Reasoning about Program Composition*

K. Mani Chandy† Beverly A. Sanders ‡

UF CISE Technical Report 96-035
November 18, 1996

Abstract

This paper presents a theory for concurrent program composition based on a predicate transformer call the the *weakest guarantee* and a corresponding binary relation *guarantees*. The theory stems from a novel view of rely-guarantee techniques for reasoning about program composition and provides a general and uniform framework for handling temporal properties as well as other kinds of program properties such as refinement and encapsulation.

1 Introduction

The contribution of this paper is a predicate-transformer based theory for reasoning about the composition of concurrent programs. This section contains the motivation for this contribution and a discussion of the central issues.

The predicate transformers *wp* and *wlp* provide an elegant basis for reasoning about sequential programs because they focus attention on the most fundamental aspects of these programs: their initial and final states [DS90]. By identifying a program with its predicate transformer, we can reason about programs using the universal notation of the predicate calculus. Predicate transformer theory is the fundamental theory of sequential programming; so extending the theory to concurrent composition is of interest.

Concurrent programs cannot be specified exclusively in terms of initial and final states. Temporal logic, which deals with computations of programs, [Pnu81, CM88, Lam94, MP91] has been used for specifying and reasoning about concurrent programs. Our theory uses concepts from temporal logic, but our contribution is to extend predicate transformer theory to concurrent programming.

*Supported in part by U.S. Air Force Office of Scientific Research and the University of Florida

†Department of Computer Science, California Institute of Technology, m/c 256-80, Pasadena, CA 91125, mani@cs.caltech.edu.

‡Department of Computer and Information Science and Engineering, P.O. Box 116120, University of Florida, Gainesville, FL 32611-6120, sanders@cise.ufl.edu.

Temporal properties are not usually compositional. Even if all computations of programs F and G satisfy a temporal logic formula, computations of $F||G$ may not (where $||$ is a program composition operator). We propose a common theoretical foundation, based on predicate transformers, for proving properties of composed programs from properties of their components.

In the literature on program composition, components have been specified with properties variously called rely/guarantee [Jon83, Sta85], hypothesis/conclusion [CM88], assumption/commitment [Col94, CK95], offers/using [LS94, LS92] and assumption/guarantee [AL93, AL95]. The common idea is that assumptions about the environment form part of the specification of a component. In this paper, we develop this idea using predicate-transformers and a different view of how to specify the environment. This leads to a simple theory of program composition.

The main aspects of our approach are summarized next:

1. We define a program property as a predicate on programs. Our theory uses predicates on both programs and states.

2. Our theory provides a uniform framework for handling temporal properties and other kinds of program properties such as refinement and encapsulation.

3. We specify program components with the dyadic operator *guarantees* on program properties where for a program F and properties X and Y, (X *guarantees* Y) is a property of program F if and only if all programs that have F as a component and have X as a property also have Y as a property. In contrast to the semantics for rely/guarantee specifications in the literature, the antecedent X is a property of the system $F||H$ in which the component F is embedded, not the environment H; likewise the consequent Y is also a property of the same system $F||H$. The *guarantees* property has many of the nice properties of implication because both X and Y in (X *guarantees* Y) are properties of the *same* system. Theories in which X refers to one program and Y refers to another program appear to be more complex than our theory and often restrict X to be safety properties.

4. In analogy with the $wp.F$ calculus, we define a property transformer wg where for a program F and properties X and Y: $wg.F.Y$ is the weakest X such that (X *guarantees* Y) is a property of F. The property transformer $wg.F$ is monotonic, universally conjunctive, and idempotent. Like wp in sequential programming, wg anchors the theory of guarantees properties in the predicate calculus.

5. We explore simple compositional properties called *all-component* and *exists-component* properties. A program has an all-component property if and

only if all its components have that property. A program has an exists-component property if there exists a component that has that property. We also propose compositional methods of proving properties that are neither all-component nor exists-component by using the guarantees operator.

6. We propose a method for specifying and reasoning about program components that has two parts:

   - The theory for guarantees properties that relies on simple properties of $\|$ such as associativity and existence of an identity and is largely independent of any particular program model.
   - Model-specific proof rules for showing that a program satisfies a guarantees property.

   This aspect of our approach is reminiscent of the algebraic specification method Larch [GHW85] which also has a two-tiered approach with language-independent and language-specific parts.

In the remainder of the paper, we develop the theory of guarantees and wg, give an example of model-specific proof rules using a programming model similar to UNITY, and develop an example program in the model. We conclude with some observations.

2 Preliminary definitions

We employ predicates on states and predicates on programs. A *program property* is a predicate on programs, and a *state predicate* is a predicate on states. We use letters X, Y and Z for program properties, p, q and r for state predicates, and F, G and H for programs.

The application of a function f to an argument x is denoted by $f.x$ and function application associates to the left. Therefore, for a program property Z and a program F, the value of $Z.F$ is boolean: $Z.F$ has value *true* if and only if property Z holds for program F.

A fundamental property of a program is whether or not it conforms to some criteria. The precise definition of conforms depends on the program model, but typically involves satisfaction of the kinds of typing and encapsulation constraints found in many programming languages and enforced by compilers. We restrict our theory to conformant programs. For instance, a conformant program can be defined as one that compiles successfully, and we do not want our theory to have to deal with programs that do not compile. We introduce a property *conforms*, where $conforms.F$ holds if and only if program F conforms to the criteria of interest.

A program is a state transition system. A computation of a program is an infinite sequence of state transitions that satisfies certain restrictions, discussed later. An important kind of program property is a predicate on computations that holds for *all* computations of the program. For example, for state predicates p and q, we define a predicate $p \leadsto q$ on computations as follows: $p \leadsto q$ holds for a computation c exactly when, for each point in c at which p holds, q holds at that point or a later point in c [Lam94, CM88]. The program property $p \leadsto q$ holds for a program F exactly when $p \leadsto q$ holds for all computations of F. The notion of a property is not, however, restricted to temporal properties such as $\leadsto$. Examples of other kinds of properties will be seen in the paper.

We use formulae that have both state-predicates and program properties, as, for instance in:

$$(p \leadsto q).F \wedge (q \leadsto r).F \Rightarrow (p \leadsto r).F$$

As in [DS90] we use $[p]$ to denote the boolean: state predicate p holds in all states. Similarly, we use $[X]$ to denote the boolean: program property X holds for all conformant programs. For instance, the previous formula which holds for all programs F can be expressed as:

$$[(p \leadsto q) \wedge (q \leadsto r) \Rightarrow (p \leadsto r)]$$

If there is any possibility of ambiguity when using $[X]$, we use the explicit notation: $(\forall F : \mathit{conforms}.F : X.F)$.

3 Program Composition and the Guarantees Operator

3.1 Parallel composition

In this section, we assume the existence of a program composition operator $||$, and use it to define an operator, called *guarantees*, on program properties. We require $||$ to be associative and have an identity *SKIP*. We also require that $||$ and *conforms* satisfy

$$(\forall F, G :: \mathit{conforms}.F||G \Rightarrow \mathit{conforms}.F. \wedge \mathit{conforms}.G) \tag{1}$$

An immediate consequence of (1) is that either *SKIP* is conformant or there are no conformant programs. Note that $||$ has higher binding power than function application.

Most theorems require that $||$ be commutative as well, and in such cases we make the commutativity assumption explicit. For commutative $||$, we postulate the following pairwise property: if the composition of each pair of programs in a set is conformant, then the composition of all programs in the set is conformant

as well:
For any set of programs $\mathcal{G}$:

$$(\forall G, G' : G, G' \in \mathcal{G} : \mathit{conforms}.G \| G') \Rightarrow \mathit{conforms}.(\| G : g \in \mathcal{G} : G) \qquad (2)$$

From (1) and (2):

$$(\forall G, G' : G, G' \in \mathcal{G} : \mathit{conforms}.G \| G') = \mathit{conforms}.(\| G : g \in \mathcal{G} : G) \qquad (3)$$

In this paper, we are only interested in conformant programs. For brevity, we often leave the restriction to conformant programs implicit. For example, we may write quantification over a set of programs as $(\forall H :: X.(F\|H) \Rightarrow Y.(F\|H))$, instead of $(\forall H : \mathit{conforms}.(F\|H) : X.(F\|H) \Rightarrow Y.(F\|H))$.

3.2 Definition of guarantees

The dyadic operator *guarantees* on program properties is defined as follows. For program properties X and Y, we define a program property $(X \textit{ guarantees } Y)$ as:

$$(X \textit{ guarantees } Y).F \equiv (\forall H : \mathit{conforms}.F\|H : X.F\|H \Rightarrow Y.(F\|H)) \qquad (4)$$

Therefore, $(X \textit{ guarantees } Y)$ is a property of a program F if and only if all conformant programs that have F as a component and have X as a property also have Y as a property. From the definition of guarantees:

$$[X \Rightarrow Y] \Rightarrow [X \textit{ guarantees } Y] \qquad (5)$$

We are, however, interested in cases where the fact that F is a component allows the left side of the guarantees properties to be weaker than the right.

In contrast to the rely/guarantee specifications mentioned in the introduction, the meaning of $(X \textit{ guarantees } Y).F$ is *not* that if the *environment* H of F has property X then the composed system $F\|H$ has property Y — it is that if the *composed* system $F\|H$ has property X then the composed system $F\|H$ has property Y. By using only properties of the composed system instead of the properties of the environment, we obtain a simple theory in which *guarantees* enjoys many of the properties of implication.

3.3 Theorems about Guarantees

The proofs of these theorems are straightforward and are not given here. In these theorems, we use $\mathcal{X}$ and $\mathcal{Y}$ to denote sets of program properties.

The next five properties are analogous to properties of implication.

Universally Disjunctive for Left Operand

$$[(\forall X : X \in \mathcal{X} : X \textit{ guarantees } Y) \;\equiv\; (\exists X : X \in \mathcal{X} : X) \textit{ guarantees } Y]$$

Universally Conjunctive for Right Operand

$$[(\forall Y : Y \in \mathcal{Y} : X \ \mathit{guarantees} \ \ Y) \ \ \equiv \ \ X \ \mathit{guarantees} \ \ (\forall Y : Y \in \mathcal{Y} : Y)]$$

Transitive

$$[(X \ \mathit{guarantees} \ \ Y) \wedge (Y \ \mathit{guarantees} \ \ Z) \Rightarrow (X \ \mathit{guarantees} \ \ Z)]$$

Shunting

$$[(X \ \mathit{guarantees} \ \ Y) \ \ \equiv \ \ (\mathit{true} \ \mathit{guarantees} \ \ (X \Rightarrow Y))]$$

Contrapositive

$$[(X \ \mathit{guarantees} \ \ Y) \ \ \equiv \ \ (\neg Y \ \mathit{guarantees} \ \ \neg X)]$$

Exists-component Guarantees properties are exists-component properties.

$$(X \ \mathit{guarantees} \ Y).G \equiv (\forall H :: (X \ \mathit{guarantees} \ Y).G \| H)$$

Therefore, guarantees properties of a program G are inherited by all programs that have G as a component.

4 Weakest guarantees

4.1 Definition of component

We define a function *component* from programs to program properties as follows: For a program F, $\mathit{component}.F$ is a property that holds for all programs that have F as a component and only such programs.

$$\mathit{component}.F.H \equiv (\exists G : \mathit{conforms}.F \| G : F \| G = H)$$

The definition of equality of programs depends on the program model and how parallel composition is defined.

From the definition of *component*:

$$\mathit{component}.G.H \wedge \mathit{component}.H.F \Rightarrow \mathit{component}.G.F$$

$$(\forall F :: \mathit{component}.F.F)$$

and for commutative $\|$:

$$[\mathit{component}.F \| H \ \ \Rightarrow \ \ \mathit{component}.F \wedge \mathit{component}.H].$$

Next, we prove the following formula that is helpful in deriving theorems about the weakest guarantee.

$$(X \ \mathit{guarantees} \ Y).F \ \ \equiv \ \ [X \Rightarrow (\mathit{component}.F \Rightarrow Y)] \tag{6}$$

Proof:

$$
\begin{array}{ll}
 & (X \ \mathit{guarantees}\ Y).F \\
\equiv & \quad \{ \text{ definition of guarantees } \} \\
 & (\forall H :: X.H \wedge \ \mathit{component}.F.H \Rightarrow Y.H) \\
\equiv & \quad \{\text{meaning of } [\,]\ \} \\
 & [X \wedge \ \mathit{component}.F \Rightarrow Y] \\
\equiv & \quad \{ \text{ predicate calculus } \} \\
 & [X \Rightarrow (\mathit{component}.F \Rightarrow Y)]
\end{array}
$$

4.2 The property transformer $wg.F$

A predicate transformer is a function from predicates to predicates. Likewise, a property transformer is a function from properties to properties. Motivated by the weakest precondition, *wp*, we define *weakest guarantee*, *wg*, and present theorems about *wg*. For a program F, $wg.F$ is a property transformer defined as follows: for a property Y, $wg.F.Y$ is the weakest property X such that $(X \ \mathit{guarantees}\ Y).F$.

$$wg.F.Y = \text{ weakest } X : (X \ \mathit{guarantees}\ Y).F$$

From (6):

$$[wg.F.Y \ \equiv \ \mathit{component}.F \Rightarrow Y] \tag{7}$$

From this definition, it follows that the property transformer $wg.F$ is monotonic, universally conjunctive, and idempotent.

4.3 Theorems about wg

We give several theorems about *wg*. (We omit straightforward proofs.)

Component theorem

$$[\mathit{component}.F \Rightarrow (wg.F.Y \equiv Y)] \tag{8}$$

Conjunction, disjunction and composition theorem For commutative $\|$, program property X, and nonempty set of programs $\mathcal{F}$:

$$(\exists F : F \in \mathcal{F} : wg.F.X).(\|F : F \in \mathcal{F} : F) \equiv X.(\|F : F \in \mathcal{F} : F)$$

and

$$(\forall F : F \in \mathcal{F} : wg.F.X).(\|F : F \in \mathcal{F} : F) \equiv X.(\|F : F \in \mathcal{F} : F)$$

Proof: Let $\overline{\mathcal{F}} = (\|F : F \in \mathcal{F} : F)$. Then

$$
\begin{array}{ll}
 & \text{true} \\
\Rightarrow & \quad \{ \ (8) \ \} \\
 & (\forall F : F \in \mathcal{F} : \mathit{component}.F.\overline{\mathcal{F}} \Rightarrow (\mathit{wg}.F.X.\overline{\mathcal{F}} \equiv X.\overline{\mathcal{F}})) \\
\Rightarrow & \quad \{ \ \| \text{ commutative, thus } F \in \mathcal{F} \Rightarrow \mathit{component}.F.\overline{\mathcal{F}} \ \} \\
 & (\forall F : F \in \mathcal{F} : (\mathit{wg}.F.X.\overline{\mathcal{F}} \equiv X.\overline{\mathcal{F}})) \\
\Rightarrow & \quad \{ \text{ predicate calculus } \} \\
 & (\forall F : F \in \mathcal{F} : \mathit{wg}.F.X.\overline{\mathcal{F}}) = X.\overline{\mathcal{F}} \\
 & \wedge \\
 & (\exists F : F \in \mathcal{F} : \mathit{wg}.F.X.\overline{\mathcal{F}}) = X.\overline{\mathcal{F}}
\end{array}
$$

Corollary: Properties in isolation The next result shows that wg is sufficient to specify a program in isolation. From the component theorem (8), and using $\mathit{component}.F.F$:

$$\mathit{wg}.F.Y.F \quad \equiv \quad Y.F$$

5 Towards compositional specifications

We seek compositional proof techniques that allow us to prove properties of composed programs from properties of their components. Next we identify classes of properties that are useful for constructing compositional proofs.

5.1 Exists-Component Properties

A program property X is an exists-component property if and only if,

$$X.G \equiv (\forall H :: X.G \| H) \tag{9}$$

Therefore, for any set $\mathcal{G}$ of programs, any exists-component property X, and commutative $\|$:

$$(\exists G : G \in \mathcal{G} : X.G) \ \Rightarrow \ X.(\|G : G \in \mathcal{G} : G) \tag{10}$$

If there exists a component of a program that satisfies an exists-component property then the program itself also satisfies that property.

Theorem For any exists-component property X and any program F:

$$X.F \ \equiv \ [\mathit{wg}.F.X]$$

Proof:

$$
\begin{array}{ll}
 & X.F \\
\equiv & \quad \{ \text{ definition of exists-component } \}
\end{array}
$$

$$
\begin{array}{ll}
 & (\forall H :: X.F \| H) \\
\equiv & \qquad \{ \text{ definition of } \textit{component} \ \} \\
 & (\forall G :: \textit{component}.F.G \Rightarrow X.G) \\
\equiv & \qquad \{ \text{ definitions of } [\,] \text{ and } \textit{wg} \ \} \\
 & [\textit{wg}.F.X]
\end{array}
$$

As noted in section 3.3, guarantees properties are exists-component properties.

For some properties and program models, the implication in (10) can be strengthened to equivalence; we call such properties *strong exists-component* properties. Thus, a property X is a strong exists-component property if and only if:

$$(\exists G : G \in \mathcal{G} : X.G) \ \equiv \ X.(\| G : G \in \mathcal{G} : G) \tag{11}$$

Setting $\mathcal{G}$ to the empty set in this equation, we observe (false $\equiv$ $X.\textit{SKIP}$) that the program *SKIP* does not have any strong exists-component properties.

We use strong exists-component properties in discussing refinement.

5.2 All-component properties

For commutative $\|$, a program property X is an all-component property if and only if, for any set $\mathcal{G}$ of programs:

$$(\forall G : G \in \mathcal{G} : X.G) \ \equiv \ X.(\| G : G \in \mathcal{G} : G) \tag{12}$$

A program has an all-component property if and only if all components of the program have that property [CS95].

From the definition, setting $\mathcal{G}$ to the empty set, it follows that all-component properties are properties of the program *SKIP*.

Theorem For any all-component property X and any program F:

$$(\exists H : \textit{component}.F.H : X.H) \equiv X.F$$

Proof is by implication in both directions:

$$
\begin{array}{ll}
 & X.F \\
\Rightarrow & \qquad \{ \ \textit{component}.F.F \ \} \\
 & (\exists H : \textit{component}.F.H : X.H) \\
\Rightarrow & \qquad \{ \text{ definition of } \textit{component} \ \} \\
 & (\exists G : F \| G = H : X.H) \\
\Rightarrow & \qquad \{ \text{ definition of all-component property } \} \\
 & (\exists G : F \| G = H : X.F \wedge X.G) \\
\Rightarrow & \qquad \{ \text{ predicate calculus } \} \\
 & X.F
\end{array}
$$

Theorem For any all-component property X and any programs F and H:

$$wg.F.X.H \equiv (\forall G : F||G \equiv H : X.F \wedge X.G)$$

Proof:

$$\begin{array}{ll} & (\forall G : F||G = H : X.F \wedge X.G) \\ \equiv & \quad \{ \text{ definition of all-component } \} \\ & (\forall G : F||G = H : X.H) \\ \equiv & \quad \{ \text{ predicate calculus } \} \\ & (\exists G :: F||G = H) \Rightarrow X.H \\ \equiv & \quad \{ \text{ definition of component } \} \\ & component.F.H \Rightarrow X.H \\ \equiv & \quad \{ \text{ definition of } wg \} \\ & wg.F.X.H \end{array}$$

5.3 Properties of the environment

5.3.1 The property transformer *env*

Often, for a program F, $conforms.F||G$ implies that program G has a property that can be exploited in proofs. A common example is a program F that encapsulates a locally-modifiable variable u, defined to be a variable that can only be modified by F. In this case $conforms.F||G$ implies that G has the property that it leaves u unchanged. To capture this, we introduce a new property transformer *environment*, denoted by *env*, where $env.X$ is a property of a program F means that X is a property of all programs that can be composed with F in a conformant way.

$$(\forall F, X : conforms.F : env.X.F \quad \equiv \quad (\forall H : conforms.F||H : X.H)). \quad (13)$$

5.3.2 Theorems about *env*

***env* is universally conjunctive with respect to properties**

$$[env.(\forall X : X \in \mathcal{X} : X) \quad \equiv \quad (\forall X : X \in \mathcal{X} : env.X)]$$

As a consequence, *env* is monotonic with respect to properties and strict with respect to property *true*.

env* is strict with respect to property *false

$$[env.false \equiv false]$$

$env.X$ **is an exists-component property**

For commutative $\|$: $env.X.F = (\forall G : conforms.F\|G : env.X.F\|G)$

Proof:
The proof uses a "ping-pong" style, proving implications in both directions.

$$
\begin{array}{ll}
 & (\forall G : conforms.F\|G : env.X.F\|G \\
= & \quad \{ \text{ definition of env (13) } \} \\
 & (\forall G : conforms.F\|G : (\forall H : conforms.F\|G\|H : X.H)) \\
= & \quad \{ \text{ predicate calculus } \} \\
 & (\forall G, H : conforms.F\|G \wedge conforms.F\|G\|H : X.H) \\
= & \quad \{\text{From (1),) } conforms.F\|G\|H \Rightarrow conforms.F\|G \ \} \\
 & (\forall G, H : conforms.F\|G\|H : X.H) \\
\Rightarrow & \quad \{ \text{ let } G := SKIP \ \} \\
 & (\forall H : conforms.F\|H : X.H) \\
= & \quad \{ \text{ definition of } env \ \} \\
 & env.X.F \\
\Rightarrow & \quad \{ \ conforms.F\|G\|H \Rightarrow conforms.F\|G, \ \| \text{ commutative } \} \\
 & (\forall G, H : conforms.F\|G\|H : X.H) \\
= & \quad \{ \text{ definition of } env \ \} \\
 & (\forall G : conforms.F\|G : env.X.F\|G
\end{array}
$$

Environment factorization For all-component properties Y and Z,

$$[env.X \wedge [X \wedge Y \Rightarrow Z] \wedge Z \quad \Rightarrow \quad Y \, guarantees \ Z] \tag{14}$$

Left-side weakening For any all-component property X,

$$[env.X \wedge X \wedge (X \, guarantees \ Y) \quad \Rightarrow (true \ guarantees \ \ Y)] \tag{15}$$

6 Refinement

We can replace one component of a program by another component if the replacement does not violate program correctness. If the specification of a program $F\|H$ is that the program must satisfy property X, then we can replace component F by G if $G\|H$ also has property X. We refer to H as the environment of F in the composed program $F\|H$.

We define refinement of a program F by a program G with respect to a property X over an environment H as follows: F is refined by G with respect to X and H if and only if:

$$X.F\|H \ \Rightarrow \ X.G\|H$$

F is refined by G with respect to X and *all* environments if the above formula holds for all programs H. F is refined by G with respect to X *in isolation* if the

above formula holds with $H = SKIP$; therefore, F is refined by G with respect to X in isolation if and only if:

$$X.F \Rightarrow X.G$$

The refinement theorem relates refinement in isolation and refinement over all environments.

The Refinement Theorem Let X be any property that is a conjunction of all-component and strong exists-component properties. G refines F with respect to X and all environments if and only if G refines F with respect to X in isolation.

$$(\forall H :: X.F \| H \Rightarrow X.G \| H) \;\equiv\; (X.F \Rightarrow X.G) \tag{16}$$

Proof: We prove the theorem by proving it for all-component properties; the proof for strong exists-component properties is similar and is omitted.

Lemma: Strong exists-component property refinement Let X be a strong exists-component property. Then

$$(\forall H :: X.F \| H \Rightarrow X.G \| H) \;\equiv\; X.F \Rightarrow X.G$$

Proof:

$$\begin{array}{ll}
 & (\forall H :: X.F \| H \;\Rightarrow\; X.G \| H) \\
\equiv & \qquad \{\ X \text{ strong exist component (11) }\} \\
 & (\forall H :: X.F \vee X.H \;\Rightarrow\; X.G \vee X.H) \\
\equiv & \qquad \{\ \text{predicate calculus }\} \\
 & (\forall H :: X.F \;\Rightarrow\; X.G \vee X.H) \\
\equiv & \qquad \{\ H := SKIP\ \} \\
 & (X.F \Rightarrow X.G \vee X.SKIP) \wedge (\forall H :: X.F \;\Rightarrow\; X.G \vee X.H) \\
\equiv & \qquad \{\ X \text{ strong exist component (11) }\} \\
 & (X.F \Rightarrow X.G \| SKIP) \wedge (\forall H :: X.F \;\Rightarrow\; X.G \vee X.H) \\
\equiv & \qquad \{\ SKIP \text{ identity of } \|\ \} \\
 & X.F \Rightarrow X.G \wedge (\forall H :: X.F \;\Rightarrow\; X.G \vee X.H) \\
\equiv & \qquad \{\ \text{predicate calculus }\} \\
 & X.F \Rightarrow X.G
\end{array}$$

Lemma: All-component refinement For any all-component property X:

$$(\forall H :: X.F \| H \Rightarrow X.G \| H) \;\equiv\; X.F \Rightarrow X.G$$

Proof:

$$\begin{array}{ll}
 & (\forall H :: X.F \| H \Rightarrow X.G \| H) \\
\equiv & \qquad \{\ X \text{ is all-component }\} \\
 & (\forall H :: X.F \wedge X.H \;\Rightarrow\; X.G \wedge X.H)
\end{array}$$

$\equiv$ { predicate calculus }

$(\forall H :: X.H \Rightarrow (X.F \ \Rightarrow \ X.G))$

$\equiv$ { predicate calculus }

$(\exists H :: X.H) \Rightarrow (X.F \ \Rightarrow \ X.G)$

$\equiv$ { X is all-component, thus $X.SKIP$, and setting $H := SKIP$ }

$true \Rightarrow (X.F \ \Rightarrow \ X.G)$

$\equiv$ { predicate calculus }

$X.F \ \Rightarrow \ X.G$

7 Model-specific theory: an example

We present a model which is a small generalization of the operational model in UNITY. This operational model helps to motivate our use of predicate transformers. The predicate transformers can also be used for models in addition to the one presented here.

7.1 Operational model

A program is a 4-tuple (V, L, C, D) where

1. V is a set of typed variables. This set of variables defines a state space in a state-transition system. Each state in the system is given by the values of the variables in V.

2. L is a subset of V. L is the set of locally modifiable variables. Local variables play a role in the definition of conformant composition.

3. C is a set of commands where each command terminates when initiated in any state, and each command has bounded nondeterminism. C includes the *skip* command which leaves the state unchanged. The computation proceeds by nondeterministically selecting any command in C, and executing it.

4. D is a subset of C. The fairness requirement is that commands in D are executed infinitely often. (There is no fairness requirement for commands in C that are not in D.)

Equality of programs Letting $F = (V_F, L_F, C_F, D_F)$ and $G = (V_G, L_G, C_G, D_G)$,

$$(F = G) \quad = \quad (V_F = V_G \ \wedge \ L_F = L_G \ \wedge \ C_F = C_G \ \wedge \ D_F = D_G)$$

Computations A program describes a state-transition system. There exists a transition from state s to state t in the state transition system if and only if there exists a command c in C where the execution of c can take the system from s to t.

A computation is an infinite sequence of states s_j, $j \geq 0$, where:

1. There exists a transition in the state transition system from each state in the computation to the next.

2. For every command d in D, there exists an infinite number of pairs of successive states s_i, s_{i+1} in the computation such that execution of d can take the system from s_i to s_{i+1}.

Parallel composition Letting $F = (V_F, L_F, C_F, D_F)$ and $G = (V_G, L_G, C_G, D_G)$, we define the parallel composition operator as follows:

$$F \| G \quad = \quad (V_F \cup V_G, L_F \cup L_G, C_F \cup C_G, D_F \cup D_G).$$

where $F \| G$ is conformant if and only if variables with the same name in $V_F \cup V_G$ have the same types, and variables in L_F are modified only by commands in C_F (which includes commands common to C_F and C_G), and likewise variables in L_G are modified only by commands in C_G.

Some consequences of the definition are that $\|$ is commutative, associative and idempotent. The unit element for parallel composition is the program *SKIP* where

$$SKIP = (\phi, \phi, skip, \phi)$$

In this model, the definition of component is

$$component.F.H \quad \equiv$$
$$(V_F \subseteq V_H) \wedge (L_F \subseteq L_H) \wedge (C_F \subseteq C_H) \wedge (D_F \subseteq D_H)$$

7.2 Program properties of interest: Safety

The State Predicate Transformer *awp*.F For a program $F = (V, L, C, D)$ define a state-predicate transformer $awp.F$ as follows:

$$awp.F \ \equiv \ (\forall c : c \in C : wp.c)$$

(The letters in *awp* stand for *all wp* because $awp.F$ is the conjunction of all the transformers $wp.c$.) Since all commands c in C terminate when initiated in any state, and since c has bounded nondeterminism, $wp.c$ is a universally-conjunctive, or-continuous, state-predicate transformer. Therefore, $awp.F$ is likewise a universally-conjunctive or-continuous, state-predicate transformer.

Since, $[wp.skip.q \equiv q]$, for all q:

$$[awp.F.q \Rightarrow q]$$

From the definition of parallel composition,

$$[awp.F \| H.q \;\equiv\; awp.F.q \wedge awp.H.q] \tag{17}$$

Next properties For state predicates p and q we define a property $(p\ next\ q)$ as follows:
For a program F,

$$(p\ next\ q).F \quad \equiv \quad [p \Rightarrow awp.F.q]$$

Since $skip \in C$, we have

$$(p\ next\ q).F \Rightarrow [p \Rightarrow q]$$

The next property describes the next state relation of a program. (This property is based on Misra's *co* property [Mis95b], but there are some slight differences due to differences in the programming model.) The property $p\ next\ q$ holds for a program exactly when for all points in all computations of the program at which (state predicate) p holds, q holds at the next point in the computation. Therefore, $p\ next\ q$ holds for a program exactly when all transitions of the program from states in which p holds are to states in which q holds.

Lemma: p next q is an all-component property.
Proof: Follows from the definition of *next*.

$$\begin{array}{ll}
 & (p\ next\ q).F \| H \\
\equiv & \qquad \{ \text{ definition of } next \ \} \\
 & [p \Rightarrow awp.F \| H.q] \\
\equiv & \qquad \{ \ 17 \ \} \\
 & [p \Rightarrow awp.F.q \wedge awp.H.q] \\
\equiv & \qquad \{ \text{ predicate calculus } \} \\
 & [p \Rightarrow awp.F.q] \wedge [p \Rightarrow awp.H.q] \\
\equiv & \qquad \{ \text{ definition of } next \ \} \\
 & (p\ next\ q).F \wedge (p\ next\ q).H
\end{array}$$

Stable We define a useful function *stable* from state predicates to program properties as follows:

$$[stable.q \quad \equiv \quad (q\ next\ q)]$$

If $stable.q$ holds for a program F then all transitions from states in which q holds are to states in which q holds. Since $q\ next\ q$ is an all-component property it follows that $stable.q$ is also an all-component property.

Initial States and Always In our theory, the predicate that holds on initial states is a property that can be asserted about a program. Thus *initially.q* is the property that asserts that all computations begin in a state satisfying q. We typically have properties of the form (*initially.q guarantees* X).

The property *always.q* is defined as follows:

$$[always.q \quad = \quad stable.q \wedge initially.q]$$

This property indicates that q holds at every point in all computations. As a result, we obtain the so-called substitution axiom [CM88, San91, SC96] that allows *true* and q to be interchanged in program properties[1]

$$[always.q \quad \Rightarrow \quad (X = X|^{q}_{true}) \wedge (X = X|^{true}_{q})]$$

A more complete discussion is given in [SC96].

7.3 Program properties of interest: Progress

Transient Transience [Mis95a] is a basic progress property of a system that follows from fairness assumptions. Let $F = (V, L, C, D)$; we define a function *transient* from state predicates to program properties, as follows:

$$transient.p.F \quad = \quad (\exists d : d \in D : [p \Rightarrow wp.d.\neg p])$$

Operationally, p is false infinitely often in every computation of every program that has F as a component because each command d in D is executed infinitely often, and there exists at least one command that establishes $\neg p$ if p holds.

From the definition of *transient*: *transient.p* is a strong exists-component property. Therefore,

$$[transient.p \Rightarrow (true \quad guarantees \quad transient.p)]$$

Leads-to ($\leadsto$) properties Leads-to properties are useful for describing progress. Operationally, $(p \leadsto q).F$ holds if in every computation of F, if p holds at some point in the computation, then q holds then or at a later point. Leads-to properties can be derived from transient and next properties. In [CM88], leads-to is defined as the strongest relation satisfying 3 axioms based on a simple progress property ensures. This axiomatization has been shown to be sound and relatively complete [San91, Rao91, Pac92]. A problem is that 'ensures", the key element in UNITY theory about progress is neither an all-component nor an exist component property. Below, we list an alternative axiomatization of leads-to that allows leads-to properties to be derived only from all-component and exist-component properties.

[1] We use the notation $X|^{q}_{r}$ to indicate textual substitution of q with r.

1. **Transient rule**

$$[\mathit{transient}\ q \quad \Rightarrow \quad q \leadsto \neg q]$$

2. **Implication rule**

$$[q \Rightarrow r] \quad \Rightarrow \quad [q \leadsto r]$$

3. **Disjunction rule** For arbitrary set of predicates $\mathcal{Q}$:

$$[(\forall q : q \in \mathcal{Q} : q \leadsto r) \quad \Rightarrow \quad (\exists q : q \in \mathcal{Q} : q) \ \leadsto \ r]$$

4. **Transitivity rule**

$$[q \leadsto r \ \ \wedge \ \ r \leadsto s \quad \Rightarrow \quad q \leadsto s]$$

5. **PSP rule**

$$[q \leadsto r \ \ \wedge \ \ s \ \mathit{next}\ t \quad \Rightarrow \quad (q \wedge s) \ \leadsto \ (r \wedge s) \vee (\neg s \wedge t)]$$

The straightforward proof that these axioms are equivalent to those of [CM88], and thus sound and relatively complete, is omitted.

From the axioms, we obtain

$$[\mathit{transient}.p \quad \Rightarrow \quad ((p \ \mathit{next}\ p \vee q) \ \ \mathit{guarantees} \ \ (p \leadsto q))] \tag{18}$$

7.4 Properties of environment

Constant expressions For any expression e, we define the property $\mathit{constant}.e$ as follows: $\mathit{constant}.e.F$ holds if and only if all computations of F leave the value of e unchanged. Therefore

$$[\mathit{constant}.e \quad = \quad (\forall k : \mathit{stable}.(e = k))]$$

Since locally modifiable variables are unchanged by other components:

$$(v \in L) \Rightarrow \mathit{env}.(\mathit{constant}.v).F$$

Local predicates For a predicate p, $\mathit{local}.p$ is a property that holds if p mentions only variables in L. We have

$$[\mathit{local}.p \ \ \Rightarrow \ \ \mathit{env}.(\mathit{constant}.p)]$$

and from (15) and $[\mathit{constant}.p \ \ \Rightarrow \ \ p \ \mathit{next}\ q]$,

$$[p \ \mathit{next}\ q \wedge \mathit{local}.p \quad \Rightarrow \quad (\mathit{true}\ \mathit{guarantees} \ \ (p \ \mathit{next}\ q))] \tag{19}$$

Program properties Let v be an integer variable. We define a property *nondecreasing*.v for a program to mean that the value of v does not decrease during computations of the program. Formally, this is expressed as

$$nondecreasing.v \equiv (\forall k :: stable.(v \geq k))$$

For two integer variables v and w, we define a property w *follows* v, as:

$$w \; follows \; v \equiv (\forall k :: v \geq k \;\rightsquigarrow\; w \geq k)$$

From the proof rules for $\rightsquigarrow$, *follows* is transitive.

These properties are useful in developing compositional proofs of properties of composed programs.

8 Example

8.1 Overview

We consider the following problem: The members of a committee, who are located at two or more sites, need to determine the earliest time, e at or after some given time t, at which they can all meet.[2]

We structure the solution by dividing the members into geographic areas and letting each area determine the earliest meeting time that all members associated with that area can meet after some proposed time. The earliest times are combined to generate a new proposed meeting time until a time is found when all members can meet.

This problem is simple enough to be handled easily without modularity. However, since different areas may use different techniques to determine the earliest meeting time of their subset of committee members, modular reasoning based on specifications of components is helpful, and also serves to illustrates our approach.

8.2 Specification

Definitions For simplicity, we represent time as a non-negative integer. For each member m there is a function $m.earliest$ mapping time to time such that

$$m.earliest.t \;=\; \text{earliest time at least } t \text{ when } m \text{ can meet.}$$

$m.earliest$ is monotonic and idempotent, and for technical reasons, we assume $m.earliest.0 = 0$. We define a boolean function

$$m.free.t \quad \equiv \quad (m.earliest.t = t)$$

thus $m.free.t$ holds if and only if m can meet at time t.

[2] This is a special case of the problem of asynchronous computation of fixed points.

Let Com be the set of members of the committee and $S \subseteq Com$. Then $S.earliest.t$ is the earliest time at least t that all members in S can meet.

$$S.earliest.t \quad = (min\ u : u \geq t : S.free.u)$$

where

$$S.free.t \quad = \quad (\forall m : m \in S : m.free.t)$$

We assume that $Com.earliest.t$ exists for all t.

Components $G(t, e, S)$ is a program that finds the earliest time e at or after t at which all members in set S can meet.

Let S_i be subsets of members so that $S = (\cup i :: S_i)$, and the cardinality of each subset is strictly smaller than that of S. (Subsets can have common members.) We propose to implement $G(t, e, S)$ as a parallel composition of components $G(u, a_i, S_i)$ for each subset S_i, and a coordinator program that computes the meeting time e for all sites from the meeting times a_i of subset S_i, all i. The coordinator is $C(t, e, u, A)$, where $A = (\cup i :: \{a_i\})$.

$$G(t, e, S) = C(t, e, u, A) \| (\| i :: G(u, a_i, S_i))$$

First we define the locally-modifiable variables and their initial values. S is a fixed set, e is a locally-modifiable variable with initial value 0, and t is unmodified by $G(t, e, S)$. The same restrictions (with appropriate actual variables) applies to $G(u, a_i, S_i)$.

Variables u and e are locally-modifiable in $C(t, e, u, A)$, and their initial values are set by C to be 0. C leaves t and A unchanged. All shared variables are passed explicitly as parameters; so, for instance, $G(u, a_i, S_i)$ does not access t.

These restrictions allow $C(t, e, u, A)$ and $G(u, a_i, S_i)$ all i to be composed in parallel.

Now, we give the guarantees properties in a table where the left column is the antecedent and the right column is the consequent. All entries in a box are to be conjoined and all items mentioning subscript i are implicitly quantified over all i.

Guarantees Properties for $G(t, e, S)$

$nondecreasing.t$	$always.(e \leq S.earliest.t)$ e *follows* $S.earliest.t$	(20)
$true$	$always.(S.free.e)$ $nondecreasing.e$	

From the above properties, the value of $S.earliest.T$ for any time T can be determined in the following way: the environment of G sets t to T, and leaves t unchanged until $e \geq t$, at which point, $e = S.earliest.T$.

Guarantees Properties for $C(t, e, u, A)$

$true$	$nondecreasing.e$ $nondecreasing.u$
$nondecreasing.t$ $nondecreasing.a_i$ $always.((a_i \leq S_i.earliest.u)$ $always.(S_i.free.a_i)$ a_i $follows$ $S_i.earliest.u$	$always.(e \leq S.earliest.t)$ $always.(S.free.e)$ e $follows$ $S.earliest.t$ (21)

8.3 Proof of the composed program

The properties of the program $G(t, e, S)$ are derived from properties of its components. The proofs are straightforward, for the purposes of illustration, we give a detailed proof of (20)

Proof of (20) : Since guarantees is an exist component property, the following component properties hold in G:

$$(\forall i :: nondecreasing.u \ guarantees \ \ (a_i \ follows \ S_i.earliest.u)) \tag{22}$$

$$\begin{aligned} & nondecreasing.t \wedge (\forall i :: (e_i \ follows \ S_i.earliest.u)) \\ & \quad guarantees \ \ (e \ follows \ S.earliest.t) \end{aligned} \tag{23}$$

$$true \ guarantees \ \ nondecreasing.u \tag{24}$$

Now, we have

$true$

$\Rightarrow$ { From (22), (24), and transitivity of $guarantees$ }

$(\forall i :: true \ guarantees \ \ (a_i \ follows \ S_i.earliest.u))$

$\Rightarrow$ {conjunctivity of $guarantees$ }

$true \ guarantees \ \ (\forall i :: (a_i \ follows \ S_i.earliest.u))$

$\Rightarrow$ {conjunction with $nondecreasing.t \ guarantees \ \ nondecreasing.t$ }

$nondecreasing.t \ guarantees \ \ nondecreasing.t \wedge (\forall i :: (a_i \ follows \ S_i.earliest.u))$

$\Rightarrow$ {transitivity with (23) }

$nondecreasing.t \ guarantees \ \ (e \ follows \ S.earliest.t)$

The previous proof illustrated how the proof rules for guarantees, which can be used to prove the properties of a program from guarantees properties of its components. Note that once we used the exist component rule to lift the properties of the components to G, the other proof rules needed were the same as rules for implication.

8.4 Implementation of the Program Components

Implementation of $C(t, e, u, A)$ Now, we give possible implementation of $C(t, e, u, A)$.

Initially $e = 0$ and $u = 0$. The program has two commands both of which are executed infinitely often:

1. if $(\forall i :: u = a_i)$ then $\{e := \max(u, e); u := \max(u, t); \}$
2. if $u < \max_i\{a_i\}$ then $u := \max_i\{a_i\}$

The proofs are straightforward, except for the progress property (21). This proof required proof of two transient properties and induction and is sketched in the appendix.

This property illustrates one of the advantages of our approach. Many rely/guarantee approaches require the left side to be a safety property, and the right side to be a property that holds for the component in isolation. For this example, that would have forced us to do the induction *after* each program composition, rather than doing it once and capturing the results of the complicated proof in the specification of C.

Implementations of G

Singleton Sets If S has only a single member, then a design for program $G(t, e, S)$ is as follows. Initially $e = 0$. The program has the following single command which is executed infinitely often:

$e < S.F.t \;\rightarrow\; e := S.F.t$

Sets with more than one member If S has more than one member, then partition S into k nonempty subsets, S_i where the cardinality of each subset S_i is strictly less than the cardinality of S. Implement $G(t, e, S)$ as a parallel composition of a program $C(t, e, u, A)$ where $A = (\cup : 0 \leq i < k : \{a_i\})$, and program $G(u, a_i, S_i)$, for $0 \leq i < k$. Process $G(u, a_i, S_i)$ computes the earliest time at or after u at which all members of set S_i can meet. Process C computes the meeting time for the entire set S from the meeting times of each of the subsets S_i.

9 Related work

Predicate transformers The relationship between Hoare logic [Hoa69] and wp was a motivation for our definition of wg. Just as the triple (or property) $\{p\}F\{q\}$ can be defined as $[p \Rightarrow wp.F.q]$ for a sequential program F, we have a defined the property X *guarantees* Y as $[X \Rightarrow wg.F.Y]$.

Since Dijkstra's initial formulation of predicate transformers for the language of guarded commands, weakest precondition predicate transformers have been developed for other sequential constructs such as recursion and various notions of fairness [Nel89, Hes92, BN94]. For concurrency, additional predicate transformers such as *sin*, *win* [Lam90] and several temporal properties [JKR89, CS95, DS96] have been proposed. Predicate transformers have also been used as a basis for defining program refinement [BvW89, Bac89, GM91, San96] for both sequential and concurrent programs. The advantage of this work is that once appropriate predicate transformers have been defined, reasoning can be carried out in the familiar context of predicate calculus.

The treatment of properties as predicates on programs, and the use of both predicates on programs and predicates on states in formulae, is found in [CS95]. This approach coupled with the results on predicate transformers led to our idea of a predicate transformer for dealing with parallel program composition.

Variants of rely/guarantee specifications have been proposed in numerous papers including [Jon83, AL93, MS96], for layered systems in [LS94, LS92], in temporal logic frameworks in [Pnu84, Sta85], and for TLA in particular in [AL95].

The temporal logic approaches are more expressive than ours in the sense that the specifications are statements about individual computations. For example, in [AL95], $E \overset{+}{\rightarrow} M$ means that for each computation, the component will satisfy M for at least one step longer than the environment satisfies E, where E is a safety property. In contrast, a similar specification in our approach would be essentially: If all computations satisfy E, than all computations satisfy M. A difference is that both the property relied upon and the property guaranteed, in our theory, are properties of the composed system; we do not use one property for the environment and the other for the component or system, and our model does not have restrictions on sharing of state between components.

An axiomatic semantics based on predicate transformers for rely/guarantee properties for UNITY with local variables has been given in [Col94, CK95]. They define properties of the form F *sat* P *w.r.t.* R where F is a program, P is a UNITY property such as leads-to, and R is an *interference predicate* constraining the next state relation of the environment. Two components *cooperate* with respect to their interference predicates if neither violates the interference predicate of the other. Proof rules allow rely/guarantee properties of compositions that cooperate with respect to the interference predicates to be derived from the rely/guarantee properties of the components. Our ideas, particularly with regard to the all-component and exists-component properties are directly influenced by UNITY. The point of departure is that our guarantees properties do not deal with properties of environments and components.

10 Summary

Predicate-transformers form the basis for axiomatic semantics for sequential programming. Many researchers have contributed to a deep and elegant theory of transformers. This paper applied the theory to an associative program composition operator $\|$ which has an identity element.

Our theory uses predicates on states of programs, and predicates on programs. We defined a program property as a predicate on programs. We introduced a dyadic operator *guarantees* on program properties where X *guarantees* Y has a different meaning than the traditional meaning of guarantee Y relying on X. The *guarantees* operator has many of the properties of implication. We identified three useful kinds of properties, *exists-component* properties, *all-component* properties, and encapsulation properties. We defined a property transformer *weakest guarantee*, and presented theorems about the property, and showed how it can help in modular reasoning about programs composed with $\|$.

Experience has shown that the theory of predicate transformers is very valuable in sequential programming. Using predicate transformers for $\|$ has the advantage that the powerful theory, developed initially for sequential programming, can be applied to a different program composition operator. A unified theory for a collection of program composition operators has benefits.

References

[AL93] Martin Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, Jan 1993.

[AL95] Martin Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[Bac89] R.J.R. Back. Refinement calculus, Part II: Parallel and reactive programs. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Lecture Notes in Computer Science Volume 430*, pages 67–93. REX Workshop on Stepwise Refinement of Distributed Systems, 1989.

[BN94] Manfred Broy and Greg Nelson. Adding fair choice to Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 16(3):924–938, 1994.

[BvW89] R.J.R. Back and J. von Wright. Refinement calculus, Part I: Sequential nondeterministic programs. In J.W. de Bakker, W.-P. de Roever,

and G. Rozenberg, editors, *Lecture Notes in Computer Science Volume 430*, pages 42–66. REX Workshop on Stepwise Refinement of Distributed Systems, 1989.

[CK95] Pierre Collette and Edgar Knapp. Logical foundations for compositional verification and development of concurrent programs in UNITY. In *Fourth International Conference on Algebraic Methodology and Software Technology*, 95.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[Col94] Pierre Collette. Composition of assumption-commitment specifications in a UNITY style. *Science of Computer Programming*, 23:107–125, 1994.

[CS95] K. Mani Chandy and Beverly A. Sanders. Predicate transformers for reasoning about concurrent computation. *Science of Computer Programming*, 24:129–148, 1995.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[DS96] Rutger Dijkstra and Beverly Sanders. A predicate transformer for the progress property 'to-always'. Technical Report 96-033, Department of Computer and Information Science and Engineering, 1996. ftp://ftp.cise.ufl.edu/cis/tech-reports/tr96/tr96-035.ps.

[GHW85] J.V. Guttag, J.J. Horning, and J.M. Wing. The larch family of specification languages. *IEEE Software*, 2(5):24–36, 1985.

[GM91] P.H.B. Gardiner and Carroll Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991.

[Hes92] Wim H. Hesselink. *Programs, Recursion, and Unbounded Choice: Predicate Transformer Semantics and Transformation Rules*. Cambridge tracts in theoretical computer science 27. Cambridge University Press, 1992.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10), October 1969.

[JKR89] C.S. Jutla, E. Knapp, and J.R. Rao. A predicate transformer approach to semantics of parallel programs. In *Proceeding of the 8th ACM Symposium on Principles of Distributed Computing*, 1989.

[Jon83] C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[Lam90] Leslie Lamport. *win* and *sin*: Predicate transformers for concurrency. *ACM Transactions on Programming Languages and Systems*, 12(3), 1990.

[Lam94] Leslie Lamport. A temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[LS92] S. S. Lam and A. U Shankar. Specifying modules to satisfy interfaces–a state transition approach. *Distributed Computing*, 6(1):39–63, July 1992.

[LS94] S. S. Lam and A. U Shankar. A theory of interfaces and modules 1: Composition theorem. *IEEE Transactions on Software Engineering*, 20(1):55–71, January 1994.

[Mis95a] J. Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.

[Mis95b] J. Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995.

[MP91] Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):97–130, 1991.

[MS96] R. Manohar and P. Sivilotti. Composing processes using modified rely-guarantee specifications. Caltech technical report CS-TR-96-22, 1996.

[Nel89] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.

[Pac92] Jan Pachl. A simple proof of a completeness result for *leads-to* in the UNITY logic. *Information Processing Letters*, 41, 1992.

[Pnu81] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[Pnu84] Amir Pnueli. In transition from global to modular temporal reasoning about programs. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 123–144. Springer-Verlag, 1984.

[Rao91] Josyula R. Rao. On a notion of completeness for the leads-to. Notes on UNITY: 24-90, July 1991.

[San91] Beverly Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2), 1991.

[San96] Beverly A. Sanders. Data refinement of mixed specification: A generalization of UNITY. *Acta Informatica*, 1996. to appear. Preliminary version available as UF CISE Technical Report 96-010 ftp://ftp.cise.ufl.edu/cis/tech-reports/tr96/tr96-010.ps.

[SC96] Beverly A. Sanders and K. Mani Chandy. Exploiting historical information about computations. Technical Report 96-034, Department of Computer and Information Science and Engineering, University of Florida. 1996. ftp://ftp.cise.ufl.edu/cis/tech-reports/tr96/tr96-034.ps.

[Sta85] Eugene W. Stark. A proof technique for rely/guarantees properties. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science*, pages 369–391, 1985. volume 206 of *Lecture Notes in Computer Science*.

11 Appendix

We sketch the proof of (21), omitting steps that are just applications of UNITY-style proof rules in order to focus on compositional reasoning as presented in this paper.

The desired property is $(e \textit{ follows } S.earliest.t).C$. In the interest of brevity, we will omit the argument C in the following.

Lemma 1

$$X \wedge Y \wedge nondecreasing.t \quad guarantees \quad e \textit{ follows } S.earliest.t$$

where

$$X \quad = \quad (u = k \quad \rightsquigarrow \quad (\forall i : a_i = u) \wedge u = S.earliest.k)$$

and

$$Y \quad = \quad ((\forall i : a_i = u) \wedge u = S.earliest.k \quad \rightsquigarrow e \geq S.earliest.k \wedge u \geq t).$$

Lemma 2 From the text of C,

$$transient.((\forall i : u = a_i) \wedge u = k \wedge \neg(e \geq k \wedge u \geq t))$$

Lemma 3 From lemma 2 and (18),

$$Z \quad guarantees \quad (\forall i : a_i = u) \wedge u = k \quad \rightsquigarrow \quad e \geq k \wedge u \geq t$$

where

$$\begin{aligned} Z \quad = \quad & (\forall i : u = a_i) \wedge u = k \wedge \neg(e \geq k \wedge u \geq t) \\ & next \\ & ((\forall i : u = a_i) \wedge u = k) \vee (e \geq k \wedge u \geq t) \end{aligned}$$

Lemma 5 From lemmas 1 and 3,

$$X \wedge Z \wedge \ \mathit{nondecreasing}.t \ \mathit{guarantees}\ e\ \mathit{follows}\ S.\mathit{earliest}.t$$

Lemma 6 From the text of C,

$$Z.C$$

Lemma 7

$$[\mathit{always}.(a_i \leq S_i.\mathit{earliest}.u) \wedge \mathit{nondecreasing}.a_i \wedge \mathit{always}.(S_i.\mathit{free}.a_i) \wedge \mathit{constant}.u \quad \Rightarrow \quad Z]$$

Lemma 8 From lemmas 6 and 7 with 14 and $\mathit{env}.\mathit{constant}.u$,

$$(\forall i :: \mathit{always}.a_i \leq S_i.\mathit{earliest}.u) \wedge (\forall i :: \mathit{nondecreasing}.a_i) \wedge (\forall i :: \mathit{always}.(S_i.\mathit{free}.a_i))$$
$$\mathit{guarantees}$$
$$Z$$

Now, it remains to find a convenient guarantees properties for X.

Lemma 9 From the text of C,

$$\mathit{transient}.(\mathit{max}_i.a_i \geq k) \wedge u < l$$

Lemma 10 From lemma 9 and (18),

$$(\forall i : \mathit{nondecreasing}.a_i) \quad \mathit{guarantees} \quad \mathit{max}_i.\{a_i\} \geq k \ \rightsquigarrow u \geq k$$

Lemma 11

$$(\forall i :: a_i\ \mathit{follows}\ S_i.\mathit{earliest}.u) \wedge (\forall i :: \mathit{nondecreasing}.a_i)$$
$$\mathit{guarantees}$$
$$u \geq k \ \rightsquigarrow \ \mathit{max}_i.\{a_i\} \geq \mathit{max}_i.\{(S_i.\mathit{earliest}.k)\}$$

Lemma 12 From lemmas 10 and 11, and induction,

$$(\forall i : a_i\ \mathit{follows}\ S_i.\mathit{earliest}.u) \wedge (\forall i : \mathit{nondecreasing}.a_i)$$
$$\mathit{guarantees}$$
$$u \geq k \ \rightsquigarrow \ u \geq S.\mathit{earliest}.k$$

Lemma 13 From lemma 12 and $(\forall l : k \leq l \leq S.earliest.k : S.earliest.l = S.earliest.k)$,

$$
\begin{aligned}
&((u \geq k \wedge u \leq S.earliest.k \wedge \neg(\forall i :: a_i = u) \;\; next \;\; u \leq S.earliest.k \\
&\wedge(\forall i :: always.(S_i.free.a_i)) \\
&\wedge(\forall i :: always.(a_i \leq S_i.earliest.a_i)) \\
&\wedge(\forall i :: a_i \; follows \; S_i.earliest.u) \\
&\wedge(\forall i :: nondecreasing.a_i)) \\
&guarantees \\
&X
\end{aligned}
$$

The right side of the next property only mentions u, which is locally-modifiable in C. The property therefore holds if it holds in C. Since this is indeed the case, we can simplify the above to

Lemma 14

$$
\begin{aligned}
&(\forall i :: always.(S_i.free.a_i)) \\
&\wedge(\forall i :: always.(a_i \leq S_i.earliest.a_i)) \\
&\wedge(\forall i :: a_i follows S_i.earliest.u) \\
&\wedge(\forall i :: nondecreasing.a_i) \\
&guarantees \\
&X
\end{aligned}
$$

From lemmas 5 and 14, we conclude (21).