Mechanization of Substitution Rule and Compositionality of UNITY in HOL

I.S.W.B. Prasetya

UU-CS-1994-19
May 1994

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Mechanization of Substitution Rule and Compositionality of UNITY in HOL

I.S.W.B. Prasetya

Technical Report UU-CS-1994-19
May 1994

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924–3275

Mechanization of Substitution Rule and Compositionality of UNITY in HOL

I.S.W.B. Prasetya
Rijksuniversiteit Utrecht, Vakgroep Informatica
Postbus 80.089, 3508 TB Utrecht, Nederland
Email: wishnu@cs.ruu.nl

Abstract

UNITY is a programming logic designed for reasoning about distributed programs. The logic has been embedded in HOL, which makes it possible to —in principle— do mechanical verification of UNITY programs. In our research we extend the existing embedding of UNITY with a substitution rule and compositionality results. The substitution rule is very useful in calculation as it enables us to simplify behavior expressions using invariants. Not only that: some behavior is even inexpressible in UNITY without exploiting invariants. Compositionality is what allows us to conclude a property of a composite program from the properties of its components. So, it allows modularity in designing. Indeed, both the substitution rule and compositionality are essential tools in designing. Therefore their mechanization in HOL increases the feasibility of HOL to verify distributed programs.

1 Introduction

UNITY, invented by Chandy and Misra in 1988 [CM88], is a programming logic which is intended to reason about liveness properties of distributed programs. Although it is not as powerful as the linear temporal logic, it does enjoy a great popularity due to its simplicity. Regardless the chosen formal framework however, reasoning about distributed programs is basically a complicated and error prone process, which is why people have tried to employ machines to aid the verification of distributed programs (this is still an ongoing research). As a first step towards this goal people have embedded many programming logics in various theorem provers. For example Interval Temporal Logic and Temporal Logic of Actions have been embedded in HOL theorem prover by, respectively, Hale [Hal88] and von Wright and Långbacka [vWL92]. As for the UNITY logic, it has also been embedded in HOL [And92, Pra93c] but the embeddings do not fully support substitutions by invariants and the application of compositionality results. In this report we will discuss an extension of the existing embedding of UNITY to support these two design techniques. We will assume the embedding of UNITY in HOL as in [Pra93c].

The results mentioned in this paper are available in a package of HOL files called U3_COMP, available at request.

By *'embedding a logic in a theorem prover'* we mean extending the theorem prover with the definition of the logic. Unless the logic is rather trivial, we will also have to extend the theorem prover with a library of theorems of that logic before we can do any verification with an acceptable convenience. A step further, is to provide the users with some tools to automate formal reasoning with the embedded logic, according to a style which is typical for the logic at hand. In HOL we have ML as a meta language which can be used to tailor such a tool. In the end of this report we will briefly discuss about some tools to support substitutions by invariants and the application of compositionality results.

HOL itself is an interactive proof environment that supports reasoning in ***higher order logic*** —hence the name HOL— ,that is, a version of predicate calculus where variables can range over functions and predicates. The logic is also *typed*. See [GM93] for an introduction to HOL. HOL has mainly been used for hardware verification. However, the higher order features make it also attractive to express and do formal reasoning about program behavior.

The source files for both HOL-UNITY and its extension with the Substitution Rule and compositionality results is available at request.

1.1 Compositionality

In UNITY a program is simply a non-empty set of atomic, always enabled statements. A program execution starts from any state satisfying a given initial condition and goes on forever; in each execution step some statement is selected nondeterministically and executed. The selection process is required to be fair, that is, each statement should be selected infinitely many times. In UNITY's view, there is no such thing as termination although it can always be simulated by *fixed points*, that is, states that are invariant under the execution any statement in the program. As there is no ordering on the execution imposed, one is free to think that the statements are distributed among a number of parallel processors. Of course then the underlying architecture should respect the atomicity and fairness assumptions of UNITY.

Compositionality is what allows one to conclude a property of a composite program from the properties of its components. In other words, compositionality tells us how to break a global specification of a program into the specifications of its components. It is thus a very useful property for designing programs.

There are two primitive program composition methods in UNITY: ***parallel composition*** and ***superposition***. The later is not going to be dealt with in this paper. There is not much use for sequential composition of programs since a UNITY program is essentially non-terminating.

Parallel composition in UNITY is simply the union of all statements in the component programs. This turns out to be very advantageous for the compositionality of safety properties as they simply distribute over parallel composition. Progress properties are however more difficult.

UNITY expresses progress by means of ENSURES operator, which exploits fairness to establish progress. More specifically, ENSURES requires the existence of a statement A that establishes, say, q. Then by fairness A will be executed and thus q will be established. Compositionality of ENSURES can be expressed elegantly. However, ENSURES does not cover all possible progress. For example progress may rely on the mutual effect of several statements instead of just one statement. To cover this, $\mapsto$ operator is introduced as a transitive and disjunctive closure of ENSURES . However, $\mapsto$ is not compositional —that is, we cannot conclude a $\mapsto$ property in the composite program based on the $\mapsto$'s in the component programs, unless we know how the $\mapsto$'s in the component programs are constructed. That is, we have to look into the *proofs* of the progress properties of the component programs. To get around this one can define a stronger closure of ENSURES which captures a proof scheme which is compositional. For example Udink, Herman, and Kok [UHJ94] has shown that a closure of ENSURES that roughly behaves like the until operator from the linear temporal logic is compositional. The trade off is however that a stronger progress operator will have a more restricted proof scheme. In this paper we will handle a more traditional approach as proposed by Singh [Sin89] who claims that *local* progress is compositional under some strict condition regarding the shared variables. It should be added that we have to be very precise in dealing with write variables and the type of permissible progress expressions in order to derive Singh's compositionality. See also [Pra93b] for more discussion on this topics.

1.2 Substitution Rule

The *Substitution Rule* of UNITY states that an invariant can be treated as a theorem that can be applied to a behavior expression of a program. So if the invariant implies $x = y$ then by the Rule it is permissible to substitute —hence the name Substitution Rule— any occurrence of x in a behavior expression by y. The Rule is very practical for calculations in UNITY logic. Also, there are properties of a program that cannot be expressed in UNITY without using Substitution Rule. However, things are rather subtle here. Only behavior expressions that take reachable states explicitly into account are permissible in the Substitution Rule, otherwise the Rule may lead to a contradiction [San91]. Reachability is however rather hard to calculate with and moreover a complete formulation of all reachable states is often not needed in practical derivations.

To make it more practical, behavior expressions can be parameterized with an invariant [San91], stating that the behavior holds, assuming the invariant holds. The Substitution Rule can be safely applied to parameterized expressions. There is also no problem with compositionality as the parameters —being invariant— are compositional.

1.3 An Overview of this Paper

Section 2 gives a brief review of the UNITY logic and how it is formally represented in HOL. Sections 3 and 4 present the formalization of parameterized UNITY operators and associated compositionality results. Section 5 gives a brief introduction about the mechanisms used in formal reasoning within HOL. Section 6 explains some tools by which application of Substitution Rule and compositionality can be done in a (semi-)automatic way in HOL.

Because the notation used here may be non-standard at some points, the reader is suggested to read the following subsection about notation.

1.4 Notation

The notation used is a mix of HOL notation and notation common to the computer science community. This is to make the formulas more readable, especially for those who are new to HOL, and at the same time to give some idea as how a given formula is represented in HOL.

Function application is denoted with a small dot: f applied to x is written as $f.x$. Some functions are treated as unary operators and an application of such a function is thus written without dot like NOT p and UNITY Pr. Function abstraction is denoted by λ. For type $\mathcal{T}$, $x : \mathcal{T}$ or $x \in \mathcal{T}$ means x is an object of type $\mathcal{T}$.

Assuming an expression language, $E(E1 \leftarrow E2)$ denotes an expression obtained by substituting all occurrences of $E1$ in E by $E2$. This is a purely syntactic operation.

Predicate calculus operators are denoted as usual by $\neg, \wedge, \vee, \Rightarrow, \forall$ and $\exists$.

Universal quantification over the whole domain is written $\forall i \centerdot P.i$. Universal quantification over a restricted part of the domain characterized by *predicate* V is written $\forall i :: V \centerdot P.i$. Notice that the *big dot* marks the end of domain specification while *small dot* denotes function application. The same notation applies for other quantified operators.

A predicate is a function from a given domain to **bool**. *Predicate operators* are defined as the point-wise lift of the predicate calculus operators:

Definition 1.1 : Predicate Operators
For predicates $p, q \in \mathcal{D} \rightarrow$ **bool**:

$$
\begin{array}{lcl}
\mathsf{TT} & = & (\lambda s \centerdot \mathsf{true}) \\
\mathsf{FF} & = & (\lambda s \centerdot \mathsf{false}) \\
\mathsf{NOT}\ p & = & (\lambda s \centerdot \neg(p.s)) \\
p\ \mathsf{AND}\ q & = & (\lambda s \centerdot (p.s) \wedge (q.s)) \\
p\ \mathsf{OR}\ q & = & (\lambda s \centerdot (p.s) \vee (q.s)) \\
p\ \mathsf{IMP}\ q & = & (\lambda s \centerdot (p.s) \Rightarrow (q.s)) \\
!!i :: W \centerdot f.i & = & (\lambda s \centerdot \forall i :: W \centerdot f.i.s) \\
??i :: W \centerdot f.i & = & (\lambda s \centerdot \exists i :: W \centerdot f.i.s)
\end{array}
$$

We use the terms 'set' and 'predicate' —and also the corresponding notations— interchangely as they are isomorphic (so for example to denote that a set P is non-empty we write $\exists x \centerdot P.x$). The reason for this is that we have chosen for predicate representation in HOL as in the current state of HOL's development predicates are still easier to work with than sets. On the other hand, people are more familiar with the set notation. So, for the benefit of the presentation we will use the set notation whenever possible without having to depart too far from the HOL notation.

For predicate p, the *everywhere* operator is denoted by $[p]$. It means that in every point s in the domain, $p.s$ holds.

We use the following binding convention:

Binding Power of the Operators
Here is a list of operators used in this paper, ordered from the most binding to the least.

1	.
2	$\neg$
3	$\wedge, \vee$
4	$\Rightarrow$
5	NOT
6	AND , OR
7	IMP
8	all other operators but '='
9	=

2 A Review on UNITY

This section briefly explains the UNITY logic and its formalization in HOL.

A UNITY program is a basically non-empty set of statements (the non-emptiness is required to guarantee progress). How the program is executed is already mentioned in Section 1. Let the reader also be reminded that: (a) each statement is atomic, never aborts and always terminates, and (b) the execution should be fair. The last point is crucial in UNITY logic, let us here repeat the definition of fairness. An execution is called fair iff each statement is executed infinitely many often during the execution. For example if a program Pr consists of three statements $\{A, B, C\}$, then:

$$A; A; A; B; A; A; A; C; A; A; A; B; A; A; A; C; \cdots$$

is a fair execution but

$$A; B; C; B; C; B; C; \cdots$$

is, while infinite, un-fair because A is only executed once. If C will color x to black then a fair execution of Pr will guarantee that x will eventually become black, even-though a moment later A or B may change the color of x.

We assume a set Var of *all available* variable names. A state is just a function $s \in \text{Var} \rightarrow \text{Val}$ where Val denotes some domain of the values that the variables may have. The space of all possible states is denoted with State.

From now on we will use the term *action* instead of *statement*. An action A is a relation on the state space, that is, $A \in \text{State} \rightarrow \text{State} \rightarrow \textbf{bool}$. For states s and t the interpretation of $A.s.t$ is that if A is executed in state s then it *can* bring the system to the state t.

Each action in a UNITY program is required to be always enabled, that is, any begin state is always related to some end states. Consequently an always enabled action does not abort. There is also no need to introduce a special constant to mark non-termination since each action is assumed to terminate.

Definition 2.1 : Always Enabled Action
For all action $A \in \text{State} \rightarrow \text{State} \rightarrow \textbf{bool}$:

$$\text{AlwaysEn}A \;=\; \forall s.\ \exists t.\ A.s.t$$

A simple assignment is always enabled. According to the conventional definition, a guarded assignment will abort on the states where the guard does not hold. In UNITY however, an if-action like if $b \to x := x + 1$ denotes a guarded-or-skip assignment. That is, if the guard fails then the effect of the action is equal to skip. Such an action is therefore always enabled.

Let Nov define a polymorphic constant. Because in HOL a type cannot be empty, Nov exists for any given type. Let us define function projection as follows.

Definition 2.2 : Function Projection
For all $f \in A \to B$:

$$(f \upharpoonright A).x = \begin{cases} f.x & \text{if } x \in A \\ \mathsf{Nov} & \text{else} \end{cases}$$

A variable set V is ignored by an action A if A does not change the value of any variable in V. A variable set V is invisible to an action A if the result of executing A does not depend on the value of any variable in V. The definition of 'ignored' and 'invisible' is given below.

Definition 2.3 : Ignored By
For any variable set V and action A:

$$V \ \mathsf{IG\_BY}\ A \ = \ \forall s, t.\ A.s.t \Rightarrow (s \upharpoonright V = t \upharpoonright V)$$

Definition 2.4 : Invisible
For any variable set V and action A:

$$V \ \mathsf{INVI}\ A \ = \ \forall s, t, s', t'.\ \left.\begin{array}{ll} s \upharpoonright (\mathsf{NOT}\ V) = s' \upharpoonright (\mathsf{NOT}\ V) & \wedge \\ t \upharpoonright (\mathsf{NOT}\ V) = t' \upharpoonright (\mathsf{NOT}\ V) & \wedge \\ s' \upharpoonright V = t' \upharpoonright V & \wedge \\ A.s.t & \end{array}\right\} \Rightarrow A.s'.t'$$

We will represent a UNITY program by a quadruple (P, In, R, W) where P, In, R, and W are predicates defining respectively: (a) a non-empty set of always enabled actions, (b) the initial states, (c) the set of read variables, and (d) the set of write variables. In the sequel, let Uprog denote the type of such quadruple and let $x \in Pred$ mean that x is a state predicate, that is $x \in$ State $\to$ **bool**. Not all objects of type Uprog are UNITY programs. Those that are, are characterized by predicate UNITY defined below.

Definition 2.5 : UNITY Program
For all $(P, In, R, W) \in$ Uprog:

$$\mathsf{UNITY}.(P, In, R, W) \ = \ \left\{\begin{array}{ll} (\exists A.\ P.A) & \wedge \\ (\forall A :: P.\ \mathsf{AlwaysEn}A) & \wedge \\ (\forall A :: P.\ (\mathsf{NOT}\ W)\ \mathsf{IG\_BY}\ A) & \wedge \\ (\forall x.\ W.x \Rightarrow R.x) & \wedge \\ (\forall A :: P.\ (\mathsf{NOT}\ R)\ \mathsf{INVI}\ A) & \end{array}\right.$$

The first condition states that a UNITY program has at least one action; the second states that all actions in a UNITY program are always enabled; the third states that a UNITY program does not write to any variable that is not declared as its write variable; the fourth states that a write variable is also readable; and the last states that the effect of a UNITY program does not depend on the value of any variable outside its declared read variables.

The third condition is called ***write constraint*** and the fifth is called ***read constraint***.

The first two conditions are explicitly mentioned in the original UNITY description [CM88] while the other conditions, those regarding variables accessibility, are left implicit. Indeed, all basic theorems mentioned in [CM88] do not require any notion of variable accessibility. However, accessibility turns out to play an important role in the compositionality of progress properties [Sin89, Pra93b, UHJ94], which is the reason that we include it in our definition.

To denote each component of a UNITY program the following destructors are introduced.

Definition 2.6 : PROG, INIT, READ, WRITE
For all quadruple $Pr = (P, In, R, W)$:

$$\begin{aligned}\mathsf{PROG}.Pr &= P\\ \mathsf{INIT}.Pr &= In\\ \mathsf{READ}.Pr &= R\\ \mathsf{WRITE}.Pr &= W\end{aligned}$$

Without being explicit about the syntax used, below we give a piece of code denoting a UNITY program for an alternating bit protocol. Actions are separated by □ symbols. The meaning of an IF action is somewhat non-conventional: it behaves like a skip if none of its guards is satisfied.

```
PROG  ALT_BIT
READ  wire, Sbit, buffer, Rbit, ack
WRITE wire, Sbit, buffer, Rbit, ack
INIT  Sbit<>ack /\ Rbit=ack

ASSIGN
   (IF Sbit = ack --> wire, Sbit := Exp, ~Sbit)
□ (buffer, Rbit := wire, Sbit)
□ (ack := buffer)
```

We will call a predicate that only restricts the value of the read variables of a program Pr a *local predicate* of Pr. For example, wire $= 3$ is a local predicate of the program ALT_BIT above whereas wire $= 3 \vee x = 0$ is not because it restricts the value of x, which is not a read variable of ALT_BIT.

Definition 2.7 : Local Predicate
For all $p \in \mathsf{Pred}$ and $Pr \in \mathsf{Uprog}$:

$$p \,\mathsf{LocPred}\ \mathsf{in}\ Pr \;=\; \forall s,t.\ (\forall x :: \mathsf{READ}.Pr.\ s.x = t.x) \Rightarrow (p.s = p.t)$$

TT and FF are always local predicates. Local predicates are also preserved by the predicate operators (NOT, AND, OR, ...).

2.1 UNITY Operators

UNITY has three primitive operators to express the behavior of a program. There is the UNLESS operator which expresses safety and the ENSURES and $\mapsto$ operators which express progress.

To remind the reader: For an action A and states descriptions (predicates) p and q, the Hoare triple $\{p\}\ A\ \{q\}$ asserts that the execution of A in any state satisfying p —if it does not abort— always ends in a state satisfying q.

For a UNITY program $Pr = (P, In, V, W)$ and predicates p and q, p UNLESS q informally means that p, once holds in Pr, will remain to hold until q holds. In other words, each action in

P should either leave p invariant, or establish q.

Definition 2.8 : Unless
Let $p, q \in \mathsf{Pred}$ and $Pr \in \mathsf{Uprog}$.

$$p \ \mathsf{UNLESS}\ q \ \mathsf{in}\ Pr \ = \ (\forall A :: \mathsf{PROG}.Pr.\ \{p \wedge \neg q\}\ A\ \{p \vee q\})$$

UNLESS is reflexive, anti-reflexive, and includes IMP. It is also IMP-monotonic in its second argument.

Theorem 2.1 : Some Properties of UNLESS
Let $Pr \in \mathsf{Uprog}$ and $p, q, r \in \mathsf{Pred}$.

1. UNLESS **Reflexivity**

$$p\ \mathsf{UNLESS}\ p\ \mathsf{in}\ Pr$$

2. UNLESS **Anti-Reflexivity**

$$p\ \mathsf{UNLESS}\ (\mathsf{NOT}\ p)\ \mathsf{in}\ Pr$$

3. IMP **Promotion**

$$\frac{[p\ \mathsf{IMP}\ q]}{p\ \mathsf{UNLESS}\ q\ \mathsf{in}\ Pr}$$

4. UNLESS **Monotonicity**

$$\frac{p\ \mathsf{UNLESS}\ q\ \wedge\ [q\ \mathsf{IMP}\ r]}{p\ \mathsf{UNLESS}\ r\ \mathsf{in}\ Pr}$$

Informally, p ENSURES q is an extension of p UNLESS q with as an additional property that there also exists an action that establishes q. By fairness, this action will eventually be executed and hence q will be established indeed. Thus ENSURES formulates a progress property.

Definition 2.9 : Ensures
Let $p, q \in \mathsf{Pred}$ and $Pr \in \mathsf{Uprog}$.

$$p\ \mathsf{ENSURES}\ q\ \mathsf{in}\ Pr \ = \ \begin{cases} \mathsf{UNITY}.Pr\ \wedge\ p\ \mathsf{UNLESS}\ q\ \mathsf{in}\ Pr\ \wedge \\ (\exists A :: \mathsf{PROG}.Pr.\ \{p\ \mathsf{AND}\ (\mathsf{NOT}\ q)\}\ A\ \{q\}) \end{cases}$$

While many properties of UNLESS still hold regardless whether a program is a UNITY program or not, this is not the case with progress properties. Here, we will consider a progress expression to be valid only if it is applied to a UNITY program. This is why we have explicitly required in the definition of p ENSURES q in Pr that Pr should be a UNITY program.

ENSURES is reflexive, includes IMP , and is monotonic in its second argument.

However, ENSURES only covers progress which is guaranteed by the execution of a single action. To express progress achieved by concerted effect of several actions, $\mapsto$ is introduced as, roughly speaking, the least transitive and left disjunctive closure of ENSURES . More specifically, $(\lambda p, q.\ p \mapsto q\ \mathsf{in}\ Pr)$ is the least relation U satisfying the following three properties:

1. $\mathsf{ENSURES}$ **Lift**

$$(p \;\mathsf{ENSURES}\; q \;\mathsf{in}\; Pr) \;\wedge\; (p, q \;\mathsf{LocPred}\;\; \mathsf{in}\; Pr) \;\Rightarrow\; U.p.q$$

2. **Transitivity**

$$U.x.y \wedge U.y.z \;\Rightarrow U.x.z$$

3. **Left Disjunctivity**
   For any non-empty W:

$$(\forall x :: W \bullet\; U.x.y) \;\Rightarrow\; U.(??\, x :: W \bullet\; x).y$$

Being a least closure $\mapsto$ induces an induction principle: it suffices to show that a relation X satisfies above three rules to prove:

$$\forall p, q \bullet\; p \mapsto q \;\mathsf{in}\; Pr \Rightarrow X.p.q$$

Also, $\mapsto$ itself satisfies above three rules.

We deviate slightly from the original definition in [CM88]. There the locality of the predicates in the first rule above is not required. By defining $\mapsto$ as we do, a progress by $\mapsto$ can only be valid in a program if the progress does not depend on the value of any local variable of whatever environment of that program. This way we can get a stronger compositionality result [Pra93b], which is why deviate from the original definition.

Other simple properties of $\mapsto$ are that it: (a) is monotonic in its second argument, (b) is anti-monotonic in its first argument, and (c) includes IMP .

3 Formalization of Parameterized Operators

There are some properties of a program that cannot be expressed in UNITY without exploiting invariants. For example suppose Pr is a UNITY program that contains

$$\mathsf{if}\; x \neq 0 \rightarrow b := \mathsf{true}$$

as the only action that modifies b. We cannot prove that ($\mathsf{NOT}\; b$) $\mathsf{UNLESS\; FF}$ —if with UNLESS here we mean the intended interpretation— unless we know that $x = 0$ is invariant.

To keep track of which invariant has been used to conclude a property, Sanders [San91] proposed to parameterize the UNITY operators with the invariant used to conclude it. This is very useful when composing programs to know which invariant the component programs should satisfy.

Let J be an invariant in program Pr which, for now, means that J holds through out the execution of Pr. Since J is invariant, whatever the behavior of Pr, J will always hold and thus can be regarded as a theorem in manipulating a behavior expression of Pr. So for example, if J implies $q = r$ then we may substitute any occurrence of q in —for example $p \mapsto (q \;\mathsf{OR}\; r)$ in Pr— with r, thus concluding $p \mapsto r$ in Pr. This is basically what the Substitution Rule asserts. That is:

Substitution Rule (This theorem is NOT formalized in HOL)
Let $p, q, I \in \mathsf{Pred}$ and $Pr \in \mathsf{Uprog}$ and $\mathsf{Op} \in \{\; \mathsf{UNLESS}\;,\; \mathsf{ENSURES}\;, \mapsto\}$. Let p' be as p but with some occurrences of subterm $E1$ is replaced by term $E2$. Similarly we define q'. Then:

$$\frac{\begin{array}{c} I \text{ is invariant in } Pr \\ [I \;\mathsf{IMP}\; (E1 = E2)] \\ p \;\mathsf{Op}\; q \;\mathsf{in}\; Pr \end{array}}{p' \;\mathsf{Op}\; q' \;\mathsf{in}\; Pr}$$

The reader should be warned that the rule above is not a sound extension of the UNITY logic defined so far (see the discussion in the paragraph below). However, if UNLESS , ENSURES and $\mapsto$ are interpreted as they intended to mean, for example as in [San91] then the Rule above is a valid theorem. The formalization of the Rule in the form as formulated above is outside the scope of this paper. We did however formalize an equivalent form of it.

One has to be careful in the use of the Substitution Rule or else one may come to a contradiction. The problem is that by definition p UNLESS q in Pr holds regardless whether p or q ever holds during the execution. On the other hands, an invariant only holds in reachable states. So, using an invariant to do substitution is only valid on the part of p and q that are reachable. See [Mis90, San91, Pra93a] for more discussions on this topic. To avoid this problem we can weaken the definition of UNITY operators by restricting them to reachable states. For example UNLESS is redefined to:

$$(\dagger)\quad p \;\mathsf{UNLESS}\; q \;\mathsf{in}\; Pr \;=\; (\forall A :: \mathsf{PROG}.Pr.\; \{\mathsf{Reach}.Pr \;\mathsf{AND}\; p \;\mathsf{AND}\; (\mathsf{NOT}\; q)\}\; A\; \{p \;\mathsf{OR}\; q\})$$

Where Reach.Pr is a predicate that characterizes all reachable states of Pr. ENSURES can be redefined in an analogous way.

Above definition captures exactly the intended meaning of UNLESS but unfortunately Reach is in general not easy to compute. The solution is to parameterize UNITY operators with an invariant to reflect that their validity is relative to the invariant. Using the invariant to do substitution is then not a problem since it is already assumed.

Before we give a formal definition of parameterized operators, first here is the formal definition of invariant.

Definition 3.1 : Invariant
For $Pr \in$ Uprog and $J \in$ Pred:

$$\mathsf{INV}.Pr.J \;=\; [\mathsf{INIT}.Pr \;\mathsf{IMP}\; J] \;\wedge\; J \;\mathsf{UNLESS}\; \mathsf{FF} \;\mathsf{in}\; Pr$$

An equivalent definition is —which can be obtained by unfolding the definition of UNLESS — is:

$$\mathsf{INV}.Pr.J \;=\; [\mathsf{INIT}.Pr \;\;\mathsf{IMP}\;\; J] \;\wedge (\forall A :: \mathsf{PROG}.Pr.\; \{J\}\; A\; \{J\})$$

So, J is an invariant iff it is preserved by all actions in the program. Another, weaker, notion is: J is an invariant iff it holds through out any computation. The reader is warned not to confuse these two notions.

TT is always an invariant and the conjunction of two invariants is also an invariant. Also, INV is not monotonic with respect to IMP (while the weaker notion of invariant mentioned above is), that is, if INV.$Pr.I$ holds and $[I$ IMP $J]$, then INV.$Pr.J$ does not necessarily hold.

The formal definition of parameterized operators is as follows.

Definition 3.2 : Parameterized Operators
Let $Pr \in$ Uprog and $p, q, J \in$ Pred.

$$\begin{aligned}
\mathsf{UNL}.Pr.J.p.q &= (p \;\mathsf{AND}\; J)\;\mathsf{UNLESS}\;(q \;\mathsf{AND}\; J)\;\mathsf{in}\; Pr \;\wedge\; \mathsf{INV}.Pr.J\\
\mathsf{ENS}.Pr.J.p.q &= (p \;\mathsf{AND}\; J)\;\mathsf{ENSURES}\;(q \;\mathsf{AND}\; J)\;\mathsf{in}\; Pr \;\wedge\; \mathsf{INV}.Pr.J\\
\mathsf{LTO}.Pr.J.p.q &= (p \;\mathsf{AND}\; J) \mapsto (q \;\mathsf{AND}\; J)\;\mathsf{in}\; Pr \;\wedge\; \mathsf{INV}.Pr.J
\end{aligned}$$

The second argument J is the *parameter*.

The definition above should correct the one in [San91] which with the Substitution Rule turns out to be unsound [Pra93a].

Since the conjunction of all invariants is again an invariant and that it precisely defines all reachable states, the definition of UNITY operators as described (by example) earlier in (†) is just an instantiation of above definition. Also, since TT is always an invariant, UNLESS , ENSURES and $\mapsto$ are just instantiations of UNL, ENS, and LTO.

Many theorems about UNITY operators can be transcribed to the corresponding theorems about parameterized operators. Section 6 briefly explains a tool to —for simple cases— automate this conversion.

A property worth mentioning of the parameterized operators is that the parameter can be strengthened by another invariant. This means that a property remains valid under a stronger invariant.

Theorem 3.1 : Strengthening Parameter
Let $Pr \in$ Uprog and $p, q, J, H \in$ Pred.

(1) $$\frac{\mathsf{INV}.Pr.J \ \wedge\ \mathsf{UNL}.Pr.H.p.q}{\mathsf{UNL}.Pr.(H \ \mathsf{AND}\ J).p.q}$$

(2) $$\frac{\mathsf{INV}.Pr.J \ \wedge\ \mathsf{ENS}.Pr.H.p.q}{\mathsf{ENS}.Pr.(H \ \mathsf{AND}\ J).p.q}$$

(3) $$\frac{\begin{array}{c} J\ \mathsf{LocPred}\ \mathsf{in}\ Pr \\ \mathsf{INV}.Pr.J \ \wedge\ \mathsf{LTO}.Pr.H.p.q \end{array}}{\mathsf{LTO}.Pr.(H \ \mathsf{AND}\ J).p.q}$$

The Substitution Rule is just a corollary of the definition. We will show it for UNL. The case for ENS and LTO can be proven in much the same way. Let J be an invariant that implies $E1 = E2$.

$\mathsf{UNL}.p.q.J.Pr$
$=$ { Definition of UNL }
$\mathsf{INV}.Pr.J \wedge ((p\ \mathsf{AND}\ J)\ \mathsf{UNLESS}\ (q\ \mathsf{AND}\ J)\ \mathsf{in}\ Pr)$
$=$ { Assumption: $[J\ \mathsf{IMP}\ (E1 = E2)]$, rewriting }
$\mathsf{INV}.Pr.J \wedge ((p(E1 \leftarrow E2)\ \mathsf{AND}\ J)\ \mathsf{UNLESS}\ (q(E1 \leftarrow E2)\ \mathsf{AND}\ J)\ \mathsf{in}\ Pr)$
$=$ { Definition of UNL }
$\mathsf{UNL}.p(E1 \leftarrow E2).q(E1 \leftarrow E2).J.Pr$

The formalization of Substitution Rule requires however a formal definition of the expression language being used, which is beyond the scope of this paper. This does not mean that applying the Substitution Rule is impossible in our HOL-formalization of UNITY. It is true that we cannot have the Rule as a theorem, but we can still have it as a HOL-tactic. This is explained further in Section 6.

4 Parallel Composition and Compositionality

Since UNITY does not require any fixed ordering on the execution of the actions in a program, parallel composition can be modeled simply by taking the union of all actions in the component programs. The formal definition is given below.

Definition 4.1 : Parallel Composition
Let $Pr, Qr \in \mathsf{Uprog}$.

$$Pr \| Qr \;=\; (\mathsf{PROG}.Pr \cup \mathsf{PROG}.Qr\ ,\ \mathsf{INIT}.Pr \ \mathsf{AND}\ \mathsf{INIT}.Qr\ ,\ \mathsf{READ}.Pr \cup \mathsf{READ}.Qr\ ,\ \mathsf{WRITE}.Pr \cup \mathsf{WRITE}.Qr)$$

Parallel composition is commutative and closed within the space of UNITY programs. A local predicate in a component program is also a local predicate in the composite program. The composition also preserves INV, UNLESS , and UNL, and hence they are compositional. The last mentioned property is displayed below.

Theorem 4.1 : Compositionality of Safety
Let $Pr, Qr \in \mathsf{Uprog}$ and $p, q, J \in \mathsf{Pred}$.

$$(1)\quad \frac{\mathsf{INV}.Pr.J \ \wedge\ \mathsf{INV}.Qr.J}{\mathsf{INV}.(Pr\|Qr).J}$$

$$(2)\quad \frac{p\ \mathsf{UNLESS}\ q\ \mathsf{in}\ Pr \ \wedge\ p\ \mathsf{UNLESS}\ q\ \mathsf{in}\ Qr}{p\ \mathsf{UNLESS}\ q\ \mathsf{in}\ (Pr\|Qr)}$$

$$(3)\quad \frac{\mathsf{UNL}.Pr.J.p.q \ \wedge\ \mathsf{UNL}.Qr.J.p.q}{\mathsf{UNL}.(Pr\|Qr).J.p.q}$$

Theorem 4.1 states that the safety of the composite program follows from the safety of its components —which is quite a natural design strategy. The proof is a matter of unfolding definitions. In fact, HOL can automatically prove Theorem 4.1.

Since ENSURES implies UNLESS , a progress property expressed by ENSURES in a composite program follows from the safety of its components and in addition one of the component should have an action that can indeed ensures the progress.

Theorem 4.2 : Compositionality of ENSURES
Let $Pr, Qr \in \mathsf{Uprog}$ and let $p, q \in \mathsf{Pred}$.

$$\frac{\begin{array}{c}\mathsf{UNITY}.Pr\\ p\ \mathsf{UNLESS}\ q\ \mathsf{in}\ Pr \ \wedge\ p\ \mathsf{ENSURES}\ q\ \mathsf{in}\ Qr\end{array}}{p\ \mathsf{ENSURES}\ q\ \mathsf{in}\ (Pr\|Qr)}$$

Notice that the condition p ENSURES q in Qr implies the existence of an action establishing q. The additional condition UNITY.Pr is required because otherwise $Pr\|Qr$ may not be a UNITY program and thus no ENSURES property is valid in it. As a corollary an analogous compositionality statement also holds for ENS.

4.1 Compositionality of $\mapsto$

Progress under $\mapsto$ is unfortunately not compositional except under some conditions as given by Ambuj K Singh [Sin89]. This is not too surprising because a progress expression such as $p \mapsto q$ does not tell us anything about how the progress is to be established, so we do not know which programs can be safely executed in parallel without destroying the progress properties.

Indeed, if we have full knowledge of the behavior upon which a progress property relies, then parallel composition can be made to preserve this property by requiring the component programs to satisfy this behavior. However, during a design process such knowledge may not be known until a later design stage.

Consider two programs: Pr and Qr. Assume that whenever Qr modifies any shared variable between Pr and Qr it also raises the flag b. Singh's theorem then states that any progress $p \mapsto q$ in Pr will be preserved by the composite $Pr[\!]Qr$, or if Qr does disrupt this progress it can only do so by modifying the shared variables, in which case b will be raised.

Singh's Theorem on the Compositionality of $\mapsto$
Let V denote the vector $v_1, v_2, \ldots$ of *all* variables read by Pr and written by Qr.

$$\frac{\begin{array}{c} p \mapsto q \text{ in } Pr \\ \forall C.\ (V = C)\ \mathsf{UNLESS}\ b \text{ in } Qr \end{array}}{p \mapsto (q\ \mathsf{OR}\ b) \text{ in } (Pr[\!]Qr)}$$

Before we can verify above theorem in HOL there are some informalities which have to be cleared first. First, the expression $V = C$ in $(V = C)$ UNLESS b that appears in the theorem is an abuse of notation because $V = C$ has the type **bool** whereas UNLESS expects a predicate in its place. What is actually meant is a predicate characterizing those states where the value of each $x \in V$ is $C.x$. We need to be more explicit to the machine about this or else it will complain about type conflicts. To formalize this we introduce angled brackets lifting: $\langle V = C \rangle$ denotes the predicate as explained above.

Definition 4.2 : Angled Brackets Lifting
For all $V \in \mathsf{Var} \to \mathbf{bool}$, $\supset$: $\mathsf{Val} \to \mathsf{Val} \to \mathsf{Bool}$, and $C \in \mathsf{Var} \to \mathsf{Val}$:

$$\langle V \supset C \rangle \;=\; (\lambda s : \mathsf{State}.\ (\forall x :: V.\ s.x \supset C.x))$$

Secondly, we have used the term *shared variables* without mentioning what we exactly mean by this. A possibility is to define it as the intersection of the read variables of the component programs. However, here we will formally define shared variables between Pr and Qr, denoted by $\mathsf{DVa}.Pr.Qr$, as the variables read by Pr and written by Qr —DVa stands for Dependent Variables. Note that DVa is not symmetric. Singh's Theorem becomes more general with this definition of shared variables.

Definition 4.3 : Shared Variables
For all $Pr, Qr \in \mathsf{Uprog}$:

$$\mathsf{DVa}.Pr.Qr \;=\; \mathsf{READ}.Pr \cap \mathsf{WRITE}.Qr$$

So, the condition in Singh's theorem can now be formally written as

$$\forall C.\ \langle \mathsf{DVa}.Pr.Qr = C \rangle\ \mathsf{UNLESS}\ b \text{ in } Qr$$

To prove Singh's theorem we need the following property of local predicates. The property is so natural that we often take it for granted. Let p be a local predicate of Pr. Being local, it does not refer to any local variables of other programs. Consequently, the only way another program can disrupt p is by writing to the shared variables.

Theorem 4.3 : Local Predicate Safety
For all $Pr, Qr \in \mathsf{Uprog}$:

$$\frac{\begin{array}{c} (\mathsf{UNITY}Pr) \wedge (\mathsf{UNITY}Qr) \\ p\ \mathsf{LocPred} \text{ in } Pr \end{array}}{\forall C.\ (p\ \mathsf{AND}\ \langle \mathsf{DVa}.Pr.Qr = C \rangle)\ \mathsf{UNLESS}\ (\mathsf{NOT}\ \langle \mathsf{DVa}.Pr.Qr = C \rangle) \text{ in } Qr}$$

For the proof the reader is referred to [Pra93b]. It suffices here to say that without having write accessibility formally present in the definition of UNITY the property would not be derivable. This is not too difficult to see: if Qr is free to write to the local variables of Pr (that is those variables in READ.Pr − WRITE.Qr) then it does not need to write to any shared variable to destroy a local predicate of Pr.

Singh's theorem can now be formalized as follows.

Theorem 4.4 : Compositionality of $\mapsto$
For all UNITY programs Pr, Qr and $p, q, b \in \mathsf{Pred}$:

$$\frac{\begin{array}{c} b \ \mathsf{LocPred} \ \ \mathsf{in}\ (Pr[\!]Qr) \\ p \mapsto q \ \mathsf{in}\ \mathrm{Pr} \\ \forall C.\ \langle \mathsf{DVa}.Pr.Qr = C\rangle \ \ \mathsf{UNLESS}\ b \ \ \mathsf{in}\ Qr \end{array}}{p \mapsto (q \ \mathsf{OR}\ b) \ \ \mathsf{in}\ (Pr[\!]Qr)}$$

For the proof of above the reader is referred to [Sin89] and [Pra93b]. Informally it can be motivated as follows. Recall that by the definition of $\mapsto$ the progress property $p \mapsto q$ can only be valid in Pr if the progress does not at any point depend on the behavior of local variables of Qr. So, by Theorem 4.3 Qr can only disturb this progress by modifying some (shared) variable in DVa.$Pr.Qr$. However, the assumption also says that every time Qr modifies DVa.$Pr.Qr$ it will also raise the flag b. So, either the progress $p \mapsto q$ is maintained in $Pr[\!]Qr$, or Qr disrupts this progress —in which case b is raised. Hence $p \mapsto (q$ OR $b)$ in $(Pr[\!]Qr)$.

A more general compositionality result than Theorem 4.4 has also been verified. It is displayed below.

Theorem 4.5 : Compositionality of $\mapsto$
For all UNITY programs Pr, Qr and $p, q, a, b \in \mathsf{Pred}$:

$$\frac{\begin{array}{c} a, b \ \mathsf{LocPred} \ \ \mathsf{in}\ (Pr[\!]Qr) \\ p \mapsto q \ \mathsf{in}\ \mathrm{Pr} \\ \forall C.\ (a \ \mathsf{AND}\ \langle \mathsf{DVa}.Pr.Qr = C\rangle) \ \ \mathsf{UNLESS}\ b \ \ \mathsf{in}\ Qr \end{array}}{(p \ \mathsf{AND}\ a) \mapsto ((q \ \mathsf{AND}\ a) \ \mathsf{OR}\ b) \ \ \mathsf{in}\ (Pr[\!]Qr)}$$

This concludes the discussion about formalizing compositionality in UNITY.

5 A Brief Review on HOL's Proving Mechanisms

To support formal reasoning HOL has *rules* and *tactics*. Rules are used in forward proving and tactics in backward proving. The base entities being manipulated are *theorems*. A theorem is a pair separated by `|-` like: `Asml |- concl`, where `Asml` is the list of the assumptions from which `concl` is provable. We also write `|- th` to stand for `[] |- th`.

A rule generates a theorem, usually from other theorems. A rule is either one of the HOL primitive inference rules, or a composition of them. For example `MP` applies Modus Ponens:

```
    |- P ==> Q      |- P
  ------------------------- MP
           |- Q
```

Another example is `CONJ`:

```
    |- P    |- Q
 ------------------  CONJ
     |- P /\ Q
```

In a backward proof one starts with a conjecture —or *goal* as it is called in HOL. A goal is a pair (Asml,concl), containing the elements of a theorem but it is yet to be validated. It is also denoted with ?- like Asml ?- concl. We also write ?- trm to stand for □ ?- trm.

A tactic transform a goal into other (usually simpler) goals —also called *subgoals*— and a function that will prove the goal once the subgoals are proven. In a sense, tactics are inverse of rules: a goal transformation by a tactic is only permissible if there is a rule that can infer the goal given the generated subgoals. A tactic proves a goal if it can transform the goal into subgoals of the form (Asml,T), which means true. To prove a conjecture, tactics are applied. In the process subgoals are created. When all subgoals are proven, the conjecture is justified and becomes a theorem.

An example of a tactic is CONJ_TAC which is the counterpart of CONJ rule mentioned earlier.

```
      ?- P /\ Q
   ================  CONJ_TAC
     ?- P     ?- Q
```

Tactics can also be composed using *tacticals*. Some examples of tacticals are THEN for sequential composition and REPEAT to apply a tactic repeatedly until it fails.

Although forward and backward styles are duals, there are cases where one style may be easier to use than the other. In this case HOL is very flexible because one can always mix both styles. Most HOL users seem however to rely more on the backward style of reasoning.

6 Tactics and Rules for Substitution Rule and Compositionality

In the earlier sections we have presented the formal definition of parameterized UNITY operators and shown that the Substitution Rule is valid indeed. Also, we have presented the formalization of various compositionality results. To apply these results one can always use the standard HOL tactics and rules. However, further automation is possible. For example to apply the compositionality results, one has to explicitly inform HOL which theorem it should use. It would be more convenient to let HOL select the theorem for us. Also, because we do not have an explicit expression language we cannot formalize the Substitution Rule as a theorem. Its application has to be simulated by performing the derivation at the end of Section 3. It would be more convenient to automate the derivation in a form of a tactic. Here we will mention some such tactics that we have made. They are available in a package of HOL files called U3_COMP, available at request.

HOL expressions are printed in the typewriter font. Function application is denoted with a space like f x. UNITY basic operators will be treated like ordinary functions. For example the p ENSURES q in Pr is denoted in HOL by ENSURES p q Pr. Parallel composition $\|$ is denoted by PAR in HOL.

The tactic PAR_TAC checks whether it can distribute a property in a composite program to the component programs according to the UNITY compositionality theorems as mentioned in Section 4. For a given goal, there can only be at most one matching compositionality theorem to apply to. Here is some example:

```
   ?- INV (Pr PAR Qr) J
 ========================  PAR_TAC
      ?- INV Pr J
      ?- INV Qr J


   ?- ENSURES (Pr PAR Qr) p q
 ============================== PAR_TAC
      ?- UNITY Pr
      ?- UNLESS p q Pr
      ?- ENSURES p q Qr
```

INV_STR_TAC applies Invariant Strengthening (Theorem 3.1) to the goal. It also automatically selects which theorem to apply. For example:

```
   ?- LTO Pr (H AND J) p q
 ===========================  INV_STR_TAC
      ?- LTO Pr H p q
      ?- INV Pr J
      ?- LocPred Pr J
```

SR_TAC simulates the application of the Substitution Rule. For example:

```
   ?- ENS Pr J p q
 ==========================  SR_TAC "a" "b"
   ?- ENS Pr J a b
   ?- q AND J = b AND J
   ?- p AND J = a AND J
```

As we said before, many theorems about UNITY operators can be transcribed to the corresponding theorems about parameterized operators. It is possible to write a rule to automate this conversion —by this we does *not* refer to the HOL's definition of conversion— but the applicability of such a function will depend on the 'regularity' of its arguments. If the theorem is simple, for example it contains no inner quantification and other operators than the UNITY operators, then the function U2UP_LIFT may be able to convert it. For example:

```
     |- !p q p' q'.
        UNLESS p q Pr /\ UNLESS p' q' Pr
        ==>
        UNLESS (p AND p') (q OR q') Pr
 ----------------------------------------  U2UP_LIFT Pred Uprog "J"
     |- !p q p' q'.
        UNL Pr J p q /\ UNL Pr J p' q'
        ==>
        UNL Pr J (p AND p') (q OR q')
```

where Pred is the type of the state predicates such as p and q; Uprog is the type of Pr; and J is a variable of type Pred not yet mentioned in the theorem we want to convert.

A theorem where U2UP_LIFT fails is for example the monotonicity of UNLESS (see Theorem 2.1). The rule fails because of the extra condition $[q \text{ IMP } r]$. The rule also fails on many $\mapsto$ basic theorems because they usually contain extra conditions. In self-tailoring a conversion to parametrized UNITY one can however use the help functions used to build U2UP_LIFT.

With above tactics formal reasoning using Substitution Rule and compositionality results should be easier.

7 Conclusion

The Substitution Rule is very useful in doing calculation. Not only that some behavior is inexpressible in UNITY without the Rule, but it also enables us to simplify behavior expressions using invariants. Compositionality allows modularity of designs as it tells us how to conclude a property of a composite program from of the properties its components. Indeed, both the Substitution Rule and compositionality are essential tools in designing.

We have extended the existing formalization of UNITY in HOL [And92, Pra93c] with the definition of parameterized operators. We have also formally proven various compositionality results —including Singh's compositionality theorem for $\mapsto$. Formal application of the Substitution Rule and compositionality results is thus now possible with HOL. Furthermore, to assist the user we provide tactics that —given arguments in proper forms— should automate the application of Substitution Rule and Compositionality. This results should increase the potential of HOL to verify distributed programs.

As an after-note we want to say that the use of theorem provers like HOL in the verification of programs is still in an experimental phase. That a programming logic, UNITY or whatever else, has been formalized in one or another theorem prover only means that it is then potentially possible to do mechanical program verification. Sometimes the formalization of a theoretical result fails and it turns out that the result carries some hidden assumptions. These hidden assumptions may imply some additional features that are not yet present in the logic. Indeed, it would be through experimentation only that we eventually come to a stable representation of the logic. Later on, other people seeking to formalize some other programming logic or language can of course benefit from the experience.

Another research course is the development of users interface. So far, an effective use of theorem provers requires a lot of practice and experience. In the end, mechanical verification should not be a branch of engineering mastered only by specialists. Instead, it should be a tool that can be easily learned. Tactics should be provided not only to automate proofs but also to protect a novice from a direct exposure to the un-human style of the underlying proof mechanisms.

The source files for both HOL-UNITY and its extension with the Substitution Rule and compositionality results is available at request.

References

[And92] Flemming Andersen. *A Theorem Prover for UNITY in Higher Order Logic.* PhD thesis, Technical University of Denmark, 1992.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design - A Foundation.* Addison-Wesley Publishing Company, Inc., 1988.

[GM93] Mike J.C. Gordon and Tom F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.

[Hal88] Roger W.S. Hale. *Programming in Temporal Logic.* PhD thesis, University of Cambridge, 1988.

[Mis90] J. Misra. Soundness of the substitution axiom. *Notes on UNITY,* 14-90, March 1990.

[Pra93a] ISWB Prasetya. Error in the unity substitution rule for subscripted operators. draft, available on request, 1993.

[Pra93b] ISWB Prasetya. Formalization of variables access constraints to support compositionality of liveness properties. In *Proceeding HUG 93, HOL User's Group Workshop*, pages 326–339. University of British Columbia, 1993.

[Pra93c] ISWB Prasetya. *UU_UNITY: a Mechanical Proving Environment for UNITY Logic.* University of Utrecht, 1993. Will appear as a technical report.

[San91] B.A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, 1991.

[Sin89] A.K. Singh. Leads-to and program union. *Notes on UNITY*, 06-89, 1989.

[UHJ94] R. Udink, T. Herman, and Kok J. Compositional local progress in unity. to appear in the proceeding of IFIP Working Conference on Programming Concepts, Methods and Calculi, 1994., 1994.

[vWL92] J. von Wright and T. Långbacka. Using a theorem prover for reasoning about concurent algorithms. In *Proc. 4th Workshop on Computer-Aided Verification*, Montreal, Canada, June 1992. Springer-Verlag.