actinf manuscript No.
(will be inserted by the editor)

Andrew M Gravell

Verification Conditions Are Code

the date of receipt and acceptance should be inserted later

Abstract This paper presents a new theoretical result concerning Hoare Logic. It is shown here that the verification conditions that support a Hoare Logic program derivation are themselves sufficient to construct a correct implementation of the given pre-, post- condition specification. This property is mainly of theoretical interest, though it is possible that it may have some practical use, for example if predicative programming methodology is adopted. The result is shown to hold for both the original, partial correctness, Hoare Logic, and also a variant for total correctness derivations.

1 Introduction

1.1 Background and Motivation

Hoare logic, and variants thereof, have been in use since Hoare's original paper [20]. In this formal system, a correctness proof typically depends on some verification conditions of the form A $\Rightarrow$ B. These must themselves be proved, using some system of predicate calculus, in order to show that the Hoare logic derivation is correct. The result here concerns the verification conditions that support a given Hoare logic derivation. Surprisingly, these alone (together with the final pre- and post-condition) are sufficient to allow a full derivation and program to be constructed.

Let us motivate this result by giving an example. Consider the following program [28] to calculate the integer square root of a natural number s.

University of Southampton, UK, E-mail: gravell@soton.ac.uk

$\{s \geq 0\}$
$q, r := s + 1, 0$
while $r + 1 \neq q$ **do**
  $p := (q + r) \div 2$
  if $s < p^2$ **then** $q := p$ **else** $r := p$ **fi**
end
$\{r^2 \leq s < (r + 1)^2\}$

The correctness of this program depends on four verification conditions

$s \geq 0 \Rightarrow 0^2 \leq s < (s + 1)^2$
$r^2 \leq s < q^2 \wedge \neg(r + 1 \neq q) \Rightarrow r^2 \leq s < (r + 1)^2$
$r^2 \leq s < q^2 \wedge r + 1 \neq q \wedge s < p^2 \Rightarrow r^2 \leq s < p^2$
$r^2 \leq s < q^2 \wedge r + 1 \neq q \wedge \neg(s < p^2) \Rightarrow p^2 \leq s < q^2$

Given these conditions, the assignment statements can be derived by finding substitutions that convert one clause to another. These can be written as Hoare triples such as

$\{0^2 \leq s < (s + 1)^2\}\ q, r := s + 1, 0\ \{r^2 \leq s < q^2\}$

and

$\{r^2 \leq s < p^2\}\ q := p\ \{r^2 \leq s < q^2\}$

These Hoare triples can then be combined using the laws of Hoare logic to construct the correctness argument, and hence the original program itself. Thus the verification conditions can be said to contain the essence of the correctness argument.

The main result of this paper will be to show that there is an algorithm to construct this program and its proof given as input the pre-, post-, and verification conditions. Alternatively, if the given conditions do not form the basis of a Hoare logic derivation, the algorithm will terminate and report that no such program exists. That is to say, it is a decision procedure.

Of course, there are other partially correct programs also based on the same verification conditions. In particular

$\{s \geq 0\}$
$q, r := s + 1, 0$
while $r + 1 \neq q$ **do**
  if $s < p^2$ **then** $q := p$ **else** $r := p$ **fi**
end
$\{r^2 \leq s < (r + 1)^2\}$

is a derivable theorm of Hoare logic. That is to say, the assignment to p is redundant when considering only partial correctness.

The algorithm described here will search for an implementation that is in some sense minimal, using only necessary assignments. There is also a variant of the algorithm that deals with total correctness.

A particular appeal of Hoare logic [20] is its simplicity. Systems with few rules are most amenable to proof theoretic approaches, which typically require structural induction (or equivalent) with at least one case per axiom or rule. Various extensions of Hoare logic have been published to cover programming language features such as procedures [21] and also including an axiomatic semantics for the Pascal programming language [22].

More recently, formal methods researchers have developed refinement calculi [4, 29, 28]. These are more suitable for top-down program derivation, with

more laws so that derivations can be abbreviated, which means they are not as well suited to the proof theoretic methods used here. In general, also, refinement calculus researchers [5] have adopted a model-theoretic approach, focussing on semantics, not axioms. The relationship between Hoare logic and refinement calculus is explained by [1], and a unifying framework for both these systems (and others) is presented by [24].

Note that as this paper is concerned with Hoare logic, the word "proof" usually refers to a Hoare logic derivation of a Hoare triple.

Finally, I would like to acknowledge the helpful and precise comments and criticisms of the anonymous referees.

1.2 Notation and Definitions

Hoare logic is, in effect, an extension of a base predicate logic, which is assumed here to be classical and first order. Issues such as undefined expressions and type correctness are not considered here, although alternative forms of the base predicate logic are discussed briefly in section 3.4. For simplicity, it is assumed here that all programming language tests used in if and while statements are (syntactically identical to) logical predicates (though not necessarily vice versa). Similarly, no distinction is made between assignment statements and substitutions. To avoid syntactic ambiguities such as the dangling else, the keyword fi is hereafter used to terminate each if statement, and end to terminate each while statement.

Other notations and notational conventions used in this paper are:

$\alpha, \beta, \gamma, \alpha'$	assignment statements
A, B, C, A′	predicates/assertions/conditions
α(A)	syntactic substitution
P, Q, R, P′	programs and sub-programs
A P B, C Q D	Hoare triples (cf. the conventional {A} P {B} etc.)
V, V′, W	sets of conditions

Note that since sets of conditions are not typically nested it is convenient to omit the usual braces. For example if V = A,B and V′ = V,C then V′ = A,B,C. Finally note that, although unconventional, omitting the braces from Hoare triples simplifies and abbreviates the later presentation.

In this paper it is assumed that substitutions can be multiple (also known as *parallel assignments*) [15]. For example $x, y := y, x$ is a classic multiple assignment that swaps the values of variables x and y.

The classical form of Hoare logic, using these notational conventions, is given in Figure 1 below. Note that there are many variants, but this (minimal) version is that used for example to show completeness of Hoare logic [9].

1.3 Basic Properties of Classical Hoare Logic

Some basic properties of classical Hoare logic are now presented.

Lemma 1 For any programs P, Q, R, conditions A, B, the Hoare triple A P;(Q;R) B is derivable if and only if A (P;Q);R B is.

Proof

A P;(Q;R) B is derivable iff
A P C and C Q;R B for some condition C, iff
A P C and C Q D and D R B for some conditions C, D, iff
A P;Q D and D R B for some condition D, iff
A (P;Q);R B
□

Note that the final step in the derivation of A P;(Q;R) B might involve an example of one of the consequence laws, such as

A P; (Q;R) B′, B′ ⇒ B ⊢ A P; (Q;R) B

This does not invalidate the argument above, however, since C (Q;R) B can also be derived from C (Q;R) B′ and B′ ⇒ B.

Also note that this property is very familiar from the study of program semantics, for example in the theory of predicate transformers, where this result would follow directly from the associativity of function composition. Working directly with program semantics is a model-theoretic approach, however, in contrast to the proof-theoretic approach adopted here. Where a property such as the one above can be expressed both model- and proof-theoretically, it is typically the case that the property can be established more easily using model-theoretic techniques. The main theorem presented here, however, concerns verification conditions, which have no equivalent in standard program semantics, so the proof-theoretic approach is more natural.

An important property of Hoare logic is the substitution principle. This is also well known in programming logics and semantics.

Lemma 2 (Substitutability) Suppose programs P and Q are such that for any conditions A and B, A P B is a derivable Hoare triple whenever A Q B is. Then program P can validly replace program Q in any Hoare proof.

Proof The Hoare triple A Q B must appear in the Hoare proof at some point. The Hoare triple A P B must also be derivable by the hypothesis. Thereafter, any proof steps that involve the use of Q as a sub-program (via the use of the sequence, conditional or iteration law) can be replaced by the equivalent proof step, but with P replacing Q. □

The model-theoretic equivalent of this substitution principle is classically called monotonicity of program refinement. For example, combining lemmas 1 and 2, the programs (P;Q);R and P;(Q;R) can freely replace each other in any Hoare logic derivation. Given this result, and the conventions given above, it is therefore safe to omit brackets and semi-colons altogether. For example, P Q is an abbreviation for the sequence P;Q. Similarly, P Q R abbreviates the program (P;Q);R, which as noted above is essentially equivalent to P;(Q;R).

⊢ α(A) α A	assignment introduction
A ⇒ B, B P C ⊢ A P C	consequence: pre-condition
A P B, B ⇒ C ⊢ A P C	consequence: post-condition
A P B, B Q C ⊢ A P;Q C	sequence
A∧B P C, A∧¬B Q C ⊢ A if B then P else Q fi C	selection
A∧B P A ⊢ A while B do P end A∧¬ B	iteration

Fig. 1 Classical Hoare Logic

Definition: a (Hoare) proof is *based on* or *derived using* a set of predicates V if and only if every instance of the rules of consequence (e.g. A ⇒ B, B P C ⊢ A P C) in the proof uses a predicate (here A ⇒ B) that is a member of V.

Definition: if P can replace Q and Q can replace P in any proof then P and Q are said to be *proof-equivalent*.

Lemma 3 if C then P else Q fi R is proof equivalent to if C then P R else Q R fi

Proof Any Hoare proof of the former must have steps such as

. . .
D R B
A∧C P D
A∧¬C Q D
A if C then P else Q fi D
A if C then P else Q fi R B

The final two inferences can equivalently be replaced by

A∧C P R B
A∧¬C Q R B
A if C then P R else Q R fi B

The converse implication is shown similarly.

□

1.4 Standard Proof-Theoretic Definitions

Some standard proof-theoretic words and phrases are now defined.

Definition: suppose A = α(B) for some substitution α. Then A is said to be an *instance* or a *specialisation* of B. B is a *generalisation* of A. Note that being an instance is a transitive relation.

Definition: also, a *renaming* is an invertible substitution: one that maps variables to variables.

Definition: the *logical matrix* of a predicate A is the syntax tree showing the structure of its Boolean operators, but ignoring the atomic formulas at the leaves. For example both B ∧ (C ∨ D) and $x = y \wedge (z > 2 \vee \forall t \bullet x(t) > 0)$ have the same logical matrix, $\_ \wedge (\_ \vee \_)$.

Definition: a *part-formula* of a predicate is any (well-formed) formula obtained by any number of applications of the following transformations: 1) replace ¬A by A; 2) replace A op B by A, for any binary boolean operator op (∨, ∧, ⇒, and so on); or 3) replace A op B by B, for any binary boolean operator op as above.

Thus the four part-formulas of C∧¬D are C∧D, C, ¬D, and D, assuming both C and D are atomic formulas. Note that being a part-formula is a transitive relation. Also note that the part-formula and instance relations commute (an instance of a part-formula is a part-formula of an instance, and vice versa).

2 Equalisation

Our basic strategy is to search for programs which use part-formulas of the given verification conditions as the intermediate pre- and post-conditions, for example the tests that occur in conditional or loop statements. There are, of course, only finitely many part-formulas of a given formula. There are, however, infinitely many instances. To limit the search space, we need a way to limit the amount of substitution required.

Let us assume therefore that predicates are ordered by i) the substitution ordering (A $\leq \alpha$(A)) or else ii) lexicographically in the case of predicates which differ only by a renaming. Note that this ordering is well-founded, so that no decreasing chain of predicates can exist.

2.1 Equalising Substitutions

Definition: *mgci*(V) gives the most general common instance of a set of conditions V, where such an instance exists. Thus A $\leq$ *mgci*(A,B), B $\leq$ *mgci*(A,B), and *mgci*(A,B) $\leq$ C for any C such that A $\leq$ C and B $\leq$ C.
Definition: *mges*(A, B) gives the most general equalising substitutions to apply to a pair of conditions A, B that result in *mgci*(A, B) where this exists. For example if *mges*(A, B) = (α, β) then α(A) = β(B) = *mgci*(A, B).

These definitions are closely related to the notion of unifiers and unification in logic programming. The main difference is that unification applies a single substitution to a set of conditions (or more typically terms) to give a single common instance. A single substitution is a natural notion in the context of declarative programming languages, which are referentially transparent. Imperative languages are not referentially transparent, so it is not surprising that multiple equalising substitutions are required.

The relationship between these notions is as follows. To find the most general common instance of a set of conditions V, first rename the variables in each condition that is a member of V so that each has a disjoint set of free variables, then unify. That is to say, suppose *rename*(V) is a function that systematically applies renaming substitutions to elements of V, and returns the resulting, name-clash-free, conditions. Then *mgci*(V) = *mgu*(*rename*(V)). We can therefore deduce that a most general common instance exists if and only if some common instance does, and that there is an algorithm (based on Robinson's unification algorithm [30]) to find the most general common instance together with the associated substitutions.
Definition: *cmp*(V) is the set of conditions arising from V through a finite number of applications of the *mgci* function and the part-formula relation, i.e. the closure of *mgci* and part-formula. That is to say, *cmp*(V) is the least condition set W such that a) V $\subseteq$ W, b) if W′ $\subseteq$ W, then *mgci*(W′) $\in$ W, and c) if A $\in$ W and B is a part-formula of A, then B $\in$ W.
Lemma 4 (*cmp*) Given a finite condition set V, *cmp*(V) is also finite. Alternatively, *cmp* is finitary. Remark: Individually, *mgci* and the part-formula relation are both finitary. It is not immediately obvious that their union is also.

Proof It is well known that a closure such as *cmp* can be constructed by iterating an appropriate set of generating operations. In this case, appropriate generators are M and S where M(V) = {*mgci*(W) | {} ≠ W ⊆ V} and S(V) = {A | A part-formula B for some B ∈ V}. Both M and S are idempotent (so that, for example, M(M(V)) = M(V)) thanks to related properties of part-formula and *mgci* so we need only consider alternating sequences of the M and S generators. Finally note that, taking a part-formula reduces the logical matrix of the original formula, whereas any instance, and in particular any *mgci*, has the same matrix[1], and thus the same number of boolean operators, as the original formula(s). The closure *cmp*(V) is therefore constructed by alternating sequences of M and S generators at most #V long, where #V is the maximum boolean operator count of any formula in V. Since *mgci* and part-formula are both finitary, and the union of a finite number of finite sets is finite, this proves *cmp* is finitary. □

2.2 Equalising Programs

We now come to the important idea of an equalising program. This, simplistically, is one in which the only assignments are ones which equalise their post-condition with that of some other assignment. A precise definition follows below.

Before giving that definition, however, it is convenient to start by introducing assertions and asserted programs. These make explicit applications of the laws of consequence, and help to reduce the difficulties in subsequent lemmas such as that noted in the proof of lemma 1.

Definition: A fully asserted program is one which is generated by the following axiom and inference rules, which are the same as in traditional Hoare logic, but incorporating an extra assertion or assertions in the consequent of each law.

Figure 2 presents asserted Hoare logic as a formal inference system.

$\vdash \alpha(A)\ \alpha\ A$	assignment introduction
$A \Rightarrow B,\ B\ P\ C \vdash A\ B\ P\ C$	consequence: pre-condition
$A\ P\ B,\ B \Rightarrow C \vdash A\ P\ B\ C$	consequence: post-condition
$A\ P\ B,\ B\ Q\ C \vdash A\ P\ B\ Q\ C$	sequence
$A \wedge B\ P\ C,\ A \wedge \neg B\ Q\ C \vdash A$ if B then $A \wedge B\ P\ C$ else $A \wedge \neg B\ Q\ C$ fi C	selection
$A \wedge B\ P\ A \vdash A$ while B do $A \wedge B\ P\ A$ end $A \wedge \neg$ B	iteration

Fig. 2 Asserted Hoare Logic

Definition: an asserted program (a.k.a. a partially asserted program) is one which can be obtained from a fully asserted program by deleting some of the assertions.

In what follows, asserted programs will be used to simplify the presentation. Observe that a fully asserted program is in some sense equivalent to

[1] Note that classical first order logic excludes boolean variables and expressions

a Hoare proof. In particular, every Hoare proof can be converted to a fully asserted program and vice versa [32] (chapter 15). Each use of the laws of consequence, for example, leads to an additional assertion being included in the fully asserted program.

Definition: the function $cmpn(\mathrm{V}) = cmp(\mathrm{V}) \cup \{\mathrm{C} = \mathrm{C}_1 \wedge \neg \mathrm{C}_2 \mid \mathrm{C}_1 \wedge \mathrm{C}_2 \in cmp(\mathrm{V})\}$. This is still finitary, as it involves one application of *cmp*() and insertion of at most one negation symbol.

Definition: an *equalising proof* is a Hoare proof, of a triple A P B say and based on verification conditions V, in which every assignment introduction is of the form C α D where both C and D are members of $cmpn(\mathrm{V},\mathrm{B})$.

Note that in an equalising proof *every* pre-condition and post-condition satisfies the condition above. This can be shown by induction on the asserted Hoare logic derivation, for example. This observation will be useful in the lemmas that follow.

Definition: an *equalising triple* is one which has an equalising proof, and similarly an *equalising program*.

Lemma 4 corollary It follows from lemma 4 and this definition that it is decidable whether an equalising program P exists, based on a given set of verification conditions V, satisfying given pre- and post-conditions A and B. A simple algorithm would be to search through the space of all proofs up to certain length. This is sufficient as $cmpn(\mathrm{V},\mathrm{B})$ is finite, and by lemma 2 (substitutability) at most one sub-program satisfying each pair of pre- and post-conditions C, D $\in cmpn(\mathrm{V},\mathrm{B})$ is required. Only proofs of length up to N^2 need be considered, therefore, where $\mathrm{N} = \#cmpn(\mathrm{V},\mathrm{B})$. More efficient algorithms than this can no doubt be devised, but this paper is concerned with existence, not efficiency.

Here is an example to show why the post-condition B is required as an argument to *cmpn* in the definition above:

$\mathrm{A}(f(x))$ while $\mathrm{B}(g(y))$ do $x, y := f(x), g(y)$ end $\mathrm{A}(f(x)) \wedge \neg \mathrm{B}(g(y))$
with verification condition $\mathrm{A}(x) \wedge \mathrm{B}(y) \Rightarrow \mathrm{A}(f(x))$

The intermediate pre-condition $\mathrm{A}(f(x)) \wedge \mathrm{B}(g(y))$ arises through equalising the post-condition with the consequent of the verification condition, but not through equalising parts of the verification condition alone. Indeed, the function $g()$ does not occur in the verification condition at all.

A useful property of equalising programs is given in the following lemma.

Lemma 5 If A P B and B Q C are both equalising triples, so too is A P Q C.

Proof Combining the proofs of the two given equalising triples with a final inference using the law of sequence gives a proof of A P Q C. In this proof, every conclusion has pre- and post-conditions occurring in $cmpn(\mathrm{V},\mathrm{B})$ or $cmpn(\mathrm{V},\mathrm{C})$. Moreover B, which is the initial pre-condition of B Q C, must occur in $cmpn(\mathrm{V},\mathrm{C})$. By closure therefore, the pre- and post-conditions of every conclusion occur in $cmpn(\mathrm{V},\mathrm{C})$ as required. □

Definition: an *a-equalising proof* is the proof corresponding the asserted program $\alpha(\mathrm{A})$ α A P B for some assignment statement α. That is to say, it consists of an equalising proof, of A P B say, followed by two further steps of the form

$\alpha(\mathrm{A})$ α A assignment axiom
$\alpha(\mathrm{A})$ α P B sequence

Similarly for *a-equalising triples* and *a-equalising programs*.

The decision procedure for the existence of an equalising program can easily be extended to a-equalising programs by checking which, if any, conditions in *cmpn*(V,B) are generalisations of the given pre-condition, and determining if it is possible to establish post-condition B from any of these.

In the following, a-equalising programs provide a "normal form" for Hoare proofs.

3 The Main Theorem

This leads us to our main result: if A P B is a derivable Hoare triple based on verification conditions V, then for some α and P′, A α P′ B is a a-equalising Hoare triple based on V. The proof of this is by induction on the depth and size of the program P, including assertions. Note that the more elegant technique of structural induction cannot be applied here because of the non-structural transformations used the proof. Also note that, for brevity, the proofs here are expressed in terms of asserted triples, which are classical Hoare triples with intermediate assertions added using the laws of asserted Hoare logic given in figure 2 above.

Definition: the *depth* of a program P is the greatest level of nesting of any statement in P. To be precise

$depth(\alpha) = 0$	assignment
$depth(\text{P Q}) = \max(depth(\text{P}), depth(\text{Q}))$	sequence
$depth(\textsf{while A do P end }) = depth(\text{P}) + 1$	iteration
$depth(\textsf{if A then P else Q fi}) = max(depth(\text{P}), depth(\text{Q})) + 1$	conditional

Definition: The *size* of a program is the number of leaves in its syntax tree, i.e. the number of assignment statements it contains plus the number of assertions.

Theorem 1 Suppose A P B is an asserted triple derived from verification conditions V. Then there exists P′, such that A P′ B is an a-equalising triple also derived from V.

Proof By induction on the depth and size of P.

Base case ($depth(\text{P}) = 0$): Note that a program of depth 0 can have neither conditionals nor loops. It follows that P must consist of a sequence of assignments and assertions. Since a sequence of assignments can be merged into a single multiple assignment [23] it follows that P can equivalently be represented as a sequence of assertions, separated by at most one assignment, where each double assertion, C D say, signifies the use of one of the consequence laws and verification condition C $\Rightarrow$ D. This means that the post-condition of each assignment must be a member of *cmpn*(V,B), and as must the pre-condition of each assignment, except possibly the first. This representation is, by definition, the asserted Hoare logic form of an a-equalising proof, as each assignment, except possibly the first, equalises parts of two verification conditions or B.

Induction step ($depth(\text{P}) > 0$): The hypothesis for this induction states that given any asserted triple C Q D such that $depth(\text{Q}) < depth(\text{P})$, or with the same depth as P but smaller in size, derived using verification conditions V,

there exist β, R such that C β R D is an asserted a-equalising triple that can also be derived from V, and with the same depth as Q. To prove the induction step, consider the structure of P.

Case 1 (P ends in an assertion, P = P_1 C say)

In this case the induction hypothesis holds of the sub-program A P_1 C, so that an a-equalising program A α P_1' C exists, satisfying the conditions above. Note also that in these circumstances C $\Rightarrow$ B $\in$ V, and therefore A α P_1' C B is an (asserted) a-equalising program as required.

Case 2 (P is a conditional)

In this case the induction hypothesis applies to both the then- and the else-parts of the conditional, which are less deep than P. It is therefore possible to derive from V an asserted triple of the form A if C then A$\wedge$C α_1 $A_1\wedge C_1$ P_1 B else A$\wedge\neg$C α_2 $A_2\wedge\neg C_2$ P_2 B fi B, where $A_1\wedge C_1$ P_1 B and $A_2\wedge\neg C_2$ P_2 B are both equalising. Now let $A'\wedge C' = mgci(A_1\wedge C_1, A_2\wedge C_2)$, which must exist because A$\wedge$C = $\alpha_1(A_1\wedge C_1) = \alpha_2(A_2\wedge C_2)$ is a common instance of these two formulas. Also let $(\alpha_1', \alpha_2') = mges(A_1\wedge C_1, A_2\wedge C_2)$, and β be such that A$\wedge$C = $\beta(A'\wedge C')$. This means that both $A'\wedge C'$ and $A'\wedge\neg C'$ are in *cmpn*(V,B) so that A β A' if C' then $A'\wedge C'$ α_1' $A_1\wedge C_1$ P_1 B else $A'\wedge\neg C'$ α_2' $A_2\wedge\neg C_2$ P_2 B fi B is an (asserted) a-equalising triple as required.

Case 3 (P is a loop)

In this case the induction hypothesis applies to the loop body. It is therefore possible to derive from V an asserted triple of the form A while C do A$\wedge$C β $A_1\wedge C_1$ R A end A$\wedge\neg$C where A$\wedge$C β $A_1\wedge C_1$ R A is a-equalising, A$\wedge$C = $\beta(A_1\wedge C_1)$, and B = A$\wedge\neg$C. It follows that A$\wedge$C $\in$ *cmpn*(V,A) $\subseteq$ *cmpn*(V,A$\wedge\neg$C) = *cmpn*(V,B). Similarly for A. Thus A while C do A$\wedge$C β $A_1\wedge C_1$ R A end B is an a-equalising program as required.

It remains to consider sequences of statements. Assume to start with that the second statement in P is an assignment, α say, and consider separately the different cases possible for the first part of P.

Case 4 (P ends in an assertion then an assignment, P = P_1 C α(B) say)

In this case the induction hypothesis applies to A P_1 C giving rise to an a-equalising program A β P_1' C of the required form. It follows that, using lemma 5, A β P_1' C α(B) is an asserted triple as required.

Case 5 (P is a conditional statement followed by an assignment)

By lemma 3, the trailing assignment α can be moved into the two legs of the conditional statement. This does not affect their depth. The induction hypothesis therefore applies to both the then- and the else-parts of the conditional, which therefore have equivalent a-equalising forms. The proof now follows as in case 2.

Case 6 (P is a loop followed by an assignment)

The induction hypothesis applies to the loop body, giving rise to the valid triple A while C do A$\wedge$C β $A_1\wedge C_1$ Q A end A$\wedge\neg$C α $A_2\wedge\neg C_2$ say, in which the loop body is a-equalising where B = $A_2\wedge\neg C_2$. If β is the empty assignment, skip, the loop body is equalising, so A$\wedge$C $\in$ *cmpn*(V,A), but note that A$\wedge$C can never be a part-formula nor instance of A, so that A$\wedge$C $\in$ *cmpn*(V), and hence A$\wedge\neg$C $\in$ *cmpn*(V), and similarly the part-formula A, so the whole program is therefore equalising. Assume therefore that β is a non-empty assignment. The result follows from lemma 6 below.

Case 7 (P is a sequential composition then an assignment, P = P_1 C P_2 α)

In this case the induction hypothesis applies to C P_2 α B, which is no deeper and smaller than P, giving rise to an a-equalising triple C β D Q B of the required form. The triple A P_1 C β D is therefore valid, no deeper and smaller than P, so that the induction hypothesis can be applied a second time to give the a-equalising triple A γ R D. This means that the triple A γ R D Q B is a-equalising, by lemma 5, and of the required form.

The final case is when the second part of P is not an assignment, but a compound statement.

Case 8 (P = P_1 C P_2, where $depth(P_2) \geq 1$, $size(P_2) > 1$)

In this case the induction hypothesis applies to C P_2 B, giving rise to an a-equalising triple of the required form C β D Q B. The induction hypothesis also applies to A P_1 C β D, giving rise to an a-equalising triple A δ R D, so that A δ R D Q B is an a-equalising triple of the required form. □

Lemma 6 Given a program in the form of a loop whose body is a-equalising followed by an assignment, and assuming the induction hypothesis of the theorem above, an a-equalising program can be derived, using the same conditions, and of the same depth as the original.

Proof

Assume the given program is of the form A **while** C **do** A∧C β A_1∧C_1 Q A **end** A∧¬C α A_2∧¬C_2 say, and without loss of generality (see case 6 above) β is a non-empty assignment. Now let A′∧C′ = $mgci(A_1 \wedge C_1, A_2 \wedge C_2)$, which must exist as A∧C is a common instance, $(\beta', \alpha') = mges(A_1 \wedge C_1, A_2 \wedge C_2)$, and γ_1 be such that A∧C = γ_1(A′∧C′). Now A γ_1 A′ **while** C′ **do** A′∧C′ β' A_1∧C_1 Q A γ_1 A′ **end** A′∧¬C′ α' A_2∧¬C_2 is a derivable Hoare triple. By the induction hypothesis, the derivable triple A_1∧C_1 Q A γ_1 A′ has an a-equalising form, A_1∧C_1 δ Q_1 A′ say. Substituting this in the previous triple gives A γ_1 A′ **while** C′ **do** A′∧C′ β' A_1∧C_1 δ Q_1 A′ **end** A′∧¬C′ α' A_2∧¬C_2. The sequence of assignments β' δ can be merged to a single assignment β_1 say (dropping the intermediate assertion A_1∧C_1). Finally, therefore, we have the derivable triple A γ_1 A′ **while** C′ **do** A′∧C′ β_1 Q_1 A′ **end** A′∧¬C′ α' A_2∧¬C_2, which is in the same form as the original program. The same sequence of steps can therefore be applied repeatedly to give a sequence of such programs. Note that in this sequence of programs, the pre-condition of the loop body is in each case a generalisation of the previous pre-condition. The sequence of these pre-conditions, A∧C, A′∧C′, A″∧C″, and so on cannot continue forever strictly decreasing, by well-foundedness of the generalisation relation. This sequence therefore reaches a fixpoint, A^*∧C^* say, resulting in a program of the form A γ A^* **while** C^* **do** A^*∧C^* β_n A_n∧C_n Q_n A^* **end** A^*∧¬C^* α^* A_2∧¬C_2, once the leading assignments γ_1, γ_2,γ_3, ... have been merged into a single assignment. The condition A^*∧C^* must be the most common instance of the post-conditions A_n∧C_n and A_2∧C_2, otherwise further transformation would be possible. In the final program, therefore

$A^*\wedge C^* = mgci(A_n\wedge C_n,\ A_2\wedge C_2)$, and
$A_n\wedge C_n\ Q_n\ A^*$ is equalising, hence
$A_n\wedge C_n \in cmpn(V,A^*)$, so that
$A^*\wedge C^* \in cmpn(V,A^*,A_2\wedge C_2)$, which means that,
since $A^*\wedge C^*$ can never be a part-formula nor instance of A^*,
$A^*\wedge C^* \in cmpn(V,A_2\wedge C_2)$, and thus
$A^* \in cmpn(V,A_2\wedge C_2)$, which means that
The sub-program A^* while C^* ... $A_2\wedge\neg C_2$ is equalising
The complete final program is therefore a-equalising as required. □

3.1 Total Correctness Logic and the Main Theorem

Next we briefly show the same result holds for a total correctness Hoare logic. Consider a Hoare logic for total correctness, with an iteration law that ensures termination and not just preservation of the invariant. For example, the classic iteration law

$A\wedge B\ P\ A \vdash A$ while B do P end $A\wedge\neg B$

could be replaced by an extended law

$A\wedge B\wedge X=E\ P\ A\wedge 0\leq E<X \vdash A$ while B do P end $A\wedge\neg B$

where X is a logical constant that does not occur free in A, nor in B, and E is an integer-valued expression.

A *logical constant* is a special kind of variable. As with other kinds, it is assumed an infinite number are available for use. Traditionally these are represented by giving them upper case names, or zero-subscripting. A logical constant is allowed to occur in logical conditions, but not in the program text. This fact has important consequences. Firstly, X can never be the target of an assignment, so it cannot be replaced via the assignment axiom. Secondly, X can never occur in the condition of a loop or conditional statement, so it cannot disappear as a result of applying the laws of iteration or selection.

There are of course many possible loop laws. This particularly simple formulation, which is inspired by law 5.5 *iteration* of Morgan [28], is amenable to proof-theoretic techniques. Note that in general systems with such laws typically use typed variables and expressions so that automatic type-checking can be used to reduce the proof burden. Of course, any well-founded relation could occur here, not just the natural number ordering. Finally note that, for brevity, no law is given here for introducing logical constants, though this would of course be required in any practical system.

Lemma 7 In a total correctness proof, any intermediate Hoare triples ending with the loop post-condition $A\wedge 0\leq E<X$, and proved using the assignment axiom, and laws of sequence, selection, and iteration, must have as its pre-condition $\alpha(A)\wedge 0\leq\alpha(E)<X$, for some assignment α.

Proof By induction on the (size of the) proof. □

A similar result holds for the loop pre-condition.

Lemma 8 In a total correctness proof, any intermediate Hoare triples beginning with the loop pre-condition $A\wedge B\wedge X=E$, and proved using assignment axiom, and laws of sequence, selection, and iteration, must have as its post-condition $\beta^{-1}(A)\wedge\beta^{-1}(B)\wedge X=\beta^{-1}(E)$, for some assignment β. (Note that

the use of the inverse substitution β^{-1} means that the post-condition here is a generalisation of the pre-condition.)
Proof By induction on the (size of the) proof. □
Combining these two results, we see that conditions $\alpha(\mathrm{A})\wedge 0\leq\alpha(E)<\mathrm{X}$ and $\beta^{-1}(\mathrm{A})\wedge\beta^{-1}(\mathrm{B})\wedge\mathrm{X}=\beta^{-1}(E)$ must each occur as part of some verification condition, before the hypothesis $\mathrm{A}\wedge\mathrm{B}\wedge\mathrm{X}=E$ P $\mathrm{A}\wedge 0\leq E<\mathrm{X}$ of the new loop law can be derived.

Matching $\alpha(E)$ and $\beta^{-1}(E)$ gives a finite number of possibilities for expression E, up to renaming. Cycling through all possible variables (the ones that occur in the closure *cmpn*(V,B)) gives a finite number of possibilities for expression E. Combining this with the techniques used for the partial correctness theorem gives an equivalent decision procedure for the total correctness variant of Hoare logic presented here – since the new iteration law contains the original one as a part-formula, the technique of equalising still applies, although the actual definitions and lemmas used must change to match the form of the new iteration law.

3.2 Requirements on the Base Predicate Logic

So far this paper has assumed that the Hoare logic in use is an extension of classical first order logic. It is now possible to review what is required of the base logic. Clearly, the boolean operators such as negation, implication and conjunction are necessary, as these used in the Hoare logic laws of inference. Similarly the axiom of assignment requires the base logic to provide a substitution mechanism. Total correctness also requires that the base logic includes natural numbers, or at least some other inductive structure. Finally, the main result here depends on equalisation, which is closely related to unification.

It is also of interest to consider richer base logics. For example, most programming languages include boolean variables and expressions, which naturally leads to consideration of logics with similar features. These logics would typically conform to the requirements listed in the paragraph above. There are two likely cases. In the first instance, consider a programming language that uses a different syntax for boolean operators (eg && instead of $\wedge$.) Boolean expressions from such a language can be converted to classical first order form, for example replacing operators by function symbols, and each boolean variable x by the semantically equivalent formula $IsTrue(x)$, where $IsTrue$ is a new predicate symbol. (Note that a boolean expression consisting of a single variable is easily identified – either using the variable's declaration, or else syntactically.) The classical unification algorithm [30] can then be applied to the resulting logical formulas. Alternatively, the programming language might use standard logical syntax for boolean operators. This also can typically be translated to standard expression syntax as in the first case above, again allowing the classical unification algorithm to be used. Alternatively, the classical algorithm could be extended to cover these richer first order logics.

Switching to a higher order base logic, however, would give rise to the problem of higher order unification [26].

4 Discussion

4.1 Verification Conditions for Total Correctness

In this section, the integer square root case study introduced earlier is considered again, with a view to indicating how verification conditions can be discovered without first having to write the code. Also, the conditions given here are for total correctness.

Following Gries' strategy for developing a loop [16]: "first develop the guard B so that A∧B ⇒ C" it is clear to see how the invariant and guard can be determined from the following theorem:

$r^2 \leq s < q^2 \wedge \neg(r+1 \neq q) \Rightarrow r^2 \leq s < (r+1)^2$

The following condition indicates how this invariant can be established:

$s \geq 0 \Rightarrow 0^2 \leq s < (s+1)^2$

Now consider a possible variant: $q - r$. One way to decrease this quantity is by finding a value between q and r. The feasibility of doing so is confirmed by the following theorem:

$r^2 \leq s < q^2 \wedge r + 1 \neq q \Rightarrow r < (q+r) \div 2 < q$

Finally note that a value between r and q can be used to re-establish the invariant and decrease the variant:

$r^2 \leq s < q^2 \wedge r < p < q \wedge s < p^2 \Rightarrow 0 \leq p - r < q - r$

$r^2 \leq s < q^2 \wedge r < p < q \wedge \neg(s < p^2) \Rightarrow 0 \leq q - p < q - r$

For book-keeping reasons these last three conditions must in fact be recorded more verbosely. The first one, for example, should not be

$r^2 \leq s < q^2 \wedge r + 1 \neq q \Rightarrow r < (q+r) \div 2 < q$

but rather

$r^2 \leq s < q^2 \wedge r + 1 \neq q \wedge \mathrm{X} = q - r \Rightarrow$
$\quad r^2 \leq s < q^2 \wedge r < (q+r) \div 2 < q \wedge \mathrm{X} = q - r$

The second one should not be

$r^2 \leq s < q^2 \wedge r < p < q \wedge s < p^2 \Rightarrow 0 \leq p - r < q - r$

but rather

$r^2 \leq s < q^2 \wedge r < p < q \wedge \mathrm{X} = q - r \wedge s < p^2 \Rightarrow$
$\quad r^2 \leq s < p^2 \wedge 0 \leq p - r < \mathrm{X}$

and similarly for the third condition.

Adding the extra constraints such as $\mathrm{X} = q - r$ means these conditions syntactically match the inference laws of Hoare logic, even though the extra clauses add no logical content.

This case study is based on Chapter 8 of Morgan's textbook on refinement [28] (see also the second edition, 1994). The development presented in that book (figure 19.2) has 11 refinement steps which use 7 different refinement laws and depend on 5 verification conditions, which are essentially the same as the ones given here. Morgan does not explicitly list these conditions, but notes that the initial development contained two errors, one a transcription error, the other a logical error. He then makes the observation that "mathematical rigour cannot eliminate mistakes entirely: nevertheless it does reduce their likelihood dramatically".

Gries [16] gives as one of his main principles: "a program and its proof should be developed hand-in-hand, with the proof usually leading the way".

Gries also has other useful heuristics, such as "four ways of weakening a predicate", by "deleting a conjunct, replacing a constant by a variable, enlarging the range of a variable, or by adding a disjunct". Other heuristics however are oriented toward programs, not predicates. (For example, "develop one leg of a conditional by finding program P such that A $\Rightarrow$ wp(P,B)".)

For a development methodology focussed exclusively on predicates, see work of Hehner [17–19].

4.2 Formal Verification Tools

Early criticism of formal program verification focussed on the likelihood of human errors, and the fallibility of social processes such as peer review [12], [13]. It is clear that to increase confidence in programs (cf. algorithms), high quality automated tool support is required for processes such as theorem proving, proof checking, and code generation.

Broadly speaking, three kinds of formal verification tools have been proposed.

Firstly there are mathematically-based toolkits, such as GYPSY [2], Atelier B [6] and Perfect Developer [11]. Programs are written using mathematically defined programming notations and formal specifications or annotations. Conditions are generated from this input, which must then be verified using interactive or automated theorem proving systems. Similar verification condition generation (VCG) systems have also been implemented that use conventional programming languages but with formal annotations, for example the Java Modelling Language JML [27], and Spec# [31].

A second approach is represented by the refinement calculator [7]: a fine-grained interactive system for program refinement. Some of these refinement steps will generate verification conditions that must be discharged by a theorem prover.

Finally, the logic compiler: Hehner's vision is of "compilers, with built-in theorem provers" which will be able to "point out the location of a logic error the same way as they do now with a syntax error" [18]. To date this vision has not been realised.

It is worthwhile comparing the VCG approach with the logic compiler approach. The latter has the advantage of being proof-driven, whereby developers verify theorems before the code is developed. Indeed, since code is generated from the theorems, development is necessarily verification first. With the VCG approach, errors may be discovered days or weeks after they are introduced, when it is finally realised that some verification condition cannot be proved because it is in fact false. A proof-driven approach would discover this fact sooner. A collection of theorems is built up, then fed to an executable code generator, for example a more efficient version of the algorithm described in this paper.

The problem that can now arise is that the supplied theorems are insufficient, preventing the derivation and output of a (correct-by-construction) program. It is crucial that in this case the "logic compiler" does not simply fail, but provides helpful diagnostics so that the developer can decide what additional theorems must be proved and supplied as additional input

to the compiler. More useful than this, in fact, would be for the compiler itself to postulate these additional theorems and pass them directly to the automated proof system: rather than requiring a complete set of verification conditions, just a partial set should suffice, or conditions that are logically equivalent but not necessarily in the exact syntactic form. As a trivial example of this, it would be helpful if the logic compiler would accept a logically equivalent formulation, for example $B \wedge A \Rightarrow C$ instead of $A \wedge B \Rightarrow C$, but also including more sophisticated logical transformations, as well as addition of the "book-keeping" constraints noted above.

A further point is that there is typically a certain amount of redundancy in the verification conditions required by Hoare logic. In the case study above, for example, each clause occurs on average about twice, ignoring renaming. This redundancy could be avoided by introducing local definitions, or by careful insertion of additional clauses by the logic compiler using appropriate heuristics, that nonetheless should not compromise decidability.

Clearly, these extra features of the logic compiler would evolve through practice and experience of actual use of such a tool.

4.3 Related and Future Work

Hoare logic has been extensively studied since it was first published [20]. Completeness [9] and incompleteness [8] results have been published, as has an extensive survey [3].

An earlier version of the main result given here was presented at a workshop [14]. This was based on a specially constructed program logic, not classical Hoare Logic.

The work here differs from previous approaches to predicative programming [17,19,24] in that the latter give a first order or predicative semantics to programs, with $\Leftarrow$ as program refinement. Although predicative semantics is quite natural for finite programs, special care is needed with iteration and recursion to give satisfactory results. The approach presented here is neutral with respect to semantics, since it is proof-theoretic. As with any axiomatic approach, no model for iteration is provided, only an inference law.

Constructive, or intuitionist, logic provides an alternative way to generating programs from proofs [25] which has been implemented in systems such as NUPRL [10] among others. The approach presented here is neutral with respect to the underlying logic, which can be either classical or constructive. A further difference is that here programs are generated from the verification conditions, not their proofs.

The result given here has been shown to apply to two versions of Hoare logic. It would be interesting to know whether it holds for larger, more realistic systems such as the Hoare Logic for Pascal [22] or the Refinement Calculus [28]. The preceding section described automated support for predicative programming based on this result, which has yet to be implemented.

References

1. Abrial JR: *The B Book*, Cambridge, 1996.
2. Ambler AL et al: *GYPSY: A Language for Specification and Implementation of Verifiable Programs*, ACM SIGPLAN Notices 12: 1-10, March 1977.
3. Apt KR: *Ten years of Hoare's logic*, ACM Transactions on Programming Languages and Systems 3: 431-483, 1981.
4. Back RJR: *On the Correctness of Refinement Steps in Program Development*, Report A-1978-5, University of Helskinki, 1978.
5. Back RJR, von Wright J: *Refinement Calculus: a systematic introduction*, Springer, 1998.
6. Behm P et al: *Meteor: A Successful Application of B in a Large Project*, LNCS 1708: 369-387, Springer, 1999.
7. Butler MJ, Långbacka: *Program Derivation Using the Refinement Calculator*, TPHOLs: 93-108, 1996.
8. Clarke EM: *Programming Language Constructs for which it is impossible to obtain good Hoare axiom systems*, JACM 26(1): 129-147, 1979.
9. Cook SA: *Soundness and Completeness of an Axiom System for Program Verification*, SIAM Journal of Computing 7: 70-90, 1978.
10. Constable RL: *Implementing Mathematics with the Nuprl Proof Development System*, Prentice Hall, 1986.
11. Crocker D: *Safe Object-Oriented Software*, Twelfth Critical Systems Symposium: 19-41, Springer, 2004.
12. de Millo RA, Lipton RJ, Perlis AJ: *Social Processes and Proof of Theorems and Programs*, CACM 22(5): 271-280, 1979.
13. Fetzer JH: *Program Verification: the very idea*, CACM 31(9): 1048-1063.
14. Gravell AM: *Logical Refinement of Imperative Programs: generating code from verified conditions*, (Constraint) Logic Programming and Software Engineering 2000, Gupta G, Ramakrishnan IV (eds.), Imperial College, London, July 2000.
15. Gries D: *The Multiple Assignment Statement*, LNCS 69: 100-112, Springer, 1978.
16. Gries D: *The Science of Programming*, Springer, 1981.
17. Hehner ECR: *Predicative Programming: I*, Communications of the ACM 27: 134-143, 1984.
18. Hehner ECR, Gupta LE, Malton AJ: *Predicative Methodology*, Acta Informatica 23:487-505, 1986.
19. Hehner ECR: *A Practical Theory of Programming*, Springer, 1993.
20. Hoare CAR: *An Axiomatic Basis of Computer Programming*, Communications of the ACM 12: 576-580, 1969.
21. Hoare CAR: *Procedures and Parameters: An Axiomatic Approach*, Symposium on Semantics of Algorithmic Languages, LNCS 188: 102-116, 1971.
22. Hoare CAR, Wirth N: *An Axiomatic Definition of the Programming Language Pascal*, Acta Informatica 2:335-355, 1973.
23. Hoare CAR et al: *Laws of Programming*, Communications of the ACM 30: 672-686, 1987.
24. Hoare CAR, He J: *Unified Theories of Programming*, Prentice Hall, 1998.
25. Howard WA: *The Formulae-as-Types notion of Construction*, To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pages 479-490, 1980.
26. Knight K: *Unification: a multidisciplinary survey*, Computing Surveys 21(1):93-124, 1989.
27. Leavens GT, Baker AL, Ruby C: *JML: a Java Modelling Language*, Formal Underpinnings of Java, OOPSLA, 1998.
28. Morgan CC: *Programming from Specifications*, Prentice Hall, 1990.
29. Morris JM: *A Theoretical Basis for Stepwise Refinement and the Programming Calculus*, Science of Computer Programming 9(3):287-306, 1987.
30. Robinson JA: *Computational Logic: The Unification Computation*, Machine Intelligence 6:63-72, 1971.
31. Rustan K, Leino M, Barnett M, Schulte W: *The Spec# Programming System: an overview*, Microsoft Research, 2004
32. Sperschneider V, Antoniou G: *Logic: a foundation for Computer Science*, Addison Wesley, 1991.