto y.m \rangle \ \Rightarrow\ (p \leadsto q)} \text{ (Discharge)}$$

Figure 3: Proof Rules for Conditional Leads-to Properties

The Assumption rule justifies the introduction of *hypothesis* properties into a proof, so long as the corresponding *unless* properties hold; the Discharge rule allows a proof with hypotheses to be converted to a proof (of an implication), without hypotheses.

Next we give two results regarding the soundness and (relative) completeness of the rules just given.

Theorem 0. (Soundness.) Let $x.m \leadsto y.m$ be a leads-to property for each m in some set. Then

$$\{x.m \leadsto y.m\} \vdash p \leadsto q \ \Rightarrow\ [\![\langle \forall\, m :: x.m \leadsto y.m \rangle \ \Rightarrow\ (p \leadsto q)]\!]$$

□

Proof. The proof is by induction on the length of the derivation of $p \leadsto q$, and uses the soundness of the underlying logic. There are two base cases (Promotion and Assumption) and three step cases (Transitivity, Disjunction, and PSP). Space constraints preclude inclusion of the proof; Tsay and Bagrodia have proved a similar result [20]. □

Theorem 1. (Completeness.) Let $x.m \leadsto y.m$ be a leads-to property for each m in some countable set, and assume the following:

$$\langle \forall\, m :: [\![x.m \ \textit{unless}\ y.m]\!] \rangle \tag{5}$$

$$[\![\langle \forall\, m :: x.m \leadsto y.m \rangle \ \Rightarrow\ (p \leadsto q)]\!] \tag{6}$$

$$[\![p \ \textit{unless}\ q]\!] \tag{7}$$

Then $\{x.m \leadsto y.m\} \vdash p \leadsto q$. □

The proof of Theorem 1 involves construction of a metric function based on the structure of the state space implied by assumptions (5)–(6) [6], and uses the completeness of the underlying UNITY logic.

4.3 Disjunction

We would like to be able to prove boolean combinations of leads-to properties, but so far our theory allows us to prove only implications. However, for each disjunction of leads-to properties, if certain *unless* properties hold, there exists an equivalent simple leads-to property.

Theorem 2. If $[\![p \ \textit{unless}\ p']\!]$ and $[\![q \ \textit{unless}\ q']\!]$, then

$$[\![(p \leadsto p') \ \vee\ (q \leadsto q') \ \equiv\ (p \wedge q \leadsto p' \vee q')]\!]$$

□

Proof. For an infinite sequence $\overline{w}$ and state predicate p, define $\overline{w} \nearrow p$ ("$\overline{w}$ converges to p") to mean that $\overline{w}$ has an infinite suffix in which every state is a p-state. The following observations follow from the hypotheses and this definition for any $\overline{w}$, p and q:

(i) $\overline{w} \nearrow (p \wedge q) \equiv (\overline{w} \nearrow p) \wedge (\overline{w} \nearrow q)$

(ii) If $[\![p \ \textit{unless}\ q]\!]$, then any behavior $\overline{w}$ satisfies $p \leadsto q$ if and only if $\neg(\overline{w} \nearrow (p \wedge \neg q))$.

(iii) $[\![p \wedge q \ \textit{unless}\ p' \vee q']\!]$

Now we calculate:

$\overline{w}$ satisfies $(p \leadsto p' \ \vee\ q \leadsto q')$
= { definition of $\vee$ }
($\overline{w}$ satisfies $p \leadsto p'$) $\vee$ ($\overline{w}$ satisfies $q \leadsto q'$)
= { Observation (ii) }
$\neg(\overline{w} \nearrow (p \wedge \neg p')) \vee \neg(\overline{w} \nearrow (q \wedge \neg q'))$
= { DeMorgan's Law }
$\neg(\overline{w} \nearrow (p \wedge \neg p') \ \wedge\ \overline{w} \nearrow (q \wedge \neg q'))$
= { Observation (i) }
$\neg(\overline{w} \nearrow (p \wedge \neg p' \wedge q \wedge \neg q'))$

$$
\begin{array}{ll}
= & \{ \text{ predicate calculus } \} \\
& \neg(\overline{w} \nearrow p \wedge q \wedge \neg(p' \vee q')) \\
= & \{ \text{ Observations (ii) and (iii) } \} \\
& \overline{w} \text{ satisfies } p \wedge q \rightsquigarrow p' \vee q'
\end{array}
$$

□

This result says that if the required *unless* properties hold, any disjunction can be replaced, in any property, by a single leads-to property. For leads-to properties P and Q for which the corresponding *unless* properties hold, we now *define* $P \ \vee \ Q$ to be $p \wedge q \rightsquigarrow p \vee q$. We need to show that this definition of $\vee$ enjoys the usual properties of disjunction—e.g., it is idempotent, associative, and monotonic. These properties are necessary to make full use of conditional progress specifications; in particular, monotonicity is needed to prove a protocol's overall progress specification from its conditional specification and a channel specification, as in our example.

It is obvious from the definition that disjunction is idempotent and associative. A general form of monotonicity would be:

$$\frac{\langle \forall\, m :: \theta \vdash P.m \ \vee \ R \rangle, \quad \theta \vdash \langle \forall\, m :: P.m \rangle \ \Rightarrow \ Q}{\theta \vdash Q \ \vee \ R}$$

where each $P.m$, and R and Q, are simple leads-to properties. Expanding $P.m$, R, and Q respectively to $p.m \rightsquigarrow p'.m$, $r \rightsquigarrow r'$, and $q \rightsquigarrow q'$, we get the first rule shown in Figure 4. A simpler form, namely the special case in which m ranges over a singleton set, is also shown. The validity of the general rule can be shown by induction on the length of the derivation of the second premise [6].

$$\frac{\langle \forall\, m :: \theta \vdash p.m \wedge r \rightsquigarrow p'.m \vee r' \rangle, \quad \theta \vdash \langle \forall\, m :: p.m \rightsquigarrow p'.m \rangle \ \Rightarrow \ (q \rightsquigarrow q')}{\theta \vdash q \wedge r \rightsquigarrow q' \vee r'}$$

$$\frac{\theta \vdash p \wedge r \rightsquigarrow p' \vee r', \quad \theta \vdash (p \rightsquigarrow p') \ \Rightarrow \ (q \rightsquigarrow q')}{\theta \vdash q \wedge r \rightsquigarrow q' \vee r'} \text{ (Simple Monotonicity)}$$

Figure 4: Monotonicity Rules for Disjunction

4.4 Putting Progress Properties in Provable Form

We say a property is in *provable form* iff it satisfies one of the following conditions:

- It is a simple leads-to property of the form $p \rightsquigarrow q$.
- It is an implication of the form $\langle \forall\, m :: x.m \rightsquigarrow y.m \rangle \ \Rightarrow \ (p \rightsquigarrow q)$, where m ranges over some set.

Using the results of the previous section, we can now define a class $\mathcal{P}$ of progress properties such that for each member of the class there is an equivalent set of provable-form properties. More precisely, for any property X in this class, there exists a collection of properties $Q.0, \ldots Q.N$ such that each $Q.i$ is in provable form, and

$$[\![X]\!] \equiv [\![Q.0 \wedge Q.1 \wedge \ldots \wedge Q.N]\!]$$

The right-hand side of the above is equivalent to

$$[\![Q.0]\!] \wedge [\![Q.1]\!] \wedge \ldots \wedge [\![Q.N]\!]$$

and thus the collection of properties constitutes a specification that is exactly equivalent to X with respect to the given program.

For a given program, define the class $\mathcal{P}$ of progress properties to be the smallest class satisfying the following conditions:

- For each leads-to property $p \rightsquigarrow q$ such that $[\![p \textit{ unless } q]\!]$, $p \rightsquigarrow q$ is in $\mathcal{P}$.
- If X and Y are in $\mathcal{P}$, then so are $X \vee Y$, $X \wedge Y$, and $X \Rightarrow Y$.

Note that $\mathcal{P}$ contains arbitrarily-nested implications like $(\cdots (P \Rightarrow Q) \cdots \Rightarrow R) \Rightarrow X$. However, we can show that for every property in $\mathcal{P}$ there is an equivalent specification consisting entirely of provable-form properties.

Theorem 3. For every property $X \in \mathcal{P}$, there exists a finite set W_X of properties such that

(i) each property in W_X is in provable form, and

(ii) any behavior satisfies X if and only if it satisfies every property in W_X.

□

Proof. We only sketch the proof here; the main details may be found in [5]. The proof proceeds in two steps. First, for each property in $\mathcal{P}$ we define an equivalent property in *conjunctive normal form* (CNF) — a conjunction of disjunctions of simple leads-to properties, in which each disjunction contains at least one simple leads-to property that is not negated. We then show that any conjunction of such disjunctions is equivalent to a set of provable-form properties.

Let X be any property in $\mathcal{P}$. We define an equivalent CNF property by induction on the structure of X (according to the definition of $\mathcal{P}$). For the basis, X is a simple leads-to property; we define the equivalent formula to be X itself. There are three step cases, according to whether X is of the form $Y \wedge Z$, $Y \vee Z$, or $Y \Rightarrow Z$. In the third case, the transformation required to apply the inductive hypothesis may result in an equivalent property that is exponentially larger than X.

The equivalent property whose existence is thus established is a conjunction with conjuncts of the form

$$Y_1 \vee \ldots \vee Y_N \vee Z_1 \vee \ldots \vee Z_M$$

where N may be 0, $M \geq 1$, and all the Ys are negated and none of the Zs are. The above is equivalent to

$$Y_1 \wedge \ldots \wedge Y_N \Rightarrow Z_1 \vee \ldots \vee Z_M$$

if $N > 0$ and to the disjunction of the Z_i's otherwise. Moreover, each Z_i is a simple leads-to property in $\mathcal{P}$, so the corresponding *unless* property holds for it; thus the disjunction $Z_0 \vee \ldots Z_M$ can be replaced by an equivalent single leads-to property Z'. If $N = 0$, the equivalent property for the conjunct is just Z'; otherwise it is

$$\langle \forall m : 1 \leq m \leq N : Y_m \rangle \Rightarrow Z'$$

In either case the equivalent property is in provable form. Thus we can construct an equivalent property in provable form for each conjunct; the conjunction of these is equivalent to X, as required. □

Because universal quantification distributes over conjunction, we can actually enlarge the class of properties having provable-form equivalents to include some quantified properties.

Theorem 4. If $P.m \in \mathcal{P}$ for each m in some set, then there exists a set of provable-form properties, the conjunction of which is equivalent to $\langle \forall m :: P.m \rangle$. □

Proof. The set is just the union of the sets of provable-form equivalents for the properties $P.m$. □

5 Example Revisited

Next we consider the application of the results of the previous section to the specification given earlier.

Using the predicate calculus, it is straightforward to show that the specification given in (4) is equivalent to the following:

$$\begin{array}{lllllll} \langle \exists m, n :: & (& & & ISR.m \vee IRS.n \vee IO.k &) & \\ \wedge & (& OSR.m & \Rightarrow & IRS.n \vee IO.k &) & \\ \wedge & (& ORS.n & \Rightarrow & ISR.m \vee IO.k &) & \\ \wedge & (& OSR.m \wedge ORS.n & \Rightarrow & IO.k &) \rangle & \end{array} \tag{8}$$

Because the corresponding *unless* property holds for each leads-to property in the original specification, we can use the disjunction operator defined earlier to remove the individual disjunctions in the above property. However, the resulting property is not in provable form because of the enclosing existential quantification. That is, the above specification has the general form

$$[\![\langle \exists m :: X(m,k) \rangle]\!]$$

but the proof theory gives no way to prove that a program satisfies specifications of that form. If the message sets G_{sr} and G_{rs} (i.e. the ranges of m and n) are finite, (8) is

actually a finite disjunction. It is thus in the set $\mathcal{P}$ defined earlier, and has a provable-form equivalent. However, as noted earlier, the transformation to provable form can result in an exponential blowup in the size of the specification: in this case the equivalent set of provable form properties contains an implication $\langle \forall\, m : m \in G' : Q.m \rangle \;\Rightarrow\; \cdots$ for every subset G' of $G_{sr} \cup G_{rs}$.

A more reasonable approach is Skolemization. The specification (8) is equivalent to one of the form

$$\langle \exists\, f :: \langle \forall\, \overline{w} :: X(f(\overline{w}, k), k) \rangle \rangle$$

where f is a function from behaviors and natural numbers to messages of the type carried on the lower-level channels, and $\overline{w}$ ranges over the behaviors of the program. Such a specification may be proved by exhibiting a particular function of the appropriate type. However, we must take care in defining that function, because in general the choice of message depends on the *entire* behavior, which is not "visible" at any state. The key observation is that the value of the function is irrelevant for any behavior in which $IO.k$ holds. In other words, we only need to identify a *particular* lower-level message for a given behavior so the protocol can "blame" the lower-level channels if the higher-level progress specification $IO.k$ is not satisfied in that behavior. We therefore define an auxiliary variable for each channel, whose value is constant in some infinite suffix of any behavior that does *not* satisfy $IO.k$ for some k. Fortunately this is typically not difficult: there is generally *some* lower-level message that will be transmitted repeatedly if progress is not made.

For the example, assume that auxiliary variables have been added so that the state functions nxt_{sr} and nxt_{rs} denote the messages that are transmitted infinitely often if $IO.k$ is not satisfied for some k. The above specification becomes

$$\begin{array}{llllll} & (& & & ISR.nxt_{sr} \;\vee\; IRS.nxt_{rs} \;\vee\; IO.k &) \\ \wedge & (& OSR.nxt_{sr} & \Rightarrow & IRS.nxt_{rs} \;\vee\; IO.k &) \\ \wedge & (& ORS.nxt_{rs} & \Rightarrow & ISR.nxt_{sr} \;\vee\; IO.k &) \\ \wedge & (& OSR.nxt_{sr} \;\wedge\; ORS.nxt_{rs} & \Rightarrow & IO.k &) \end{array} \tag{9}$$

Now we need only replace the disjunctions with simple leads-to properties in order to have a provable-form specification. Thanks to the fact that the corresponding *unless*-properties hold for *ISR*, *IRS* and *IO*, this is a valid transformation. The final specification then consists of the following four properties:

$$|ins| \geq k \rightsquigarrow sr\text{-}in.nxt_{sr} \vee rs\text{-}in.nxt_{rs} \vee |outs| \geq k \tag{10}$$

$$(true \rightsquigarrow sr\text{-}out.nxt_{sr}) \;\Rightarrow\; (|ins| \geq k \rightsquigarrow rs\text{-}in.nxt_{rs} \vee |outs| \geq k) \tag{11}$$

$$(true \rightsquigarrow rs\text{-}out.nxt_{rs}) \;\Rightarrow\; (|ins| \geq k \rightsquigarrow sr\text{-}in.nxt_{sr} \vee |outs| \geq k) \tag{12}$$

$$(true \rightsquigarrow sr\text{-}out.nxt_{sr}) \;\wedge\; (true \rightsquigarrow rs\text{-}out.nxt_{rs}) \;\Rightarrow\; (|ins| \geq k \rightsquigarrow |outs| \geq k) \tag{13}$$

It remains to show that the desired high-level progress property R can be proved from the above and the channel specifications (1) and (2). The proof follows:

0. $|ins| \geq k \rightsquigarrow sr\text{-}out.nxt_{sr} \vee rs\text{-}in.nxt_{rs} \vee |outs| \geq k$ { (1), (10), monotonicity }

1.	$\|ins\| \geq k \rightsquigarrow rs\text{-}out.nxt_{rs} \vee sr\text{-}in.nxt_{sr} \vee \|outs\| \geq k$	{ (2), (10), monotonicity }
2.	$\|ins\| \geq k \rightsquigarrow rs\text{-}in.nxt_{rs} \vee \|outs\| \geq k$	{ **0**, (11), mono., idemp. }
3.	$\|ins\| \geq k \rightsquigarrow sr\text{-}in.nxt_{sr} \vee \|outs\| \geq k$	{ **1**, (12) simple mono.,idemp. }
4.	$\|ins\| \geq k \rightsquigarrow rs\text{-}out.nxt_{rs} \vee \|outs\| \geq k$	{ **2**, (2), simple monotonicity }
5.	$\|ins\| \geq k \rightsquigarrow sr\text{-}out.nxt_{sr} \vee \|outs\| \geq k$	{ **3**, (1), simple monotonicity }
6.	$\|ins\| \geq k \rightsquigarrow \|outs\| \geq k$	{ **4**, **5**, (12), monotonicity }

6 Discussion and Conclusions

The results presented here suggest the following approach to specification of conditional progress properties of protocols:

1. Write down the specification in the form $X \Rightarrow Y$, where X follows from the properties expected of the environment, and Y implies the properties required of the protocol.

2. Using predicate calculus, try to massage the specification into an equivalent conjunction of provable form properties and conditional properties. Whether this is possible depends on the scopes of the quantifiers appearing in the specification.

3. If the result is in provable form except for enclosing existential quantifications, add auxiliary variables and state functions to enable replacement of the existential quantifier with a state function.

This approach is not limited to protocols; indeed, various examples of programs exhibiting progress dependencies like those considered here have appeared in the literature [7, 9, 17]. The approach is most likely to be useful for specifying and verifying *algorithms*, as opposed *implementations*, e.g. of "real" protocols. The latter typically have features such as timeouts and fixed bounds on the number of retransmissions before giving up on delivery. In designing and verifying an algorithm, however, it is better to abstract from such details and characterize the conditions needed for correctness as precisely and generally as possible. On the other hand, the ability to precisely state the progress expected of the environment may make it easier in some cases to specify less-abstract forms of an algorithm.

In related work, Tsay and Bagrodia [19] have given a (different) relatively complete inference rule for proving UNITY conditional properties of the form

"Hypothesis: $true \rightsquigarrow p$; Conclusion: $true \rightsquigarrow q$"

which may be used to specify strong fairness.

The results given here assume that p *unless* q hold for each leads-to property in the specification. However, this condition can be relaxed, using techniques described elsewhere [2, 5]. While this requirement does not seem unreasonable, it does require that any progress obligation of the system (or its environment) remain "visible" at the interface to the system until it is discharged. If a progress requirement $p \rightsquigarrow q$ is

"distributed"—in the sense that p can only become true at one location in the system and q can only become true at a different location—then p *unless* q may introduce additional synchronization requirements, because it requires that whenever p is falsified, q holds or becomes true in the same program step.

The results presented here were developed for use with a *compositional* theory of module specifications [3, 4]. (A compositional theory is one in which the properties that can be proved of a component are not changed by composition with other components.) The work was motivated by the need to express the properties "expected" from the environment of a component in the component specification itself, and in such a way that correctness of the component can be proved *in isolation*, i.e. without having the complete specification of the environment. Abadi and Lamport [1] have studied the *semantic* conditions under which this is possible. Lam and Shankar [11] proposed a theory of modules and interfaces which supports isolated proofs of correctness of individual modules, for systems in which the dependencies among components form a well-founded structure. The theory presented here is aimed at removing the latter restriction, i.e. to allow proof of correctness of a composite without imposing any order on the component structure.

References

[1] Martín Abadi and Leslie Lamport. Composing specifications. In *Stepwise Refinement of Distributed Systems (LNCS 430)*. Springer-Verlag, 1990.

[2] Ken Calvert. Eliminating disjunctions of leads-to properties. *Information Processing Letters*, 49:189–194, 1994.

[3] Kenneth L. Calvert. Module composition and refinement: Extending the Lam-Shankar theory. Technical Report GIT–CC–91/58, College of Computing, Georgia Institute of Technology, 1991.

[4] Kenneth L. Calvert. Module composition and refinement with applications to protocol conversion. In *Proceedings XII Symposium on Protocol Specification, Testing, and Verification, Orlando, Florida*. North-Holland, June 1992.

[5] Kenneth L. Calvert. Specifying progress properties with leads-to. Technical Report GIT–CC–92/59, College of Computing, Georgia Institute of Technology, December 1992. available via anonymous FTP from ftp.cc.gatech.edu.

[6] Kenneth L. Calvert. Reasoning about conditional progress properties. Technical Report GIT–CC–94/03, College of Computing, Georgia Institute of Technology, March 1994.

[7] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[8] Rob Gerth and Amir Pnueli. Rooting UNITY. In *Proceedings of the Fifth International Workshop on Software Specification and Design, Pittsburgh*, May 1989.

[9] Simon S. Lam and A. Udaya Shankar. Specifying modules to satisfy interfaces: A state transition system approach. Technical Report TR88-30, University of Texas at Austin, Department of Computer Sciences, August 1988. (revised September 1990).

[10] Simon S. Lam and A. Udaya Shankar. A relational notation for state transition systems. *IEEE Transactions on Software Engineering*, 16(7):755–775, July 1990.

[11] Simon S. Lam and A. Udaya Shankar. Understanding interfaces. In *Proceedings of the Fourth International Conference on Formal Description Techniques (FORTE), Sydney, Australia*, November 1991.

[12] Leslie Lamport. A temporal logic of actions. Technical Report Research Report 57, DEC Systems Research Center, April 1990.

[13] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings ACM Symposium on Principles of Distributed Computing, Vancouver, BC.*, 1987.

[14] Zohar Manna and Amir Pnueli. *Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[15] Jan Pachl. A simple proof of a completeness result for *leads-to* in the UNITY logic. *Information Processing Letters*, 41:35–38, 1992.

[16] Beverly A. Sanders. Eliminating the substitution axiom from UNITY logic. Technical Report 128, Eidgenössische Technische Hochschule Zürich, Institut für Computersysteme, May 1990.

[17] Beverly A. Sanders. Stepwise refinement of mixed specifications of concurrent programs. In *Proceedings of IFIP TC2/WG2.3 Working Conference on Programming Concepts and Methods, Sea of Gallilee, Isreal, April 1990*. Elsevier Science Publishers B.V., 1990.

[18] A. Udaya Shankar and Simon S. Lam. Time-dependent distributed systems: Proving safety, liveness, and real-time properties. *Distributed Computing*, 1987(2):61–78, 1987.

[19] Yih-Kuen Tsay and Rajive Bagrodia. Deducing fairness properties in UNITY logic—a new completeness result. to appear in ACM TOPLAS.

[20] Yih-Kuen Tsay and Rajive Bagrodia. Operational implication of conditional UNITY properties. In *Proceedings of DIMACS Workshop on Specifications of Parallel Algorithms*, May 1994.