Bull. Nov. Comp. Center, Comp. Science, 26 (2007), 39–63

Effective generation of verification conditions for non-deterministic unstructured programs*

I.S. Anureev, E.V. Bodin, N.V. Shilov

Abstract. We address a problem of efficiency of verification condition generation for unstructured non-deterministic imperative programs. Importance of the study is based upon two arguments:

- industrial programming is very often unstructural (e.g., extensive use of 'go to' in C-programs);
- program analysis techniques (like abstraction) introduce unstructural and non-deterministic control flow to programs.

In the structural deterministic case, verification condition generation is driven by the rules of axiomatic semantics that are well known due to axiomatic semantics for Pascal by A. Hoare. In this case, generation of verification condition seems to be trivial, since its complexity is linear in the number of control structures due to modularity of structured imperative programs. In contrast, in the unstructured non-deterministic case, the complexity seems a priori to be exponential in the number of operators due to the exponential number of acyclic paths through a program. We present in this paper an efficient, sound and complete verification condition generator that can be used in both cases (i.e. in unstructured non-deterministic, as well as in structural deterministic). The generator complexity is linear in the overall number of control constructs and operators.

1. Introduction

Formal program verification is computer-aided mathematically-rigorous proving of program correctness. A verifying compiler is a system program that can parse, verify and build a safe executable code for an input high-level source program. According to a Turing Prize Laureate sir C.A.R. Hoare [9], it could be "a grand challenge for computing research".

In this paper we address a problem of efficiency of verification condition generation for unstructured non-deterministic input-output imperative programs annotated by loop invariants in the style of R.W. Floyd [7].

Importance of the study is based upon two arguments:

- industrial programming is very often unstructural (e.g., extensive use of 'go to' in C-programs);

*This research is supported in parts by RFBR grants 06-01-00464-a and 05-07-90162.

- program analysis techniques (like abstraction) introduce unstructural and non-deterministic control flow to programs.

In the structural deterministic case, verification condition generation is driven by the rules of axiomatic semantics that are well known due to axiomatic semantics for Pascal by C.A.R. Hoare and N. Wirth [8]. In this case, generation of verification condition in a classical weakest-precondition style of E.W. Dijkstra [5] seems to be trivial, since its complexity is linear in the number of control structures due to modularity of structured imperative programs. In contrast, in the unstructured non-deterministic case, the complexity seems a priori to be exponential in the number of operators due to the exponential number of acyclic paths through a program.

But in reality, even in the structured deterministic case, classical verification condition generation requires exponential space and time. To the best of our knowledge, all verification research and experiments fulfilled before new millennium put up with this exponential explosion as inevitable payoff.

The first research that successfully attacked this unfeasible complexity was reported in paper [6]. The cited paper remarks that there are two reasons for the blow-up:

- every assignment may cause multiple instantiations of a term instead of a variable in a postcondition,
- every choice duplicates a condition and a postcondition since it couples it with *then*-branch and *else*-branch.

For avoiding these term multiplicity and condition duplication, paper [6] uses fresh auxiliary variables and two special constructs **assert** and **assume**. These constructs provide an opportunity to translate (in a linear time) any non-deterministic guarded loop-free program into a so-called passive form. A program in the passive form does not change its variable values but can terminate normally or abnormally. For any program in a passive form, the weakest precondition for normal termination can be constructed in a linear time and the weakest precondition for abnormal termination can be constructed in a quadratic time. Translation to the passive form and normal/abnormal termination condition generation altogether resulted in a quadratic verification condition generation algorithm. This algorithm is sound and complete for loop-free guarded structured programs. Later this approach was extended to the unstructured case [2].

We present in this paper an efficient, sound and complete verification condition generator that can be used in both cases (i.e. in non-deterministic unstructured, as well as in structural deterministic). The generator complexity is linear in the overall number of control constructs and operators.

The rest of the paper is organized as follows. Section 2 defines syntax and semantics of assertions and Hoare triple. Section 3 defines syntax and semantics of two model programming languages: mini-Pascal and

mini-NIL. Section 4 defines axiomatic semantics for mini-Pascal, syntax and verification conditions for annotated mini-Pascal. This section also discuss complexity of the problem of verification condition generation. After that annotated mini-NIL programs and Floyd verification method are discussed in section 5. Verification (or Floyd) conditions for annotated mini-NIL are discussed in Section 6. Our main result – the system of verification conditions – is presented in Section 7. Concluding remarks, research background and topics for further research are discussed in Section 8.

2. Background: assertion language and Hoare triples

A many-sorted system is a (non-empty) set of values that is a disjoint union of sorts (or types) provided with a collection of named ground operations[1] and a disjoint collection of named ground properties[2] with fixed signatures, i.e.

- for every operation, the types of its arguments and results are defined;
- for every ground property, the types of its arguments are defined.

Thus, a data type is a set of values provided with a list of all ground operations, that can use these values as arguments or return these values as their values, and provided with a list of all ground properties, that can use these values as arguments.

Assume that we are given a many-sorted system and a set of typed variables[3]. Terms (and their types) are type-correct expressions constructed from variables and operation names in accordance with standard rules: every variable is a term whose type is the type of the variable; if $n \geq 0$ is an arity of some ground operation named 'ρ', and types of its arguments and its result are $type_1, \ldots type_n$, $type$, and $\tau_1, \ldots \tau_n$ are terms of types $type_1, \ldots type_n$, respectively, then $\rho(\tau_1, \ldots \tau_n)$ is a term and $type$ is its type. Atoms are type-correct expressions constructed from the terms and ground properties in accordance with standard rules: if $n \geq 0$ is an arity of some ground property named 'π', the types of its arguments are $type_1, \ldots type_n$, and $\tau_1, \ldots \tau_n$ are the terms of types $type_1, \ldots type_n$, respectively, then $\pi(\tau_1, \ldots \tau_n)$ is an atom.

Let us assume in the sequel that all terms are represented as strings. For any term τ let $|\tau|$ be the total number of variables and operations in the term.

A state is a mapping σ that assigns to every variable its value compatible with its type. Let us denote by Σ the set of all states. For every state $\sigma \in \Sigma$,

[1] including the names of constants, i.e. the names for nullary operations

[2] that includes typed and/or polymorphic equality '='

[3] i.e. an alphabet of variable names (identifiers) with assigned types: for every variable x its assigned data type is called an x-type

every variable x, and every value v of x-type, let[4] $upd(\sigma, x, v)$ be a state that is equal to σ everywhere but x, and equals v on x:

$$upd(\sigma, x, v)(y) = \begin{cases} \sigma(y), \text{ if } y \text{ is not } x; \\ v \text{ otherwise.} \end{cases}$$

Every state $\sigma \in \Sigma$ can be extended on all terms in the standard manner: for every type-correct term $\rho(\tau_1, \ldots \tau_n)$ let $\sigma(\rho(\tau_1, \ldots \tau_n))$ be the result of application of the operation named ρ to arguments $\sigma(\tau_1)$, ... $\sigma(\tau_n)$.

Definition 1. An assertion language is a first-order typed language of formulas that are constructed from atoms in accordance with the standard rules:

- Every atom is a formula.
- Negation $(\neg\phi)$, conjunction $(\phi \wedge \psi)$, and disjunction $(\phi \vee \psi)$ of formulas are formulas also.
- For every variable x, universal $(\forall x.\ \phi)$ and existential $(\exists x.\ \phi)$ quantifications of a formula are formulas.

Implication $(\phi \to \psi)$ and equivalence $(\phi \leftrightarrow \psi)$ of formulas are standard abbreviations for $((\neg\phi) \vee \psi)$ and $((\phi \to \psi) \wedge (\psi \to \phi))$, respectively.

Let us assume in the sequel, that all formulas are represented as strings. For any formula ϕ, let $|\phi|$ be the total number of constructs '$\neg$', '$\wedge$', '$\vee$', '$\forall$', and '$\exists$' in ϕ, and $\tau(\phi)$ be the maximal value of $|\tau|$ such that a term τ occurs in ϕ. An instance of a variable x is free/bound in a formula if it is out of/in THE range of any '$\forall x$' or '$\exists x$'. For every formula ϕ, variable x and term τ, let $\phi_{\tau/x}$ denote a formula that results from ϕ after substitution of τ instead of all free instances of x.

Definition 2. Semantics of assertions is defined in terms of entailment relation '$\models$' between states and formulas in the standard way.

- For atoms: $\sigma \models \pi(\tau_1, \ldots \tau_n)$ iff
  a tuple of values $(\sigma(\tau_1), \ldots \sigma(\tau_n))$ satisfies the ground property named 'π'.
- $\sigma \models (\neg\phi)$ iff it is not the case $\sigma \models \phi$,
  $\sigma \models (\phi \wedge \psi)$ iff $\sigma \models \phi$ and $\sigma \models \phi$,
  $\sigma \models (\phi \vee \psi)$ iff $\sigma \models \phi$ or $\sigma \models \phi$.
- $\sigma \models (\forall x.\ \phi)$ iff $upd(\sigma, x, v) \models \phi$ for every value v of x-type;
  $\sigma \models (\exists x.\ \phi)$ iff $upd(\sigma, x, v) \models \phi$ for some value v of x-type.

An assertion ϕ is valid $(\models \phi)$, if it holds in all states, i.e. $\sigma \models \phi$ for every $\sigma \in \Sigma$.

[4] 'upd' = 'update'

Proposition 1. *For every formula ϕ, variable v, term τ, and state $\sigma \in \Sigma$, the following equivalence holds: $\sigma \models \phi_{\tau/x}$ iff $upd(\sigma, x, \sigma(\tau)) \models \phi$.*

A programming language is a collection of 'programs'. IO-semantics[5] for a programming language $\mathcal{L}$ is a mapping $\mathcal{S} : \mathcal{L} \to 2^{\Sigma \times \Sigma}$ that assigns a binary relation $\mathcal{S}(\mathcal{P}) \subseteq \Sigma \times \Sigma$ to every program $\mathcal{P} \in \mathcal{L}$.

Definition 3. Let $\mathcal{L}$ be a programming language. A (Hoare) triple is a sentence of the form $\{\phi\}\mathcal{P}\{\psi\}$, where ϕ and ψ are assertions, and $\mathcal{P} \in \mathcal{L}$ is a program. If $\mathcal{S}$ is an IO-semantics for $\mathcal{L}$, then a triple $\{\phi\}\mathcal{P}\{\psi\}$ is said to be $\mathcal{S}$-valid[6] ($\mathcal{S} \models \{\phi\}\mathcal{P}\{\psi\}$)[7], if for every states $\sigma', \sigma'' \in \Sigma$ such that $(\sigma', \sigma'') \in \mathcal{S}(\mathcal{P})$, $\sigma' \models \phi$ implies $\sigma'' \models \psi$.

3. Programming languages: mini-Pascal and mini-NIL

Below we define two toy programming languages: a structural deterministic language mini-Pascal and an unstructural non-deterministic language mini-NIL.

Mini-Pascal is a subset of the Pascal language extended by guards (or tests) and an explicit empty (or skip) action[8]. It is designed for research purposes and is free from all 'cumbersome' programming constructs and types like 'go to', procedures and definable functions, pointers. The language is provided with structural operational semantics.

Definition 4. Mini-Pascal programs are constructed from assignments, guards, and the empty operator by means of sequential composition ';', deterministic choice 'if-then-else', and deterministic iteration 'while-do' as follows:

- The empty (or skip) action is an expression '*skip*'. This expression is a program.
- An assignment is an expression of the form '$x := \tau$', where x is a variable and τ is a term of x-type. Every assignment is a program.
- A guard (or test) is an expression of the form $\xi?$, where ξ is a quantifier-free assertion. Every guard is a program.
- If α and β are programs then $(\alpha\ ;\ \beta)$ is also a program.
- If ξ is a quantifier-free formula, and α and β are programs, then $(if\ \xi\ then\ \alpha\ else\ \beta)$ is also a program.

[5] i.e. Input-Output semantics

[6] or simply valid when semantics is implicit

[7] $\models \{\phi\}\mathcal{P}\{\psi\}$ when semantics is implicit

[8] Guards and the empty action are syntactic sugar, but they makes many ideas more transparent.

- If ξ is a quantifier-free formula and α is a program, then $(while\ \xi\ do\ \alpha)$ is also a program.

Let us assume in the sequel, that all mini-Pascal programs are represented as trees that correspond to their grammar structure: all leaves are empty actions, guards and assignments, all other nodes are constructs ';', 'if-then-else', and 'while-do'.

Definition 5. Structural Operational Semantics (SOS) for mini-Pascal is defined by the following inference system (σ, σ', σ'', σ''' are states within Σ):

$\frac{}{\sigma\langle\xi?\rangle\sigma}$, if $\sigma \models \xi$	$\frac{}{\sigma\langle skip\rangle\sigma}$
$\frac{}{\sigma\langle x{:=}\tau\rangle upd(\sigma,\ x,\ \sigma(\tau))}$	$\frac{\sigma'\langle\alpha\rangle\sigma''\ ,\ \sigma''\langle\beta\rangle\sigma'''}{\sigma'\langle(\alpha\ ;\ \beta)\rangle\sigma'''}$
$\frac{\sigma'\langle\alpha\rangle\sigma''}{\sigma'\langle(if\ \xi\ then\ \alpha\ else\ \beta)\rangle\sigma''}$, if $\sigma' \models \xi$	$\frac{\sigma'\langle\beta\rangle\sigma''}{\sigma'\langle(if\ \xi\ then\ \alpha\ else\ \beta)\rangle\sigma''}$, if $\sigma' \models (\neg\xi)$
$\frac{\sigma'\langle\alpha\rangle\sigma'',\ \sigma''\langle(while\ \xi\ do\ \alpha)\rangle\sigma'''}{\sigma'\langle(while\ \xi\ do\ \alpha)\rangle\sigma''}$, if $\sigma' \models \xi$	$\frac{}{\sigma\langle(while\ \xi\ do\ \alpha)\rangle\sigma}$, if $\sigma \models (\neg\xi)$

For any mini-Pascal program α, let $SOS(\alpha)$ be the following binary relation:

$$\{(\sigma'\ ,\ \sigma'') \in \Sigma \times \Sigma\ :\ \sigma'\langle\alpha\rangle\sigma'' \text{ is provable in the above inference system}\}.$$

The mini-NIL language is designed for approbation of the basic concepts of a verification project F@BOOL@[9] [3, 4]. The language is provided with a small step operational semantics.

Definition 6. Let labels be unsigned integers 0, 1, 2, ... An assignment operator is an expression of the form '$l:\ x := \tau\ goto\ L$', where l is a label, x is a variable, τ is a term of x-type, and L is a finite set[10] of labels; an expression '$x := \tau$' is called a body of this operator. A condition operator is an expression of the form '$l:\ if\ \xi\ then\ L^+\ else\ L^-$', where l is a label, ξ is a quantifier-free formula, L^+ and L^- are finite sets[10] of labels; a formula 'ξ' is called a body of this operator. A mini-NIL program is a finite set of

[9] Please refer to the concluding section 8 for some details about F@BOOL@.

[10] The empty set is admissible.

operators[11] such that any label marks one operator at most. A label '0' (zero) is called an initial (or start) of any mini-NIL program. A final (or terminal) label of a mini-NIL program is any label that has an instance in the program but does not mark any operator[12]. If S is a mini-NIL program, then let us denote the set of its final labels by $F(S)$ (or simply F when S is implicit).

Let us also assume in the sequel, that all mini-NIL programs are represented in a linear form, i.e. as lists of operators.

Informally speaking, execution of a mini-NIL program starts from any operator marked by the label '0' and finishes with a pass of control to any label that does not mark any operator in the program.

Definition 7. A configuration is a pair of the form (l, σ), where l is a label and σ is a state. A firing of an assignment operator '$l: \ x := \tau \ goto \ L$' is a pair of configurations $((l, \sigma), (l', \ \sigma'))$, where $l' \in L$ and $\sigma' = upd(\sigma, x, \sigma(\tau))$. A firing of a condition operator '$l: \ if \ \xi \ then \ L^+ \ else \ L^-$' is a pair of configurations $(l, \sigma) \ \ (l', \sigma)$, where $l' \in L^+$, if $\sigma \models \xi$, or $l' \in L^-$ otherwise.

Definition 8. Assume that we are given a mini-NIL program P. A step (or small step) of P is a firing of any operator in P. A start configuration of P is any configuration with the label 0. A final configuration of P is any configuration with a label that does not mark any operator in P. A trace of P is any finite sequence of configurations such that every consequential pair of configurations within the sequence is a step of P. A computational trace of P is a trace that starts with a start configuration and finishes with a final configuration. For any mini-NIL program P, let $SSS(P)$ be the following binary relation:

$\{(\sigma' \ , \ \sigma'') \in \Sigma \times \Sigma \ :$ there is a computational trace of P
that starts with the state σ' and finishes with the state $\sigma''\}$.

4. Axiomatic semantics and annotated mini-Pascal

The Pascal language has been provided with the axiomatic semantics in [8]. This semantics is presented as an inference system for reasoning about Hoare triples for Pascal programs. It is sound in the following sense: for every inference tree, if all leaves of the tree are either axioms or valid assertions, then the tree infers a valid Hoare triple. Below we present a similar inference system but for mini-Pascal.

[11] i.e. the assignment and condition operators

[12] It means that a terminal label occur in '*goto*', '*then*', or '*else*' section(s) of some operator(s) but does not mark any operator in the program.

Definition 9. The axiomatic semantics (AxS) for mini-Pascal is defined by the following inference system[13] :

$T: \ \frac{}{\{(\xi \to \psi\}\xi?\{\psi\}}$	$E: \ \frac{}{\{(\psi\}skip\{\psi\}}$
$A: \ \frac{}{\{\psi_{\tau/x}\}x{:=}\tau\{\psi\}}$	$S: \ \frac{\{\phi\}\alpha\{\chi\},\ \{\chi\}\beta\{\psi\}}{\{\phi\}(\alpha\ ;\ \beta)\{\psi\}}$
$C: \ \frac{\{(\phi \wedge \xi)\}\alpha\{\psi\},\ \{(\phi \wedge (\neg\xi))\}\beta\{\psi\}}{\{\phi\}(if\ \xi\ then\ \alpha\ else\ \beta)\{\psi\}}$	$L: \ \frac{\{(\iota \wedge \xi)\}\alpha\{\iota\}}{\{\iota\}(while\ \xi\ do\ \alpha)\{(\iota \wedge (\neg\xi))\}}$
$M_{pre}: \ \frac{(\phi \to \chi),\ \{\chi\}\alpha\{\psi\}}{\{\phi\}\alpha\{\psi\}}$	$M_{pst}: \ \frac{\{\phi\}\alpha\{\chi\},\ (\chi \to \psi)}{\{\phi\}\alpha\{\psi\}}$

A semi-proof is an inference tree in the axiomatic semantics such that all its leaves are either instances of axioms T, E, and A, or (program-free) assertions; all assertions in a semi-proof are called verification conditions for this semi-proof. A proof in the axiomatic semantics is a semi-proof such that all its verification conditions are valid assertions. A triple $\{\phi\}\alpha\{\psi\}$ is said to be provable in the axiomatic semantics ($\vdash_{AxS} \{\phi\}\alpha\{\psi\}$), if there exists a proof with $\{\phi\}\alpha\{\psi\}$ in the root. A set of assertions VC is a set of verification conditions for a triple, if the triple has a semi-proof with VC as its verification conditions.

The axiomatic semantics of mini-Pascal is sound in the following sense.

Proposition 2. *Every provable Hoare triple is SOS-sound.*

Proof. A standard induction on the height of a proof-tree. ■

The major problem with proving Hoare triples is to construct a set of verification conditions. To overcome the trouble, it makes sense to ask a developer to provide a program with 'invariants' for all loops. It leads to the notion of annotated mini-Pascal programs.

Definition 10.
An annotated mini-Pascal program is a program, where every 'while-do' loop is tagged by an assertion. This assertion is called an invariant of the corresponding loop. If ι is an invariant of a loop, then it is tagged to the

[13] 'T'= 'Test', 'E'= 'Empty', 'A'= 'Assignment', 'S'= 'Sequencing', 'C'= 'Choice', 'L'= 'Loop', 'M'= 'Modification'.

corresponding '*while*' as a superscript '$while^{\iota}$'. Hoare triple with annotated mini-Pascal program is Hoare-valid, if it has a proof in the axiomatic semantics such that all instances of the Loop rule L in the proof use tagged loop invariants: $\frac{\{(\iota\wedge\xi)\}\alpha\{\iota\}}{\{\iota\}(while^{\iota}\ \xi\ do\ \alpha)\{(\iota\wedge(\neg\xi))\}}$.

Let us define below two recursive algorithms wp and vc. Both algorithms have two arguments: an annotated mini-Pascal program and an assertion. The algorithm wp returns an assertions, the algorithm vc returns a set of assertions.

Algorithm 1.

- $wp(\xi?,\ \psi)\ =\ (\xi \rightarrow \psi)$
- $wp(skip,\ \psi)\ =\ \psi$
- $wp(x := \tau,\ \psi)\ =\ \psi_{\tau/x}$
- $wp((\alpha;\beta),\ \psi)\ =\ wp(\alpha,\ wp(\beta,\ \psi))$
- $wp((if\ \xi\ then\ \alpha\ else\ \beta),\ \psi)\ =\ ((\xi\wedge wp(\alpha,\ \psi))\ \vee\ ((\neg\xi)\wedge wp(\beta,\ \psi)))$
- $wp((while^{\iota}\ \xi\ do\ \alpha),\ \psi)\ =\ \iota$

Algorithm 2.

- $vc(\xi?,\ \psi)\ =\ \emptyset$
- $wp(skip,\ \psi)\ =\ \emptyset$
- $vc(x := \tau,\ \psi)\ =\ \emptyset$
- $vc((\alpha;\beta),\ \psi)\ =\ vc(\alpha,\ wp(\beta,\ \psi))\ \cup\ vc(\beta,\ \psi)$
- $vc((if\ \xi\ then\ \alpha\ else\ \beta),\ \psi)\ =\ \ vc(\alpha,\ \psi)\ \cup\ vc(\beta,\ \psi)$
- $vc((while^{\iota}\ \xi\ do\ \alpha),\ \psi)\ =$
  $vc(\alpha,\ \iota)\ \cup\{((\iota\wedge(\neg\xi))\rightarrow\psi)\ ,\ ((\iota\wedge\xi)\rightarrow wp(\alpha,\iota))\}$

Proposition 3.

1. *For every annotated mini-Pascal program α and every assertion ψ, the cardinality of a set $vc(\alpha,\ \psi)$ is the doubled number of while-loops in α.*
2. *There are some exponential functions $T(x,y)$ and $S(x,y)$ of integer arguments such that for every annotated mini-Pascal program α and every assertion ψ the run-time of the algorithm $wp(\alpha,\ \psi)$ and the size of the assertion $wp(\alpha,\ \psi)$ have upper bounds $T(m,n)$ and $S(m,n)$, where m and n are the sizes of α and ψ, respectively.*

Table 1. Sample programs that are exponentially hard for Algorithms 1 and 2

α_k	β_k
$x_1 := f(x_0, x_0);$	$if\ p(x)$ $\quad then\ x := g(x)$ $\quad\quad else\ x := h(x);$
$x_2 := f(x_1, x_1);$	$if\ p(x)$ $\quad then\ x := g(x)$ $\quad\quad else\ x := h(x);$
.....	.....
$x_{n-1} := f(x_{k-2}, x_{k-2});$	$if\ p(x)$ $\quad then\ x := g(x)$ $\quad\quad else\ x := h(x);$
$x := f(x_{k-1}, x_{k-1});$	$if\ p(x)$ $\quad then\ x := g(x)$ $\quad\quad else\ x := h(x);$

3. *There exists an assertion ψ and an infinite series of annotated mini-Pascal programs α_k, $k \geq 0$, where programs are of the size proportional to k, such that the run-time of the algorithm $wp(\alpha_k,\ \psi)$ and the size of terms in the assertion $wp(\alpha_k,\ \psi)$ have lower bounds 2^k.*

4. *There exists an assertion ψ and an infinite series of annotated mini-Pascal programs β_k, $k \geq 0$, where programs are of the size proportional to k, such that the run-time of the algorithm $wp(\beta_k,\ \psi)$ and the size of assertions $wp(\beta_k,\ \psi)$ have lower bounds 2^k, while all terms in these assertions have a constant size.*

Proof. The first statement (about cardinality) is trivial to prove by structural induction. The second statement also can be proved by structural induction. A very detailed parametric analysis of these functions has been carried out in [1]. The last two statements (lower bounds) can be proved by two parameterized examples that are presented in Tab. 1. It is easy to see that $wp(\alpha_k, c = x)$ is $c = \tau_k$, where τ_k is a term that is an isomorphic complete binary tree of height k. Similarly, $wp(\beta_k, c = x)$ is a formula ϕ_k being represented by a decision tree, it contains a subtree that is isomorphic to the complete binary tree of height k. Both facts are easy to prove by induction. Please refer to Figure 1 for illustration: the left part depicts τ_3, the right part – ϕ_2. ■

The above algorithms 1 and 2 are very important for formal verification due to the following proposition.

Proposition 4.

- *For every Hoare triple* $\{\phi\}\alpha\{\psi\}$ *with an annotated mini-Pascal program, the triple* $\{\phi\}\alpha\{\psi\}$ *is Hoare-valid iff it has a proof (in the axiomatic semantics) with a set of verification conditions* $\{(\phi \to wp(\alpha,\ \psi))\} \cup vc(\alpha,\ \psi)$.
- *For every Hoare triple* $\{\phi\}\alpha\{\psi\}$ *with a loop-free mini-Pascal program, the triple* $\{\phi\}\alpha\{\psi\}$ *is SOS-valid iff its single verification condition* $(\phi \to wp(\alpha,\ \psi))$ *is valid.*

Proof. The first statement has a trivial and non-so-trivial parts. A trivial part is: if a triple $\{\phi\}\alpha\{\psi\}$ has a proof with a set of verification conditions $\{(\phi \to wp(\alpha,\ \psi))\} \cup vc(\alpha,\ \psi)$, then it is Hoare-valid. A non-so-trivial part is the backward implication. But this implication can be proved by induction on the height of a proof tree for a Hoare-valid triple.

The second stament also comprises two implications. The trivial implication states that SOS-validity follows from validity of the single verification condition. It follows from the first statement of Proposition 3. The backward implication states that validity of the verification condition follows from SOS-validity of a loop-free triple. A proof can be carried out by induction on the height of a syntax tree of a loop-free program. It has been proved first in more general settings in [5]. ■

The above proposition implies that for any triple $\{\phi\}\alpha\{\psi\}$ with annotated mini-Pascal program, the proof-search in the axiomatic semantics is equivalent to validation of the following set of assertions: $\{(\phi \to wp(\alpha,\ \psi))\} \cup vc(\alpha,\ \psi)$. It means that the algorithms *wp* and *vc* provide us with a method of verification condition generation for Hoare triples with annotated mini-Pascal programs. Proposition 4 states that the method is *sound and complete with respect to Hoare-validity.* Together with the soundness of the axiomatic semantics (Proposition 2), it implies the next corollary.

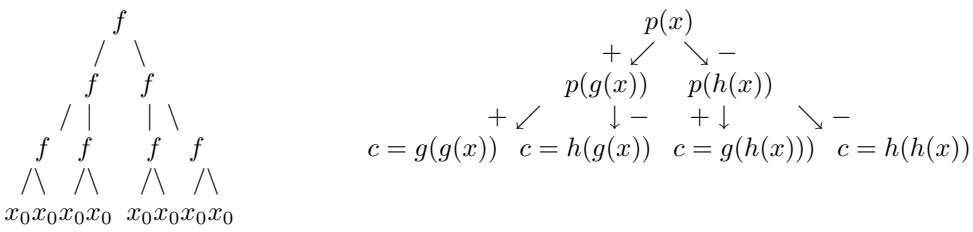

Figure 1. Tree representation of a term τ_3 and a formula ϕ_2

Corollary 1. *For every Hoare triple* $\{\phi\}\alpha\{\psi\}$ *with an annotated mini-Pascal program, if all assertions in a set* $\{(\phi \to wp(\alpha,\ \psi))\} \cup vc(\alpha,\ \psi)$ *are valid, then the triple* $\{\phi\}\alpha\{\psi\}$ *is SOS-valid.*

But the exponential space and time complexity (Proposition 3) of Algorithms 1 and 2 makes them impractical. To the best of our knowledge, all verification research and experiments fulfilled before new millennium put up with this exponential explosion as inevitable payoff.

The first research that successfully attacked this unfeasible complexity was reported in paper [6]. The cited paper remarks that there are two reasons for the blow-up:

- every assignment $x := \tau$ may cause multiple instantiations of the term τ instead of the variable x in a postcondition ψ;
- every choice $if\ \xi\ then\ \alpha\ else\ \beta$ duplicates the condition ξ and postcondition ψ, since it couples it with *then*-branch α and *else*-branch β.

For avoiding these term multiplicity and condition duplication, paper [6] uses fresh auxiliary variables and two special constructs **assert** and **assume** (the last is equivalent to our guard). These constructs provide the opportunity to translate (in a linear time) any non-deterministic guarded loop-free program into the so-called passive form. A program in the passive form does not change its variable values but can terminate normally or abnormally. For any program in the passive form, the weakest precondition for normal termination can be constructed in the linear time and the weakest precondition for abnormal termination can be constructed in the quadratic time. Translation to the passive form and normal/abnormal termination condition generation resulted in the quadratic verification condition generation algorithm. This algorithm is sound and complete for loop-free guarded structured programs. Later this approach was extended to the unstructured case [2].

We can not explain the method of [6, 2] in full details, since mini-Pascal has no explicit **assert** construct. But we still can provide some 'flavor' of the basic ideas of [6, 2]. Roughly speaking, to avoid exponential explosion, we need to modify the *wp* algorithm 1. Algorithm 3 below is an optimized[14] version of Algorithm 1. The revised algorithm *rewp* also has two arguments (an annotated mini-Pascal program and an assertion) and returns an assertions. It is assumed that the algorithm has an access to a global stack of auxiliary variables with an unbounded amount of fresh variables of any legal type.

[14] It is just an optimized version! We do not claim that it is quadratic in time and space!

Algorithm 3.

- $rewp(\xi?,\ \psi)\ =\ (\xi \to \psi)$;
- $rewp(skip,\ \psi)\ =\ \psi$;
- $rewp(x := \tau,\ \psi)\ =\ \exists y.\ (y = \tau \wedge \psi_{y/x})$, *where*
  - y *is a fresh variable of* x*-type;*
- $rewp((\alpha;\beta),\ \psi)\ =\ rewp(\alpha,\ wp(\beta,\ \psi))$;
- $rewp((if\ \xi\ then\ \alpha\ else\ \beta),\ \psi)\ =$
  $=\ \exists\overline{y}.(\psi_{\overline{y}/\overline{x}}\ \wedge\ ((\xi \wedge rewp(\alpha,\ \overline{y} = \overline{x}))\ \vee\ ((\neg\xi) \wedge rewp(\beta,\ \overline{y} = \overline{x}))))$, *where*
  - $\overline{x}$ *is a vector of all free variables in the formula* ψ,
  - $\overline{y}$ *is a vector of fresh variables component-wise type-compatible with* $\overline{x}$,
  - $\psi_{\overline{y}/\overline{x}}$ *denotes a component-wise substitution of* $\overline{y}$ *instead of* $\overline{x}$ *in* ψ,
  - $\overline{y} = \overline{x}$ *is a conjunction of component-wise equalities;*
- $rewp((while^{\iota}\ \xi\ do\ \alpha),\ \psi)\ =\ \iota$.

Proposition 5. *For every annotated mini-Pascal program* α *and every assertion* ψ*, the formulas* $wp(\alpha, \psi)$ *and* $rewp(\alpha, \psi)$ *are equivalent.*

Let us note that, while this algorithm is still exponential in size of the constructed assertion (due to duplication of a post-condition '$\overline{y} = \overline{x}$ ' in choice), it is not exponential in the size of terms, and it duplicates a fixed postcondition (that is a system of equalities between variables).

5. Floyd method for annotated mini-NIL

There are several very useful graphs related to unstructured programs. We are most interested in flowcharts and control-passing graphs. A flowchart representation of a mini-NIL program is comprehensive. In contrast, a control-passing graph represents only a part of information about a mini-NIL program.

Definition 11. A flowchart of a mini-NIL program is a directed graph, where nodes correspond (in 1-to-1 manner) to labels and operator's bodies, and edges (also in 1-to-1 manner) – to control-passing: from a label – to a body of an operator that is marked by this label (if any), from an operator body – through '*goto*' or '*then*' and '*else*' – to other labels (if any); edges that correspond to control-passing through '*then*' are marked by '+' and edge that and correspond to control-passing through '*else*' are marked by '–'.

Definition 12. A control-passing[15] graph of a mini-NIL program is a directed graph, where nodes correspond (in 1-to-1 manner) to labels, and edges to control-passing between labels (ignoring operator bodies): there is an edge from a label 'k' to a label 'l' iff an operator marked by 'k' passes a control to the label 'l' through '*goto*', '*then*' or '*else*'.

Due to these representations, we can extend the graph-theoretic terminology to mini-NIL programs. For example, we can speak about a path between two labels (nodes) of a program, a cycle, etc.

Definition 13.
An annotated mini-NIL program is a program, where some labels are tagged by assertions in such a way that the following annotation conditions hold:

- The initial label '0' is tagged by an assertion.
- All final labels are tagged by some assertion (one and the same for all).
- Every cycle through the program flowchart contains a tagged label.

If a label l has a tagged assertion, then the label is called a contract point and the assertion is called an invariant of the label. If ϕ is an invariant of a label l in an annotated mini-NIL program, then it is tagged to the label as a superscript 'l^{ϕ}'. If a label l of an annotated mini-NIL program has an invariant, then let us refer to this invariant as ϕ_l. Let us refer to a common invariant of all final labels as ϕ_F.

The primary topic of our paper is efficient generation of verification conditions. So it is very important to evaluate the complexity of all related algorithms. For this let us fix a number of parameters that contribute to the complexity of the algorithms. For any assertion ψ, two parameters $|\psi|$ and $\tau(\psi)$ are already in use. Let us define similar parameters for (annotated) mini-NIL programs. For any annotated mini-NIL program P, let $|P|$ be the size of P (i.e. the overall number of symbols in P as a word including invariants), let $n(P)$ be the number of operators that occur in P, and let $l(P)$ be the number of labels that occur in P. In the sequel we assume that mini-NIL programs are represented in a linear form as lists of operators.

Proposition 6. *Annotation correcteness for mini-NIL programs can be checked in the cubic time* $O(|P| \times l(P)^2)$.

[15] We use the term 'control-passing' instead of a more conventional 'control-flow' to avoid ambiguities with 'flow-chart'.

Proof. It is very easy to check (in a time linear in $|P|$) that the initial label '0' has an invariant. It is a little bit more complicated to check (in a time squared in $|P|$) that all final labels have a joint invariant. It is more challenging to check whether every cycle through a flowchart contains a contract point. But it can be carried out as follows.

The control-passing graph G for P can be constructed in time $O(|P| \times l(P)^2)$ by scanning P. Then in time $O(l(P)^2)$ it is possible to assign an infinite weight ∞ to all edges that lead to contract points, and weight 1 to all other edges. After that we are in conditions of Floyd-Warshall's algorithm that computes the weights of the lightest paths between all pairs of nodes of a weighted graph. This algorithm requires the cubic time $O(l(P)^3)$. A criterion follows: every cycle through a flowchart contains a contract point iff for every label (i.e. a node of G) the lightest path from this label to itself has infinite weight. ■

Let us define below a recursive algorithm mP that converts every non-empty path between any two labels (maybe equal) in a flowchart of a mini-NIL program to a corresponding mini-Pascal program that is a sequential composition of a single empty construct and a number of assignments and tests. Observe that every non-empty path of this kind should begin with a label and then repeat several times the following pattern: 'an edge' - 'a body of operator' - 'an edge (marked by '+' or '–' maybe)' - 'a label'. Due to this reason, the algorithm has four patterns for a path:

- 'prefix' - 'edge' - 'assignment body' - 'unmarked edge' - 'label';
- 'prefix' - 'edge' - 'quantifier-free formula' - 'edge marked by +' - 'label';
- 'prefix' - 'edge' - 'quantifier-free formula' - 'edge marked by –' - 'label';
- 'label'.

Algorithm 4.

- $mP(\pi \ \to x := \tau \ \to l) \ = \ (mP(\pi) \ ; \ x := \tau)$;
- $mP(\pi \ \to \xi \ \stackrel{+}{\to} l) \ = \ (mP(\pi) \ ; \ \xi?)$;
- $mP(\pi \ \to \xi \ \stackrel{-}{\to} l) \ = \ (mP(\pi) \ ; \ (\neg\xi)?)$;
- $mP(l) \ = \ skip$.

Definition 14. Let P be an annotated mini-NIL program. A contract-free path (through P) is any path through the flowchart that has no instances of contract points inside (but that can start and/or end in a contract point). The program P is said to be Floyd-valid if for every pair of contract points l and k, for every contract-free path π from k to l through the flowchart, the following triple $\{\phi_k\}\ mP(\pi)\ \{\phi_l\}$ is valid: $\models \{\phi_k\}\ mP(\pi)\ \{\phi_l\}$.

Proposition 7. *For every Floyd-valid annotated mini-NIL program P, Hoare triple $\{\phi_0\}P\{\phi_F\}$ is SSS-valid.*

Proof. This proposition follows from the next statement: for every trace that starts and finishes in any contract points k and l of a Floyd-valid program P, if this trace starts in a state $\sigma' \models \phi_k$, then it finishes in a state $\sigma'' \models \phi_l$. This statement can be proved by induction on the number of instances of contract points in the trace. In deterministic settings this statement has been proved for the first time in [7]. ■

6. Floyd and verification conditions for mini-NIL

The next proposition follows from part 2 of Proposition 4.

Proposition 8. *For every annotated mini-NIL program P, P is Floyd-valid iff for every pair of contract points l and k, for every non-empty contract-free path π from k to l through the flowchart, the following assertion $(\phi_k \to wp(mP(\pi),\ \phi_l))$ is valid.*

This proposition leads to the next definition.

Definition 15. For every annotated mini-NIL program P, a set of Floyd conditions is any set of assertions of the following kind:

$\{(\phi_k \to \theta(l,\pi))$: k and l are contract points,
π is a non-empty contract-free path from k to l through the flowchart,
and the assertion $\theta(l,\pi)$ is equivalent to $wp(mP(\pi),\ \phi_l)\}$.

The next proposition follows immediately from Proposition 8 and Definition 15.

Proposition 9. *For every annotated mini-NIL program P and any set of Floyd conditions FC, P is Floyd-valid iff all assertions in FC are valid.*

Every annotated mini-NIL program P has the following canonical set of Floyd conditions:

$\{(\phi_k \to wp(mP(\pi), \phi_l))$: k and l are contract points,
π is a non-empty contract-free path from k to l through the flowchart$\}$.

Unfortunately, construction of the canonical Floyd conditions can be exponential in time and in space due to

- consequential substitutions of terms instead of variables,
- exponential number of contract-free paths between contract points.

We already know how to cope with the first problem: it is possible to use $rewp$ instead of wp. In particular, we can define another set of Floyd conditions

$\{(\phi_k \to rewp(mP(\pi), \phi_l)) \ : \ k$ and l are contract points,
π is a non-empty contract-free path from k to l through the flowchart$\}$.

Let us refer to this set as the improved Floyd conditions (for P). Every assertion in the improved Floyd set has a linear size. But we still do not know how to reduce the exponential number of improved Floyd conditions. The problem looks like a reincarnation of the exponential explosion through the choice constructs in mini-Pascal.

An alternative to exponentially large sets of Floyd conditions are the sets of verification conditions that always consist of a linear number of assertions.

Definition 16.

- For every annotated mini-NIL program P, for every label[16] k within this program, let $wp(k)$ be the following formula

$$\bigwedge_{\substack{l \text{ is a contract point,} \\ \pi \text{ is a non-empty contract-free path to } l}} wp(mP(\pi),\ \phi_l).$$

- For every annotated mini-NIL program P, a set of verification conditions is any set of assertions

$\{(\phi_k \to \theta_k) \ : \ k$ is a contract point, and assertion θ_k is equivalent to $wp_k\}$.

Every annotated mini-NIL program P has the following canonical set of verification conditions $\{(\phi_k \to wp_k) \ : \ k$ is a contract point$\}$. The next proposition immediately follows from Definition 16 and Proposition 8.

Proposition 10. *For every annotated mini-NIL program P and any set of verification conditions VC, P is Floyd-valid iff all assertions in VC are valid.*

As follows from the above proposition, efficient generation of a set of verification conditions is very important for verification of mini-NIL programs. By definition, the number of assertions in every set of verification conditions for an annotated mini-NIL program is not greater than the number of contract points in the program, i.e. cardinality of a set of verification conditions is 'tiny' with respect to exponential cardinality of a set of Floyd

[16] not necessary a contract point

conditions. But construction of assertions in the canonical set can be exponential in time and in space due to the same reasons as for the canonical Floyd conditions. Again we can use *rewp* instead of *wp*. But we still do not know how to handle exponential explosion that is due to the exponential number of paths.

7. System of verification conditions

Lemma 1. *Let P be an annotated mini-NIL program. For any label k that is not a contract point, the following holds:*

- *if k marks an assignment operator '$k:\ x := \tau$ goto L' then*

$$wp_k \leftrightarrow \exists y.(y = \tau \wedge \bigwedge_{l \in L} (wp'_l)_{y/x}),$$

- *if k marks a condition operator '$k:\ if\ \xi\ then\ L^+\ else\ L^-$' then*

$$wp_k \leftrightarrow (\xi \rightarrow \bigwedge_{l \in (L^+)} wp\prime_l) \ \wedge\ ((\neg\xi) \rightarrow \bigwedge_{l \in (L^-)} wp'_l),$$

where 'y' is any fresh variable of x-type, and

$$wp'_l \ = \begin{cases} wp_l \ , \text{ if } l \text{ is not a contract point;} \\ \phi_l \ , \text{ if } l \text{ is a contract point} \end{cases}$$

Please refer to Appendix A for proof.

The above lemma leads to the next definition, where we assume that we have the unbounded amount of uninterpreted predicate symbols of any signature (i.e. argument types).

Definition 17. Let P be an annotated mini-NIL program. Let *var* be a list $z_0, z_1, \ldots$ of all variables that occur in P (but are not bounded in any invariant). For every label k in P, let θ_k be a fresh uninterpreted predicate symbol with a signature that component-wise matches the types of variables in *var*; for every list of terms *ter* that is component-wise type-compatible with *var*, let $\theta_k(ter)$ be a formula that results from instantiating terms from *ter* instead of the corresponding formal arguments in θ. For every label k within P, let a verification equation EQ_k be the following formula with uninterpreted symbols:

- if k marks an assignment operator '$k:\ x := \tau$ *goto* L' then

$$\theta_k(var) \leftrightarrow \exists y.(y = \tau \wedge \bigwedge_{l \in L} (\theta'_l(var))_{y/x}),$$

- if k marks a condition operator '$k:\ if\ \xi\ then\ L^+\ else\ L^-$' then

$$\theta_k(var) \leftrightarrow (\xi \rightarrow \bigwedge_{l \in (L^+)} \theta\prime_l(var)) \ \wedge \ ((\neg\xi) \rightarrow \bigwedge_{l \in (L^-)} \theta'_l(var)),$$

where 'y' is any fresh variable of x-type, and

$$\theta'_l \ = \begin{cases} \theta_l \text{ , if } l \text{ is not a contract point;} \\ \phi_l \text{ , if } l \text{ is a contract point} \end{cases}$$

Let the system of verification equations (for P) be the following universally quantified conjunction of all verification equations:

$$\forall var.(\bigwedge_{k \text{is a labell in } P} EQ_k)$$

where a quantifier prefix '$\forall var.$' is a shorthand for $\forall z_0.\forall z_1 \ldots$.

Definition 18. Let P be an annotated mini-NIL program. A solution of the system of verification equations for P is any interpretation I of predicate symbols θ_k (k is a label) that makes the system valid: $I \models \forall var.(\bigwedge_{k \text{is a labell in } P} EQ_k)$.

Lemma 2. *For every annotated mini-NIL program, its system of verification equations has a unique solution. This solution is $I(\theta_k(ver)) = wp_k$ for every label k within the program.*

Please refer to Appendix B for proof.

Theorem 1. *Let P be an annotated mini-NIL program and S_P be its system of verification equations with uninterpreted symbols $\{\theta_l : l$ is a label within $P\}$.*

1. *The system S_P consists of a linear number of equations (at most $n(P)$) and can be constructed in quadratic space and time $O(n(P) \times |P|)$.*
2. *The following assertion $S_P \rightarrow (\bigwedge_{k \text{ is a contract point in } P}(\phi_k \rightarrow \theta_k)$) is valid iff P is Floyd valid.*

Proof. The first statement is quite simple. The system consists of $n(p)$ equations at most, since final labels have no corresponding equations. Then each equation is constructed by scanning the corresponding operator in $O(|P|)$ time and space. The second state follows from Lemma 2 and Proposition 10: for any interpretation I of uninterpreted symbols $\{\theta_l : l$ is a label within $P\}$,

- if $I \models S_P$ holds, then I is a solution of the system and (by Lemma 2) $I(\theta_k) = wp_k$ for every contract point;
- hence, $I \models (\bigwedge_{k \text{ is a contract point in } P}(\phi_k \rightarrow \theta_k)$) is equivalent to

$$\models \Big(\bigwedge_{k \text{ is a contract point in } P} (\phi_k \to wp_k) \Big),$$

i.e. to Floyd validity (according to Proposition 10).

■

8. Concluding remarks

Let us sketch first how to expand the efficient verification condition generation to deterministic structured programs, to cover both mini-Pascal and mini-NIL. Assume that we are given Hoare triple $\{phi\}\alpha\{psi\}$ with an annotated mini-Pascal program. Translate this *triple* to an annotated mini-NIL program in accordance with the following 'algorithm'.

Algorithm 5.

1. *Translate α to a mini-NIL program $mN(\alpha)$ ignoring annotations. In this translation, every loop 'while ξ do ...' corresponds to some set of operators with some 'loop-heading' conditional operator 'l : if ξ then $\{start\_body\}$ else $\{exit\_loop\}$'.*
2. *Annotate mini-NIL program $mN(\alpha)$ as follows: tag the initial label '0' as ϕ, the final label(s) in $P(\alpha)$ as ψ, and tag labels of loop-heading conditional operators as invariants of corresponding while-loops. Let $aN(\{phi\}\alpha\{psi\})$ be the resulting annotated mini-NIL program.*

Hoare triple with annotated mini-Pascal program	annotated mini-NIL program
$\{a \geq 1\}$	$0^{a \geq 1}$: $x := a$ *goto* $\{1\}$
$x := 1$; $y := 1$;	1 : $y := 1$ *goto* $\{2\}$
while$^{y=x^2 \wedge (x-1)^2<a}$ $y \leq a$ *do*	$2^{y=x^2 \wedge (x-1)^2<a}$: *if* $y \leq a$ *then* $\{3\}$ *else* $\{5\}$
$(y := y + 2 \times x + 1$; $x := x + 1)$	3 : $y := y + 2 \times x + 1$ *goto* $\{4\}$
; $x := x - 1$	4 : $x := x + 1$ *goto* $\{2\}$
$\{x^2 \leq a \ \wedge \ (x+1)^2 > a\}$	5 : $x := x - 1$ *goto* $\{6^{x^2 \leq a \ \wedge \ (x+1)^2 > a}\}$

Figure 2. Example of translation

We do not provide the details how to translate mini-Pascal to mini-NIL ignoring annotations. But Figure 2 presents an example of translation of Hoare triple with an annotated mini-Pascal program to an annotated mini-NIL program. Nevertheless, we can claim that in the general case this translation can be carried out in a time linear in size of a triple so that the following equivalence holds: $\{phi\}\alpha\{psi\}$ is Hoare-valid iff the program

$aN\{phi\}\alpha\{psi\}$) is Floyd-valid. In this way we can expand the efficient verification condition generation to deterministic structured programs.

Then let us sketch a background of this research on efficient generation of compact verification conditions presented in this paper. This research is a part of the F@BOOL@ project. The primary target of the project is to develop a transparent, compact, and portable verifying compiler for component-based programs. It is assumed that F@BOOL@ will exploit efficient decision procedures for validation of correctness conditions (instead of semi-automatic theorem provers). In particular, F@BOOL@ will extensively use SAT-solvers for validation of Boolean encoding of correctness conditions. The target group of F@BOOL@ users comprises Computer Science, Information Technology and Mathematics students willing to comprehend program verification.

The major outcome of the presented research is the following explicit alternatives for verification of annotated mini-NIL programs:

- to generate and verify exponentially many Floyd conditions of linear size;
- to generate and verify a small set (linear in the number of elements) of exponentially big verification conditions;
- to generate (in accordance with theorem 1) and verify a single and relatively small (quadratic in size) assertion with uninterpreted predicate symbols.

Which alternative is better in practice is a topic for further experimental research. But in the context of the project F@BOOL@, generation and verification of improved canonical Floyd conditions should be more efficient. F@BOOL@ approach assumes propositional encoding of first-order formulas over a ring of residuals and then validation by SAT-solvers. Hence the size of formulas and the absence of uninterpreted predicate symbols could be very critical for this approach.

In contrast, in a more conventional verification approach based on automatic theorem proving, generation and verification of a single and small assertion could be better in spite of auxiliary uniterpreted predicate symbols. In particular, the idea of theorem 1 comes from the project Spectrum-II that is oriented on verification of C#-programs [10]. In this project, the auxiliary uniterpreted predicate symbols are called 'lazy invariants' and they are extensively used in translation of C#-programs from C#-light to C#-kernel levels[17].

[17]Both C#-light and C#-kernel are subsets of C#. More precisely, C#-kernel includes additionally metainstructions which handle directly the virtual machine for the C#-light language.

References

[1] Anureev I.S., Bodin E.V., Shilov N.V. Complexity of Axiomatic Semantics Verification Conditions for mini-Pascal. — Manuscript, available upon demand by email.

[2] Barnett M., Leino K. R. M. Weakest-Precondition of Unstructured Programs // Proc. of Workshop on Program Analysis For Software Tools and Engineering (PASTE). — 2005. — P. 82–87.

[3] Bodin E., Kalinina N., Shilov N. Verifying Compiler F@BOOL@ Part I: Outlines of F@BOOL@ project in the context of component-based programming. Mini-NIL: a prototype of F@BOOL@ virtual machine language. — Novosibirsk, 2005. — (Prepr. / IIS SB RAS; N 131).

[4] Bodin E., Kalinina N., Shilov N. Verifying Compiler F@BOOL@ Part II: Logical annotations in mini-NIL, their static and run-time semantics. — Novosibirsk, 2006. — (Prepr. / IIS SB RAS; N 138).

[5] Dijkstra W.E. The Dicsipline of programming. — Prentice Hall, 1976.

[6] Flanagan C., Saxe J.B. Avoiding Exponential Explosion: Generating Compact Verification Conditions // The 28th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. — 2001. — P. 193–205.

[7] Floyd R.W. Assigning Meanings to Programs // Proc. of a Symposium in Applied Mathematics. Mathematical Aspects of Computer Science. Vol. 19. — Providence, R. I.: American Math. Society, 1967. — P. 19-32.

[8] Hoare, C.A.R. and Wirth N. An Axiomatic Definition of the Programming Language PASCAL // Acta Informatica. — 1973. — N 2. — P. 335–355.

[9] Hoare C. A. R. The Verifying Compiler: A Grand Challenge for Computing Research // Proc. of PSI'2003. — Lect. Notes in Comput. Sci. — 2003. — Vol. 2890. — P. 1–12.

[10] Nepomniaschy V.A., Anureev I.S., Dubranovskii, I.V. Promsky A.V. Towards Verification of C# Programs: A Three-Level Approach // Programming and Computer Software. — 2006. — Vol. 32(4). — P. 190–202.

A. Proof of Lemma 1

Let us discuss the assignment case only. Any contract-free path π that starts at k and goes to a contract point without visiting any other contract point matches the following pattern:

$$k \to x := \tau \to \pi'$$

where π' is a contract-free path that starts from some label $l \in L$. Hence wp_k can be represented as a conjunction of the following two formulas:

$$\bigwedge_{\substack{l \in L,\\ l \text{ is a contract point}}} wp(x := \tau,\ \phi_l))$$

$$\bigwedge_{\substack{l \in L,\\ l \text{ is not a contract point}}} \Big(\bigwedge_{\substack{l' \text{ is a contract point,}\\ \pi' \text{ is a non-empty contract-free path from } l \text{ to } l'}} wp((x := \tau\ ;\ mP(\pi'))\ , \phi_{l'})\ \Big).$$

The first formula can be processed directly as follows:

$$\bigwedge_{\substack{l \in L,\\ l \text{ is a contract point}}} wp(x := \tau,\ \phi_l)) \Leftrightarrow \bigwedge_{\substack{l \in L,\\ l \text{ is a contract point}}} rewp(x := \tau,\ \phi_l)) \Leftrightarrow$$

$$\Leftrightarrow \bigwedge_{\substack{l \in L,\\ l \text{ is a contract point}}} \exists y.(y = \tau \wedge (\phi_l)_{y/x})) \Leftrightarrow$$

$$\Leftrightarrow \exists y.(y = \tau \wedge \bigwedge_{\substack{l \in L,\\ l \text{ is a contract point}}} (\phi_l)_{y/x}).$$

For the internal conjunction of the second formula we have:

$$\bigwedge_{\substack{l' \text{ is a contract point,}\\ \pi' \text{ is a non-empty contract-free path from } l \text{ to } l'}} wp((x := \tau\ ;\ mP(\pi'))\ , \phi_{l'}) \Leftrightarrow [18]$$

$$\Leftrightarrow \bigwedge_{\substack{l' \text{ is a contract point,}\\ \pi' \text{ is a non-empty contract-free path from } l \text{ to } l'}} wp(x := \tau,\ wp(mP(\pi')\ , \phi_{l'})) \Leftrightarrow [19]$$

$$\Leftrightarrow \bigwedge_{\substack{l' \text{ is a contract point,}\\ \pi' \text{ is a non-empty contract-free path from } l \text{ to } l'}} rewp(x := \tau,\ wp(mP(\pi')\ , \phi_{l'})) \Leftrightarrow [20]$$

[18] by definition of *wp* for sequential composition
[19] since *wp* is equivalent to *rewp*
[20] by definition of *rewp*

$$\Leftrightarrow \bigwedge_{\substack{l' \text{ is a contract point,} \\ \pi' \text{ is a non-empty contract-free path from } l \text{ to } l'}} \exists y.(y = \tau \ \wedge (wp(mP(\pi')\ , \phi_{l'}))_{y/x}) \ \Leftrightarrow$$

$$\Leftrightarrow \exists y.(y = \tau \ \wedge \bigwedge_{\substack{l' \text{ is a contract point,} \\ \pi' \text{ is a non-empty contract-free path from } l \text{ to } l'}} (wp(mP(\pi')\ , \phi_{l'}))_{y/x}) \ \Leftrightarrow$$

$$\Leftrightarrow \exists y.(y = \tau \ \wedge \ (\bigwedge_{\substack{l' \text{ is a contract point,} \\ \pi' \text{ is a non-empty contract-free path from } l \text{ to } l'}} wp(mP(\pi')\ , \phi_{l'}))_{y/x}) \ \Leftrightarrow$$ [21]

$$\Leftrightarrow \ \exists y.(y = \tau \ \wedge \ (wp_l)_{y/x}).$$

This finishes the proof of Lemma 1.

B. Proof of Lemma 2

Let P be an annotated program. By Lemma 1, the system of verification equations for P has a solution $I(\theta_k(var)) = wp_k$ for every label k. Let us prove that this solution is unique.

Let us define the following binary relation R on labels within P: for any labels k and l, kRl iff $k = l$, or l is not a contract point and there exists a contract-free path from k to l in the control-passing graph. Reflexivity and transitivity of the relation R is obvious. Antisymmetry of R can be proved by contradiction: if kRl and lRk hold for non-equal labels k and l, then both labels are not contract points, and there exists a cycle in the control-passing graph without any contract point. This contradicts the assumption that P is an annotated program. Hence R is a partial order.

Let us sort topologically all labels within P in accordance with this partial order R. Then the set of all labels within P can be partitioned as follows:

- let the layer L_0 comprise all minimal elements;
- for every $i \geq 0$ let the layer L_{i+1} comprise all elements that have predecessors at the layers L_0, ... L_i.

By this definition of R-layers,

[21] by definition of assertion wp_l

- for any label in L_0, all its successors via '*goto*', '*then*' or '*else*' are contract points;
- for every $i \geq 0$, for any label in L_{i+1} , all its successors via '*goto*', '*then*' or '*else*' are either contract points or labels at the layers L_0, ... L_i.

The partial order R induces a partial order on uninterpreted symbols $\theta_{...}$ within the system of verification equations: for any θ_k and θ_l, $\theta_k R \theta_l$ iff kRl holds for labels.

Observe that all equations in the system of verification equations have the following form '$\theta_k \leftrightarrow B(\{\theta'_l : l \in L\})$', where L is the set of all labels that are '*goto*'-, '*then*'- or '*else*' direct successors of k, and $B(\{\theta'_l : l \in L\})$ is a first-order combination of variables $\{\theta_l : l \in L,\ l$ is not a contract point$\}$ and assertions $\{\phi_l : l \in L,\ l$ is a contract point$\}$. Due to this we can sort topologically all equations in accordance with the partial order R on left-hand sides of equations.

After this sorting, all equations are partitioned in accordance with the layers L_0, L_1, ... so that

- for any equation $\theta_k \leftrightarrow B(\dots)$ in L_0, its right-hand side $B(\dots)$ is a first-order combination of some assertions $\{\phi_l : l$ is a contract point$\}$;
- for every $i \geq 0$, for any equation $\theta_k \leftrightarrow B(\dots)$ in L_{i+1}, its right-hand side $B(\dots)$ is a first-order combination of some variables $\{\theta_l : l \in L_j,\ 0 \leq j \leq i\}$ and some assertions $\{\phi_l : l$ is a contract point$\}$.

Let I be any solution of the system, i.e. $I \models \forall var.(\bigwedge_{k \text{is a labell in } P} EQ_k)$. Observe that, for every equation $\theta_k \leftrightarrow B(\{\theta'_l : l \in L\})$ within this system, $I(\theta_k) = \mathcal{B}(\{I(\theta'_l) : l \in L\})$, where $\mathcal{B}$ is a predicate transformer that corresponds to the first-order combination B. Hence, due to the partial order R, that layered all equations, and Lemma 1, for evey $i \geq 0$, for every $k \in L_i$, we have $I(\theta_k) = wp_k$. That is we have proved uniqueness of the solution I and finished the proof of Lemma 2.