# Mechanizing a Theory of Program Composition for UNITY

LAWRENCE C. PAULSON
University of Cambridge

Compositional reasoning must be better understood if non-trivial concurrent programs are to be verified. Chandy and Sanders [2000] have proposed a new approach to reasoning about composition, which Charpentier and Chandy [1999] have illustrated by developing a large example in the UNITY formalism. The present paper describes extensive experiments on mechanizing the compositionality theory and the example, using the proof tool Isabelle. Broader issues are discussed, in particular, the formalization of program states. The usual representation based upon maps from variables to values is contrasted with the alternatives, such as a signature of typed variables. Properties need to be transferred from one program component's signature to the common signature of the system. Safety properties can be so transferred, but progress properties cannot be. Using polymorphism, this problem can be circumvented by making signatures sufficiently flexible. Finally the proof of the example itself is outlined.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical Verification*

General Terms: Theory, Verification

Additional Key Words and Phrases: UNITY, Isabelle, concurrency, compositional reasoning

Contents

## 1. INTRODUCTION

This paper reports experiments on mechanizing a theory of UNITY program composition. The key ingredients are mechanized proofs, compositional reasoning, and the UNITY formalism.

—Mechanical proof tools provide rigour and power, but they are highly sensitive to small changes in the definitions. The gap between the "hand proof" and "mechanical proof" communities makes communication difficult. Unstated assumptions in a hand formalism can cause major problems during its mechanization. Notations designed for hand proofs may not be suitable for mechanical tools.

—Compositional reasoning means deriving a system's properties from those of its components. It requires reconciling any differences in how those components are formalized. This paper uses the **guarantees** primitive [Chandy and Sanders 2000], but almost any theory of compositionality poses similar problems.

—UNITY [Chandy and Misra 1988] is a simple formalism for expressing and verifying shared-variable concurrent programs. Properties can be proved from program texts, and program components can be specified abstractly without giving an implementation. The issues discussed below are relevant to other shared-variable formalisms, such as the Temporal Logic of Actions [Lamport 1994].

*Paper outline.* The paper begins with a brief review of UNITY (Section 2) and an overview of the **guarantees** primitive (Section 3). Then it discusses a basic issue: how to formalize the states of a system component (Section 4). It argues against the standard practice of assuming a uniform state representation based on a flat name space, preferring instead to give each program component its own signature of strongly-typed variables. The system's signature is the union of those of its components, so component properties must be transferred to the system's signature.

Pursuing the typed-signature approach, we find that the **guarantees** primitive can be made to work with safety properties: a complete solution is available (Section 5). For progress properties, the situation is different: no complete solution can exist, but two partial solutions are described (Section 6). Although they are not used in the final example, the findings presented in these two sections are important.

The inability to transfer progress properties forces a new approach. By including a polymorphic dummy variable in each component, we can ensure that between the system's signature and a suitable instance of the component's signature there is an isomorphism (Section 7). This new approach necessitates changes to the existing treatment of process arrays (Section 8). The resource allocation system of Charpentier and Chandy [1999] is introduced, with remarks about how some of the proofs were mechanized (Section 9). Finally, a brief concluding section considers the implications of this work (Section 10).

## 2. REVIEW OF THE UNITY FORMALISM

For the sake of completeness, let us recall the main elements of UNITY. More detail can be found elsewhere [Misra 1995a; Misra 1995b].

A UNITY program consists of a set of atomic *actions* that operate upon a declared set of *variables*. An execution step applies an action (chosen nondeterministically) to the current state, resulting in a new state. The set of actions always includes **skip**, which leaves the state unchanged. An *initial condition* specifies the states in which execution may begin.

Safety properties are proved of the atomic actions according to a weakest precondition semantics. Progress (or liveness) properties can also be proved, typically by assuming that the actions are executed under weak fairness. UNITY includes an elegant calculus for proving program properties, and authors have published numerous worked examples. Although UNITY proofs were originally done by hand, researchers have attempted to use mechanical proof tools. The present paper is a continuation of my earlier work [Paulson 2000] on mechanizing UNITY using Isabelle [Paulson 1994].

The Isabelle formalization is based on higher-order logic. It represents state predicates as sets of states. It represents a guarded command as a relation on states. It represents a program property as the set of programs satisfying that property. The theory is polymorphic in the type of states, making it independent of any particular type of states.

The safety properties include constrains (**co**), **stable** and **invariant**:

$$A \textbf{ co } B \triangleq \{F \mid A \text{ is a precondition for } B \text{ in } F\}$$
$$\textbf{stable } A \triangleq A \textbf{ co } A$$
$$\textbf{invariant } A \triangleq \{F \mid \textbf{Init } F \subseteq A \wedge F \in \textbf{stable } A\}$$

A program satisfies $A$ **co** $B$ if each command takes any state in $A$ to a state in $B$. A special case is **stable** $A$: once execution enters the set $A$ it can never leave. A stable predicate that holds at the start is called **invariant**. We write **Init** $F$ for $F$'s set of initial states and **Acts** $F$ for $F$'s set of actions.

Let $\mathcal{R}(F)$ denote the set of states reachable in the program $F$. We can define *weak* forms of the safety properties by restricting attention to reachable states [Sanders 1991]:

$$A \textbf{ co}_w B \triangleq \{F \mid F \in (\mathcal{R}(F) \cap A) \textbf{ co } B\}$$
$$\textbf{stable}_w A \triangleq A \textbf{ co}_w A$$
$$\textbf{always } A \triangleq \{F \mid \textbf{Init } F \subseteq A \wedge F \in \textbf{stable}_w A\}$$

These weak properties satisfy many of the same laws as the strong ones. The presence of $\mathcal{R}(F)$ complicates the meta-theory but makes the primitives more usable in practice: program proofs can appeal to previously-proved invariants.

Progress properties include **transient**, **ensures** and $\mapsto$ ("leads-to"). A program satisfies **transient** $A$ if some action takes $A$ to $\overline{A}$: informally, it falsifies $A$. The program satisfies $A$ **ensures** $B$ if it takes $A$ to $B$ by an atomic action. This concept is expressed as the conjunction of **transient**$(A \setminus B)$ — we cannot have $A$ without $B$ forever — with $(A \setminus B)$ **co** $(A \cup B)$ — the state remains in $A$ until it enters $B$. The main progress assertion, $A \mapsto B$, is defined as the transitive and disjunctive closure of $A$ **ensures** $B$. "Disjunctive closure" means that if $F \in A_i \mapsto B$ for all $i$ in $I$ then $F \in (\bigcup_{i \in I} A_i) \mapsto B$. There is also a weak form of leads-to, which restricts attention to reachable states:

$$F \in A \mapsto_w B \triangleq \{F \mid F \in (\mathcal{R}(F) \cap A) \mapsto B\}$$

The Isabelle mechanization proves hundreds of theorems of the meta-theory and develops several examples from the literature.

## 3. A MECHANIZED THEORY OF COMPOSITION

When components are joined to form a system, we ought to be able to combine the properties of the components without reasoning directly about their underlying actions. The

Isabelle UNITY theory defines program composition following Misra [1994a], who calls it *union*. Composing two programs $F$ and $G$ yields a new program, written $F \sqcup G$, defined by

$$\mathbf{Init}(F \sqcup G) = \mathbf{Init}\, F \cap \mathbf{Init}\, G$$
$$\mathbf{Acts}(F \sqcup G) = \mathbf{Acts}\, F \cup \mathbf{Acts}\, G.$$

Intuitively, the program texts are merely combined. Although $F \sqcup G$ should be undefined when **Init** $F$ and **Init** $G$ are disjoint, all Isabelle functions are total, yielding in this situation $\mathbf{Init}(F \sqcup G) = \emptyset$: there are no legal initial states. On the other hand, the logic is strongly typed, so $F \sqcup G$ can be written only if $F$ and $G$ have the same state type. This point will be crucial below.

The Isabelle UNITY theory proves many laws such as these:

$$\bot \sqcup F = F \tag{1}$$
$$\bigsqcup_{i \in I \cup J} F_i = \left(\bigsqcup_{i \in I} F_i\right) \sqcup \left(\bigsqcup_{i \in J} F_i\right)$$
$$F \sqcup G \in A \textbf{ co } B \iff (F \in A \textbf{ co } B) \wedge (G \in A \textbf{ co } B) \tag{2}$$
$$F \sqcup G \in \textbf{transient } A \iff (F \in \textbf{transient } A) \vee (G \in \textbf{transient } A) \tag{3}$$

Here $\bot$ is the null program, which has only a **skip** action and allows all initial states. Laws (2) and (3) are compositional, but there are no similar ones for weak safety ($\mathbf{co}_w$) or for leads-to. To allow compositional reasoning more generally, a component's specification must somehow describe the assumptions it makes of its environment.

### 3.1 The **guarantees** Operator

Many authors have put forward theories for compositional reasoning. Misra [1994b] defines *closure properties*. A program in Coq-UNITY must specify the maximal set of actions it may be composed with [Heyd and Crégut 1996]. The **guarantees** primitive of Chandy and Sanders [2000] is attractive. It is simple and remarkably general; the artefacts being specified do not even have to be programs.

In attempting to formalize **guarantees** in Isabelle, I discovered many implicit assumptions that had to be identified, clarified and formalized. Most of these were assumptions about UNITY itself that had not caused problems hitherto. The effort needed to mechanize a straightforward example [Charpentier and Chandy 1999] was entirely out of proportion to what one might have expected from reading the paper. I describe a small fraction of this effort below.

A **guarantees** assertion specifies a conditional assurance, typically to perform some service. By definition, $F \in (X \textbf{ guarantees } Y)$ means for all $G$, if $F \sqcup G$ satisfies $X$, then $F \sqcup G$ also satisfies $Y$. Here $X$ and $Y$ are program properties: safety, progress, or even other guarantees properties. Guarantees assertions provide a general means of proving safety and progress properties of systems that take $F$ as a component. Unlike many other rely-guarantees theories, the relied-upon assertion ($X$) and the guaranteed assertion ($Y$) refer to the same system, which avoids complications when reasoning about a system built of many components.

The **guarantees** theory requires the *component* relation, which defines a partial ordering

on programs:

$$F \le G \triangleq \exists H\,[F \sqcup H = G]$$

An equivalent definition is

$$F \le G \triangleq (\mathbf{Init}\,G \subseteq \mathbf{Init}\,F) \wedge (\mathbf{Acts}\,F \subseteq \mathbf{Acts}\,G).$$

Among the facts proved are $F_i \le \bigsqcup_{j\in I} F_j$ for $i \in I$ and the obvious $\bot \le F$. Once the component relation has been proved to be reflexive, anti-symmetric and transitive, it can be installed as a partial ordering using Isabelle's type class mechanism. All theorems proved about partial orderings then become applicable to the component relation.

The **guarantees** operator is defined such that

$$F \in (X\ \mathbf{guarantees}\ Y) \iff \forall G\,[F \sqcup G \in X \to F \sqcup G \in Y]$$

Note that in $F \in X$ **guarantees** $Y$ the assertion as a whole specifies $F$, but $X$ and $Y$ specify programs of the form $F \sqcup G$ for arbitrary $G$.

### 3.2 Theorems About **guarantees**

Chandy and Sanders [2000] state many facts about guarantees. In Isabelle, some of these facts are best expressed as derived rules of inference. If $X$ is stronger than $Y$ then any program trivially satisfies $X$ **guarantees** $Y$:

$$\frac{X \subseteq Y}{F \in X\ \mathbf{guarantees}\ Y}$$

Guarantees properties can be chained. A corollary of the following rule, putting $X = \emptyset$, is that **guarantees** is transitive:

$$\frac{F \in V\ \mathbf{guarantees}\ (X \cup Y) \quad F \in Y\ \mathbf{guarantees}\ Z}{F \in V\ \mathbf{guarantees}\ (X \cup Z)}$$

Other properties of **guarantees** are best expressed as equational laws. Here $\mathcal{X}$ and $\mathcal{Y}$ range over *sets* of program properties.

$$\Big(\bigcap_{X\in\mathcal{X}} X\ \mathbf{guarantees}\ Y\Big) = \Big(\bigcup_{X\in\mathcal{X}} X\Big)\ \mathbf{guarantees}\ Y$$
$$\Big(\bigcap_{Y\in\mathcal{Y}} X\ \mathbf{guarantees}\ Y\Big) = X\ \mathbf{guarantees}\ \Big(\bigcap_{Y\in\mathcal{Y}} Y\Big) \tag{4}$$

These equations could also be expressed as logical equivalences, and we could abbreviate $\bigcap_{Y\in\mathcal{Y}} Y$ as $\bigcap \mathcal{Y}$, which expresses the conjunction of all the properties in $\mathcal{Y}$. Equation (4) becomes

$$\forall_{Y\in\mathcal{Y}}\,[F \in X\ \mathbf{guarantees}\ Y] \iff F \in X\ \mathbf{guarantees}\ \bigcap \mathcal{Y},$$

saying for $F$ to guarantee every property in $\mathcal{Y}$ is equivalent to $F$'s guaranteeing $\bigcap \mathcal{Y}$.

### 3.3 On Local Variables

Before we look at the Isabelle definitions, we must consider a further issue: local variables. Published examples often use them, but UNITY has no notion of locality. At best, we can

```
preserves :: "('a=>'b) => 'a program set"
  "preserves v == ⋂z. stable {s. v s = z}"

guar :: ['a program set, 'a=>'b, 'a program set] => 'a program set
        ("(_/ guarantees[_]/ _)" [55,0,55] 55)
 "X guarantees[v] Y ==
      {F. ∀G ∈ preserves v. F⊔G ∈ X → F⊔G ∈ Y}"
```

Fig. 1. The **guarantees** Primitive in Isabelle

declare that certain variables may only be updated by certain processes and forbid compositions that violate these restrictions. In general, we need a treatment of *compatibility*: the relation that holds between components that may be composed.

I am developing a formalization of compatibility, but the experiments reported in this paper are based on the simplest treatment of locality. Assuming nothing about the structure of states, we can represent a variable $v$ by a function to inspect the state: a function from states to values. A collection of $n$ variables is represented similarly: by a function from states to an $n$-tuple of values. If $v$ is a function over states then **preserves** $v$ is the set of programs that do not modify the part of the state inspected by $v$. Intuitively, $v$ is a variable and **preserves** $v$ is the set of programs that do not modify $v$. The corresponding version of **guarantees** takes $v$ as an additional argument to restrict the quantification over environments:

$$\textbf{preserves}\, v \triangleq \bigcap z \left(\textbf{stable}\{s \mid v(s) = z\}\right)$$

$$X\ \textbf{guarantees}_v\ Y \triangleq \{F \mid \forall_{G \in \textbf{preserves}\, v}\, [F \sqcup G \in X \to F \sqcup G \in Y]\}$$

The Isabelle declarations appear in Fig. 1, including types and syntactic information for $X$ **guarantees**$_v$ $Y$. Type `'a program` is the type of programs; the type variable `'a` allows it to be used with any type of states. Type `'a program set` is the type of program properties, which are sets of programs. The discussion below ignores **preserves** to avoid obscuring the main points.

## 4. REPRESENTING COMPONENT STATES

Consider a system built from separate components, each a UNITY program operating on a fixed set of variables. A component may modify some of the variables, while others are read-only. A component's state is determined by the values of all the variables it can read. To use mechanical proof tools requires formalizing (among many other things) each component's state space.

### 4.1 Possible Choices

States can be formalized in various ways. To allow composition, it is convenient if the state type is common to all components. Obviously the components may be designed independently of each other, and while they must communicate by an agreed-upon set of shared variables, we cannot expect the combined set of variables (which includes each component's local variables) to be agreed upon in advance. For maximum flexibility, we should permit a variety of state types and to provide a means of transferring properties from one type to another. At the opposite extreme, we could impose a uniform "ISO standard" state type on all programmers now and forevermore, ensuring a flat, global name space.

In her PhD dissertation, Vos [1999] discusses state representations. She notes that most authors represent the state by a function from variables to values. Andersen et al. [1994], using the HOL system, define the types of variables and their values separately for each program. Heyd and Crégut [1996], using the Coq system, represent a state by a dependent function from variables to types. The type of variables is defined by enumeration and the type of states can map each variable to a distinct type. Both the HOL and Coq approaches permit a variety of state types. Prasetya [1995] adopts a uniform but unacceptably restrictive representation: a state is a function from variables to natural numbers. Vos generalizes Prasetya's approach to allow variables of other types. Her representation is also uniform: a state is a function from variables to a large, recursively defined value space.

### 4.2 Drawbacks of a Global Value Space

Of all the representations discussed above, Vos's map from variables to a global value space is the best. It is uniform and reasonably flexible. However, it has drawbacks:

—It is potentially restrictive. Her values are the disjoint sum of natural numbers, booleans, reals, strings and (recursively) lists, trees and finite sets of values. That is quite comprehensive, but adding a new type (such as bags) would be a major undertaking. Users must live with the selection provided.

—It clutters expressions with constructor and destructor functions: injections into the disjoint sum and their (partial) inverses. Vos alleviates this problem somewhat by defining a notation for expressions over her value space. But we would like to use the full notation of our proof tool, not a subset of it.

—It ignores the relation between variables and their types, yielding a weakly-typed formalism. Instead of specific types such as **int** and **bool**, all variables effectively have type **any**.

Here is an example to illustrate the last two points. Types in higher-order logic behave much as they do in programming languages. For instance, if $i$ and $j$ have type **int** then so does the expression $i + j$, and the only values they can have are integers. With weak typing, $i$ and $j$ have type **any** and can take on various values such as **make_int**(3) and **make_bool**(**false**). We must write the integer sum of $i$ and $j$ as **make_int**(**dest_int**($i$) + **dest_int**($j$)). The constructor and destructor functions are ugly and they complicate proofs. Reasoning about **dest_int**($i$) requires showing that $i$ always has a value constructed using **make_int**, in effect an assertion that $i$ is well-typed. The conjunction of the typing assertions for all the program variables forms a typing invariant. We must prove this typing invariant and appeal to it frequently. Proofs become more complex and less can be proved automatically. Certainly, we should try to avoid weak typing.

Lamport [1994, Section 2.1] adopts the uniform approach for TLA, decreeing "A state is an assignment of values to variables—that is, a mapping from the set **Var** of variable names to the collection Val of values." However, when mechanizing TLA, Merz [1999, Section 1] felt compelled to find another representation of states: the "universal state space that underlies TLA is not well supported by the type system of Isabelle/HOL." The problem is not specific to one proof tool; most of them use strongly-typed logics.

Most UNITY proofs are done by hand, where none of these problems arise. There is no need to impose a limited selection of types. The constructor and destructor functions are implicit, as is the typing invariant. Informal discussions with Charpentier indicate that he

regards the representation of states to be uniform — and thus, common to all components. As noted above, Lamport [1994] takes the same view, which I believe is common among practitioners of hand proofs.

### 4.3 Representing States as Records

For mechanical proof, a better representation of states is a record type. This claim requires justification, for a record is essentially the same as a tuple; as Vos notes, tuples are suitable only for trivial programs. The key difference between records and tuples is that records are equipped with syntactic sugar that keeps track of variable names, preventing confusion between different variables or state types. Some proof tools allow record types to be combined, easily expressing new record types.

The typed variable declarations in a UNITY program correspond in an obvious way to a record type, which we can view as the program's signature. The resulting formal proofs are strongly typed and need no typing invariants. It is attractive to formalize each component with respect to its own signature. When building the system, we form the union of the signatures of the components.

Typically we have to add variables to each signature. Component $F$ has been implemented (or specified) with a state type involving variables $v$ and $x$, say, and must now be combined with a component $G$ that uses a third variable, $y$. In previous work, I have described an approach to this problem [Paulson 2000, Section 10]. A generalized pairing function $h$ expresses how the signature has been extended. The type of $h$ is $\alpha \times \beta \Rightarrow \gamma$, where $\alpha$ is the old type of states, $\beta$ is a type comprising the additional variables and $\gamma$ is the new type of states. The function $h$ — which must be injective in its first argument and surjective — is supplied to other functions that extend properties and programs to the new signature:

—If $A$ is a set of states (equivalently, a state predicate) for the original signature then **extend_set** $h\ A$ is the corresponding set for the extended signature. This set does not depend on the new parts of the signature: in our example, the variable $y$.

$$\textbf{extend\_set}\, h\ A \triangleq \{h(x, y) \mid x \in A \text{ and } y \text{ ranging over type } \beta\}$$

—If $act$ is an action of $F$, then **extend_act** $h\ act$ has the corresponding effect on the old part of the state while leaving the new part unchanged.

$$\textbf{extend\_act}\, h\ act \triangleq \{(h(s, y), h(s', y)) \mid (s, s') \in act \text{ and } y \text{ ranging over type } \beta\}$$

—If $F$ is a program for the original signature then **extend** $h\ F$ is the corresponding program for the extended signature. This program lets $y$ take on any initial value and never updates it.

$$\textbf{Init}(\textbf{extend}\ h\ F) = \textbf{extend\_set}\, h\ (\textbf{Init}\ F)$$
$$\textbf{Acts}(\textbf{extend}\ h\ F) = \textbf{extend\_act}\, h\ `\ (\textbf{Acts}\ F)$$

Here $f\ `\ A$ is $\{f(x) \mid x \in A\}$, the image of the set $A$ under the function $f$, and thus **extend_act** $h\ `\ (\textbf{Acts}\ F)$ is the set of extended $F$-actions.

The previous paper sketches the proofs that the main program properties are preserved. We

have, for example, these equivalences:

$$\textbf{extend}\ h\ F \in (\textbf{extend\_set}\ h\ A)\ \textbf{co}\ (\textbf{extend\_set}\ h\ B) \iff F \in A\ \textbf{co}\ B \quad (5)$$
$$\textbf{extend}\ h\ F \in (\textbf{extend\_set}\ h\ A) \mapsto (\textbf{extend\_set}\ h\ B) \iff F \in A \mapsto B$$
$$\textbf{extend}\ h\ F \in \textbf{invariant}(\textbf{extend\_set}\ h\ A) \iff F \in \textbf{invariant}\ A$$

We also have the analogous laws for the weak versions of these operators. Of course, they do not tell us anything about properties of **extend** $h\ F$ that involve the extra variable $y$, but why should we care about such properties when **extend** $h\ F$ never uses $y$?

It is easy to prove a law for **guarantees**. It resembles the others, but — appearances are deceiving — it is almost useless. The extended guarantee only applies to programs that neither modify nor even read the new variable $y$:

$$\textbf{extend}\ h\ F \in \Big((\textbf{extend}\ h\ `\ X)\ \textbf{guarantees}\ (\textbf{extend}\ h\ `\ Y)\Big) \iff F \in X\ \textbf{guarantees}\ Y \quad (6)$$

The image **extend** $h\ `\ X$ denotes the set of programs of the form **extend** $h\ H$ for $H \in X$. These programs do not mention the variable $y$. When $G$ is a program component that uses $y$, this guarantee tells us nothing about the composed system (**extend** $h\ F$) $\sqcup\ G$. We need a more general theorem.

## 5. TRANSFERRING SAFETY GUARANTEES

Our problem is essentially one of linguistics. The guarantee for $F$ is formalized in the language of state type $\alpha$, in which (say) it can refer to variables $v$ and $x$. To include $F$ in a system involving another variable $y$ requires transferring $F$'s guarantee to this extended language. Naturally, $F$ cannot do anything with $y$.

A uniform state type would avoid these problems by providing a common language for expressing components' guarantees. But, as argued above, a flat name space has drawbacks. Therefore, let us see how far we can go using signatures. Progress properties, as we shall discover in the next section, present severe problems. For safety, the problems can be solved; safety properties can be transferred.

### 5.1 The Extended Guarantee Theorem

Extending a guarantee from one signature to another requires transferring properties from the extended signature down to the original signature and back again. (This observation follows directly from the definition of **guarantees**.) Some terminology may be useful in the discussion below. The original signature is the state type $\alpha$, and programs over that type are called $\alpha$-programs. The extended state type is $\gamma$, whose programs are called $\gamma$-programs. The generalized pairing function $h$, which has type $\alpha \times \beta \Rightarrow \gamma$, relates the two signatures. A key result is the *extended guarantee theorem*:

THEOREM 1. *Let $F$ be an $\alpha$-program and let $p$ be a function that maps $\gamma$-programs to $\alpha$-programs. Let $X$ and $Y$ be $\alpha$-program properties and let $X'$ and $Y'$ be $\gamma$-program properties. Then if for all $\gamma$-programs $G$ we have*

$$(\textbf{\textit{extend}}\ h\ F) \sqcup G \in X' \Longrightarrow F \sqcup p(G) \in X \quad \text{(i)}$$
$$F \sqcup p(G) \in Y \Longrightarrow (\textbf{\textit{extend}}\ h\ F) \sqcup G \in Y' \quad \text{(ii)}$$

*and if $F \in X$ **guarantees** $Y$ then we have **extend** $h\ F \in X'$ **guarantees** $Y'$.*

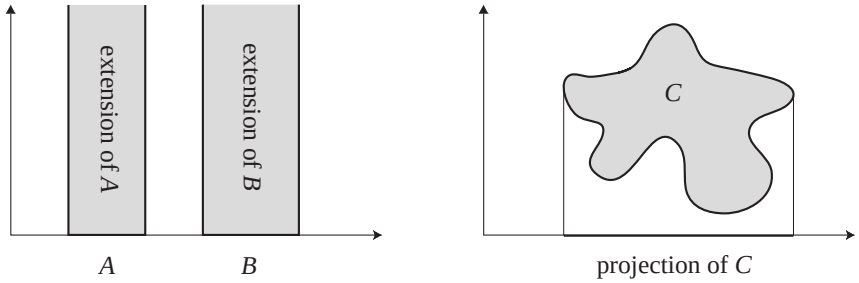


Fig. 2. Examples of **extend_set** and **project_set**

The proof is trivial by the definition of **guarantees**, which as we recall expands to a universally quantified implication. We assume the antecedent of the guarantee we are trying to derive, project it (using the function $p$) down to type $\alpha$, apply the existing guarantee, then extend the conclusion upwards. (The Isabelle proofs also consider the **preserves** property, but that does not concern us here.)

In order to apply this theorem, we start with a guarantee for $F$, for type $\alpha$, that we need to extend to some other type $\gamma$. We must choose program properties $X'$ and $Y'$ that adequately represent the guarantee we require and choose a function $p$ such that (i) and (ii) hold.

Note that this theorem places no constraints on the function $p$ apart from its type and conditions (i) and (ii). Plainly $p$ must be an inverse of the function **extend**. If we imagine a program $F$ executing in an extended signature, and consider what $F$ will see of actions performed by its environment, then we arrive at the obvious definition of projection. Program $F$ sees the effect of changes to variables in its signature (say $v$ and $x$), and does not see changes to other variables ($y$). To be precise, the action $act$ is projected from $\gamma$-states down to $\alpha$-states by

$$\{(x, x') \mid \exists y\, y'\, [(h(x, y), h(x', y')) \in act]\}. \tag{7}$$

When trying to develop a worthwhile theory, I considered some alternative definitions. The most plausible idea is to require $y = y'$ above, projecting only actions that leave the extended part of the signature unchanged:

$$\{(x, x') \mid \exists y\, [(h(x, y), h(x', y)) \in act]\}. \tag{7'}$$

Taking this idea to the extreme, we could even arrange for projection to ignore actions other than those created by **extend_act**. These variants still make $p$ an inverse of **extend**, but they do not transfer program properties as well as (7), the version adopted.

A $\gamma$-state predicate $A$ is projected to an $\alpha$-state predicate as follows:

$$\{x \mid \exists y\, [h(x, y) \in A]\}$$

Figure 2 illustrates two sets $A$ and $B$ being extended and a set $C$ being projected. In both cases, $h$ is the ordinary pairing function, so projection is the obvious map from the plane down to a line.

## 5.2 Weak Safety: An Example

The obvious projection function for programs projects the initial condition and actions as described above. From this definition of $p$, we can satisfy the conditions of Theorem 1, provided the guarantee we want to extend concerns strong safety. Naturally we want more: at least, weak safety. Transferring weak program properties requires complicating the projection function, adding an argument to give control over the set of reachable states. Let us see why this step is necessary.

Recall that a weak safety property is a strong safety property that has been relativized with respect to reachability. Naive projection, as proposed above, does not work for weak safety. Consider the following program over an extended signature.

$$\begin{aligned} &\textbf{initially } x, y = 0, 1 \\ &\qquad x, y := x + y, y + 2 \end{aligned}$$

Execution yields $(x, y) = (0, 1)$, (1,3), (4,5), (9,7), ...; the variable $y$ enumerates the odd numbers while $x$ enumerates perfect squares. (Note that $1 + \cdots + (2n-1) = n^2$.) Now "$y$ is an odd number" is a strong invariant, since it is preserved by the assignment. However, "$x$ is a perfect square" is only an **always** property (a weak invariant). When we project this program to the signature containing $x$ alone, we obtain the following:

$$\begin{aligned} &\textbf{initially } x = 0 \\ &\qquad x := x + \langle \textit{anything} \rangle \end{aligned}$$

The projected program allows any number to be added to $x$. It has lost the weak safety property "$x$ is a perfect square." We can correct this problem by projecting a state transition only if the starting state is reachable in the original program. In our example, the projected program only has the instances of $x := x+y$ such that $(x, y)$ is reachable, namely if $x = n^2$ and $y = 2n + 1$ for some $n \geq 0$. The result of the projection is a program consisting of the infinitely many assignments

$$\begin{aligned} x &:= x + 1 \quad \text{if} \quad x = 0 \\ [] \; x &:= x + 3 \quad \text{if} \quad x = 1 \\ [] \; x &:= x + 5 \quad \text{if} \quad x = 4 \\ [] \; x &:= x + 7 \quad \text{if} \quad x = 9 \\ [] \; x &:= \cdots \end{aligned}$$

We are no longer dealing with program texts in any practical sense, but with program semantics. Formally, our modified projection function involves restricting the domain of each action.

## 5.3 The Isabelle Theory of the Projection Operators

Figure 3 presents, in Isabelle theory syntax, the definitions used for the projection operations. As described above, to transfer weak safety properties requires an operator to restrict an action's domain. This operator satisfies laws such as the following:

$$\begin{aligned} (x, y) \in \textbf{Restrict}\, A\, r &\iff (x, y) \in r \wedge x \in A \\ \textbf{Restrict UNIV}\, r &= r \\ \textbf{Restrict}\, A\, (\textbf{Restrict}\, B\, r) &= \textbf{Restrict}(A \cap B)\, r \end{aligned}$$

The projection functions for sets and actions were described above. Each takes as its first argument the function $h$, which describes how the signature is extended with new variables. The set **project_set** $h\ A$ is the result of ignoring the new variables. The action **project_act** $h\ act$ is like $act$ but ignores its effect on the new variables. Obviously **project_act** $h\ Id = Id$ (where $Id$, the identity relation, represents the **skip** action). As we might expect, both projection functions are left inverses:

$$\textbf{project\_set}\ h\ (\textbf{extend\_set}\ h\ A) = A$$
$$\textbf{project\_act}\ h\ (\textbf{extend\_act}\ h\ act) = act$$

Now we might expect to have **project** $h$ (**extend** $h\ F$) $= F$, but there are complications. In Section 5.2 we saw that **project_act** does not preserve weak safety properties; a solution is to restrict each action to the set of reachable states before projecting it. Strong safety does not require restricting the actions, but since **Restrict UNIV** $r = r$ we can arrange for restriction to do nothing.

An additional argument, the $\gamma$-set $C$, accommodates these options. If $G$ is a $\gamma$-program then **project** $h\ C\ G$ expresses its projections. Each action of the projected program is given by

$$\textbf{project\_act}\ h\ (\textbf{Restrict}\ C\ act) \qquad \text{for } act \in \textbf{Acts}\ G,$$

which restricts an action's domain to $C$ before projecting it.

The function $p$ needed to apply Theorem 1 is (by currying) **project** $h\ C$. To extend a guarantee involving strong safety, we put $C = \textbf{UNIV}$; projecting a program then projects its initial condition and actions. For weak safety, we put $C = \mathcal{R}((\textbf{extend}\ h\ F) \sqcup G)$ where $G$ is the environment mentioned in conditions (i) and (ii). (Formally, $G$ is a universally quantified variable whose scope comprises those conditions.) If the guarantee combines strong and weak safety properties, then the conflicting constraints on $C$ prevent our applying Theorem 1.

Provided $C$ is big enough — if **project_set** $h\ C = \textbf{UNIV}$ — we have

$$\textbf{project}\ h\ C\ (\textbf{extend}\ h\ F) = F;$$

it suffices if $C = \textbf{UNIV}$. In general, the program **project** $h\ C$ (**extend** $h\ F$) is derived from $F$ by restricting the domains of its actions. Another result shows that this sort of restriction preserves safety properties.

$$\frac{F \in A\ \textbf{co}\ B}{\textbf{project}\ h\ C\ (\textbf{extend}\ h\ F) \in A\ \textbf{co}\ B} \tag{8}$$

We now come to a series of results that let us apply Theorem 1 to guarantees of safety. Because all the proofs have been mechanized, I merely sketch the arguments below. None of these results are deep. As so often in mathematics, the difficulty consists in knowing what theorems to prove from what definitions; the proofs themselves are straightforward.

### 5.4 Strong Safety and Projection

We begin with a rule that tells us what we can conclude when the program **extend** $h\ F$ satisfies a **co** property relating $\gamma$-state predicates $A$ and $B$. It is proved by a direct appeal to the definitions of the constants mentioned. The converse fails because $A$ and $B$ may disagree in the new parts of the signature; for instance, if $A$ has the form $P(x) \wedge y = 1$

```
Restrict :: "['a set, ('a*'b) set] => ('a*'b) set"
  "Restrict A r == r ∩ (A × UNIV)"

project_set :: "['a*'b => 'c, 'c set] => 'a set"
  "project_set h A == {x. ∃y. h(x,y) ∈ A}"

project_act :: "['a*'b => 'c, ('c*'c) set] => ('a*'a) set"
  "project_act h act == {(x,x'). ∃y y'. (h(x,y), h(x',y')) ∈ act}"

project :: "['a*'b => 'c, 'c set, 'c program] => 'a program"
  "project h C F ==
       mk_program (project_set h (Init F),
                   project_act h ' Restrict C ' Acts F)"
```

Fig. 3. The Projection Operators in Isabelle

and $B$ has the form $Q(x) \wedge y = 2$ then we cannot have **extend** $h\ F \in A$ **co** $B$ because the program cannot change the value of $y$.

$$\frac{\textbf{extend}\ h\ F \in A\ \textbf{co}\ B}{F \in (\textbf{project\_set}\, h\, A)\ \textbf{co}\ (\textbf{project\_set}\, h\, B)} \tag{9}$$

Next we have an equivalence used in several proofs about **project** and constraints. The proof is straightforward, again by appeal to the definitions. The second conjunct is necessary; if $C = \emptyset$ then **project** $h\ C\ G$ degenerates to a null program, and the left side reduces to $A \subseteq B$.

$$\textbf{project}\, h\, C\, G \in A\ \textbf{co}\ B \iff G \in (C \cap \textbf{extend\_set}\ h\ A)\ \textbf{co}\ (\textbf{extend\_set}\ h\ B) \wedge A \subseteq B \tag{10}$$

Strong safety properties can be transferred downwards. This is condition (i) of Theorem 1. The premise refers to the system in the extended signature and the conclusion refers to the same system projected to signature $\alpha$. It is proved using (2) and (10).

$$\frac{(\textbf{extend}\ h\ F) \sqcup G \in (\textbf{extend\_set}\ h\ A)\ \textbf{co}\ (\textbf{extend\_set}\ h\ B)}{F \sqcup (\textbf{project}\, h\, C\, G) \in A\ \textbf{co}\ B}$$

Strong safety properties can also be transferred upwards. This is condition (ii) of Theorem 1. It too is proved using (2) and (10). The **UNIV** argument cannot be replaced by an arbitrary $C$, for if $C = \emptyset$ then the premise degenerates to $F \in A$ **co** $B$.

$$\frac{F \sqcup (\textbf{project}\, h\ \textbf{UNIV}\, G) \in A\ \textbf{co}\ B}{(\textbf{extend}\ h\ F) \sqcup G \in (\textbf{extend\_set}\ h\ A)\ \textbf{co}\ (\textbf{extend\_set}\ h\ B)}$$

Since we can satisfy both conditions, we can apply Theorem 1 to guarantees whose preconditions and postconditions involve strong safety (**co**). Naturally enough, if the original (type $\alpha$) safety property was $A$ **co** $B$, then the extended safety property is

$$(\textbf{extend\_set}\ h\ A)\ \textbf{co}\ (\textbf{extend\_set}\ h\ B).$$

### 5.5 Weak Safety and Projection

To transfer weak safety properties requires further laws. The variable $C$ plays an important role. Some proofs refer to the function $f$ that satisfies $f(h(x, y)) = x$, which exists

because $h$ is injective in its first argument. It is the obvious projection from the extended signature to the original signature: it has type $\gamma \Rightarrow \alpha$.

This equivalence generalizes (10) to the projected system $F \sqcup (\mathbf{project}\, h\, C\, G)$.

$$(F \sqcup (\mathbf{project}\, h\, C\, G)) \in A\ \mathbf{co}\ B \iff (\mathbf{extend}\ h\ F) \sqcup G \in (C \cap \mathbf{extend\_set}\ h\ A)\ \mathbf{co}\ (\mathbf{extend\_set}\ h\ B) \wedge F \in A\ \mathbf{co}\ B \quad (11)$$

Note that the second conjunct has changed from $A \subseteq B$ to $F \in A\ \mathbf{co}\ B$. Given that $C$ can be instantiated with a set of reachable states and is intersected in the precondition of the **co** operator, we can see how this can give rise to weak safety properties. Once again the proof appeals to (2) and (10).

We need some results about the connection between **project** and the reachability function $\mathcal{R}$. This theorem says that if the state $h(x, y)$ is reachable in the $\gamma$-system then state $x$ is reachable in the corresponding $\alpha$-system. In practice, we satisfy the first premise by putting $C = \mathcal{R}((\mathbf{extend}\ h\ F) \sqcup G)$.

$$\frac{\mathcal{R}((\mathbf{extend}\ h\ F) \sqcup G) \subseteq C \quad h(x, y) \in \mathcal{R}((\mathbf{extend}\ h\ F) \sqcup G)}{x \in \mathcal{R}(F \sqcup (\mathbf{project}\, h\, C\, G))} \quad (12)$$

To prove the theorem, we must reformulate it in terms of the function $f$ mentioned above. In the second premise, replace $h(x, y)$ by $z$, and in the conclusion, replace $x$ by $f(z)$. The proof is by induction over the reachability of $z$.

Now we obtain condition (ii) of Theorem 1: weak safety properties can be transferred upwards. The proof uses (11) and (12).

$$\frac{C = \mathcal{R}((\mathbf{extend}\ h\ F) \sqcup G) \quad F \sqcup \mathbf{project}\, h\, C\, G \in A\ \mathbf{co}_w\ B}{(\mathbf{extend}\ h\ F) \sqcup G \in (\mathbf{extend\_set}\ h\ A)\ \mathbf{co}_w\ (\mathbf{extend\_set}\ h\ B)} \quad (13)$$

Since (12) can be applied with $C = \mathbf{UNIV}$, we could have got this far without having $C$ in our theory at all. Now we come to results that depend crucially upon $C$. The first is a converse to (12). It says that if state $x$ is reachable in the $\alpha$-system, then some extension of $x$ is reachable in the corresponding $\gamma$-system.

$$\frac{C \subseteq \mathcal{R}((\mathbf{extend}\ h\ F) \sqcup G) \quad x \in \mathcal{R}(F \sqcup (\mathbf{project}\, h\, C\, G))}{\exists y\, [h(x, y) \in \mathcal{R}((\mathbf{extend}\ h\ F) \sqcup G)]} \quad (14)$$

As before, we always apply this theorem with $C = \mathcal{R}(\mathbf{extend}\ h\ F \sqcup G)$. The proof is by induction over the reachability of $x$.

Finally, we obtain condition (i) of Theorem 1: weak safety properties can be transferred downwards. The rather involved proof appeals to (2), (9), (11) and (14). Below, $C_{FG}$ is an abbreviation for $\mathcal{R}((\mathbf{extend}\ h\ F) \sqcup G)$.

$$\frac{(\mathbf{extend}\ h\ F) \sqcup G \in (\mathbf{extend\_set}\, h\ A)\ \mathbf{co}_w\ (\mathbf{extend\_set}\, h\ B)}{F \sqcup (\mathbf{project}\, h\, C_{FG}\, G) \in A\ \mathbf{co}_w\ B}$$

Again we can satisfy both conditions and therefore can apply Theorem 1 to guarantees involving weak safety. The property $A\ \mathbf{co}_w\ B$ is transferred as

$$(\mathbf{extend\_set}\ h\ A)\ \mathbf{co}_w\ (\mathbf{extend\_set}\ h\ B).$$

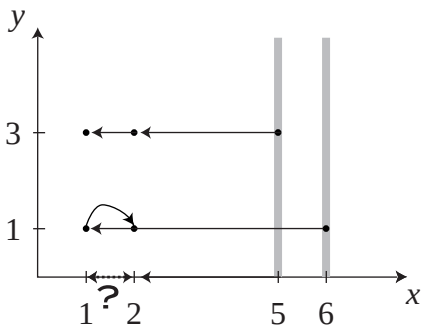


Fig. 4. A Progress Property That Cannot Be Transferred Downwards

### 5.6 Safety Guarantees: Summary

Let us review the situation. We formalize states not by a uniform representation based on variable to value maps, but as records. A program component's state type specifies a finite set of variables and their types, which constitute the component's signature. To combine components, first we must merge their signatures and then we must transfer each component's guarantees to the common signature. The equivalence (6) is useless here, but we can transfer guarantees using Theorem 1.

An appeal to Theorem 1 involves expressing the original $\alpha$-program properties $X$ and $Y$ as $\gamma$-program properties $X'$ and $Y'$, and then showing (i) that $X'$ can be transferred down to $X$ and (ii) that $Y$ can be transferred up to $Y'$. In the previous section we saw that this could be done for program properties of the form $A$ **co** $B$ and $A$ **co**$_w$ $B$. Boolean expressions and quantifications over these properties can be transferred too, provided the properties occur only in positive positions. For example, if $X$ is a conjunction (actually, an intersection in our formalism) then $X'$ will also be a conjunction; to satisfy (i), the conjuncts are transferred separately.

This is a complete solution for safety properties. The only thing we cannot do, for reasons explained in Section 5.3, is combine strong and weak properties in a single guarantee. This is seldom required anyway.

## 6. TRANSFERRING PROGRESS GUARANTEES

Safety has been solved, but no complete solution exists for progress. Safety is tractable because a **co** property involves a single state change. A leads-to property, however, may involve an arbitrarily long sequence of state changes. Theorem 1 turns out to be of little use for transferring progress guarantees.

We can still seek partial solutions, typically based on stronger notions of progress. We can restrict attention to certain key variables, and we can transfer **ensures** assertions, which like **co** involves one state change only.

### 6.1 The Problem with Progress

Let us see why no general solution exists, starting with condition (i): transferring a leads-to property downwards. Here the environment (process $G$) is influenced by the extra part of the signature (the variable $y$), which is lost after projection.

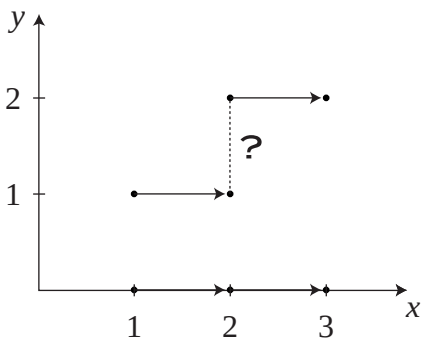


Fig. 5. A Progress Property That Cannot Be Transferred Upwards

Figure 4 presents a typical case. Component $G$ has the actions

$$\begin{array}{rll} x := x-1 & \text{if} & x < 5 \wedge y = 3 \\ [] \; x := x+1 & \text{if} & x < 5 \wedge y = 1 \\ [] \; x, y := 2, 3 & \text{if} & x = 5 \\ [] \; x, y := 1, 1 & \text{if} & x = 6 \end{array}$$

and therefore satisfies

$$\begin{array}{rl} & \{x = 5\} \mapsto \{x = 2 \wedge y = 3\} \mapsto \{x = 1\} \\ \text{and} & \{x = 6\} \mapsto \{x = 1 \wedge y = 1\} \mapsto \{x = 2\}. \end{array}$$

After projection, the program loses the variable $y$:

$$\begin{array}{rll} x := x-1 & \text{if} & x < 5 \\ [] \; x := x+1 & \text{if} & x < 5 \\ [] \; x := 2 & \text{if} & x = 5 \\ [] \; x := 1 & \text{if} & x = 6 \end{array}$$

Since this program can both increase and decrease $x$, it satisfies neither $\{x = 5\} \mapsto \{x = 1\}$ nor $\{x = 6\} \mapsto \{x = 2\}$.

Condition (ii) involves transferring a leads-to property upwards. Here the projected system may be able to make progress when the original system cannot because the $y$-parts do not match. The progress we see in the projected system is illusory.

Figure 5 illustrates a typical situation. The system might have the actions

$$\begin{array}{rll} x := x+1 & \text{if} & x \leq 1 \wedge y = 1 \\ [] \; x := x+1 & \text{if} & x > 1 \wedge y = 2. \end{array}$$

After projection, these become

$$\begin{array}{rll} x := x+1 & \text{if} & x \leq 1 \\ [] \; x := x+1 & \text{if} & x > 1 \end{array}$$

so the projected system satisfies $\{x = 1\} \mapsto \{x = 3\}$. The original system does not, since the two increment actions are guarded by different values of $y$.

These examples show there is no general way to apply Theorem 1 with leads-to. But what if we impose special conditions on the environment or on the progress properties? If only condition (ii) could be satisfied, then we could at least transfer guarantees with a safety precondition and a progress postcondition. But there is no point in satisfying condition (i) alone, since no guarantee needs a progress precondition to establish a safety postcondition.

### 6.2 Variable-Restricted Progress

To illustrate the sort of special case worth investigating, this section presents a variable-restricted form of leads-to. The intuition is simple: since new variables can cause the transfer to fail, strengthen leads-to so that it specifies precisely which variables are relevant to making progress. Such progress assertions can be transferred — and we seem to have solved the problem of reasoning in a compositional way about progress. That a flaw remains (the details appear below) shows again how slippery this problem is.

The idea is to restrict the base case of leads-to in its use of **ensures**, for that is where progress actually occurs. Let $CC$ be a set of state predicates and consider the relation $A \mapsto_{CC} B$, defined to be the transitive and disjunctive closure of the following base case:

$$\frac{F \in A \textbf{ ensures } B \quad A \setminus B \in \{\emptyset\} \cup CC}{A \mapsto_{CC} B}$$

Here $A \setminus B$ denotes set difference and **ensures** is defined by

$$A \textbf{ ensures } B \triangleq ((A \setminus B) \textbf{ co } (A \cup B)) \cap \textbf{transient}(A \setminus B).$$

If $F \in A \mapsto_{CC} B$ then $F$ makes progress from $A$ to $B$ with each use of **ensures** confined to state predicates drawn from the set $CC$. Including $\emptyset$ among the allowed state predicates allows $A \mapsto_{CC} B$ to enjoy precondition strengthening and postcondition weakening, which are needed in order to derive other standard progress theorems, such as PSP.

The plan is to let $CC$ be the set of state predicates that can be expressed using certain variables alone. Section 3.3 noted how a function $v$ from states to $n$-tuples can represent a collection of $n$ variables. Consider all state predicates of the form $\{s \mid P(v(s))\}$, where $P$ is a predicate over $n$-tuples. Such state predicates are "given by $v$," in the sense that every access to the state has the form $v(s)$; there is no direct access to $s$. We can neatly define this collection of state predicates:

$$A \in \textbf{givenBy } v \iff \exists P\, [A = \{s \mid P(v(s))\}]$$

This definition is polymorphic: it need not mention $n$-tuples or specify the types of $v$ and $P$.

Now $A \mapsto_{\textbf{givenBy } v} B$ specifies the programs that progress from $A$ to $B$ while depending only on the variables in $v$: changes to other variables cannot interfere. Typically, assertions about a primitive component $F$ involve all its local variables. As it is combined with other components, its progress assertions (now about **extend** $h\, F$) will depend on its local variables only, ignoring the new ones.

From the definitions above, a substantial proof effort establishes conditions (i) and (ii) for Theorem 1. Both theorems refer to the function $f$, which as we recall from Section 5.5 is the first projection of $h$. It has type $\gamma \Rightarrow \alpha$, mapping extended states to the corresponding original states.

This theorem satisfies condition (i), transferring the progress property downward:

$$\frac{\textbf{extend}\ h\ F \sqcup G \in (\textbf{extend\_set}\ h\ A) \mapsto_{\textbf{givenBy}\ f}\ (\textbf{extend\_set}\ h\ B)}{F \sqcup \textbf{project}\ h\ \textbf{UNIV}\ G \in A \mapsto B}$$

If the $\gamma$-system makes progress from $A$ to $B$ depending only on $f$ — and therefore *not* depending on the additional variables in type $\gamma$ — then its projection satisfies $A \mapsto B$. The conclusion uses the standard leads-to relation.

This theorem satisfies condition (ii), transferring the progress property upward:

$$\frac{F \sqcup \textbf{project}\ h\ \textbf{UNIV} G \in A \mapsto_{\textbf{givenBy}\ v}\ B}{\textbf{extend}\ h\ F \sqcup G \in (\textbf{extend\_set}\ h\ A) \mapsto_{\textbf{givenBy}(v \circ f)}\ (\textbf{extend\_set}\ h\ B)}$$

If the projected (type $\alpha$) system makes progress from $A$ to $B$ depending only on the variable $v$, then its extended version satisfies $A \mapsto B$, depending now on $v \circ f$; the composed function refers to the variable $v$ of the original state signature.

Both theorems are available for the weak version of $A \mapsto_{CC} B$ too, so we can freely use the variable-restricted progress relation in guarantees. Unfortunately, it is less useful than it looks. We can reason about a component that makes progress on its own. However, when several components co-operate to make progress, we cannot restrict attention to one component's local variables. Section 9 will describe the resource allocator example and indicate where the proof fails.

### 6.3 Transferring an Ensures Property

We can handle the special case where a component makes progress by an atomic action: an **ensures** property. Provided the environment is well-behaved as given by (16) below, we can transfer the **ensures** property upwards. This satisfies condition (ii) and lets us extend a guarantee that has a safety precondition and a progress postcondition.

The first lemma involves **unless**, where as we recall $A$ **unless** $B$ abbreviates $(A \setminus B)$ **co** $(A \cup B)$.

$$\frac{G \in \textbf{stable}\ C \quad \textbf{project}\ h\ C\ G \in (\textbf{project\_set}\ h\ C \cap A)\ \textbf{unless}\ B}{G \in (C \cap \textbf{extend\_set}\ h\ A)\ \textbf{unless}\ (\textbf{extend\_set}\ h\ B)} \tag{15}$$

This result is one of the few with a tolerably easy proof. Things get pretty horrendous from now on, though as before the details are not illuminating.

We can now formalize what it means for the environment to be well-behaved. For the system to progress from $C \cap$ **extend_set** $h\ A$ to **extend_set** $h\ B$, the environment must keep the system in $C \cap$ **extend_set** $h\ A \setminus$ **extend_set** $h\ B$. Formally, this is a **stable** property:

$$G \in \textbf{stable}((C \cap \textbf{extend\_set}\ h\ A) \setminus (\textbf{extend\_set}\ h\ B)) \tag{16}$$

All the results below depend on this assumption.

The first step is to prove that a non-trivial **transient** property cannot have originated with the projection of $G$. It is for this result that condition (16) is assumed; a more natural condition such as $G \in (C \cap$ **extend_set** $h\ A)$ **unless** (**extend_set** $h\ B$) is too weak, letting $G$ make progress.

$$\frac{\textbf{project}\ h\ C\ G \in \textbf{transient}(\textbf{project\_set}\ h\ C \cap A \setminus B)}{C \cap \textbf{extend\_set}\ h\ A \setminus \textbf{extend\_set}\ h\ B = \emptyset} \tag{17}$$

The alternative would be to allow the projection of $G$ to satisfy a **transient** property and to show that $G$ itself satisfies the corresponding property on $\gamma$-states. Transferring a **transient**

property upwards requires exhibiting an action that, among other things, is enabled over all of $C \cap$ **extend_set** $h\ A \setminus$ **extend_set** $h\ B$. This requirement is hard to satisfy, for (as a glance at Fig. 2 will remind you) applying **extend_set** yields a large set. If the action has such a large domain, presumably it does not use the additional variable at all.

The next theorem has a second premise, **extend** $h\ F \sqcup G \in$ **stable** $C$, which is trivial to satisfy: we only apply the theorem with $C =$ **UNIV** or $C = \mathcal{R}$(**extend** $h\ F \sqcup G$).

$$\frac{F \sqcup \textbf{project}\ h\ C\ G \in (\textbf{project\_set}\ h\ C \cap A)\ \textbf{ensures}\ B}{\textbf{extend}\ h\ F \sqcup G \in (C \cap \textbf{extend\_set}\ h\ A)\ \textbf{ensures}\ (\textbf{extend\_set}\ h\ B)}$$

The proof is messy. The safety aspect of **ensures** is proved by appeals to (2), (8) and (15). The **transient** aspect refers to (3) and (17). This result directly establishes condition (ii) for both strong and weak **ensures**.

For strong **ensures**, we assume (16) with $C =$ **UNIV**.

$$\frac{F \sqcup \textbf{project}\ h\ \textbf{UNIV}\ G \in A\ \textbf{ensures}\ B}{\textbf{extend}\ h\ F \sqcup G \in (\textbf{extend\_set}\ h\ A)\ \textbf{ensures}\ (\textbf{extend\_set}\ h\ B)}$$

For weak **ensures**, abbreviate $C = \mathcal{R}$(**extend** $h\ F \sqcup G$):

$$\frac{F \sqcup \textbf{project}\ h\ C\ G \in A\ \textbf{ensures}_w\ B}{\textbf{extend}\ h\ F \sqcup G \in (\textbf{extend\_set}\ h\ A)\ \textbf{ensures}_w\ (\textbf{extend\_set}\ h\ B)}$$

These rules give a partial treatment of progress, but it is unsatisfactory in several respects. It only covers progress made by a single component; the progress must be made atomically (by **ensures**); proving the well-behavedness assumption (16) can be tedious. Luckily, there is much simpler way of transferring guarantees — *all* guarantees — while retaining the benefits of signatures. However, it requires abandoning Theorem 1.

## 7. POLYMORPHIC STATE TYPES

Our problems with **guarantees** are caused by the use of projection. Program properties are not easily preserved under projection, especially in the case of progress. If we are to transfer guarantees from one signature to another, the signatures have to be isomorphic. Fortunately, we can ensure that they will be isomorphic by taking advantage of polymorphism. We get the best of both worlds: strong typing and the ability to combine any two signatures. No global state type is required, but we must put up with an extra component in each signature.

Each component is given an extra variable that it does not use. Initially this variable is unconstrained, and each action preserves its value. Crucially, the variable's type is not specified, but is left polymorphic. In essence, the component stands for an infinite family of components, all identical but for the type they give this variable. Everything we prove of this component, including **guarantees** properties, will be polymorphic in the dummy variable's type.

When we combine several components to build a system, we first work out what the common signature should be. Then, for each component $F$, we form the Cartesian product of all the new variables and give this type to the dummy variable. More precisely, we specify the bijection from the $F$'s signature to the common signature; the domain of this bijection will be an instance of the $F$'s polymorphic type. The properties previously proved of $F$ are easily transferred over a bijection between states. This device solves all our

problems. While it introduces some complications, it is much simpler overall than any approach relying on Theorem 1.

Polymorphism is essential here. An obvious alternative is to introduce a global value space: a recursive disjoint sum over integers, booleans, lists, etc. Such a value space has many drawbacks (recall Section 4 above). And it does not help here, because the resulting state types will not be isomorphic. If component $F$ has an dummy variable that belongs to this value space, then including $F$ in a system will mean replacing its dummy variable by tuple of variables belonging to other system components. The tuple cannot take on all possible elements of the value space, so some states have been lost. It turns out (and I have devoted some effort to the attempt) that to transfer program properties through this sort of mapping is as hard as transferring them through a projection.

A new constant, **rename**, streamlines the mechanization. It applies **extend** using `unit`, the one-element type. This extends type $\alpha$ with nothing and yields an isomorphism between signatures:

```
rename :: "['a => 'b, 'a program] => 'b program"
  "rename h == extend (λ(x,u::unit). h x)"
```

Here `λ(x,u::unit). h x` is a function with pattern-matching; we could have written **rename** $h$ = **extend** $(h \circ \mathbf{fst})$, where **fst** is the first projection function for ordered pairs. The function $h$ must be a bijection; this condition is implicit in all the theorems presented below.

To transfer guarantees under **rename**, two equivalences can be used:

$$\mathbf{rename}\ h\ F \in (\mathbf{rename}\ h\ `\ X)\ \mathbf{guarantees}\ (\mathbf{rename}\ h\ `\ Y) \iff F \in X\ \mathbf{guarantees}\ Y \quad (18)$$

$$\mathbf{rename}\ h\ F \in X\ \mathbf{guarantees}\ Y \iff F \in (\mathbf{rename}(h^{-1})\ `\ X)\ \mathbf{guarantees}\ (\mathbf{rename}(h^{-1})\ `\ Y) \quad (19)$$

Each is a simple consequence of equation (6), which as you may recall from Section 4 was our first attempt at extending guarantees. The use of $h^{-1}$ in (19) does not cause problems. When we apply this equation, we have in mind a particular bijection $h$. Typically it is a renaming of record fields and we can easily express its inverse.

To see how we deal with the program properties in this framework, consider **co**. The following equivalence is an immediate consequence of the similar result (5) for **extend**:

$$\mathbf{rename}\ h\ F \in (h\ `\ A)\ \mathbf{co}\ (h\ `\ B) \iff F \in A\ \mathbf{co}\ B$$

Using this, we can prove another equivalence that helps us eliminate expressions of the form **rename** $h\ `\ X$, which are introduced by (18) and (19):

$$\mathbf{rename}\ h\ `\ (A\ \mathbf{co}\ B) = (h\ `\ A)\ \mathbf{co}\ (h\ `\ B)$$

To rename a **co** property is to rename its precondition and postcondition. Formally, each of these is a set, hence the use of the image operator. A similar equivalence can be proved for all the other program properties, such as **stable**, **invariant**, $\mapsto$ and their weak counterparts. These equivalences can be installed so that Isabelle's simplifier will apply them automatically. Not only can arbitrary guarantees be transferred; they can be transferred almost without effort. In contrast, invoking Theorem 1 requires several proof steps, even if we have already proved theorems for its conditions (i) and (ii).

## 8. ARRAYS OF PROCESSES, REVISITED

Arrays of processes sometimes arise in examples. The allocation system of Charpentier and Chandy [1999] comprises a resource allocator, a network, and a family of similar clients. In the previous UNITY paper, I described how the replication of a process could be handled [Paulson 2000, Section 11]. Now, replication requires taking care to keep the dummy-variable convention and to ensure that the renaming is a bijection.

As an aside, note that arrays of processes also pose a challenge for the uniform state representation based on the traditional variable-to-value map. If a single client has a variable called $x$, then each replicated client will have a subscripted variable $x_i$. So the type of variable names needs to admit subscripting; more generally, it must anticipate every kind of variable-name structuring that might be encountered. This is analogous to a problem already discussed, namely the need to anticipate every type of value that might be needed.

The treatment of the array of clients illustrates some features of our new approach, in particular, the treatment of the dummy variable. The client must be specified according to our new convention; it has a polymorphic dummy variable whose purpose is to anticipate all ways of extending the signature with new variables. When we insert this component into an array, the other array elements count as new variables. And to allow further extensions of the signature, the array must have its own dummy. So the original client's dummy variable is instantiated to range over pairs, consisting of an array of client states coupled with the array's dummy.

There is a further difficulty. Under the old approach, to make an array of processes was simple. To make the array element $i$, we used the obvious embedding from the value $v$ into the functions $f$ such that $f(i) = v$. However embeddings are no longer good enough: we must have bijections. This requires a careful construction in which the client's value is inserted into an already existing array, with the other elements pushed upwards so that no information is lost. That way the insertion operation becomes a bijection. The definitions and proofs are all straightforward, but compared with the previous approach, we have certainly lost some elegance. Also, the old approach allowed the array to be indexed by any type, but now the type has to be infinite, so I have fixed it to be that of natural numbers.

The necessary Isabelle declarations appear in Fig. 6. Here are informal descriptions of them:

—Function **insert_map** is like the familiar function update $f(i := z)$, except that the previous value of $f(i)$ is not lost but is moved to position $i + 1$, with further elements moved up.

—Function **delete_map** is the obvious inverse of **insert_map**, which discards a function's value at position $i$ and moves the other elements down.

—Function **lift_map** $i$, where $i$ is a natural number, is the state renaming to make element $i$ of an array. It expects the client's state to have the form $(s, (f, uu))$ where $s$ is the "real" state and $(f, uu)$ is its view of the dummy variable. The resulting state consists of the array built from $s$ and $f$, which supplies the other array elements; the new state has $uu$ as its dummy variable.

—Function **drop_map** $i$ is the inverse of **lift_map** $i$.

—Function **lift_set** $i$ translates a state predicate about clients into a state predicate about element $i$ of an array of clients.

—Function **lift** $i$ converts a program over the client signature into one over the signature

```
insert_map :: "[nat, 'b, nat=>'b] => (nat=>'b)"
  "insert_map i z f k == if k<i then f k
                         else if k=i then z
                         else f(k-1)"

delete_map :: "[nat, nat=>'b] => (nat=>'b)"
  "delete_map i g k == if k<i then g k else g (Suc k)"

lift_map :: "[nat, 'b * ((nat=>'b) * 'c)] => (nat=>'b) * 'c"
  "lift_map i == λ(s,(f,uu)). (insert_map i s f, uu)"

drop_map :: "[nat, (nat=>'b) * 'c] => 'b * ((nat=>'b) * 'c)"
  "drop_map i == λ(g, uu). (g i, (delete_map i g, uu))"

lift_set :: "[nat, ('b * ((nat=>'b) * 'c)) set]
             => ((nat=>'b) * 'c) set"
  "lift_set i A == lift_map i ` A"

lift :: "[nat, ('b * ((nat=>'b) * 'c)) program]
         => ((nat=>'b) * 'c) program"
  "lift i == rename (lift_map i)"

PLam  :: "[nat set, nat => ('b * ((nat=>'b) * 'c)) program]
          => ((nat=>'b) * 'c) program"
  "PLam I F == ⨆i:I. lift i (F i)"
```

Fig. 6. Isabelle Declarations for Arrays of Processes

for an array of clients, with the supplied program as element $i$. Used with the image operator, it translates program properties.

—Function **PLam** forms an array of processes over a supplied index set of natural numbers. The notation $\{F[i]\}_{i\in I}$ for **PLam** $I$ $F$ appeared in my previous paper [Paulson 2000].

Equations for these operators are easy to derive, culminating in a crucial property, namely that **lift_map** is bijective:

$$\begin{aligned}
\textbf{delete_map}\, i\, (\textbf{insert_map}\, i\, x\, f) &= f \\
(\textbf{insert_map}\, i\, x\, (\textbf{delete_map}\, i\, g)) &= g(i := x) \\
\textbf{drop_map}\, i\, (\textbf{lift_map}\, i\, s) &= s \\
\textbf{lift_map}\, i\, (\textbf{drop_map}\, i\, s) &= s
\end{aligned}$$

Naturally, since **lift** is ultimately based on **extend**, program properties can be transferred. For instance, here is the rule for weak progress:

$$(\textbf{lift}\, i\, F \in (\textbf{lift\_set}\, i\, A) \mapsto_w (\textbf{lift\_set}\, i\, B)) = (F \in A \mapsto_w B)$$

Guarantees can be transferred in two steps. Suppose that we have a process $F$ satisfying $F \in X$ **guarantees** $Y$. First, we transfer this guarantee so that it holds of an array containing $F$ as element $i$:

$$\textbf{lift}\, i\, F \in (\textbf{lift}\, i\, `\, X)\ \textbf{guarantees}\ (\textbf{lift}\, i\, `\, Y)$$

Then we appeal to the meaning of **guarantees**: every system that contains **lift** $i$ $F$ as a component and satisfies **lift** $i$ ‘ $X$ also satisfies **lift** $i$ ‘ $Y$. Since by definition

$$\{F[i]\}_{i\in I} \triangleq \bigsqcup_{i\in I} \textbf{lift}\ i\ F,$$

we find that **lift** $i$ $F$ is a component of $\{F[i]\}_{i\in I}$ and obtain

$$\{F[i]\}_{i\in I} \in (\textbf{lift}\, i \text{ ‘ } X)\ \textbf{guarantees}\ (\textbf{lift}\, i \text{ ‘ } Y).$$

In order to transfer a guarantee this way, we need the following equivalence, which is simply an instance of (18).

$$\textbf{lift}\, i\ F \ \in\ (\textbf{lift}\, i \text{ ‘ } X)\ \textbf{guarantees}\ (\textbf{lift}\, i \text{ ‘ } Y) \iff F \in X\ \textbf{guarantees}\ Y$$

In practice, it is often easier to transfer the guarantee backwards. That is, we start with the final system and a property we require of it. Then we reduce the property to one that must hold for an individual client. This equivalence, an instance of (19), does the job.

$$\begin{aligned}\textbf{lift}\, i\ F \in X\ \textbf{guarantees}\ Y \\ \iff F \in (\textbf{rename}(\textbf{drop\_map}\, i) \text{ ‘ } X) \\ \textbf{guarantees}\ (\textbf{rename}(\textbf{drop\_map}\, i) \text{ ‘ } Y)\end{aligned}$$

In proofs, this two-stage translation turns out to require a bit of effort. It does not work purely by logical equivalences, so it cannot be done purely by rewriting. A theorem of the form $\{F[i]\}_{i\in I} \in X'$ **guarantees** $Y'$ has to be stated explicitly and then proved from the assumption $F \in X$ **guarantees** $Y$. When a guarantee can be transferred by rewriting alone, it can be conveniently generated on the fly.

## 9. EXAMPLE: THE ALLOCATION SYSTEM

In the resource allocation system of Charpentier and Chandy [1999], several clients request and return resources (represented by tokens), while an allocator attempts to satisfy the requests. The design is compositional: the allocator, network and a typical client are specified separately. Properties of the allocator are proved under the assumption that the network and clients behave well; for instance, they return resources eventually. A typical client is verified under analogous assumptions. Then the allocation system is verified by reasoning from the components' properties.

### 9.1 System Overview

First, we must discuss the formalization of communication. In UNITY, processes communicate via shared variables, but the allocator and clients communicate over a network. The communication channels are modelled as histories, that is, as lists of transmitted messages. The network process has the task of copying data from the output of one process to the input of another process. For this purpose, Charpentier and Chandy use a modified version of **follows**, a temporal operator introduced by Sivilotti [1997] in his thesis. (History variables are ordered by extension. Variable $x$ follows $y$ if both are increasing over time, $x$ never exceeds $y$ and $x$ eventually reaches any value reached by $y$. This is expressed formally using **co** and $\mapsto$.) They specify the network to arrange that each input history follows the corresponding output history. Each component has full access to its history, so part of its specification is that it only extends its history. With this approach, no variable may be

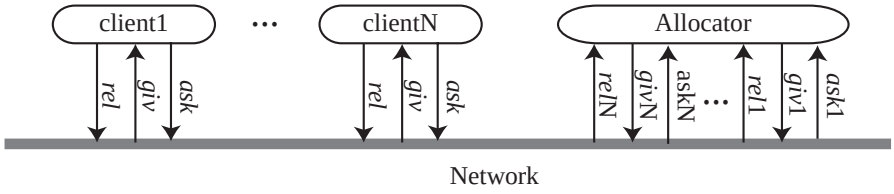


Fig. 7. The Allocation System

updated by more than one process. Each process writes to its output variables, which other processes can only read. The network is itself a process. This deviates from the shared-variable mutual-exclusion flavour of traditional examples, but (according to Charpentier) it makes proofs easier.

Figure 7 shows the system's structure. The $N$ clients each meet the same specification. (They form an array of processes, as described in Section 8.) Each client has three variables. It sends requests along *ask*; tokens are given to it along *giv*; it releases tokens after use along *rel*. The allocator communicates with its clients via the network. Its variables $ask_i$ and $rel_i$ receive data from *ask* and *rel* of client $i$, whose variable *giv* receives data from the allocator's variable $giv_i$. Each component is specified formally.

—The network's specification is simple, using the **follows** operator: each of its output histories follows that of the corresponding input history. For example, $ask_i$ follows variable *ask* of client $i$.

—The client's specification comprises three guarantees. History variables *ask* and *rel* are increasing. (The client makes no such guarantee for *giv*, a variable outside its control.) No element of *ask* exceeds *NbT*, the maximum permitted number of tokens. Finally, there is a progress guarantee: if the history *giv* is also increasing, and if the client has been given at least as many tokens as it has asked for, then it will eventually release all tokens it was given. The formal statements of these guarantees are formidable, involving operators for histories and prefixes. Figure 9 presents them in Isabelle syntax.

—The allocator meets three analogous guarantees. Variable $giv_i$ is increasing, for $1 = 1, \ldots, N$. The total number of tokens given and not released never exceeds *NbT*. The progress guarantee states that provided $ask_i$ and $rel_i$ are increasing ($1 = 1, \ldots, N$), no client asks for more than *NbT* tokens, and clients eventually return their tokens, then the allocator will meet all requests. These preconditions match the client's specifications.

Charpentier and Chandy [1999] put these components together to form a system, which they prove always to meet client requests. Intuitively, progress depends on the mutual co-operation of all components.

Finally we can see why the variable-restricted leads-to operator (Section 6.2) is insufficient. The network is the only component with direct access to all the variables in the system. Clients and the allocator have their signatures extended, so we express their progress guarantees using variable-restricted leads-to. The client's progress guarantee can be transferred, and we can combine it with the network's progress property, proving that the allocator's variable $rel_i$ behaves correctly: clients eventually return their tokens. It looks like we have proved the allocator's precondition. But we have not, for using variable-restricted leads-to in the allocator's specification allows progress to depend only on the allocator's

```
record clientState =
  giv :: nat list   (*client's INPUT history:  tokens GRANTED*)
  ask :: nat list   (*client's OUTPUT history: tokens REQUESTED*)
  rel :: nat list   (*client's OUTPUT history: tokens RELEASED*)

record 'a clientState_d =
  clientState +
  dummy :: 'a       (*dummy field for new variables*)

record allocState =
  allocGiv :: nat => nat list   (*OUTPUT history: source of giv[i] *)
  allocAsk :: nat => nat list   (*INPUT: allocator's copy of ask[i] *)
  allocRel :: nat => nat list   (*INPUT: allocator's copy of rel[i] *)

record 'a allocState_d =
  allocState +
  dummy    :: 'a                (*dummy field for new variables*)

record 'a systemState =
  allocState +
  client :: nat => clientState  (*states of all clients*)
  dummy  :: 'a                  (*dummy field for new variables*)
```

Fig. 8. The Resource Allocator: Process Signatures

local variables. Actually, the progress made by $rel_i$ depends on that made by client $i$'s local variable *rel*. Variable-restricted leads-to is simply too strong to express a guarantee's precondition.

## 9.2 Remarks on the Proof Effort

The allocation system is a substantial example. Its specification comprises nearly 200 lines of Isabelle text. (Figures 8 and 9 present extracts.) The proof script comprises nearly 600 lines, proving around 40 theorems with an average of just over two commands per proof. Much of the script concerns routine properties of translations between different signatures. A typical translation function maps one record to another, copying fields across and moving information in some trivial way. These functions are obviously bijections and their inverses can be read off from their definitions, but all these properties must, at present, be stated and proved explicitly. Routine reasoning of this sort amounts to fully half of the proof script. Specialized automation could save the user a lot of work.

I have mechanized most of the proofs in Charpentier and Chandy [1999], with the exception of some large ones in the appendices. A typical example is the first composition proof from section 4.2. This proof amounts to a page and a half; the reasoning is given in some detail. The effort needed to undertake this proof is difficult to quantify. Merely to formalize and enter the allocation system specification took the better part of a day. Some effort went into developing the **follows** operator, the generalized prefix relation, and other mathematical notions. The great majority of the effort was devoted to exploring the question of transferring guarantees, discussed at length in this paper. This exploration was interleaved with mechanizing the proof itself.

The Isabelle proofs follow the hand proofs quite well, but the level of automation is disappointing. At issue is the amount of effort needed to mechanize a given amount of

```
client_increasing :: 'a clientState_d program set
  "client_increasing ==
       UNIV guarantees[funPair rel ask]
       (Increasing ask) ∩ (Increasing rel)"

client_bounded :: 'a clientState_d program set
  "client_bounded ==
       UNIV guarantees[ask]
       Always {s. ∀elt ∈ set (ask s). elt ≤ NbT}"

client_progress :: 'a clientState_d program set
  "client_progress ==
         Increasing giv
         guarantees[funPair rel ask]
         (⋂h. {s. h ≤ giv s & h pfixGe ask s}
                 LeadsTo {s. tokens h ≤ (tokens o rel) s})"
```

Fig. 9. The Resource Allocator: Client Specification

proof text. To formalize a page from a mathematics book may take weeks; Paulson and Grąbczewski [1996, Section 5.2] discuss a sentence in a proof that took two days to mechanize. In contrast, a typical paper about weakest precondition calculi presents highly detailed proofs of elementary theorems. The UNITY meta-theory, including extensions such as the **guarantees** operator, falls into this category; many of the published proofs can be ignored, since the auto-tactic proves them in seconds. This is because most UNITY laws reduce to simple properties of sets, where Isabelle's reasoners are highly effective. Many UNITY examples can also be done easily, at least when proving safety. With the allocator, the need to combine the properties of the components makes automation difficult. The hand proofs appear to be written in great detail, but each line makes several implicit inferences.

For progress proofs, the situation is different. Progress proofs have always been (in my experience) hard to automate. They often appeal to transitivity and similar laws that cause any mechanical search to diverge. Even in the meta-theory, some progress laws (such as the completion and PSP theorems) are hard to prove. Per line of proof text, the difficulty of mechanization seems about the same whether the proof concerns program composition or the progress properties of a piece of code.

### 9.3 Signatures of the Components and the System

Since much of this paper has discussed signatures and maps between them, let us examine some of the signatures in the allocator example (Fig. 8). Record `clientState` declares the three variables of a client: the histories `giv`, `ask` and `rel`. The dummy variable is added separately, extending the record with a new field `dummy` to create the record type `clientState_d`. Record `allocState` declares the three analogous variables, but as they represent requests from all the clients, they are functions from clients (designated by natural numbers) to histories. Again, record `allocState_d` extends this state with a new field `dummy`. Finally, the record `systemState` extends `allocState` with a family of clients and a `dummy` field; the dummy is not strictly necessary, but it allows the system to become a component of some larger system later.

Here is a translation function from `allocState_d` to `systemState`, which illus-

trates how dummy variables work:

```
sysOfAlloc :: "((nat => clientState) * 'a) allocState_d
               => 'a systemState"
  "sysOfAlloc == λs. let (cl,xtr) = allocState_d.dummy s
                     in (| allocGiv = allocGiv s,
                           allocAsk = allocAsk s,
                           allocRel = allocRel s,
                           client   = cl,
                           dummy    = xtr|)"
```

The . . . _d records are polymorphic in the type of their dummy variable. This function requires the allocator's dummy to be a pair consisting of a family of clients (`cl`) and another value (`xtr`) of polymorphic type, which serves as the system's dummy. So the function is a bijection that simply re-arranges these values in the obvious way. One trivial lemma (needed for other proofs) presents this function's inverse, which simply returns the fields to their original places.

```
inv sysOfAlloc s =
            (| allocGiv = allocGiv s,
               allocAsk = allocAsk s,
               allocRel = allocRel s,
               allocState_d.dummy = (client s, dummy s) |)
```

Figure 9 presents the specification of a client. The system description assumes the existence of such a client, along with an allocator and network, satisfying their specifications:

```
consts
    Alloc   :: 'a allocState_d program
    Client  :: 'a clientState_d program
    Network :: 'a systemState program
    System  :: 'a systemState program
rules
    Alloc   "Alloc   : alloc_spec"
    Client  "Client  : client_spec"
    Network "Network : network_spec"
```

Here is how we combine these components to form a system. Process `Network` already belongs to the system signature, so it requires no renaming. Process `Alloc` belongs to the allocator's signature, so it must be renamed using the bijection `sysOfAlloc`. Process `Client` is replicated, forming an array of similar processes, using the `plam` binder. Informally, we might write something like $\{Client_i\}_{i<Nclient}$. The client needs renaming both after the replication (to add the other system variables) and before (to put its signature into the form that `plam` expects).

```
System == rename sysOfAlloc Alloc ⊔ Network ⊔
          rename sysOfClient
                 (plam x: lessThan Nclients. rename client_map Client)
```

## 10. CONCLUSIONS

The Isabelle mechanization of **guarantees** is successful in so far as a substantial example can be verified in a similar style to the hand proofs. However, this treatment of compositionality is not ideal, whether we use Theorem 1 or polymorphic state types (Section 7). The former approach gives a general treatment of safety, but it allows only a partial treatment of progress, and (having experimented at length) I do not see much hope of handling

progress in a guarantee's precondition. The dummy variable device is general, but it complicates definitions somewhat; and some theorem provers, such as ACL2 and PVS, do not have polymorphism.

There is a further problem, again concerning components' signatures. The client is specified to operate on three variables. However its implementation uses a fourth variable [Charpentier and Chandy 1999, Section 6.1]. Charpentier has pointed out (in an e-mail) that the original specification admits implementations with any number of additional variables. This view presupposes the uniform state representation, in which all variables are available. But a flat name space is not satisfactory: if several components are designed in this way, we run the risk of name clashes.

The **guarantees** primitive is not tied to UNITY. A process formalism that hides local variables — I do not know of a suitable one — would work better. The problems described in this paper would disappear. When a component was combined with the rest of the system, the latter's local variables would no longer appear in the signature; the two signatures would be identical; we would not have to transfer properties from one signature to another. UNITY remains valuable as a simple formalism in which fundamental issues can be examined easily.

An alternative conclusion is that we should indeed use a uniform state type. In this view, the difficulties described in this paper outweigh the arguments given in Section 4 against having a global name space. This claim can be examined experimentally by formalizing a uniform state type within the Isabelle UNITY environment. Performing two sets of proofs in the same environment would identify the respective benefits of the two state representations. Stephan Merz has suggested (in a private e-mail) yet another representation. Declare an abstract type of states and introduce selector functions for new variables as they are required. We get a universal state type and strong typing simultaneously. A drawback is that we are forced to assume axioms. Clearly there is room for much more experimentation.

The present work illustrates the hazards of formalization. I had been rendering the UNITY theory into Isabelle in a straightforward way; for a long time, this approach proceeded smoothly. But when I came to Chandy and Sanders [2000], definitions that looked correct proved to have the wrong properties. My treatment of states made **guarantees** too weak, highlighting the underlying assumption of a global name space. Formalization makes implicit assumptions explicit; that is one of its benefits.

The Isabelle formalization of UNITY is useful for investigating proposed constructs such as **guarantees**, both the meta-theory and possible applications. With a bit more development, the Isabelle environment could be used to verify larger examples involving program composition.

## ACKNOWLEDGMENTS

Michel Charpentier and Jayadev Misra gave much advice by electronic mail. Andrew Pitts and Glynn Winskel offered some tips on how to approach this work. Hector Andrade, Sara Kalvala, Michael Norrish and Beverly Sanders gave valuable comments on the paper, as did the referees.

## REFERENCES

ANDERSEN, F., PETERSEN, K. D., AND PETTERSSON, J. S. 1994. Program verification using HOL-UNITY. In *Higher Order Logic Theorem Proving and Its Applications: HUG '93*, J. Joyce and C. Seger, Eds. LNCS 780. Springer, 1–15.

CHANDY, K. M. AND MISRA, J. 1988. *Parallel Program Design: A Foundation*. Addison-Wesley.

CHANDY, K. M. AND SANDERS, B. A. 2000. Reasoning about program composition. Tech. Rep. 2000-003, CISE, University of Florida. available via http://www.cise.ufl.edu/~sanders/pubs/composition.ps.

CHARPENTIER, M. AND CHANDY, K. M. 1999. Towards a compositional approach to the design and verification of distributed systems. In *World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, J. M. Wing, J. Woodcock, and J. Davies, Eds. LNCS 1708. Springer, 570–589.

HEYD, B. AND CRÉGUT, P. 1996. A modular coding of UNITY in COQ. In *Theorem Proving in Higher Order Logics: TPHOLs '96*, J. von Wright, J. Grundy, and J. Harrison, Eds. LNCS 1125. Springer, 251–266.

LAMPORT, L. 1994. The temporal logic of actions. *ACM Trans. Program. Lang. Syst. 16*, 3 (May), 872–923.

MERZ, S. 1999. An encoding of TLA in Isabelle. http://www.pst.informatik.uni-muenchen.de/~merz/isabelle/TLA/doc/IsaTLADesign.ps.

MISRA, J. 1994a. Asynchronous compositions of programs. At URL ftp://ftp.cs.utexas.edu/pub/psp/unity/new_unity/composition.ps.Z.

MISRA, J. 1994b. Closure properties. At URL ftp://ftp.cs.utexas.edu/pub/psp/unity/new_unity/closure.ps.Z.

MISRA, J. 1995a. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering 3*, 2, 273–300. Also at URL ftp://ftp.cs.utexas.edu/pub/psp/unity/new_unity/progress.ps.Z.

MISRA, J. 1995b. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering 3*, 2, 239–272. Also at URL ftp://ftp.cs.utexas.edu/pub/psp/unity/new_unity/safety.ps.Z.

PAULSON, L. C. 1994. *Isabelle: A Generic Theorem Prover*. Springer. LNCS 828.

PAULSON, L. C. 2000. Mechanizing UNITY in Isabelle. *ACM Transactions on Computational Logic 1*, 1, 3–32.

PAULSON, L. C. AND GRĄBCZEWSKI, K. 1996. Mechanizing set theory: Cardinal arithmetic and the axiom of choice. *Journal of Automated Reasoning 17*, 3 (Dec.), 291–323.

PRASETYA, I. S. W. B. 1995. Mechanically supported design of self-stabiliting algorithms. Ph.D. thesis, University of Utrecht.

SANDERS, B. 1991. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing 3*, 2, 189–205.

SIVILOTTI, P. A. G. 1997. A method for the specification, composition, and testing of distributed object systems. Ph.D. thesis, California Institute of Technology, 256-80 Caltech, Pasadena, California 91125.

VOS, T. E. J. 1999. UNITY in Diversity, a stratified approach to the verification of distributed algorithms. Ph.D. thesis, Utrecht University.

1 November 2000