Mechanizing UNITY in Isabelle

Lawrence C. Paulson

UNITY is an abstract formalism for proving properties of concurrent systems, which typically are expressed using guarded assignments [Chandy and Misra 1988]. UNITY has been mechanized in higher-order logic using Isabelle, a proof assistant. Safety and progress primitives, their weak forms (for the substitution axiom) and the program composition operator (union) have been formalized. To give a feel for the concrete syntax, the paper presents a few extracts from the Isabelle definitions and proofs. It discusses a small example, two-process mutual exclusion. A mechanical theory of unions of programs supports a degree of compositional reasoning. Original work on extending program states is presented and then illustrated through a simple example involving an array of processes.

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

General Terms: Theory, Verification

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

1. INTRODUCTION

Reasoning about concurrent systems is difficult. Many formalisms have been introduced for this purpose, some involving hand methods and others supported by various tools. The present paper concerns a recent version of the UNITY formalism [Misra 1995a; Misra 1995b]. It describes preliminary experiments in reasoning about UNITY using the interactive proof system Isabelle [Paulson 1994].

A novel aspect of the work is its combination of interactive and automatic theorem-proving. Much recent research concerns fully automatic methods, which are inherently limited in scope. The present approach is to employ interactive proof while exploiting Isabelle's automatic tools to minimize the user's effort. Many classic examples [Chandy and Misra 1988] have been done with a modest effort. Elaborate Isabelle programming has not been required, merely the use of the built-in classical reasoner and simplifier. Another novelty is the use of set theory to formalize the many notational conventions adopted by UNITY researchers.

The research was funded by the U.K.'s Engineering and Physical Sciences Research Council, grant GR/K57381 'Mechanizing Temporal Reasoning.'
Address: Computer Laboratory, University of Cambridge, Cambridge CB2 3QG, England
email `lcp@cl.cam.ac.uk`

Paper outline. The paper begins with overviews of UNITY (§2), other UNITY tools (§3) and Isabelle (§4). It describes the Isabelle formalization of safety and progress primitives (§§5, 6), including their weak counterparts that satisfy the substitution axiom (§7). A small example is given: verification of a two-process mutual exclusion algorithm (§8). The theory of program composition is described (§9). An operator for changing the representation of states are described (§10) and then applied to an example involving arrays of processes (§11). Finally, the paper concludes (§12).

2. AN OVERVIEW OF UNITY

UNITY is a simple model of concurrent programming. There is a single, global state. A program consists of a collection of guarded atomic commands that are repeatedly selected and executed under some fairness constraint. There are no program structures; we cannot even write $C_1; C_2$ to express sequential execution. Typically a program manipulates its program counter in order to direct the flow of control. UNITY is of enduring interest: a decade after its introduction, it is still the focus of much research.

In the original version [Chandy and Misra 1988], called classic UNITY below, the only commands were simultaneous assignments to variables. New UNITY [Misra 1995b] allows any terminating commands. It imposes the trivial assumption that each program includes a **skip** command (which has no effect on the state). Commands do not have to be deterministic; they have a simple relational semantics.

UNITY includes a small fragment of temporal logic. While primitive compared with TLA [Lamport 1994] for instance, the formalism can express basic safety and progress properties. There is a set of elegant laws for proving such properties.

Safety properties are expressed using the *constrains* operator, **co**. The assertion A **co** B expresses the usual precondition-postcondition relationship.[1] If A holds then B will hold after the execution of any command of the program. (Since all commands terminate, partial and total correctness coincide.) If A **co** A then the predicate A holds forever once established, and is called **stable**. A stable predicate that holds initially is called **invariant**. Classic UNITY adopted a different safety primitive, **unless**, but **co** has a simpler theory; either operator can be defined in terms of the other.

Progress properties are expressed using the *leads-to* relation, $\mapsto$. The meaning of $A \mapsto B$ is that if A holds now then B is guaranteed to hold eventually. Leads-to properties depend upon which fairness policy is adopted. Misra [1995a] describes three: minimal progress, weak fairness and strong fairness. Most work assumes weak fairness, which says that a command will eventually be executed if it is enabled continuously. Strong fairness says that a command will be executed if it is enabled infinitely often. Minimal progress says that a command other than **skip** will be executed if at least one such command is enabled.

3. EXISTING UNITY TOOLS

UNITY originated as a hand formalism. Recent papers continue to present proofs found by hand. However, hand proofs often contain errors. Automation would be

[1] Some authors, such as Charpentier and Chandy [1999], use A **next** B synonymously.

beneficial.

Model checking has been applied to concurrent systems with great success. It is fully automatic, but its scope for handling traditional UNITY problems appears to be limited. Model checking examines fixed, finite-state systems, while typical UNITY proofs concern infinite-state systems or infinite families of systems (for example, a token ring with n nodes). Misra [1995b] supplies many examples where the program to be verified is not given as specific commands but is described abstractly to be any program satisfying certain low-level constraints. Despite these objections, Kaltenbach [1996] has developed a model checker for UNITY. Others have investigated alternative means of verifying UNITY programs automatically, using decision procedures for example [Thirioux 1998].

However promising automatic methods may be, a general treatment requires formal proof. A most impressive effort is HOL-UNITY [Andersen et al. 1994a], which implements classic UNITY and is based upon the HOL system [Gordon and Melham 1993]. It provides a good degree of automation and sports a graphical interface for proving progress properties using Owicki-Gries proof lattices [Andersen et al. 1994b]. Also using HOL, Prasetya [1995] has formalized a UNITY variant allowing read/write annotations on variables. Building on this work, Vos [1999] has added a deep embedding of the programming language and devoted special attention to finding representations of algorithms that simplify their proofs. Heyd and Crégut [1996] report an encoding of UNITY in the type theory tool Coq. They add a notion of context to allow program composition. They have investigated UNITY's meta-theory, proving that the safety and progress primitives are complete with respect to an operational semantics of program execution. All of these efforts, like the present work, derive the UNITY laws from definitions rather than taking them as axioms.

Private discussions with UNITY researchers indicate that proof support still needs to be improved, particularly for program composition. The aim of the Isabelle formalization is to provide that improved support. One strand of the research is to seek a better formalization, exploiting the power of set theory. Another is to continue to seek better forms of automation. A third strand is to mechanize a new theory for reasoning about program composition [Chandy and Sanders 1998]. This part of the work, still under development, will be useful in its own right while giving valuable feedback to the theoreticians.

4. OVERVIEW OF ISABELLE

Isabelle [Paulson 1994] is an interactive proof tool. Unlike similar tools, Isabelle is *generic*: the underlying mechanisms allow many different logics to be supported. Examples include Isabelle/HOL, a mechanization of higher-order logic, and Isabelle/ZF, a mechanization of untyped set theory [Paulson and Grąbczewski 1996]. Higher-order syntax allows new quantifiers to be defined. A variant of Horn clause resolution supports for user-derived inference rules.

Let us examine Isabelle's treatment of inference rules by an example. The traditional way of expressing that the relation $\prec$ is transitive is by the predicate calculus formula

$$\forall xyz.\, x \prec y \wedge y \prec z \rightarrow x \prec z.$$

Reasoning with such a formula requires many proof steps to manipulate the quantifiers and connectives. In Isabelle, transitivity can also be expressed by the rule

$$\frac{x \prec y \qquad y \prec z}{x \prec z.}$$

Isabelle primitives build proofs from rules, working either forwards or backwards. *Forward* proof means concluding $a \prec c$ from the theorems $a \prec b$ and $b \prec c$. *Backward* proof means reducing the goal $t \prec u$ to the subgoals $t \prec ?y$ and $?y \prec u$, where $?y$ can later be instantiated with any term. Usually the instantiation is performed automatically: Isabelle uses unification rather than matching when applying a rule.

UNITY proofs traditionally proceed in the forward direction. Isabelle mainly supports backward proof, for when working interactively it is natural to start with a claim and break it down. Commands for reducing goals to subgoals are called *tactics*. The simplest tactics, used for proof checking, merely apply a rule or list of rules to a subgoal. These could be familiar UNITY rules, perhaps modified to suit the backward style.

The UNITY development is based upon Isabelle/HOL. It relies on the typed set theory inherent in higher-order logic. Sets over type α have type $\alpha\,\mathbf{set}$; membership is written $x \in A$ and comprehension is written $\{x \mid \phi\}$. Conceptually sets are just predicates, but expressing membership as $A(x)$ and comprehension as $\lambda x.\,\phi$ is unpleasant. Set operators such as $A \cup B$, $A \cap B$ and $A \setminus B$ (difference) are defined in the obvious way using comprehension. Binding operators such as general union are defined and rules such as the following are derived:

$$\frac{x \in A \qquad b \in B(x)}{b \in \left(\bigcup_{x \in A} B(x)\right)} \tag{1}$$

Isabelle/HOL accepts inductive definitions of sets, which it translates to least fixed-points. The set of states reachable in a UNITY program is defined inductively, as is the leads-to relation (Fig. 2).

Isabelle provides powerful automatic tactics. The *simplifier* performs conditional, contextual and permutative rewriting. The *classical reasoner* (`Blast_tac`) proves subgoals using tableau methods [Paulson 1999]. It is generic, applying user-supplied rules such as (1) above and UNITY rules. `Auto_tac` attempts to prove as many subgoals as possible, using both the simplifier and the classical reasoner. An arithmetic tactic proves many goals involving linear arithmetic.

Polymorphism simplifies the treatment of set theory. The set-theoretic operators are polymorphic in the type of elements. Isabelle automatically uses the correct type, for example when rules are applied by the classical reasoner. (Dozens of rules are installed for this purpose.) The UNITY primitives are also polymorphic, as discussed below.

For ease of reading below, ASCII symbols will be replaced by their mathematical equivalents in Isabelle texts, so `x : A Un B` will appear as `x ∈ A∪B`. Rules are often presented entirely in mathematical notation.

```
co        :: "['a set, 'a set] => 'a program set"       (infixl 60)
  "A co B == {F. ∀act∈Acts F. act``A ⊆ B}"

stable    :: "'a set => 'a program set"
  "stable A == A co A"

invariant :: "'a set => 'a program set"
  "invariant A == {F. Init F ⊆ A} ∩ stable A"
```

Fig. 1. Declaring UNITY's Safety Primitives in Isabelle

5. FORMALIZING SAFETY PROPERTIES

UNITY deals with programs and their properties. A UNITY program is a set of guarded commands, while a property typically expresses a relationship between two predicates over states. How are these concepts best formalized?

I have chosen to work with sets of states rather than predicates over states. The difference is purely notational. Formulas are easier to read when they are expressed using familiar operators such as union rather than unusual operators such as "disjunction lifted over state variables." Other set-theoretic notions turn out to be useful, such as taking the image of a set under a function. Equally important is that Isabelle can prove many theorems about set theory automatically. Predicate notation is used below whenever it is clearer, typically when we are discussing particular programs instead of meta-theory.

Another fundamental decision concerns the representation of actions. A UNITY action can be executed only if it is enabled: if its guard is satisfied. Classic UNITY [Chandy and Misra 1988] includes only deterministic actions, with nondeterminism arising only in the choice of action. Accordingly, some researchers [Andersen et al. 1994a; Heyd and Crégut 1996] model actions as total functions over the state space. If there is a guard, then the function behaves as the identity unless the guard is satisfied. Thus we have the odd situation that the following two commands become indistinguishable:

$$x := f(x)$$
$$x := f(x) \qquad \text{if} \quad x \neq f(x)$$

These commands are certainly different in the operational semantics, since the first is always enabled and the second need not be. However, a **skip** action (required in all UNITY programs) can be executed if $x = f(x)$. So, the same sequences of states can be generated regardless of which of the two commands is included in a program.

Like Prasetya [1995], I prefer to represent actions by relations. One reason is that Misra [1995b] allows actions to be nondeterministic. Another reason is that relations are sets and will work smoothly with the set-theoretic formalization. In particular, the guard of an action is simply the domain of the corresponding relation. A relation will often be expressed as a set comprehension such that its guard can be determined trivially by rewriting; in mechanical proof, this will be easier than working with guards of the form $\{x \mid x \neq f(x)\}$.

And so, an action is a relation over states: a set of (σ, σ') pairs where σ is a state before execution of the command and σ' is a possible state after execution.

Relational semantics does not distinguish between divergence and abortion, but UNITY actions are atomic and admit neither. In the relational semantics, a command is enabled in state σ provided it can move to some next state σ'. As noted above, the relation's domain is the set of enabled states. There is little support for conventional program syntax: commands are expressed in set-theoretic notation.

The Isabelle formalization is polymorphic: a program has the type $\alpha\,\mathbf{program}$, where α is the type of states. For example, the constrains operator takes two arguments of type $\alpha\,\mathbf{set}$ and returns a result of type $\alpha\,\mathbf{program}$. Given the precondition and postcondition, it yields the set of programs satisfying that specification.

A UNITY program comprises a set of initial states, **Init** and a set of actions, **Acts**. Among the actions must be **skip**, which is formalized by the built-in identity relation, *Id*. The abstract type $\alpha\,\mathbf{program}$ embodies these conditions; it is essentially a subtype of a product type and has a single constructor function, **mk_program**, which satisfies two equations:

$$\mathbf{Init}(\mathbf{mk\_program}(\mathit{init}, \mathit{acts})) = \mathit{init}$$
$$\mathbf{Acts}(\mathbf{mk\_program}(\mathit{init}, \mathit{acts})) = \{\mathit{Id}\} \cup \mathit{acts}$$

Figure 1 presents the Isabelle definitions of the constrains primitive and related safety operators. The formalization follows Misra [1995b]. The assertion $F \in A\,\mathbf{co}\,B$ relates A and B, which are sets of states, with respect to the program F. A specification denotes a set of programs, as in the "propositions-as-types" paradigm; this allows a smooth formalization, especially for working with programs comprising several components, as in §11.

The meaning of $F \in A\,\mathbf{co}\,B$ is that if any command of F is executed in some state belonging to A then the next state will belong to B. The Isabelle theory defines constrains as follows:

$$A\,\mathbf{co}\,B \triangleq \{F \mid \forall_{act \in \mathbf{Acts}\,F}\,(act\text{“}A \subseteq B)\}$$

Here, $act\text{“}A$ denotes the image of the set A under the relation act: the set of all y such that $(x, y) \in act$ for $x \in A$. The assertion is that B includes these images for every possible action. It is not essential to use the image operator; there are many equivalent ways of expressing **co**. However, the image operator reduces the need for quantifiers while allowing the use of laws such as $R\text{“}(A \cup B) = R\text{“}A \cup R\text{“}B$.

The definition of $\mathbf{stable}(A)$ as $A\,\mathbf{co}\,A$ is standard. The theory also defines $A\,\mathbf{unless}\,B$ as $(A \setminus B)\,\mathbf{co}\,(A \cup B)$, where $A \setminus B$ denotes set difference. This set-theoretic definition, omitted from the figure, is equivalent to the usual definition: if first we have A and not B, then afterwards A or B will hold. The notion of invariant is defined such that

$$F \in \mathbf{invariant}\,A \iff \mathbf{Init}\,F \subseteq A \wedge F \in \mathbf{stable}(A). \tag{2}$$

The Isabelle theory proves some forty theorems about these safety operators. Nearly all are proved in one step, typically by calling `Auto_tac` or `Blast_tac`. These theorems include a few technical lemmas, but most of them express well-known properties of the operators **co**, **stable** and **invariant**. Let us review a few of them.[2]

[2]The full list can be viewed on the Internet at URL `http://www.cl.cam.ac.uk/Research/HVG/`

The relation **co** is transitive:

$$\frac{F \in A \,\mathbf{co}\, B \qquad F \in B \,\mathbf{co}\, C}{F \in A \,\mathbf{co}\, C}$$

The relation satisfies a disjunction property, stated here in terms of union:

$$\frac{F \in A \,\mathbf{co}\, A' \qquad F \in B \,\mathbf{co}\, B'}{F \in (A \cup B) \,\mathbf{co}\, (A' \cup B')}$$

If co holds between two I-indexed families of sets then it holds between the corresponding intersections:

$$\frac{\forall_{i \in I}\, (F \in A_i \,\mathbf{co}\, A'_i)}{F \in \left(\bigcap_{i \in I}\, A_i\right) \mathbf{co} \left(\bigcap_{i \in I}\, A'_i\right)}$$

The *elimination theorem* concerns a fixed program variable, say x. In set theory, it takes on an unconventional form. Here $\{s \mid s(x) = m\}$ is the set of all states such that the program variable x holds the value m.

$$\frac{\forall_{m \in M}\, (F \in \{s \mid s(x) = m\} \,\mathbf{co}\, B_m)}{F \in \{s \mid s(x) \in M\} \,\mathbf{co}\, (\bigcup_{m \in M}\, B_m)}$$

Misra [1995b] presents a detailed proof of the elimination theorem, but Isabelle proves it automatically.[3] The automation owes something to `Blast_tac` but more to the formalization itself: the standard UNITY safety laws correspond to elementary statements in set theory.

6. FORMALIZING PROGRESS PROPERTIES

Figure 2 presents part of the Isabelle formalization of the progress primitives **transient**, **ensures** and $\mapsto$. Misra [1995a] notes that **transient** is the only primitive whose definition depends upon the form of fairness adopted. Its definition in Isabelle specifies a policy of weak fairness, which the other constants inherit. Other fairness policies, such as minimal progress or strong fairness, might be provided in the future; one possibility is to supply the fairness policy to each progress primitive as an argument.

Fairness applies to all actions. To deny fairness to an action, simply form its union with the identity relation, allowing the action to do nothing. This is a further advantage of allowing nondeterministic actions, an issue discussed at the beginning of §5.

A program satisfies **transient**(A) provided whenever the state belongs to A eventually an atomic action will take it out of A. Under weak fairness, it suffices that some command is guaranteed to take any state in A to a state outside that set. The Isabelle definition of **transient** expresses those conditions using the operations of domain, image and set complement. The conjunct $A \subseteq \mathbf{dom}\, act$ makes explicit

`Isabelle/library/HOL/UNITY/UNITY.html`

[3] M corresponds to p in Misra's paper, a predicate that concerns only x, while B_m corresponds to q.

```
  transient :: "'a set => 'a program set"
    "transient A ==
        {F. ∃act∈Acts F. A ⊆ Domain act & act`A ⊆ -A}"

  ensures :: "['a set, 'a set] => 'a program set"       (infixl 60)
     "A ensures B == (A-B co A∪B) ∩ transient (A-B)"

  leadsTo :: "['a set, 'a set] => 'a program set"       (infixl 60)
    "A leadsTo B == {F. (A,B) ∈ leads F}"

  wlt :: "['a program, 'a set] => 'a set"
    "wlt F B == Union {A. F ∈ A leadsTo B}"


(*relation symbol for the inductive definition*)
  leads :: "'a program => ('a set * 'a set) set"

inductive "leads F"
 intrs

  Basis  "F ∈ A ensures B ==> (A,B) ∈ leads F"

  Trans  "[| (A,B) ∈ leads F;  (B,C) ∈ leads F |]
          ==> (A,C) ∈ leads F"

  Union  "{(A,B) | A. A∈S} ∈ Pow(leads F) ==> (Union S, B) ∈ leads F"

 monos Pow_mono
```

Fig. 2. Declaring UNITY's Progress Primitives in Isabelle

that the command is enabled over the whole of A, and is required since otherwise the empty relation would trivially satisfy **transient** A.

$$\mathbf{transient}\, A \triangleq \{F \mid \exists_{act \in \mathbf{Acts}\, F} (A \subseteq \mathbf{dom}\, act \wedge act``A \subseteq \overline{A})\} \tag{3}$$

A program satisfies A **ensures** B provided whenever the state belongs to A eventually an atomic action will take it into B. Misra's definition is straightforwardly expressed with set operators. Note that the **co**-assertion is just A **unless** B.

$$A\, \mathbf{ensures}\, B \triangleq ((A \setminus B)\, \mathbf{co}\, (A \cup B)) \cap \mathbf{transient}(A \setminus B)$$

A program satisfies $A \mapsto B$ provided whenever the state belongs to A execution will eventually take it into B. The leads-to relation is defined inductively to be the closure of the **ensures** relation under transitivity and arbitrary unions. (For technical reasons, the Isabelle declaration requires the constant `leads` and use of the powerset operator, as can be seen in Fig. 2.)

Progress properties are harder to establish than safety properties. Although `Blast_tac` remains invaluable, many of the proofs require induction over the definition of leads-to. In a few cases it was essential to follow the proofs of Misra [1995a]. The Isabelle theory for leads-to has some sixty theorems, proved with an average of under three tactic calls each. Most of them are properties of leads-to,

sometimes proved in several variant forms. Let us consider four of these theorems.[4]

The first is a version of the *cancellation law*. Proofs involving these laws can be hard to mechanize, since first the goal must be massaged until it contains a union of the right form.

$$\frac{F \in A \mapsto (A' \cup B) \qquad F \in (B \setminus A') \mapsto B'}{F \in A \mapsto (A' \cup B')}$$

The second example is *PSP* (progress-safety-progress). The six-step Isabelle proof uses the cancellation law and requires, as a lemma, the analogous property for the **ensures** relation. Compared with Misra [1995a], the following version has a slightly stronger conclusion.

$$\frac{F \in A \mapsto A' \qquad F \in B \,\mathbf{co}\, B'}{F \in (A \cap B') \mapsto ((A' \cap B) \cup (B' \setminus B))}$$

This rule may require an explanation. Given that $A \mapsto A'$, we consider when an execution first reaches a state outside B. It might stay in B persistently, so that we eventually find ourselves in $(A' \cap B)$. If we leave B, then given $B \,\mathbf{co}\, B'$ the state we next reach must belong to B', or $B' \setminus B$ to be precise. The start state might already belong to $B' \setminus B$; note that execution starts in B'.

The third example is *leads-to induction*. This rule lets us conclude $A \mapsto B$ from a complicated premise involving a measure: if A holds then execution will reach a state in which either A remains true and the measure decreases or else B holds. Given that the measure cannot decrease for ever, we conclude that A leads to B. The rule, which should not be confused with induction over the definition of the leads-to relation, takes an unusual form in set notation. Let r be a well-founded relation over type α and f a function of type $\beta \Rightarrow \alpha$; the combination of r and f constitute a measure on states (of type β). The quantified premise expresses that the program, if started in A, either remains in A while decreasing its measure or else reaches a state in B. Here $f^{-1}\text{“}M$ is the inverse image of M under f: the set of all states that f maps into M. The Isabelle derivation of this rule comprises eleven steps, including a lemma.

$$\frac{\forall m.\, F \in (A \cap f^{-1}\text{“}\{m\}) \mapsto ((A \cap f^{-1}\text{“}(r^{-1}\text{“}\{m\})) \cup B)}{F : A \mapsto B}$$

In practice, this rule is less formidable than it looks. Note that $f^{-1}\text{“}\{m\}$ is just $\{s.\, fs = m\}$, and the inverse image often disappears after simplification.

The *completion theorem*, our fourth example, is a major result about progress for conjunctions of properties. As a special case, if $A_1 \mapsto B_1$ and $A_2 \mapsto B_2$, where B_1 and B_2 are stable properties, then $A_1 \cap A_2 \mapsto B_1 \cap B_2$. Intuitively, if the program starts in a state belonging to both A_1 and A_2 then makes progress towards B_1 and B_2, until eventually both hold. (In particular, if B_1 and B_2 are disjoint, then so are A_1 and A_2.) We can obviously extend this theorem to an arbitrary finite index

[4]The full list can be viewed on the Internet at URL `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/library/HOL/UNITY/WFair.html`

set I (this version is needed in an example later):

$$\frac{\forall_{i\in I}\, F \in (A_i \mapsto B_i) \qquad \forall_{i\in I}\, F \in \mathbf{stable}(B_i)}{F \in (\bigcap_{i\in I}\, A_i) \mapsto (\bigcap_{i\in I}\, B_i)} \tag{4}$$

The completion theorem appears obvious, but I know of no simple proof. Misra [1995a] establishes the theorem by first defining **wlt**, the *weakest leads-to* predicate transformer: if B is a set of states then $\mathbf{wlt}\ B$ is the largest set A such that $A \mapsto B$. Misra's proof makes use of many lemmas about **wlt**, such as his property W5:

$$F \in (\mathbf{wlt}\, F B \setminus B)\ \mathbf{co}\ (\mathbf{wlt}\, F B)$$

Given the theory of **wlt** we can finally prove the completion theorem in its general form.

$$\frac{F \in A \mapsto A' \cup C \quad F \in A'\ \mathbf{co}\ A' \cup C \quad F \in B \mapsto B' \cup C \quad F \in B'\ \mathbf{co}\ B' \cup C}{F \in A \cap B \mapsto (A' \cap B') \cup C}$$

The special case mentioned above follows by setting $C = \emptyset$.

7. WEAK SPECIFICATION PRIMITIVES AND THE SUBSTITUTION AXIOM

The substitution axiom, which was part of the original UNITY logic [Chandy and Misra 1988], asserts that an invariant can be replaced by *true* in any program property. The Isabelle formalization is definitional rather than axiomatic, so rather than assuming the substitution axiom, we must define specification primitives for which substitution is valid.

The definitions of **co** and $\mapsto$ given above do not satisfy the substitution axiom. For example, consider the program whose initial condition is $x = 0$ and that has only a **skip** command. The program trivially satisfies $x > 0\,\mathbf{co}\, x > 0$ and $x > 0 \mapsto x > 0$, and obviously $x = 0$ is an invariant. Substituting with $x = 0$ in the postconditions yields the invalid assertions $x > 0\ \mathbf{co}\ \mathit{false}$ and $x > 0 \mapsto \mathit{false}$.

To make versions of **co** and $\mapsto$ that satisfy the substitution axiom, it suffices to intersect the precondition with the set of reachable states, an observation first made by Sanders [1991]. Let $\mathcal{R}(F)$ stand for the set of states reachable in program F. This set is defined inductively to Isabelle:

```
consts reachable :: "'a program => 'a set"
inductive "reachable F"
  intrs
    Init  "s ∈ Init F ==> s ∈ reachable F"

    Acts  "[| act ∈ Acts F;  s ∈ reachable F;  (s,s') ∈ act |]
           ==> s' ∈ reachable F"
```

An elementary induction over this definition shows that $\mathcal{R}(F)$ is the strongest invariant of the program F. With the help of $\mathcal{R}(F)$, we define new safety and progress operators:

$$F \in A\ \mathbf{co}_w\ B \iff F \in (\mathcal{R}(F) \cap A)\ \mathbf{co}\ B$$
$$F \in A \mapsto_w B \iff F \in (\mathcal{R}(F) \cap A) \mapsto B$$

These are called the *weak* specification operators because by precondition strengthening

$$F \in A \,\mathbf{co}\, B \text{ implies } F \in A \,\mathbf{co}_w\, B$$
$$\text{and} \qquad F \in A \mapsto B \text{ implies } F \in A \mapsto_w B.$$

The Isabelle theory defines $\mathbf{always}(A)$ to satisfy

$$F \in \mathbf{always}(A) \iff \mathbf{Init}\, F \subseteq A \wedge F \in \mathbf{stable}_w(A),$$
$$\text{where} \qquad \mathbf{stable}_w(A) \triangleq A \,\mathbf{co}_w\, A.$$

This definition resembles that of $\mathbf{invariant}(A)$ but uses the weak form of stable: $\mathbf{always}(A)$ is the weak form of $\mathbf{invariant}(A)$. A program belongs to $\mathbf{always}(A)$ if all its reachable states belong to A. One way of proving $F \in \mathbf{always}(A)$ is to show that $I \subseteq A$ where I is an invariant of F. Recall from law (2) that a program belongs to $\mathbf{invariant}(A)$ provided A includes its initial states and all of its commands preserve A.

Both the weak and strong specification operators are necessary. The strong forms have simpler definitions and are easier to reason about. Theorems such as PSP and completion are proved first about the strong version of $\mapsto$ and then transported to the weak version. Statically-checked properties of a program, such as that it does not update a certain variable, are best expressed using the strong operators: they hold regardless of what states are reached at run time.

But for general reasoning about programs, it seems natural that specifications should only be concerned with reachable states, so we also need the weak operators. The main additional properties they satisfy are closely related to the substitution axiom. If I is a weak invariant (an **always** property) then we may assume it to hold in the precondition:

$$\frac{F \in \mathbf{always}\, I}{F \in (I \cap A) \,\mathbf{co}_w\, A' \iff F \in A \,\mathbf{co}_w\, A'}$$

Dually, we may conclude that I holds in the postcondition:

$$\frac{F \in \mathbf{always}\, I}{F \in A \,\mathbf{co}_w\, (I \cap A') \iff F \in A \,\mathbf{co}_w\, A'}$$

Similar rules hold for $\mapsto_w$ and the other weak specification operators.

8. EXAMPLE: TWO-PROCESS MUTUAL EXCLUSION

In one of the "Notes on UNITY," Misra [1990] discusses Peterson's algorithm for mutual exclusion. He shows how it can be derived from a high-level algorithm, replacing a shared queue by three shared booleans during the refinement. Then, he modifies the refinement to obtain a new program. Figure 3 presents the final UNITY program. The integer variables m and n are program counters for processes u and v, respectively.[5] Misra proves safety and progress properties for this program. His

[5] I simplify the program by removing the predicates $u.h$ and $v.h$, which determine when u and v complete their noncritical sections, since their formal semantics is unclear. Fairness alone will cause these events to occur.

initially $u, v, m, n = \mathit{false}, \mathit{false}, 0, 0$

$$
\begin{array}{lll}
\{\text{program for } u\} & & \\
u, m := \mathit{true}, 1 & \text{if} & m = 0 \\
[\!] \; p, m := v, 2 & \text{if} & m = 1 \\
[\!] \; m := 3 & \text{if} & \neg p \wedge m = 2 \\
[\!] \; u, m := \mathit{false}, 4 & \text{if} & m = 3 \qquad (*u\text{'s critical section}*) \\
[\!] \; p, m := \mathit{true}, 0 & \text{if} & m = 4 \\
[\!] & & \\
\{\text{program for } v\} & & \\
v, n := \mathit{true}, 1 & \text{if} & n = 0 \\
[\!] \; p, n := \neg u, 2 & \text{if} & n = 1 \\
[\!] \; n := 3 & \text{if} & p \wedge n = 2 \\
[\!] \; v, n := \mathit{false}, 4 & \text{if} & n = 3 \qquad (*v\text{'s critical section}*) \\
[\!] \; p, n := \mathit{false}, 0 & \text{if} & n = 4
\end{array}
$$

Fig. 3. The Mutual Exclusion Program in UNITY

invariants are flawed, but fortunately the proofs are still basically correct [Dappert-Farquhar 1990]. Misra's error makes our example more realistic: when the property being "verified" is false, proof tools should help us find out what is wrong. The failed Isabelle proof describes a state that violates the faulty invariant.

This program can be analyzed trivially using a model checker because its state space is finite. The example is nonetheless instructive, and the mechanical proofs can be compared with the UNITY discussion documents mentioned above. With its ten actions, the mutual exclusion program is more complicated than many published UNITY programs, which often consist of a single parametric assignment. The proof procedures demonstrated below work by logical inference, not by state enumeration, and they are equally effective against infinite-state problems.

8.1 Formalization

Figure 4 presents the Isabelle version of the program. The type *state* defines program states to be records whose components are the program variables p, m, n, u and v. Here is a summary of the record notation:

—To inspect a record field, use function application. For example, $m(s)$ returns the value of field m of record s.

—To update a record field, use the $(|\ldots|)$ notation. For example, $s(|m := 1|)$ is the record obtained from s by changing the value of field m to be 1. Several fields can be changed at the same time, as in $s(|m := 1, n := 1|)$.

The constants $U0$–$U4$ denote commands 0–4, respectively, for process u, while $V0$–$V4$ have the analogous meaning for v. The constant *Mutex* denotes the program with its initial state ($u = v = \mathit{false}$, $m = n = 0$) and commands 0–4 for both

```
record state =
  p :: bool
  m :: int
  n :: int
  u :: bool
  v :: bool


(** The program for process U **)

  "U0 == {(s,s'). s' = s (|u:=True, m:=#1|) & m s = #0}"

  "U1 == {(s,s'). s' = s (|p:= v s, m:=#2|) & m s = #1}"

  "U2 == {(s,s'). s' = s (|m:=#3|) & ~ p s & m s = #2}"

  "U3 == {(s,s'). s' = s (|u:=False, m:=#4|) & m s = #3}"

  "U4 == {(s,s'). s' = s (|p:=True, m:=#0|) & m s = #4}"

(** The program for process V **)

  "V0 == {(s,s'). s' = s (|v:=True, n:=#1|) & n s = #0}"

  "V1 == {(s,s'). s' = s (|p:= ~ u s, n:=#2|) & n s = #1}"

  "V2 == {(s,s'). s' = s (|n:=#3|) & p s & n s = #2}"

  "V3 == {(s,s'). s' = s (|v:=False, n:=#4|) & n s = #3}"

  "V4 == {(s,s'). s' = s (|p:=False, n:=#0|) & n s = #4}"

  "Mutex == mk_program ({s. ~ u s & ~ v s & m s = #0 & n s = #0},
                        {U0, U1, U2, U3, U4, V0, V1, V2, V3, V4})"
```

Fig. 4. The Mutual Exclusion Program in Isabelle

```
  "IU == {s. (u s = (#1 ≤ m s & m s ≤ #3)) &
             (m s = #3 --> ~ p s)}"

  "IV == {s. (v s = (#1 ≤ n s & n s ≤ #3)) &
             (n s = #3 --> p s)}"

  "bad_IU == {s. (u s = (#1 ≤ m s & m s ≤ #3)) &
                 (#3 ≤ m s & m s ≤ #4 --> ~ p s)}"
```

Fig. 5. Good and Bad Invariants

processes. The function **mk_program** implicitly adds a **skip** command, required in all UNITY programs. (The figure omits the types of constants; recall that commands are binary relations on states and that program properties are sets of states.)

As a notation for commands, set theory leaves much to be desired. The definition of u's first command is

$$U0 \triangleq \{(s, s') \mid s' = s(|u := \mathit{true}, m := 1|) \wedge m(s) = 0\}.$$

Here s is the initial state, s' is the final state and $\ldots \wedge m(s) = 0$ formalizes the command's guard. The explicit references to states hinder readability. We can even express impossible commands, which is not necessarily a drawback: sometimes we may wish to describe a command independently from any possible implementation.

Figure 5 presents the Isabelle specifications of the invariants. The constants IU and IV are the correct invariants for processes u and v respectively:

$$\begin{aligned} IU &\iff (u \equiv 1 \leq m \leq 3) \wedge (m = 3 \rightarrow \neg p) \\ IV &\iff (v \equiv 1 \leq n \leq 3) \wedge (n = 3 \rightarrow p) \end{aligned}$$

The constant bad_IU is Misra's original, flawed invariant for u:

$$bad\_IU \iff (u \equiv 1 \leq m \leq 3) \wedge (3 \leq m \leq 4 \rightarrow \neg p)$$

8.2 Safety

The safety property for this program is mutual exclusion. The critical section of each process is command 3, so we must show that the program can never reach a state in which $m = n = 3$. The Isabelle proof is simple. Both invariants are proved with three tactic calls each; from their conjunction, the proof that no reachable states satisfy $m = n = 3$ is automatic.

To illustrate the proof style for simple safety proofs, here is the complete proof script for u's invariant. Using the weak specification form, we enter the goal $\mathit{Mutex} \in \mathbf{always}\, IU$. Isabelle's response is shown in a *slanted* typeface.

```
Goal "Mutex ∈ Always IU";
  Level 0 (1 subgoal)
  Mutex ∈ Always IU
  1. Mutex ∈ Always IU
```

The first step is to refine by the rule `AlwaysI`, which reduces the goal to showing that IU includes the initial states and is (weakly) stable:

```
by (rtac AlwaysI 1);
  Level 1 (2 subgoals)
  Mutex ∈ Always IU
  1. Init Mutex ⊆ IU
  2. Mutex ∈ Stable IU
```

The tactic `constrains_tac` attempts to prove safety properties. It unfolds the program definition in a controlled manner, performing a case split over the different commands, and simplifies the resulting subgoals. Here, as often, it proves the subgoal automatically:

```
by (constrains_tac 2);
  Level 2 (1 subgoal)
  Mutex ∈ Always IU
  1. Init Mutex ⊆ IU
```

The remaining subgoal, concerning the initial state, is proved automatically:

```
by Auto_tac;
  Level 3 (0 subgoal)
  Mutex ∈ Always IU
  No subgoals!
```

The invariant for process v, namely $Mutex \in \mathbf{always}\, IV$, is proved in the same way. Mutual exclusion is expressed as $Mutex \in \mathbf{always}\{s \mid \neg(ms = 3 \wedge ns = 3)\}$. Its proof consists of three further steps: invocation of a weakening rule, combining the previous two invariants and calling `Auto_tac`.

Isabelle identifies the command that falsifies the original, flawed invariant. Applying the three tactics shown above leaves an unproved subgoal in which $n = 1$, $p = \mathit{false}$, $m = 4$ and $u = \mathit{false}$. Recall that n is a program counter. Looking at the program for v, we find that if $n = 1$ then it can execute $p, n := \neg u, 2$. This assignment sets p to *true* and violates the assertion's second conjunction, namely $(3 \leq m \leq 4 \rightarrow \neg p)$.

8.3 Progress

The key progress properties are that each process can reach its critical section: $m = 1 \mapsto m = 3$ and $n = 1 \mapsto n = 3$. The proofs follow Misra's note precisely. For each process there are eight lemmas, which are proved with fifteen tactic calls. Combining them to reach the main result requires several further calls.

For an impression of how progress properties are tackled, here is part of a proof of $m = 3 \mapsto p = \mathit{true}$:

```
Goal "Mutex ∈ {s. m s = #3} LeadsTo {s. p s}";
  Level 0 (1 subgoal)
  Mutex ∈ {s. m s = #3} LeadsTo {s. p s}
  1. Mutex ∈ {s. m s = #3} LeadsTo {s. p s}
```

The first step is by transitivity of $\mapsto$, giving $m = 4$ as an intermediate assertion:

```
by (res_inst_tac [("B", "{s. m s = #4}")] LeadsTo_Trans 1);
  Level 1 (2 subgoals)
  Mutex ∈ {s. m s = #3} LeadsTo {s. p s}
  1. Mutex ∈ {s. m s = #3} LeadsTo {s. m s = #4}
  2. Mutex ∈ {s. m s = #4} LeadsTo {s. p s}
```

The two new subgoals both follow by the corresponding **ensures** property. (In other words, they will eventually be established by an atomic action.) Command $U3$ takes a state satisfying $m = 3$ to one in which $m = 4$ and command $U4$ takes a state satisfying $m = 4$ to one in which $p = \mathit{true}$. The UNITY package defines a tactic `ensures_tac` to prove **ensures** properties. As discussed in §6 above, $F \in A\ \mathbf{ensures}\ B$ provided $F \in (A \setminus B)\ \mathbf{co}\ (A \cup B)$ and $F \in \mathbf{transient}(A \setminus B)$. The tactic uses `constrains_tac` to tackle the safety property and, given the relevant command, attempts also to prove the **transient** property. Any unproved subgoals are given to the user, but typically everything is proved automatically. To complete

our proof of $m = 3 \mapsto p = \mathit{true}$ requires invoking `ensures_tac` twice, with $U3$ and $U4$ as arguments.

8.4 Other Examples

The lift example from Andersen, Petersen, and Pettersson [1994a] has become the standard benchmark for UNITY implementations. It has also been done using Isabelle. The system consists of a lift and an arbitrary number of floors, with a button for each floor. Among the safety properties are that if the door is open then the lift is stopped; also, the lift car must stay within the available floors. The key progress property is that all requests to visit a floor are eventually met. Progress is hard to prove because the lift may have to go further away from a requested floor; the formal measure of distance involves many case splits.

The example is too big to present in detail, but some statistics may be illuminating. The Isabelle mechanization consists of around thirty theorems proved using three tactics per theorem; it runs in well under three minutes.

The lift example has a serious problem: the model omits the passengers who make floor requests. A trivial invariant is that no buttons are ever pressed; the lift never moves. Andersen informs me that the progress proofs consider all possible combinations of floor requests. However, extending the program with a "request" action falsifies several progress theorems expressed using the **ensures** primitive. Recall that an **ensures** property is established by an atomic operation. Many actions of the lift depend on the set of outstanding requests, that is, on which buttons have been pushed. In a given state, if a new request can influence the lift's behaviour then progress should not be expressed in terms of **ensures**. The most we can say about the lift is that it will service any static set of floor requests. Considering dynamic requests will require new proofs, which could be a significant exercise.

Many minor examples have been mechanized, such as common meeting time, deadlock and the token ring [Misra 1995a]. Another small example is reachability in directed graphs [Chandy and Misra 1988, p. 121]. The task is to flag all the vertices that are reachable from a given initial vertex in a finite graph. Verifying the program involves showing that all executions terminate and that if the program terminates then the task has been accomplished. (A UNITY program is considered to have terminated if it has reached a fixed point: no action can change the state.) Progress is proved by leads-to induction, using as a metric the number of unflagged vertices. The reachability example involves 14 theorems proved with 46 tactic calls, and it runs in eight seconds. Mechanizing this example took me four days.

UNITY can also be applied to the verification of security protocols. The inductive method [Paulson 1998] maps into UNITY straightforwardly, using the same theory of messages and trace model. The inductive definition is replaced by a UNITY program, replacing the rules by actions. (Thus, the program has one action for each message round and a further action to model the adversary.) This approach opens new possibilities and requires careful thought. Protocol specialists usually assume that messages can be blocked, but with weak fairness, every protocol step must eventually be executed. The disagreement makes little real difference: fairness allows an arbitrary finite delay, and delaying the honest agents gives the adversary time to mount an attack. Protocol analyses normally consider safety only; UNITY

```
JOIN  :: ['a set, 'a => 'b program] => 'b program
  "JOIN I F == mk_program (⋂i∈I. Init (F i), ⋃i∈I. Acts (F i))"

Join :: ['a program, 'a program] => 'a program
        (infixl 65)
  "F Join G == mk_program (Init F ∩ Init G, Acts F ∪ Acts G)"

SKIP :: 'a program
  "SKIP == mk_program (UNIV, {})"
```

Fig. 6. Some Program Composition Operators in Isabelle

gives us a theory of progress, which could become useful if protocols arise that have a non-trivial control structure.

One thing most of these examples share is a high degree of automation. Most safety and **ensures** properties are proved easily. Occasionally a theorem has a lengthy proof, as when progress lemmas are combined using transitivity, union and PSP laws to derive a leads-to property. The average theorem is proved using three or four tactic calls.

9. UNIONS OF PROGRAMS

Systems are built from components. We should prove the properties of a system from those of its components; reasoning about the system as one huge program is unfeasible. Many issues are involved here. For a start, we must formalize program composition itself. How to reason about composed systems is a separate matter.

9.1 Formalizing Program Union

The Isabelle mechanization follows Misra [1994]. The composition of programs F and G, written $F \sqcup G$, is the program whose set of actions is the union of those of F and G and whose set of initial states is the intersection of those of F and G. (Other authors use other symbols; because composition is the join of a lattice, I prefer $\sqcup$.) The composition for $i \in I$ of F_i is written $\bigsqcup_{i\in I} F_i$ and represents a program built from a family of similar components. We are seldom interested in infinite programs; typically I is a finite, non-empty set.

Composition can go wrong in two ways. (1) The set of initial states could become empty; then the program cannot be executed. (2) The expression $F \sqcup G$ is well-typed only if F and G have the same type of states; in the resulting program, all variables are available to both components. Variable naming and renaming leads to complex issues that are discussed in §10 below.

Figure 6 shows the Isabelle declarations of the main primitives. The constant `JOIN` formalizes $\bigsqcup_{i\in I} F_i$ while the infixed constant `Join` formalizes $F \sqcup G$. Finally, `SKIP` denotes the null program, which has no commands other than **skip** and allows any initial state. (`UNIV` is a polymorphic constant denoting the universal set.) The null program is written as $\bot$ below because it is the bottom element of the lattice of programs. Note that $\bot$ is polymorphic: it can denote the null program for any state type.

9.2 Basic Properties of Unions

The first results proved are mostly trivial. There are equations to specify the initial states and commands of composite programs:

$$\mathbf{Init}\Big(\bigsqcup_{i\in I} F_i\Big) = \bigcap_{i\in I} \mathbf{Init}\, F_i \tag{5}$$

$$\mathbf{Acts}\Big(\bigsqcup_{i\in I} F_i\Big) = \{Id\} \cup \bigcup_{i\in I} \mathbf{Acts}\, F_i \tag{6}$$

The inclusion of $\{Id\}$ on the right-hand side of equation (6) is necessary in case the index set, I, is empty.

The lattice properties of the operators are proved. For example, $\sqcup$ has $\bot$ for its identity element; it is associative and commutative and satisfies $F \sqcup F = F$. Laws governing the join of a family of programs include the following:

$$\bigsqcup_{i\in\emptyset} F_i = \bot$$

$$F_k \sqcup \Big(\bigsqcup_{i\in I} F_i\Big) = \bigsqcup_{i\in I} F_i \qquad (\text{if } k \in I)$$

$$\bigsqcup_{i\in I\cup J} F_i = \Big(\bigsqcup_{i\in I} F_i\Big) \sqcup \Big(\bigsqcup_{i\in J} F_i\Big)$$

$$\bigsqcup_{i\in I} (F_i \sqcup G_i) = \Big(\bigsqcup_{i\in I} F_i\Big) \sqcup \Big(\bigsqcup_{i\in I} G_i\Big)$$

Such laws are not required in typical published examples, but they give the theory a convenient algebraic structure. They assume extensional equality: two UNITY programs that have the same initial states and the same sets of commands are equal (by definition).

9.3 Strong Safety Properties

The next group of results cover what Misra [1994] calls the Union Theorem and its corollaries. These include theorems for the inheritance of strong safety properties. If $F \in A\ \mathbf{co}\ B$ and $G \in A\ \mathbf{co}\ B$ then $F \sqcup G \in A\ \mathbf{co}\ B$, since an action of $F \sqcup G$ is either an action of F or an action of G and therefore takes the precondition A to the postcondition B. The converse direction holds too: we have the equivalence

$$F \sqcup G \in A\ \mathbf{co}\ B \iff (F \in A\ \mathbf{co}\ B) \wedge (G \in A\ \mathbf{co}\ B) \tag{7}$$

and the version for indexed families ($I \neq \emptyset$),

$$\Big(\bigsqcup_{i\in I} F_i\Big) \in A\ \mathbf{co}\ B \iff \forall_{i\in I}\, (F_i \in A\ \mathbf{co}\ B). \tag{8}$$

Among the other safety properties is the inheritance of strong invariants:

$$\frac{F \in \mathbf{invariant}\, A \qquad G \in \mathbf{invariant}\, A}{F \sqcup G \in \mathbf{invariant}\, A}$$

9.4 Weak Safety Properties

Weak safety properties do not have the same rules for inheritance. For a counterexample, let programs F and G have among their variables the booleans p and q. The commands of F include the assignment $q := \mathit{true}$; all other commands are guarded by p. Similarly, G has $p := \mathit{true}$ and its other commands are guarded by q. Each program has p and q initially false; in isolation neither can reach a state in which its other commands are enabled. So if A is any program property not involving p and q then A will be weakly stable, giving $F \in A\,\mathbf{co}_w\,A$ and $G \in A\,\mathbf{co}_w\,A$. Once F and G are put together, each can set the other program's guard to *true*, enabling their commands, and (for most A) causing $F \sqcup G \in A\,\mathbf{co}_w\,A$ to fail.

Misra [1994, §5.4.1] points out that when applying the substitution axiom to $F \sqcup G$, the invariant used must hold for the program as a whole rather than only for one component. The guarantees primitive of Chandy and Sanders [1998] also makes weak specifications of program units refer to the reachable states of the program as a whole.

The root of the problem is there is no obvious relation between states reachable in $F \sqcup G$ and those reachable in F and G. There is no simple expression for $\mathcal{R}(F \sqcup G)$ in terms of $\mathcal{R}(F)$ and $\mathcal{R}(G)$. The initial condition of $F \sqcup G$ is the intersection of those for F and G, which could make fewer states reachable, but the actions of $F \sqcup G$ comprise those of F and G, which could make more states reachable.

9.5 Progress Properties

For progress properties, inheritance is a complex matter. Law (7) has no analogue for leads-to. Having $F \in A \mapsto B$ and $G \in A \mapsto B$ does not guarantee $F \sqcup G \in A \mapsto B$ because the two programs might interfere with each other. For example, F might increment the shared variable x while G decrements it. Each program satisfies $\mathit{true} \mapsto |x| > 10$, but $F \sqcup G$ does not: it can alternately increment and decrement x forever.

The most useful lemma proved about progress is an equivalence for transient properties,

$$F \sqcup G \in \mathbf{transient}\,A \iff (F \in \mathbf{transient}\,A) \vee (G \in \mathbf{transient}\,A).$$

and its analogue for indexed families,

$$\Big(\bigsqcup_{i\in I} F_i\Big) \in \mathbf{transient}\,A \iff \exists_{i\in I}\,(F_i \in \mathbf{transient}\,A). \tag{9}$$

I have also proved some of Misra's laws relating safety and progress, such as

$$\frac{F \in \mathbf{stable}(A) \qquad G \in A\,\mathbf{ensures}\,B}{F \sqcup G \in A\,\mathbf{ensures}\,B,}$$

and used them in a small example, the handshake protocol [Misra 1994, §5.3.2]. Section 11 will present a small example of inheritance of a progress property.

10. PROGRAM STATES

As mentioned in §5, the formalization does not specify a particular representation of program states. Various representations exist, each with their own advantages.

```
"extend_set h A == h `` (A×UNIV)"

"extend_act h act == ⋃(s,s')∈act. ⋃y. {(h(s,y), h(s',y))})"

"extend h F == mk_program (extend_set h (Init F),
                           extend_act h `` Acts F)"
```

Fig. 7. Primitives for Changing the State Representation

States can be functions from variable names to values; this natural idea, however, requires giving all variables the same type, or else resorting to dependent types. States can be records with one field for each program variable; this treatment is more complicated but allows mixed types. The states of the system consisting of F and G running in parallel with no shared variables could be ordered pairs of states (s_F, s_G). A similar representation is best for modelling an array of processes.

Sometimes the representation of states has to be changed. Programs F and G can be composed only if their representations of states are identical; all variables are shared by the two components. If F reads data from the variable *in* and G writes data to the variable *out*, then combining the producer and consumer requires identifying the variables, which is most easily done by renaming them both to a common name such as *io*. Also, any local variables of F will have to be present in G's representation of states, and *vice versa*. Ideally, we should be able to prove properties of F using a representation of states that includes only F's variables, adding further variables later if they are required in order to compose F with other programs.

Renaming or adding variables should preserve program properties between the two representations. Marques [1998] has shown that the new representation inherits the properties of the old. He works in the UNITY theory of Heyd and Crégut [1996], where actions are functions; straightforward modifications make his approach work in the Isabelle theory, where actions are relations. His idea is invaluable for reconciling differences in naming between program components.

His work can be strengthened by showing that program properties are preserved in both directions. In other words, the new representation should satisfy precisely the same properties (modulo the renaming) as the old. Equivalences can be expressed as rewrite rules and performed automatically, which allows better automation. However, the equivalence is surprisingly hard to prove for progress properties. The proofs are easy for the atomic progress properties, namely transient and ensures; the difficulty lies in expressing an induction over the definition of leads-to.

A function h of type $\alpha \times \beta \Rightarrow \gamma$ formalizes a change of state representation. Here α is the old type of states, β is a type representing the items (if any) being added to states and γ is the new type of states. The function h is required to be surjective (it can express all target states) as well as injective in its first argument (it faithfully copies the source state). It does not have to be injective in its second argument. The requirements on h are left implicit in the theorems stated below.

Figure 7 presents the Isabelle definitions of the primitives for extending the representation of states. The function **extend_set** uses h to map a set of old states to the corresponding set of new states; **extend_act** is similar, transforming an action

over old states to an action over new states. Here are their definitions in standard mathematical notation:

$$\begin{aligned}\mathbf{extend\_set}\, h\, A &\triangleq \{h(x,y) \mid x \in A\}\\ \mathbf{extend\_act}\, h\, act &\triangleq \{(h(s,y), h(s',y)) \mid (s,s') \in act\}\end{aligned}$$

The function **extend** uses these two functions to transform a program over old states to a program over new states. If the effect of h is to add a variable v to the state, then the new program's initial condition will be independent of v and its actions will preserve the value of v. Defining **extend** as a function is another departure from Marques [1998], who formalizes extending the state space as a relation between programs.

Many facts about **extend** are proved easily. For example, it distributes over composition:

$$\mathbf{extend}\, h \left(\bigsqcup_{i\in I} F_i\right) = \left(\bigsqcup_{i\in I} \mathbf{extend}\, h\, F_i\right)$$

Since I can be empty, a corollary is that $\mathbf{extend}\, h\, \bot = \bot$.

Safety properties are preserved in both directions:

$$\begin{aligned}\mathbf{extend}\, h\, F \in (\mathbf{extend\_set}\, h\, A)\ \mathbf{co}\ (\mathbf{extend\_set}\, h\, B)) &\iff F \in A\ \mathbf{co}\ B\\ \mathbf{extend}\, h\, F \in \mathbf{invariant}(\mathbf{extend\_set}\, h\, A) &\iff F \in \mathbf{invariant}\, A\end{aligned}$$

Proving $\mathcal{R}(\mathbf{extend}\, h\, F) = \mathbf{extend\_set}\, h\, (\mathcal{R}(F))$, which requires two separate inductions over the definition of reachability, yields analogous results for the weak specification operators:

$$\begin{aligned}\mathbf{extend}\, h\, F \in (\mathbf{extend\_set}\, h\, A)\ \mathbf{co}_w\ (\mathbf{extend\_set}\, h\, B) &\iff F \in A\ \mathbf{co}_w\ B\\ \mathbf{extend}\, h\, F \in \mathbf{always}(\mathbf{extend\_set}\, h\, A) &\iff F \in \mathbf{always}\, A\end{aligned}$$

Among the progress properties, it is easy to show that **transient** and **ensures** are preserved:

$$\begin{aligned}\mathbf{extend}\, h\, F \in \mathbf{transient}(\mathbf{extend\_set}\, h\, A) &\iff F \in \mathbf{transient}\, A\\ \mathbf{extend}\, h\, F \in (\mathbf{extend\_set}\, h\, A)\ \mathbf{ensures}\ (\mathbf{extend\_set}\, h\, B) &\iff F \in A\ \mathbf{ensures}\ B\end{aligned}$$

An elementary induction over the definition of leads-to establishes that it is preserved in one direction:

$$\frac{F \in A \mapsto B}{\mathbf{extend}\, h\, F \in (\mathbf{extend\_set}\, h\, A) \mapsto (\mathbf{extend\_set}\, h\, B)} \tag{10}$$

To prove the opposite direction also requires induction. Using

$$(\mathbf{extend\_set}\, h\, A) \mapsto (\mathbf{extend\_set}\, h\, B)$$

in the induction formula will not work; it must be generalized to have the form $A' \mapsto B'$, relating arbitrary sets and expressing the conclusion using an inverse of $\mathbf{extend\_set}\, h$. Since h is injective in its first argument, there is a function f such that $f(h(x,y)) = x$ for all x and y. An obvious inverse of $\mathbf{extend\_set}\, h$ is to take a set's image under f, which has the effect of forgetting the new variables and

projecting down to the original set. Unfortunately, the base case of the induction fails: the projection preserves neither **transient** nor **ensures**.

Here is a counterexample. For A to be transient, some action must include A in its domain and map every element of A out of A; recall definition (3) of §6. Consider the action $i := 3-i$. The singleton set of states $\{i = 1\}$ is transient under that action, as is $\{i = 2\}$. Now extend the state to have another variable, j. The extended action updates i by $i := 3 - i$ and leaves j unchanged. It maps the two-element set of states $\{(i = 1, j = 1), (i = 2, j = 2)\}$ to $\{(i = 2, j = 1), (i = 1, j = 2)\}$ and back again. Since these sets are disjoint, both are transient. Projecting either set down to states over i yields $\{i = 1, i = 2\}$, which is not transient: the assignment maps it to itself.

The solution is not to project from the extended set as a whole, but to regard it as the union of "slices" for fixed values of y:

$$\mathbf{slice}\, A\, y \triangleq \{x \mid h(x, y) \in A\}$$

Now we can easily prove that transient properties are preserved if we project down to a slice on the old state type:

$$\frac{\mathbf{extend}\, h\, F \in \mathbf{transient}\, A}{F \in \mathbf{transient}(\mathbf{slice}\, A\, y)}$$

A complicated argument is necessary to show that ensures properties are similarly preserved. Note that the postcondition is given by a simple projection, $f\text{“}B$; having a weaker postcondition makes the result easier to prove.

$$\frac{\mathbf{extend}\, h\, F \in A\ \mathbf{ensures}\ B}{F \in (\mathbf{slice}\, A\, y)\ \mathbf{ensures}\ (f\text{“}B)}$$

Using the rule above for the base case, we can finally apply induction over the definition of leads-to, obtaining

$$\frac{\mathbf{extend}\, h\, F \in A \mapsto B}{F \in (\mathbf{slice}\, A\, y) \mapsto (f\text{“}B)}$$

Since **slice** is an inverse of **extend_set** h, this result yields the converse of the rule (10): leads-to properties are preserved in both directions. The analogous for weak leads-to is immediate.

11. ARRAYS OF PROCESSES

Charpentier and Chandy [1999] verify an abstract system consisting of a resource allocator and clients communicating over a network. Clients request resources from the allocator, receive them and eventually return them. The specification describes the properties of a single client, but elsewhere the example assumes that there are many clients, each meeting the same specification. Generalizing from this example, we arrive at the notion of a process array.

The **extend** operator, described in the previous section, can express such arrays of processes. Suppose that we know some properties of a program F. Perhaps F has been given as code from which we have proved the properties, or perhaps F has been specified abstractly. If $\{F[i]\}_{i \in I}$ is an array of processes, each executing program F, what properties does the array inherit from its components?

An array of processes can be modelled by a UNITY program containing many copies of F, indexed by the set I. For example, suppose that F has just two variables, v and w. An F-state might be modelled as a record with two suitably-named fields. (By F-states I mean F's state type, not its set of reachable states.) In the array program, a state is a function mapping I to F-states, intuitively an array of records. If instead we wish to model v and w as array variables, a second use of **extend** is needed to perform the renaming.

An array of processes is formalized by first defining the meaning of $F[i]$ in terms of that of F. Then, the array $\{F[i]\}_{i\in I}$ is defined to be the indexed join of its elements, $\bigsqcup_{i\in I} F[i]$. To define $F[i]$ using **extend** requires a suitable state-transforming function h_i. Let s be an F-state and let f be an array state (a function from I to F-states). Then $h_i(s,f)$ forms an $F[i]$-state by taking its ith component to be s and the other components from f. It is easy to check that h_i is surjective and is injective in its first argument.

In Isabelle, we can formalize h_i as a curried function, **lift_map**. The definition utilizes the `:=` notation for function updating, where $f[y := z]$ denotes the function that maps y to z and maps x to $f(x)$ if $x \neq y$.

```
lift_map i (s,f) == f(i := s)
```

The operators **lift_set** and **lift_act**, analogues of **extend_set** and **extend_act**, can be defined as follows:

```
lift_set i A   == {f. f i ∈ A}
lift_act i act == {(f,f'). f(i:= f' i) = f' & (f i, f' i) ∈ act}
```

Note that **lift_set** $i\,A$ is the set of $F[i]$-states whose ith component belongs to A, while **lift_act** $i\,act$ is the $F[i]$-action that transforms the ith component using the supplied F-action while leaving the other components unchanged.

Now **lift_prog** can be defined in the obvious way, giving its initial condition using **lift_set** and its actions using **lift_act**. It is easy to prove that **lift_set** i equals **extend_set** (**lift_map** i). Simply to define **lift_set** in terms of **extend_set** would have been more direct; however, it then would have been natural to ask what the definition simplifies to, so either approach involves proving that **extend_set** (**lift_map** i) $= \{f \mid f\,i \in A\}$. Analogous equations hold for **lift_act** and **lift_prog**, and so their properties follow immediately from those of **extend_act** and **extend**. Here are two examples:

$$\textbf{lift\_prog}\ i\,F \in (\textbf{lift\_set}\ i\,A)\ \textbf{co}\ (\textbf{lift\_set}\ i\,B) \iff F \in A\ \textbf{co}\ B \tag{11}$$

$$\textbf{lift\_prog}\ i\,F \in \textbf{transient}(\textbf{lift\_set}\ i\,A) \iff F \in \textbf{transient}\,A \tag{12}$$

We can do better. If **lift_prog** $i\,F \in$ **transient**(**lift_set** $j\,A$) then $i = j$ provided A is not the empty set (the state predicate *false*), for if $i \neq j$ then actions of **lift_prog** $i\,F$ never affect component j. This result is used to derive rule (15) below.

Arrays of programs are defined by

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

Safety properties of arrays can be proved using the equivalence

$$\{F[i]\}_{i\in I} \in (\mathbf{lift\_set}\ i\ A)\ \mathbf{co}\ (\mathbf{lift\_set}\ i\ B) \iff (F \in A\ \mathbf{co}\ B), \tag{13}$$

which holds for $i \in I$; the component i determines the safety property for the whole array because **lift_prog** j affects only component j, while **lift_set** i lets the other components take on any values. This law is a consequence of equations (8) and (11).

Elementary progress properties of arrays can be proved using the equivalence

$$\{F[i]\}_{i\in I} \in \mathbf{transient}\ A \iff \exists_{i\in I}\,(F \in \mathbf{transient}\ A), \tag{14}$$

which is an immediate consequence of equation (9). We next can derive a rule concerning **ensures** properties:

$$\frac{i \in I \qquad F \in A\ \mathbf{ensures}\ B}{\{F[i]\}_{i\in I} \in (\mathbf{lift\_set}\ i\ A)\ \mathbf{ensures}\ (\mathbf{lift\_set}\ i\ B)} \tag{15}$$

Removing the occurrences of **ensures** from the premise and conclusion yields a rather specialized rule for deducing progress properties:

$$\frac{i \in I \qquad F \in (A \setminus B)\ \mathbf{co}\ (A \cup B) \qquad F \in \mathbf{transient}(A \setminus B)}{\{F[i]\}_{i\in I} \in (\mathbf{lift\_set}\ i\ A) \mapsto (\mathbf{lift\_set}\ i\ B)} \tag{16}$$

These laws reason in terms of existential and universal properties, following Chandy and Sanders [1998]. The program property X is *existential* provided that if F_i satisfies X for some $i \in I$ then $\bigsqcup_{i\in I} F$ satisfies X. It is *universal* provided that if F_i satisfies X for all $i \in I$ then $\bigsqcup_{i\in I} F$ satisfies X. In this terminology, laws (8) and (9) tell us that **co** properties are universal while **transient** properties are existential. Although leads-to properties are neither existential nor universal, they can be derived from combinations of existential and universal properties, using the **ensures** primitive, the PSP law, etc.

A simple example will demonstrate this reasoning style. Let a *timer* be a program whose state consists of a single natural number and whose action decrements this number provided it is positive. Obviously, a timer will eventually reach zero regardless of its starting value, and a finite array of timers will eventually reach a state in which all the timers are zero.

Here is the Isabelle description of the timer:

```
decr  == ⋃ n. {(Suc n, n)}
Timer == mk_program (UNIV, {decr})
```

The statement to be proved takes the following form, where `plam i∈I. F i` is the array construction $\{F[i]\}_{i\in I}$:

```
finite I ==> (plam i∈I. Timer) ∈ UNIV leadsTo {s. ∀ i∈I. s i = 0}
```

Provided the index set is finite, starting the array of timers (`plam i∈I. Timer`) in any initial state (expressed by `UNIV`) will reach a state in which all the timers are zero. Here are some extracts from the 12-step proof.

First, we appeal to the completion theorem (4), which was described at the end of §6. Simplification yields two subgoals:

```
finite I ==> (plam i∈I. Timer) ∈ UNIV leadsTo {s. ∀ i∈I. s i = 0}
 1. !!i. i ∈ I ==> (plam i∈I. Timer) ∈ UNIV leadsTo lift_set i {0}
 2. !!i. i ∈ I ==> Timer ∈ stable {0}
```

The second of these states that once a timer reaches zero, it stays there. Note that it refers to a single timer, not an array. The tactic `constrains_tac` proves it instantly.

The first subgoal states that the array will eventually reduce timer i to zero, where $i \in I$. Here, we invoke leads-to induction, and after a few steps are left with the successor case:

```
finite I ==> (plam i∈I. Timer) ∈ UNIV leadsTo {s. ∀ i∈I. s i = 0}
 1. !!i n.
       i ∈ I
       ==> (plam i∈I. Timer)
           ∈ lift_set i {Suc n} leadsTo lift_set i (lessThan (Suc n))
```

Here **lessThan** k abbreviates $\{j.\, j < k\}$. If timer i holds $n+1$, the array reduces this value. To prove this fact, we appeal to rule (16). After simplification, we are left with two new subgoals:

```
finite I ==> (plam i∈I. Timer) ∈ UNIV leadsTo {s. ∀ i∈I. s i = 0}
 1. !!i n. i ∈ I ==> Timer ∈ {Suc n} co lessThan (Suc (Suc n))
 2. !!i n. i ∈ I ==> Timer ∈ transient {Suc n}
```

The first, which states that a timer value never increases, falls to `constrains_tac`. The second, which states that a timer cannot hold the value $n+1$ forever, has a two-step proof.

This example illustrates how assertions about an array of processes can be proved by reasoning about the elements. We never regard the array as a monolithic program. However, reducing progress properties to combinations of existential and universal properties can be criticised as being unnatural and implementation-dependent. Chandy and Sanders [1998] also propose a **guarantees** operator that allows compositional reasoning about program properties in a more general way. I have formalized this operator and proved its basic properties, but using it in combination with **extend** and **lift_prog** requires much additional theory that will have to wait for another paper.

12. CONCLUSIONS

UNITY was designed for hand proofs. Making sense of its many notational conventions within the rigid discipline of a mechanical proof tool is a challenge. Like most program verification formalisms, UNITY leaves program states implicit and makes little distinction between program variables and variables of the logic. This convention makes assertions concise and readable, but causes other notational problems. Program variables are not variables at all, but constants denoting locations in the state. Leaving states implicit means that formulas often denote state predicates rather than truth values. Implication might stand for three different concepts:

(1) Ordinary logical implication, yielding a truth value.

(2) Implication lifted over states, yielding a state predicate.

(3) Implication between state predicates, yielding a truth value.

The scope for confusion is endless. Even if the human reader can work out what is intended, heavily overloaded syntax makes parsers struggle.

The solution to these notational problems, as I have argued in this paper, lies in the language of set theory. Ordinary implication (item 1 above) is written $\phi \to \psi$, as usual. Lifted implication (item 2) is expressed using complement and union, $\overline{A} \cup B$. Implication between state predicates (item 3) is expressed using inclusion, $A \subseteq B$. An alternative solution, used by Chandy and Sanders [1998], is to adopt Dijkstra's formalism. The *everywhere* operator provides a means of distinguishing state predicates from relations between state predicates. Unfortunately, everywhere is ambiguous: it denotes universal quantification over the domain of interest, which must be understood from the context. It is familiar to a minority, while elementary set theory is covered in most undergraduate discrete mathematics courses.

Much work remains to be done with the Isabelle/UNITY development. I have started to mechanize a case study: the resource allocation system of Charpentier and Chandy [1999].

ACKNOWLEDGMENTS

J. Misra commented extensively on a draft of this paper. J. Margetson, M. Norrish, B. Sanders, M. Staples and the referees also made suggestions. F. Andersen, M. Chandy and M. Charpentier gave help and advice by electronic mail.

REFERENCES

Andersen, F., Petersen, K. D., and Pettersson, J. S. 1994b. A graphical tool for proving UNITY progress. In T. F. Melham and J. Camilleri Eds., *Higher Order Logic Theorem Proving and Its Applications*, LNCS 859 (1994), pp. 17–3. Springer.

Andersen, F., Petersen, K. D., and Pettersson, J. S. 1994a. Program verification using HOL-UNITY. In J. Joyce and C. Seger Eds., *Higher Order Logic Theorem Proving and Its Applications: HUG '93*, LNCS 780 (1994), pp. 1–15. Springer.

Chandy, K. M. and Misra, J. 1988. *Parallel Program Design: A Foundation*. Addison-Wesley.

Chandy, K. M. and Sanders, B. A. 1998. Reasoning about program composition. preprint.

Charpentier, M. and Chandy, K. M. 1999. Towards a compositional approach to the design and verification of distributed systems. preprint.

Dappert-Farquhar, A. 1990. A correction on "a family of 2-process mutual exclusion algorithms". `ftp://ftp.cs.utexas.edu/pub/psp/unity/notes/22-90.ps.Z`. Notes on UNITY: 22-90.

Gordon, M. J. C. and Melham, T. F. 1993. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press.

Heyd, B. and Crégut, P. 1996. A modular coding of unity in coq. In J. von Wright, J. Grundy, and J. Harrison Eds., *Theorem Proving in Higher Order Logics: TPHOLs '96*, LNCS 1125 (1996), pp. 251–266.

Kaltenbach, M. 1996. *Interactive Verification Exploiting Program Design Knowledge: A Model-Checker for UNITY*. Ph. D. thesis, University of Texas at Austin.

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

Marques, F. 1998. Program composition in COQ-UNITY. In J. Grundy and M. Newey Eds., *Theorem Proving in Higher Order Logics: Emerging Trends. Supplementary proceedings, TPHOLs '98* (1998), pp. 95–104. Technical report 98-08, Department of Computer Science, Australian National University.

Misra, J. 1990. A family of 2-process mutual exclusion algorithms. `ftp://ftp.cs.utexas.edu/pub/psp/unity/notes/13-90.ps.Z`. Notes on UNITY: 13-90.

Misra, J. 1994. Asynchronous compositions of programs. At URL `ftp://ftp.cs.utexas.edu/pub/psp/unity/new_unity/composition.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. 1998. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security 6*, 85–128.

Paulson, L. C. 1999. A generic tableau prover and its integration with Isabelle. *Journal of Universal Computer Science 5*, 3.

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.

Thirioux, X. 1998. Automatically proving Unity safety properties with arrays and quantifiers. In J. Rolim Ed., *Parallel and Distributed Processing*, LNCS 1388 (1998).

Vos, T. E. J. 1999. *UNITY in Diversity, a Stratified Approach to the Verification of Distributed Algorithms*. Ph. D. thesis, Utrecht University.