Trustworthy Tools for Trustworthy Programs: A Verified Verification Condition Generator

Peter V. Homeier and David F. Martin

Computer Science Department
University of California, Los Angeles
homeier@cs.ucla.edu and dmartin@cs.ucla.edu

Abstract. Verification Condition Generator (VCG) tools have been effective in simplifying the task of proving programs correct. However, in the past these VCG tools have in general not themselves been mechanically proven, so any proof using and depending on these VCGs might have contained errors. In our work, we define and rigorously prove correct a VCG tool within the HOL theorem proving system, for a standard **while**-loop language, with one new feature not usually treated: expressions with side effects. Starting from a structural operational semantics of this programming language, we prove as theorems the axioms and rules of inference of a Hoare-style axiomatic semantics, verifying their soundness. This axiomatic semantics is then used to define and prove correct a VCG tool for this language. Finally, this verified VCG is applied to an example program to verify its correctness.

1 Introduction

The most common technique used today to produce quality software without errors is testing. However, even repeated testing cannot reliably eliminate all errors, and hence is incomplete. To achieve a higher level of reliability and trust, programmers may construct proofs of correctness, verifying that the program satisfies a formal specification. This need be done only once, and eliminates whole classes of errors. However, these proofs are complex, full of details, and difficult to construct by hand, and thus may themselves contain errors, which reduces trust in the program so proved. Mechanical proofs are more secure, but even more detailed and difficult.

One solution to this difficulty is partially automating the construction of the proof by a tool called a *Verification Condition Generator* (VCG). This VCG tool writes the proof of the program, modulo a set of formulas called *verification conditions* which are left to the programmer to prove. These verification conditions do not contain any references to programming language phrases, but only deal with the logics of the underlying data types. This twice simplifies the programmer's burden, reducing the volume of proof and level of proof, and makes the process more effective. However, in the past these VCG tools have not in general themselves been proven, meaning that the trust of a program's proof rested on the trust of an unproven VCG tool.

In this work we define a VCG within the Higher Order Logic (HOL) theorem proving system [6], and prove that the truth of the verification conditions it returns suffice to verify the asserted program submitted to the VCG. This theorem stating the VCG's correctness then supports the use of the VCG in proving the correctness of individual programs with complete soundness assured. The VCG automates much of the work and detail involved, relieving the programmer of all but the essential task of proving the verification conditions. This enables proofs of programs which are both effective and trustworthy to a degree not previously seen together.

2 Previous Work

There has been very little work done on proving the correctness of expressions; an exception is Sokolowski's paper on a "term-wise" approach to partial correctness [10]. Even he does not treat expressions with side effects. Side effects appear commonly in "real" programming languages, such as in C, with the operators ++ and get_ch. In addition, several interesting functions are most naturally designed with a side effect; an example is the standard method for calculating random numbers, based on a seed which is updated each time the random number generator is run.

In this paper, we define a "verified" verification condition generator as one which has been proven to correctly produce, for any input program and specification, a set of verification conditions whose truth implies the consistency of the program with its specification. Preferably, this verification of the VCG will be mechanically checked for soundness, because of the many details and deep issues that arise. Many VCG's have been written but not verified; there is then no assurance that the verification conditions produced are properly related to the original program, and hence no security that after proving the verification conditions, the correctness of the program follows. Gordon's work below is an exception in that the security is maintained by the HOL system itself.

Igarashi, London, and Luckham in 1973 gave an axiomatic semantics for a subset of Pascal, and described a VCG they had written in MLISP2 [7]. The soundness of the axiomatic semantics was verified by hand proof, but the correctness of the VCG was not rigorously proven. The only mechanized part of this work was the VCG itself.

Larry Ragland, also in 1973, verified a verification condition generator written in Nucleus, a language Ragland invented to both express a VCG and be verifiable [9]. This was a remarkable piece of work, well ahead of its time. The VCG system consisted of 203 procedures, nearly all of which were less than one page long. These gave rise to approximately 4000 verification conditions. The proof of the generator used an unverified VCG written in Snobol4. The verification conditions it generated were proven by hand, not mechanically. This proof substantially increased the degree of trustworthiness of Ragland's VCG.

Michael Gordon in 1989 did the original work of constructing within HOL a framework for proving the correctness of programs [5]. He introduced new constants in the HOL logic to represent each program construct, defining them as functions directly denoting the construct's semantic meaning. This is known as a "shallow" embedding of the programming language in the HOL logic. The work included defining verification condition generators for both partial and total correctness as tactics. This approach yielded tools which could be used to soundly verify individual programs. However, the VCG tactic he defined was not itself proven. If it succeeded, the resulting subgoals were soundly related to the original correctness goal by the security of HOL itself. Fundamentally, there were certain limitations to the expressive power and proven conclusions of this approach, as recognized by Gordon himself:

> "$\mathcal{P}[\mathcal{E}/\mathcal{V}]$ (substitution) is a meta notation and consequently the assignment axiom can only be stated as a meta theorem. This elementary point is nevertheless quite subtle. In order to prove the assignment axiom as a theorem within higher order logic it would be necessary to have types in the logic corresponding to formulae, variables and terms. One could then prove something like:
>
> $\vdash\ \forall P\ E\ V.\ \textbf{Spec}\ (\textbf{Truth}(\textbf{Subst}(P, E, V)), \textbf{Assign}(V, \textbf{Value}\ E),$
> $\textbf{Truth}\ P)$
>
> It is clear that working out the details of this would be a lot of work." [5]

In 1991, Sten Agerholm [1] used a similar shallow embedding to define the weakest preconditions of a small **while**-loop language, including unbounded nondeterminism and blocks. The semantics was designed to avoid syntactic notions like substitution. Similar to Gordon's work, Agerholm defined a verification condition generator for total correctness specifications as an HOL tactic. This tactic needed additional information to handle sequences of commands and the **while** command, to be supplied by the user.

This paper explores the alternative approach described but not investigated by Gordon. It turns out to yield great expressiveness and control in stating and proving as theorems within HOL concepts which previously were only describable as meta-theorems outside HOL, as above. For example, we are able to prove the assignment axiom that Gordon cannot:

$$\vdash \ \forall\, q\, x\, e.\ \{\ q \lhd [x := e]\ \}\ x := e\ \{q\}$$

where $q \lhd [x := e]$ is a substituted version of q, described later.

To achieve this expressiveness, it is necessary to create a deeper foundation than that used previously. Instead of using an extension of the HOL Object Language as the programming language, we create an entirely new set of datatypes within the Object Language to represent constructs of the programming language and the associated assertion language. This is known as a "deep" embedding, as opposed to the shallow embedding developed by Gordon. This allows a significant difference in the way that the semantics of the programming language is defined. Instead of defining a construct *as* its semantic meaning, we define the construct as simply a syntactic constructor of phrases in the programming language, and then separately define the semantics of each construct in a structural operational semantics [12]. This separation means that we can now decompose and analyze syntactic program phrases at the HOL Object Language level, and thus reason within HOL about the semantics of purely syntactic manipulations, such as substitution or verification condition generation, since they exist *within* the HOL logic.

This has definite advantages because syntactic manipulations, when semantically correct, are simpler and easier to calculate. They encapsulate a level of detailed semantic reasoning that then only needs to be proven once, instead of having to be repeatedly proven for every occurrence of that manipulation. This will be a recurring pattern in this paper, where repeatedly a syntactic manipulation is defined, and then its semantics is described and proven correct within HOL.

3 Higher Order Logic

Higher Order Logic (HOL) [6] is a version of predicate calculus that allows variables to range over functions and predicates. Thus denotable values may be functions of any higher order. Strong typing ensures the consistency and proper meaning of all expressions. The power of this logic is similar to set theory, and it is sufficient for expressing most mathematical theories.

HOL is also a mechanical proof development system. It is secure in that only true theorems can be proved. Rather than attempting to automatically prove theorems, HOL acts as a supportive assistant, mechanically checking the validity of each step attempted by the user.

The primary interface to HOL is the polymorphic functional programming language ML ("Meta Language") [4]; commands to HOL are expressions in ML. Within ML is a second language OL ("Object Language"), representing terms and theorems by ML abstract datatypes **term** and **thm**. A shallow embedding represents program constructs by new OL functions to combine the semantics of the constituents to produce the semantics of the combination. Our approach is to define a *third* level of language, contained within OL as concrete recursive datatypes, to represent the constructs of the programming language PL being studied and its associated assertion language AL. We begin with the definition of variables.

4 Variables and Variants

A variable is represented by a new concrete type `var`, with one constructor, VAR`:string->num->var`. We define two deconstructor functions, Base(VAR *str n*) = *str* and Index(VAR *str n*) = *n*. The number attribute eases the creation of variants of a variable, which are made by (possibly) increasing the number.

All possible variables are considered predeclared of type `num`. In future versions, we hope to treat other data types, by introducing a more complex state and a static semantics for the language which performs type-checking. Some languages distinguish between program variables and logical variables, which cannot be changed by program control. In this simple language, this is unnecessary. In our more recent work with procedure calls, we support logical variables; this is treated in our Category B paper being presented at this conference.

The *variant* function has type `var->(var)set->var`. *variant x s* returns a variable which is a variant of *x*, which is guaranteed not to be in the "exclusion" set *s*. If *x* is not in the set *s*, then it is its own variant. This is used in defining proper substitution on quantified expressions.

The definition of *variant* is somewhat deeper than might originally appear. To have a constructive function for making variants in particular instances, we wanted

variant x s = (*x* IN *s* => *variant* (*mk_variant x* 1) *s* | *x*) (*)

where *mk_variant* (VAR *str n*) *k* = VAR *str* (*n*+*k*). For any finite set *s*, this definition of *variant* will terminate, but unfortunately, it is not primitive recursive on the set *s*, and so does not conform to the requirements of HOL's recursive function definition operator. As a substitute, we wanted to define the *variant* function using `new_specification` by specifying its properties, as

1) (*variant x s*) *is_variant x*, and
2) ~(*variant x s* IN *s*), and
3) ∀z. if (*z is_variant x*) ∧ ~(*z* IN *s*), then Index(*variant x s*) ≤ Index(*z*),

where *y is_variant x* = (Base(*y*) = Base(*x*) ∧ Index(*x*) ≤ Index(*y*)).

But even the above specification did not easily support the proof of the existence theorem, that such a variant existed for any *x* and *s*, because the set of values for *z* satisfying the third property's antecedent is infinite, and we were working strictly with finite sets. The solution was to introduce the function *variant_set*, where *variant_set x n* returns the set of the first *n* variants of *x*, all different from each other, so CARD (*variant_set x n*) = *n*. The definition of *variant_set* is

variant_set x 0 = EMPTY ∧
variant_set x (SUC *n*) = INSERT (*mk_variant x n*) (*variant_set x n*).

Then by the pigeonhole principle, we are guaranteed that there must be at least one variable in *variant_set x* (SUC (CARD *s*)) which is not in the set *s*. This leads to the needed existence theorem. We then defined *variant* with the following properties:

1') (*variant x s*) IN *variant_set x* (SUC (CARD *s*)), and
2') ~(*variant x s* IN *s*), and
3') ∀z.if *z* IN *variant_set x* (SUC (CARD *s*)) ∧ ~(*z* IN *s*),
then Index(*variant x s*) ≤ Index(*z*).

From this definition, we then proved both the original set of properties (1)–(3), and also the constructive function definition given above (*), as theorems.

5 Programming and Assertion Languages

The syntax of the programming language PL is

`exp:`	e	$::=$	$n \mid x \mid {++}x \mid e_1 + e_2 \mid e_1 - e_2$
`bexp:`	b	$::=$	$e_1 = e_2 \mid e_1 < e_2 \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \mid {\sim}b$
`cmd:`	c	$::=$	**skip** $\mid$ **abort** $\mid x := e \mid c_1; c_2 \mid$ **if** b **then** c_1 **else** c_2 $\mid$ **assert** a **while** b **do** c

Table 1. Programming Language Syntax

Most of these constructs are standard. n is an unsigned integer; x is a variable; ++ is the increment operator; **abort** causes an immediate abnormal termination; the **while** loop requires an invariant assertion to be supplied. The notation used above is for ease of reading; each phrase is actually formed by a constructor function, e.g., `ASSIGN:var->exp->cmd` for assignment. We overload the same operator in different languages, asking the reader to disambiguate by context.

The syntax of the associated assertion language AL is

`vexp:`	v	$::=$	$n \mid x \mid v_1 + v_2 \mid v_1 - v_2 \mid v_1 * v_2$
`aexp:`	a	$::=$	**true** $\mid$ **false** $\mid v_1 = v_2 \mid v_1 < v_2 \mid a_1 \wedge a_2 \mid a_1 \vee a_2 \mid {\sim}a \mid a_1 \Rightarrow a_2 \mid a_1 = a_2 \mid a_1 \Rightarrow a_2 \,\vert\, a_3 \mid$ **close** $a \mid \forall x.\ a \mid \exists x.\ a$

Table 2. Assertion Language Syntax

Again, most of these expressions are standard. a_1 => a_2 | a_3 is a conditional expression, yielding the value of a_2 or a_3 depending on the value of a_1. **close** a forms the universal closure of a, which is true when a is true for all possible assignments to its free variables. Again, the notation is for readability; e.g., the constructor `AVAR:var->vexp` creates a `vexp` from a variable.

6 Operational Semantics

The semantics of the programming language is expressed by the following three relations, where a state is a mapping from variables to `num`:

$E\ e\ s_1\ n\ s_2$:	numeric expression e`:exp` evaluated in state s_1 yields numeric value n`:num` and state s_2.
$B\ b\ s_1\ t\ s_2$:	boolean expression b`:bexp` evaluated in state s_1 yields truth value t`:bool` and state s_2.
$C\ c\ s_1\ s_2$:	command c`:cmd` evaluated in state s_1 yields state s_2.

Here is the structural operational semantics [12] of the programming language PL, given as rules inductively defining the three relations E, B, and C. These relations are defined within HOL using Tom Melham's excellent rule induction package [2,8]. The notation $s[v/x]$ indicates the state s updated so that $(s[v/x])(x) = v$.

E	*Number:* $\frac{}{E\,(n)\,s\;n\;s}$ *Variable:* $\frac{}{E\,(x)\,s\;s(x)\;s}$	*Increment:* $\frac{E\;x\;s_1\;n\;s_2}{E\,(\text{++}x)\,s_1\,(n+1)\,s_2[(n+1)/x]}$
	Addition: $\frac{E\;e_1\;s_1 n_1 s_2, \quad E\;e_2\;s_2 n_2 s_3}{E\,(e_1+e_2)\,s_1\,(n_1+n_2)\,s_3}$	*Subtraction:* $\frac{E\;e_1\;s_1 n_1 s_2, \quad E\;e_2\;s_2 n_2 s_3}{E\,(e_1-e_2)\,s_1\,(n_1-n_2)\,s_3}$
B	*Equality:* $\frac{E\;e_1\;s_1 n_1 s_2, \quad E\;e_2\;s_2 n_2 s_3}{B\,(e_1=e_2)\,s_1\,(n_1=n_2)\,s_3}$	*Less Than:* $\frac{E\;e_1\;s_1 n_1 s_2, \quad E\;e_2\;s_2 n_2 s_3}{B\,(e_1<e_2)\,s_1\,(n_1<n_2)\,s_3}$
	Conjunction: $\frac{B\;b_1\;s_1\;t_1\;s_2, \quad B\;b_2\;s_2 t_2 s_3}{B\,(b_1\wedge b_2)\,s_1\,(t_1\wedge t_2)\,s_3}$ *Disjunction:* $\frac{B\;b_1\;s_1\;t_1\;s_2, \quad B\;b_2\;s_2 t_2 s_3}{B\,(b_1\vee b_2)\,s_1\,(t_1\vee t_2)\,s_3}$	*Negation:* $\frac{B\;b\;s_1 t\;s_2}{B\,({\sim}b)\,s_1\,({\sim}t)\,s_2}$
C	*Skip:* $\frac{}{C\;\mathbf{skip}\;s\;s}$	*Conditional:* $\frac{B\;b\;s_1\;\mathrm{T}\;s_2, \quad C\;c_1\;s_2\;s_3}{C\,(\mathbf{if}\;b\;\mathbf{then}\;c_1\;\mathbf{else}\;c_2)\;s_1 s_3}$
	Abort: (no rules)	$\frac{B\;b\;s_1\;\mathrm{F}\;s_2, \quad C\;c_2\;s_2\;s_3}{C\,(\mathbf{if}\;b\;\mathbf{then}\;c_1\;\mathbf{else}\;c_2)\;s_1 s_3}$
	Assignment: $\frac{E\,(e)\;s_1\;n\;s_2}{C\,(x := e)\;s_1 s_2[n/x]}$	*Iteration:* $\frac{B\;b\;s_1\;\mathrm{T}\;s_2, \quad C\;c\;s_2\;s_3 \quad C\,(\mathbf{assert}\;a\;\mathbf{while}\;b\;\mathbf{do}\;c)\;s_3\;s_4}{C\,(\mathbf{assert}\;a\;\mathbf{while}\;b\;\mathbf{do}\;c)\;s_1\;s_4}$
	Sequence: $\frac{C\;c_1 s_1 s_2, \quad C\;c_2 s_2 s_3}{C\,(c_1\,;\,c_2)\;s_1 s_3}$	$\frac{B\;b\;s_1\;\mathrm{F}\;s_2}{C\,(\mathbf{assert}\;a\;\mathbf{while}\;b\;\mathbf{do}\;c)\;s_1\;s_2}$

Table 3. Programming Language Structural Operational Semantics

The semantics of the assertion language AL is given by recursive functions defined on the structure of the construct, in a directly denotational fashion:

$V\,v\,s$: numeric expression v`:vexp` evaluated in state s, yields a numeric value in `num`.
$A\,a\,s$: boolean expression a`:aexp` evaluated in state s, yields a truth value in `bool`.

V	$V\,n\,s = n$ $V\,x\,s = s(x)$ $V\,(v_1 + v_2)\,s = V\,v_1\,s + V\,v_2\,s$ (–, * treated analogously)
A	A **true** s = T A **false** s = F $A\,(v_1 = v_2)\,s = (V\,v_1\,s = V\,v_2\,s)$ (< treated analogously) $A\,(a_1 \wedge a_2)\,s = (A\,a_1\,s \wedge A\,a_2\,s)$ ($\vee$, ~, $\Rightarrow$, a_1=a_2, a_1=>a_2|a_3 treated analogously) A (**close** a) $s = (\forall s_1.\ A\,a\,s_1)$ $A\,(\forall x.\ a)\,s = (\forall n.\ A\,a\,s[n/x])$ $A\,(\exists x.\ a)\,s = (\exists n.\ A\,a\,s[n/x])$

Table 4. Assertion Language Denotational Semantics

7 Substitution

We define proper substitution on assertion language expressions using the technique of *simultaneous substitutions*, following Stoughton [11]. The usual definition of proper substitution is a fully recursive function. Unfortunately, HOL only supports primitive recursive definitions. To overcome this, we use simultaneous substitutions, which are represented by functions of type `subst = var->aexp`. This describes a family of substitutions, all of which are considered to take place simultaneously. This family is in principle infinite, but in practice all but a finite number of the substitutions are the identity substitution ι. The virtue of this approach is that the application of a simultaneous substitution to an assertion language expression may be defined using only primitive recursion, not full recursion, and then the normal single substitution operation of $[v/x]$ may be defined as a special case:

$$[v/x] \;=\; \lambda y.(y{=}x \;\Rightarrow\; v \;\mid\; \mathsf{AVAR}\; y).$$

We apply a substitution by the infix operator ◁. Thus, $a \triangleleft ss$ denotes the application of the simultaneous substitution ss to the expression a, where a can be either `vexp` or `aexp`. Therefore $a \triangleleft [v/x]$ denotes the single substitution of the expression v for the variable x wherever x appears free in a. Finally, there is a dual notion of applying a simultaneous substitution to a state, instead of to an expression; this is called *semantic substitution*, and is defined as $s \triangleleft ss = \lambda y.(V\,(ss\;y)\;s)$.

Most of the cases of the definition of the application of a substitution to an expression are simply the distribution of the substitution across the immediate subexpressions. The interesting cases of the definition of $a \triangleleft ss$ are where a is a quantified expression, e.g.:

$$\begin{array}{rl}(\forall x.\ a) \triangleleft ss = & \textbf{let}\ \mathit{free} = \displaystyle\bigcup_{z \in (FV\,a) - \{x\}} FV\,(ss\ z) \qquad \textbf{in} \\ & \textbf{let}\ y = \mathit{variant}\ x\ \mathit{free}\ \textbf{in} \\ & \forall\, y.\ \ a \triangleleft (ss[(\mathsf{AVAR}\ y)\,/\,x])\end{array}$$

Here *FV* is a function that returns the set of free variables in an expression, and *variant x free* is a function that yields a new variable as a variant of *x*, guaranteed not to be in the set *free*.

Once we have defined substitution as a syntactic manipulation, we can then prove the following two theorems about the semantics of substitution:

$$\vdash \forall v\, s\, ss.\ \ V\,(v \lhd ss)\; s \;=\; V\, v\, (s \lhd ss)$$

$$\vdash \forall a\, s\, ss.\ \ A\,(a \lhd ss)\; s \;=\; A\, a\, (s \lhd ss)$$

This is our statement of the Substitution Lemma of logic, and essentially says that syntactic substitution is equivalent to semantic substitution.

8 Translation

Expressions have typically not been treated in previous work on verification; there are some exceptions, notably Sokolowski [10]. Expressions with side effects have been particularly excluded. Since expressions did not have side effects, they were often considered to be a sublanguage, common to both the programming language and the assertion language. Thus one would see expressions such as $p \wedge b$, where p was an assertion and b was a boolean expression from the programming language.

One of the key realizations of this work was the need to carefully distinguish these two languages, and not confuse their expression sublanguages. This then requires us to *translate* programming language expressions into the assertion language before the two may be combined as above. In fact, since we allow expressions to have side effects, there are actually two results of translating a programming language expression e:

- an assertion language expression, representing the value of e in the state "before" evaluation, *and*
- a simultaneous substitution, representing the change in state from "before" evaluating e to "after" evaluating e.

For example, the translator for numeric expressions is defined using a helper function:

VE1: `exp -> subst -> (aexp # subst)`:

$VE1\ (n)\ ss$	=	$n,\ ss$	*(where comma (,) makes a pair)*
$VE1\ (x)\ ss$	=	$ss\ x,\ ss$	
$VE1\ (\texttt{++}x)\ ss$	=	$(ss\ x) + 1,\ ss[((ss\ x) + 1) / x]$	
$VE1\ (e_1 + e_2)\ ss$	=	$(VE1\ e_1 \rightarrow \lambda v_1.\ (VE1\ e_2 \rightarrow \lambda v_2\ ss_2.\ (v_1 + v_2,\ ss_2)))\ ss$	
$VE1\ (e_1 - e_2)\ ss$	=	$(VE1\ e_1 \rightarrow \lambda v_1.\ (VE1\ e_2 \rightarrow \lambda v_2\ ss_2.\ (v_1 - v_2,\ ss_2)))\ ss$	

where $\rightarrow$ is a "translator continuation" operator, defined as

$$(f \rightarrow k)\ ss \;=\; \textbf{let}\ (v, ss') = f\ ss\ \ \textbf{in}\ \ k\ v\ ss'$$

Then define

$VE\ e \;=\; \text{fst}\ (\ VE1\ e\ \iota\ )$ *(where* ι *is the identity substitution*

$VE\_state\ e \;=\; \text{snd}(\ VE1\ e\ \iota\ )$ *and* fst *and* snd *select the members of a pair)*

We can then prove that these translation functions, as syntactic manipulations, are semantically correct, according to the following theorem:

$$\vdash\ \forall\, e\, s_1\, n\, s_2.\ (E\, e\, s_1\, n\, s_2) = (\ \ n = V(\,VE\, e)\ s_1 \quad \wedge \quad s_2 = s_1 \lhd (VE\_state\ e)\,)$$

A similar set of functions are used to translate boolean expressions. We define the helper function *AB1* and the main translation functions *AB* and *AB_state*, and prove their correctness as

$$\vdash \ \forall\, b\, s_1\, t\, s_2.\ (B\, b\, s_1\, t\, s_2) = (\ \ t = A\,(AB\, b)\ s_1 \ \ \wedge \ \ s_2 = s_1 \lhd (AB\_state\ b)\,)$$

These theorems mean that every evaluation of a programming language expression has its semantics completely captured by the two translation functions for its type. These are essentially small compiler correctness proofs.

As a product, we may now define the simultaneous substitution that corresponds to an assignment statement, overriding the expression's state change with the change of the assignment:

$$[x := e] \ = \ (VE\_state\ e)[(VE\ e)\ /\ x]$$

9 Axiomatic Semantics

We define the semantics of Floyd/Hoare partial correctness formulae as follows:

`aexp:`	$\{a\}$	=	**close** a	(the universal closure of a)
		=	$\forall s.\ A\ a\ s$	(a is true in all states)
`exp:`	$\{p\}e\{q\}$	=	$\forall p\ q\ e\ n\ s_1\ s_2.\ \ A\ p\ s_1 \wedge E\ e\ s_1\ n\ s_2 \Rightarrow A\ q\ s_2$	
`bexp:`	$\{p\}b\{q\}$	=	$\forall p\ q\ b\ t\ s_1\ s_2.\ \ A\ p\ s_1 \wedge B\ b\ s_1\ t\ s_2 \Rightarrow A\ q\ s_2$	
`cmd:`	$\{p\}c\{q\}$	=	$\forall p\ q\ c\ s_1\ s_2.\ \ A\ p\ s_1 \wedge C\ c\ s_1\ s_2 \Rightarrow A\ q\ s_2$	

Table 5. Floyd/Hoare Partial Correctness Formulae Semantics

Given these formulae, we can now express the axiomatic semantics of the programming language, and *prove* each rule as a theorem from the previous structural operational semantics:

Skip:

$$\frac{}{\{q\}\ \textbf{skip}\ \{q\}}$$

Abort:

$$\frac{}{\{\textbf{false}\}\ \textbf{abort}\ \{q\}}$$

Assignment:

$$\frac{}{\{q \lhd [x := e]\}\ x := e\ \{q\}}$$

Sequence:

$$\frac{\{p\}\ c_1\ \{r\}, \qquad \{r\}\ c_2\ \{q\}}{\{p\}\ c_1\ ;\ c_2\ \{q\}}$$

Conditional:

$$\frac{\begin{array}{c}\{p \wedge AB(b)\}\ b\ \{r_1\} \\ \{p \wedge \sim\! AB(b)\}\ b\ \{r_2\} \\ \{r_1\}\ c_1\ \{q\} \\ \{r_2\}\ c_2\ \{q\}\end{array}}{\{p\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{q\}}$$

Iteration:

$$\frac{\begin{array}{c}\{a \wedge AB(b)\}\ b\ \{p\} \\ \{a \wedge \sim\! AB(b)\}\ b\ \{q\} \\ \{p\}\ c\ \{a\}\end{array}}{\{a\}\ \textbf{assert}\ a\ \textbf{while}\ b\ \textbf{do}\ c\ \{q\}}$$

Table 6. Programming Language Axiomatic Semantics

The most interesting of these proofs was that of the **while**-loop rule. It was necessary to prove a subsidiary lemma first, by the strong version of rule induction for command semantics provided by Tom Melham's rule induction package. This lemma thus used versions of itself for "lower levels" in the relation built up by rule induction to prove each instance, and so needed strong induction to present as a usable assumption each hypothesized lower-level tuple in the relation. The subsidiary lemma was necessary because the **while**-loop rule as a theorem was not in the right syntactic form for the induction tactic. The lemma we proved is

$$\begin{array}{l} \vdash\ \forall a\, b\, c\, p\, q.\ \{p\}c\{a\} \ \wedge\ \{a \wedge (AB\ b)\}b\{p\} \ \wedge\ \{a \wedge \sim\!(AB\ b)\}b\{q\} \ \Rightarrow \\ \qquad \forall w\, s_1\, s_4.\ C\ w\ s_1\ s_4 \Rightarrow \\ \qquad\qquad ((w\ =\ \textbf{assert}\ a\ \textbf{while}\ b\ \textbf{do}\ c\ ) \Rightarrow (A\ a\ s_1 \Rightarrow A\ q\ s_4)) \end{array}$$

Although we did prove analogous theorems as an axiomatic semantics for both the numeric and boolean expressions in the programming language, it turned out that there was a better way to handle them provided through the use of the translation functions. Using these translation functions, we may define functions to compute the appropriate precondition to an expression, given the postcondition, as follows.

`vexp`	ve_pre $e\ v$ = $v \lhd (VE\_state\ e)$ vb_pre $b\ v$ = $v \lhd (AB\_state\ b)$
`aexp`	ae_pre $e\ a$ = $a \lhd (VE\_state\ e)$ ab_pre $b\ a$ = $a \lhd (AB\_state\ b)$

Table 7. Expression Precondition Functions

We may now prove the following axiomatic semantics for expressions:

Numeric expression precondition:	*Boolean expression precondition:*
$\dfrac{}{\{\text{ae\_pre}\ e\ q\}\ e\ \{q\}}$	$\dfrac{}{\{\text{ab\_pre}\ b\ q\}\ b\ \{q\}}$

Table 8. Programming Language Expression Axiomatic Semantics

These precondition functions now allow us to revise the rules of inference for conditionals and loops, as follows.

Conditional:

$$\frac{\{r_1\}\ c_1\ \{q\}, \qquad \{r_2\}\ c_2\ \{q\}}{\{AB\ b\ \ \text{=>}\ \ \text{ab\_pre}\ b\ r_1 \mid \text{ab\_pre}\ b\ r_2\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{q\}}$$

Iteration:

$$\frac{\begin{array}{c}\{a \wedge AB\ b \Rightarrow \text{ab\_pre}\ b\ p\} \\ \{a \wedge \sim\!(AB\ b) \Rightarrow \text{ab\_pre}\ b\ q\} \\ \{p\}\ c\ \{a\}\end{array}}{\{a\}\ \textbf{assert}\ a\ \textbf{while}\ b\ \textbf{do}\ c\ \{q\}}$$

Table 9. Programming Language Axiomatic Semantics (revisions)

10 Verification Condition Generator

We now define a verification condition generator for this programming language. To begin, we first define a helper function *vcg1*, of type `cmd->aexp->(aexp # (aexp)list)`. This function takes a command and a postcondition, and returns a precondition and a list of verification conditions that must be proved in order to verify that command with respect to the precondition and postcondition. This function does most of the work of calculating verification conditions.

This is called by the main verification condition generator function, *vcg*, defined with type `aexp—>cmd—>aexp—>(aexp)list`. *vcg* takes a precondition, a command, and a postcondition, and returns a list of the verification conditions for that command.

In these definitions, comma (,) makes a pair of two items, square brackets ([]) delimit lists, semicolon (;) within a list separates elements, and ampersand (&) is an infix version of HOL's `APPEND` operator to join two lists.

vcg1	*vcg1* (**skip**) *q* = *q*, []
	vcg1 (**abort**) *q* = **true**, []
	vcg1 (*x* := *e*) *q* = *q*◁[*x* := *e*], []
	vcg1 (c_1 ; c_2) *q* = **let** (r,h_2) = *vcg1* c_2 *q* **in**
	let (p,h_1) = *vcg1* c_1 *r* **in**
	p, (h_1 & h_2)
	vcg1 (**if** *b* **then** c_1 **else** c_2) *q* =
	let (r_1,h_1) = *vcg1* c_1 *q* **in**
	let (r_2,h_2) = *vcg1* c_2 *q* **in**
	(*AB b* => ab_pre *b* r_1 | ab_pre *b* r_2), (h_1 & h_2)
	vcg1 (**assert** *a* **while** *b* **do** *c*) *q* =
	let (*p*,*h*) = *vcg1 c a* **in**
	a, [*a* ∧ *AB b* ⇒ ab_pre *b p* ;
	a ∧ ~(*AB b*) ⇒ ab_pre *b q*] & *h*
vcg	*vcg p c q* = **let** (*r*,*h*) = *vcg1 c q* **in**
	[*p* ⇒ *r*] & *h*

Table 10. Verification Condition Generator

The correctness of these VCG functions is established by proving the following theorems from the axioms and rules of inference of the axiomatic semantics:

`VCG1_THM`	⊢ ∀*c q*.	**let** (*p*,*h*) = *vcg1 c q* **in** (**every close** *h* ⇒ {*p*}*c*{*q*})
`VCG_THM`	⊢ ∀*p c q*.	**every close** (*vcg p c q*) ⇒ {*p*}*c*{*q*}

Table 11. Verification of Verification Condition Generator

every *P lst* is defined in HOL as being true when for every element *x* in the list *lst*, the predicate *P* is true when applied to *x*. Accordingly, **every close** *h* means that the universal closure of every verification condition in *h* is true.

These theorems are proven from the axiomatic semantics by induction on the structure of the command involved. This verifies the VCG. It shows that the *vcg* function is *sound*, that the correctness of the verification conditions it produces suffice to establish the correctness of the annotated program. This does not show that the *vcg* function is *complete*, that if a program is correct, then the *vcg* function will produce a set of verification conditions sufficient to prove the program correct from the axiomatic semantics [3]. However, this soundness result is quite useful, in that we may directly apply these theorems in order to prove individual programs partially correct within HOL, as seen in the next section.

11 Example Programs

Given the *vcg* function defined in the last section and its associated correctness theorem, proofs of program correctness may now be partially automated with security. This has been implemented in an HOL tactic, called `VCG_TAC`, which transforms a given program correctness goal to be proved into a set of subgoals which are the verification conditions returned by the *vcg* function. These subgoals are then proved within the HOL theorem proving system, using all the power and resources of that theorem prover, directed by the user's ingenuity.

As an example, we will take the quotient/remainder algorithm for integer division by repeated subtraction. The program to be verified, with the annotations of the loop invariant and pre- and postconditions, is

```
{$x0 = x \wedge y0 = y$}
    $r := x$;
    $q := 0$;
    assert $x0 = q * y0 + r \wedge y0 = y$
    while $\sim(r < y)$ do
        $r := r - y$;
        $q := {++}q$
    od
{$x0 = q * y0 + r \wedge r < y0$}
```

With the assumption that the program cannot change the values of *x0* and *y0*, this specification means that if the program terminates, the final value of *q* must be the quotient of the division of *x0* by *y0*, and *r* the remainder. (We could also prove that the final values of *x* and *y* are unchanged by this algorithm.)

Applying `VCG_TAC` to this goal produces the following three verification conditions:

VC1: $x0 = x \wedge y0 = y \Rightarrow x0 = 0 * y0 + x \wedge y0 = y$

VC2: $x0 = q * y0 + r \wedge y0 = y \wedge \sim(r < y)$
$\Rightarrow x0 = (q + 1) * y0 + (r - y) \wedge y0 = y$

VC3: $x0 = q * y0 + r \wedge y0 = y \wedge r < y$
$\Rightarrow x0 = q * y0 + r \wedge r < y0$

Here is a transcript of the application of `VCG_TAC` to this problem. We have written a parser for the subject language, using the parser library in HOL, invoked using the delimiters "[[" and "]]". The partial correctness goal is parsed and converted into the abstract syntax form used internally, which is printed as the current goal. `VCG_TAC` then converts that goal into verification conditions in the Object Language of HOL.

```
#g [[ {x0 = x /\ y0 = y}
#        r := x;
#        q := 0;
#        assert x0 = q * y0 + r /\ y0 = y
#        while ~(r < y) do
#           r := r - y;
#           q := ++q
#        od
#     {x0 = q * y0 + r /\ r < y0}
#  ]];;
"CSPEC
 (AAND
  (AEQ(AVAR(VAR `x0` 0))(AVAR(VAR `x` 0)))
  (AEQ(AVAR(VAR `y0` 0))(AVAR(VAR `y` 0))),
  SEQ
  (SEQ
   (ASSIGN(VAR `r` 0)(PVAR(VAR `x` 0)))
   (ASSIGN(VAR `q` 0)(NUM 0)))
  (WHILE
   (AAND
    (AEQ
     (AVAR(VAR `x0` 0))
     (APLUS
      (AMULT(AVAR(VAR `q` 0))(AVAR(VAR `y0` 0)))
      (AVAR(VAR `r` 0))))
    (AEQ(AVAR(VAR `y0` 0))(AVAR(VAR `y` 0))))
   (NOT(LESS(PVAR(VAR `r` 0))(PVAR(VAR `y` 0))))
   (SEQ
    (ASSIGN(VAR `r` 0)(MINUS(PVAR(VAR `r` 0))(PVAR(VAR `y` 0))))
    (ASSIGN(VAR `q` 0)(INC(VAR `q` 0))))),
  AAND
  (AEQ
   (AVAR(VAR `x0` 0))
   (APLUS
    (AMULT(AVAR(VAR `q` 0))(AVAR(VAR `y0` 0)))
    (AVAR(VAR `r` 0))))
  (ALESS(AVAR(VAR `r` 0))(AVAR(VAR `y0` 0))))"

() : void
Run time: 6.1s

#e(VCG_TAC);;
OK..
3 subgoals
"!x0 q y0 r y.
  ((x0 = (q * y0) + r) /\ (y0 = y)) /\ r < y ==>
  (x0 = (q * y0) + r) /\ r < y0"

"!x0 q y0 r y.
  ((x0 = (q * y0) + r) /\ (y0 = y)) /\ ~r < y ==>
  (x0 = ((q + 1) * y0) + (r - y)) /\ (y0 = y)"

"!x0 x y0 y.
  (x0 = x) /\ (y0 = y) ==> (x0 = (0 * y0) + x) /\ (y0 = y)"

() : void
Run time: 80.3s
Intermediate theorems generated: 5643
```

These verification conditions are each solved as a subgoal by normal HOL techniques.

The Object Language variables involved in these verification conditions are constructed to have names similar to the original program variable names; if there is a non-zero variant number, it is appended to the variable name. Thus, if one changed the name of program variable x to z in the example above, the verification conditions would be the same but with the OL variable z in place of x.

Here is the HOL definition of the `VCG_TAC` tactic:

```
         let VCG_TAC =
(a)          MATCH_MP_TAC vcg_THM
(b)          THEN REWRITE_TAC[vcg;vcg1]
             THEN CONV_TAC (DEPTH_CONV let_CONV)
(c)          THEN REWRITE_TAC[ab_pre;assign]
(d)          THEN REPEAT (CHANGED_TAC
                     (BETA_TAC THEN
                      REWRITE_TAC[VE1_DEF;VE_DEF;VE_state_DEF;
                                  AB1_DEF;AB_DEF;AB_state_DEF;
                                 IDENT_SS_var;trans_cont]))
             THEN REWRITE_TAC[a_subst_IDENT]
(e)          THEN CONV_TAC vcg_CONV
(f)          THEN REWRITE_TAC[APPEND_INFIX;APPEND;EVERY_DEF;CLOSE]
(g)          THEN CONV_TAC (TOP_DEPTH_CONV INTERPRET_aexp_CONV)
(h)          THEN REWRITE_TAC[V_DEF]
             THEN CONV_TAC (DEPTH_CONV var_BND_CONV)
             THEN REPEAT CONJ_TAC
             THEN ( GEN_TAC ORELSE ALL_TAC )
(i)          THEN INTERPRET_PROG_VARS_TAC;;
```

The `VCG_TAC` tactic first (a) applies the theorem `VCG_THM`, the last theorem of Table 11 of the previous section, to the current goal using the HOL tactic `MATCH_MP_TAC` to reason backwards from the program correctness statement to the invocation of the *vcg* function. By the theorem, the proof of these verification conditions will establish the proof of the original program correctness statement.

The next step of `VCG_TAC` is to "execute" the various syntactic manipulation functions mentioned in the current goal by symbolically rewriting the goal using the definitions of the functions. This applies (b) to the *vcg* function, (c) to the operators that create substitutions, (d) to the translation functions, (e) to the substitution functions, and others. Because the rewriting process is done symbolically, instead of actually executing a program, it is relatively slow, but complete soundness is assured. This "execution" converts the invocation of the *vcg* function on the annotated program into the actual set of verification conditions that the *vcg* function returns.

The tactic makes use at (e) of a set of conversions, culminating in `vcg_CONV`, to test the equality of variables (`var_EQ_CONV`), lookup a variable in a simultaneous substitution (`var_BND_CONV`), calculate a variant of a variable (`variant_CONV`), apply a substitution to an expression (`subst_CONV`), and reduce a term with nested "let" and substitution operators in an efficient order (`vcg_CONV`), among others.

After performing these conversions, the program correctness goal is left as a set of "constant" verification conditions in the assertion language. `VCG_TAC` then (f–i) uses the definitions of the semantics of the assertion language to rewrite these verification conditions into equivalent statements in the Object Language of HOL, beginning with (f) the definition of **close**, then proceeding with (g) the definitions of A and (h) V. In particular, (g) all quantification over assertion language variables, and (i) all references to assertion language variables within program states, are converted to references to similarly-named OL variables. These verification conditions are then presented to the user as the necessary subgoals that need to be solved in order to complete the proof of the program originally presented.

12 Future Work

In the future, we intend to extend this work to include several more language features, principally mutually recursive procedures and concurrency. In addition, we also intend to cover total correctness, beyond the partial correctness issues dealt with in this paper.

The work on mutually recursive procedures requires many new concepts and techniques to define the semantics and perform verification condition generator proofs. These include declarations of procedures, their collection into environments, their verification independent of actual use of the procedures, well-formedness conditions on programs, and the very delicate issue of parameter passing. We wish to find a method of proving the total correctness of systems of mutually recursive procedures which is efficient and suitable for processing by a VCG.

Concurrency raises a whole host of new issues, ranging from the level of structural operational semantics ("big-step" versus "small-step"), to dealing with assertions describing temporal sequences of states instead of single states, to issues of fairness. We believe that a proper treatment of concurrency will exhibit qualities of modularity and compositionality. *Modularity* means that a specification for a process should state both (a) the assumptions under which it should operate, and (b) the task (or commitment) which it should meet, given those assumptions. *Compositionality* means that the specification of a system of processes should be verifiable in terms of the specifications of the individual constituent processes.

13 Summary and Conclusions

The fundamental contribution of this work is the exhibition of a tool to ease the task of proving programs which is itself proven to be sound. This verification condition generator tool performs an automatic, syntactic transformation of the annotated program into a set of verification conditions. The verification conditions produced are themselves proven within HOL, establishing the correctness of the program within the same system wherein the VCG was verified.

This proof of the correctness of the VCG may be considered as an instance of a compiler correctness proof, with the VCG translating from annotated programs to lists of verification conditions. Each of these has its semantics defined, and the VCG correctness theorem closes the commutative diagram, showing that the truth of the verification conditions implies the truth of the annotated program.

The programming language and its associated assertion language are represented by new concrete recursive datatypes. This implies that they are completely independent of other data types and operations existing in the HOL system, without any hidden associations that might affect the validity of proof. This requires substantial work in defining their semantics and in proving the axioms and rules of inference of the axiomatic semantics from the operational semantics. However, this deeply embedded approach yields great expressiveness, ductility, and the ability to prove as theorems within HOL the correctness of various syntactic manipulations, which could only be stated as meta-theorems before. These theorems encapsulate a level of reasoning which now does not need to be repeated every time a program is verified, raising the level of proof from the semantic level to the syntactic. But the most important part of this work is the degree of trustworthiness of this syntactic reasoning. Verification condition generators are not new, but we are not aware of any other proofs of their correctness to this level of rigor. This enables program proofs which are both trustworthy and effective to a degree not previously seen together.

References

1. Sten Agerholm, “Mechanizing Program Verification in HOL”, in *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, Davis, August 1991*, edited by M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley (IEEE Computer Society Press, 1992), pp. 208–222.

2. J. Camilleri and T. Melham, “Reasoning with Inductively Defined Relations in the HOL Theorem Prover”, Technical Report No. 265, University of Cambridge Computer Laboratory, August 1992.

3. Stephen A. Cook, “Soundness and Completeness of an Axiom System for Program Verification”, in *SIAM Journal on Computing*, Vol. 7, No. 1, February 1978, pp. 70–90.

4. G. Cousineau, M. Gordon, G. Huet, R. Milner, L. Paulson, and C. Wadsworth, *The ML Handbook* (INRIA, 1986).

5. Michael J. C. Gordon, “Mechanizing Programming Logics in Higher Order Logic”, in *Current Trends in Hardware Verification and Automated Theorem Proving*, ed. P.A. Subrahmanyam and Graham Birtwistle, Springer-Verlag, New York, 1989, pp. 387–439.

6. Michael J. C. Gordon, and T. F. Melham, *Introduction to HOL, A theorem proving environment for higher order logic*, Cambridge University Press, Cambridge, 1993.

7. S. Igarashi, R. L. London, and D. C. Luckham, “Automatic Program Verification I: A Logical Basis and its Implementation”, *ACTA Informatica* 4, 1975, pp. 145–182.

8. Tom Melham, “A Package for Inductive Relation Definitions in HOL”, in *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, Davis, August 1991*, edited by M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley (IEEE Computer Society Press, 1992), pp. 350–357.

9. L. C. Ragland, “A Verified Program Verifier”, Technical Report No. 18, Department of Computer Sciences, University of Texas at Austin, May 1973.

10. Stefan Sokolowski, “Partial Correctness: The Term-Wise Approach”, *Science of Computer Programming*, Vol. 4, 1984, pp. 141–157.

11. Allen Stoughton, “Substitution Revisited”, *Theoretical Computer Science*, Vol. 59, 1988, pp. 317–325.

12. Glynn Winskel, *The Formal Semantics of Programming Languages, An Introduction*, The MIT Press, Cambridge, Massachusetts, 1993.