Interactive Verification of Safety-Critical Software

Daniela da Cruz, Pedro Rangel Henriques
CCTC & Universidade do Minho
Braga, Portugal
{*danieladacruz,prh*}*@di.uminho.pt*

Jorge Sousa Pinto
HASLab/INESC TEC & Universidade do Minho
Braga, Portugal
jsp@di.uminho.pt

***Abstract*—A central issue in program verification is the generation of *verification conditions* (VCs): proof obligations which, if successfully discharged, guarantee the correctness of a program vis-à-vis a given specification. While the basic theory of program verification has been around since the 1960s, the late 1990s saw the advent of practical tools for the verification of realistic programs, and research in this area has been very active since then. Automated theorem provers have contributed decisively to these developments.**

This paper establishes a basis for the generation of verification conditions combining forward and backward reasoning, for programs consisting of mutually-recursive procedures annotated with contracts and loop invariants. We introduce also a *visual technique* to verify a program, in an interactive way, using *Verification Graphs* (VG), where a VG is a Control Flow Graph (CFG) whose edges are *labeled* with contracts (pre- and postconditions). This technique intends to help a software engineer to find statements that are not valid with respect to the program's specification.

I. Introduction

The importance of Formal Software Verification in the safety-critical domain cannot be overstated. Rather than repeating exhausted arguments in favor of the adoption of rigorous principles in the validation of code, it suffices to cite the recommendations contained in industry norms such as ISO/IEC-15408 (known as Common Criteria, for security-sensitive applications), CENELEC EN 50128 (in the railway domain), or DO-178C (in the aerospace domain), which recommend or even enforce the adoption of such principles in the development of critical applications. The DO-178C standard emphasizes the crucial role of formal methods in the area of safety critical systems. This does not however imply a diminished importance of testing, which is and will continue to be a paramount method for verifying if the software does what it is expected to do and, does not do what it is expected not to do.

Verification activities are essential to achieve the required level of confidence expected in critical software, but they are becoming increasingly costly as they require the development and maintenance of a large body of tests on larger and more complex applications. Formal program verification offers a way to reduce these costs while providing stronger guarantees than testing.

The goal of the tools known as Verification Condition Generators (VCGens) is precisely to ensure that a software satisfies its *formal specification*. A specification provides an abstract and rigorous representation of the program meaning or goal; it says *what the program is supposed to do*, rather than *how to do it*. A VCGen reads a piece of code together with a specification and computes a set of verification conditions (VCs) that are then sent to a theorem prover. If all the verification conditions of a program are correct, then the program is said to be correct. This approach can of course be used to establish the correctness of the code with respect to the contracts embedded in it.

There are two well-established methods for producing sets of verification conditions, based respectively on *backward propagation* (BWP) and *forward propagation* (FWP) of assertions. Verifying the behavior of programs is of course in general an undecidable problem, and as such automation is necessarily limited. Typical tools require the user to provide additional information, in particular *loop invariants*, and one could be tempted to think that if appropriate annotations are provided, then generating and proving verification conditions will be straightforward. It would however be wrong to think that the method used for generating VCs is not important, for the following reasons:

- The choice of employing BWP or FWP of assertions is not indifferent. Program verification tools typically resort to the former, since the latter introduces existential quantifiers in the generated VCs. The use of forward propagation has however been gaining momentum in a series of recent papers (see below).
- There are serious efficiency issues involved: a naive algorithm can produce verification conditions of exponential size on the length of the program, thus compromising any hope of automating the verification process [1]. Fortunately, it is now well understood how to prevent this explosion. It has also been understood that splitting VCs into smaller formulas may help the performance of theorem provers [2].
- The above issues have to do with the "quality" of the sets of VCs generated, with impact on the feasibility of the automated proofs. Other aspects have to do with being able to identify the particular execution paths that cause them. A method for producing VCs that facilitates this is advisable for debugging applications.

- Finally, let us recall that a failed verification may be due not only to an incorrect program, but also possibly to inadequate annotations (the user may have provided an 'invariant' annotation that is not in fact an invariant). Complementing the automated verification techniques with *interactive features* can help the users find the appropriate annotations.

Contributions of this paper: The main goal of the present paper is to investigate how forward propagation and backward propagation methods can be combined to generate verification conditions (*for the sake of space, all the propositions, lemmas and their proofs are ommitted but can be found in [3]*).

In particular, the properties of labeled CFGs (CFGs whose edges have labels corresponding to assertions propagated from the specification of a given block) are studied in the context of contract-annotated procedures. It is shown how these graphs can be used for generating verification conditions in a way that:

1) combines forward propagation of preconditions and backward propagation of postconditions, thus bringing together the advantages of forward and backward reasoning;
2) allows for a closer relation with error paths;
3) can be implemented either based on pre-defined strategies or interactively (user-guided VC generation);
4) lends itself to visualization applications.

Structure of the paper: Section II introduces our setting; Section III introduces the definition of Labeled Control Flow Graph; Section IV introduces the definition of Verification Graph and how to use it in interactive verification; Section V introduces GamaSlicer and explains how interactive verification works in practice; Section VI describes a number of applications of our approach; and Section VII closes the paper.

II. Setting

The general framework to which we adhere in this paper is the verification of programs based on their contracts. The *Design by Contract* (DbC) approach to software development [4] facilitates modular verification and certified code reuse. The contract of a component (or procedure, or method) can be regarded as a form of enriched software documentation that specifies the behaviour of that component. The development and broad adoption of annotation languages for the most popular programming languages reinforces the importance of using DbC principles in the development of programs. These include for instance the Java Modeling Language (JML) [5]; Spec# [6], a formal language for C# contracts; and the SPARK [7] subset of Ada.

The Language Syntax: To illustrate the ideas presented in the following sections we use a simple programming language. Its syntax is defined in Figure 2, in two levels (x and $\mathbf{p}$ range over sets of variables and procedure names respectively). First we form blocks (or sequences) of commands, which correspond to programs of a standard *While* programming language. These include $\mathbf{skip}$, assignment, conditionals, loops, and a procedure call command. Each loop is additionally annotated with an assertion, interpreted as a *loop invariant*. The language of assertions extends boolean expressions with implication and first-order quantification. Procedures can then be defined, consisting of a block of code annotated with two assertions that form the procedure's specification, or *contract*. A *program* is a non-empty sequence of (mutually recursive) procedure definitions.

Verification Conditions: In practice, working program verification systems require users to provide the invariants as annotations in the code, and then employ a *backward propagation strategy* to construct derivations. In fact, these derivations do not even need to be explicitly constructed: an algorithm can be used that takes as input a piece of annotated code together with a specification, and produces a set of first-order proof obligations ensuring that a derivation exists. Such an algorithm is usually known as a *verification conditions generator* (VCGen).

A VCGen for the annotated programs of Figure 2 is defined in Figure 1. In brief, the function wprec calculates the weakest precondition of a block, except in the case of loops, for which it simply returns the annotated invariant (hopefully an approximation of the weakest precondition, which would be given as a least-fixpoint solution to a recursive equation). Note that it may well be the case that this annotation is not in fact an invariant; even if it is an invariant, it is possible that it is not sufficiently strong to allow Q as a postcondition of the loop; on the other hand, the annotation does not need to be the weakest of all sufficiently strong invariants, and often is not.

The function wvc collects, for every loop in the block, additional conditions required to establish that the annotation is indeed an invariant, and that it is sufficiently strong for the loop to attain the desired postcondition. An additional verification condition is added to the set $\mathsf{VCG}^w(P, S, Q)$, stating that the precondition P must be stronger than the assertion propagated backward from Q through the block S.

We remark that this set of VCs is constructed following a strategy that propagates the postcondition Q backwards; the function wvc collects the side conditions of an implicit Hoare logic tree. When considering a block of the form $C\,;\,S$, the corresponding Hoare logic rule dictates that two derivations should be recursively considered for C and S. The strategy first considers the rightmost branch corresponding to the block S with the given postcondition Q, and then the

VERIFICATION CONDITIONS OF A BLOCK:

BACKWARD PROPAGATION:

$$\mathsf{VCG}^w(P,\ S,\ Q) = \{P \to \mathsf{wprec}(S,Q)\} \ \cup\ \mathsf{wvc}(S,Q)$$

$$\begin{aligned}
\mathsf{wprec}(\mathbf{skip}, Q) &= Q \\
\mathsf{wprec}(x := e, Q) &= Q[e/x] \\
\mathsf{wprec}(\mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f, Q) &= (b \to \mathsf{wprec}(S_t, Q)) \wedge (\neg b \to \mathsf{wprec}(S_f, Q)) \\
\mathsf{wprec}(\mathbf{while}\ b\ \mathbf{do}\ \{I\}\ S, Q) &= I \\
\mathsf{wprec}(\mathbf{call}\,\mathbf{p}, Q) &= \forall\, \overline{x_f}.\ (\forall\, \overline{y_f}.\ \mathbf{pre}(\mathbf{p})[\overline{y_f}/\overline{y}] \to \mathbf{post}(\mathbf{p})[\overline{y_f}/\overline{y}, \overline{x_f}/\overline{x}]) \to Q[\overline{x_f}/\overline{x}] \\
\mathsf{wprec}(C; S, Q) &= \mathsf{wprec}(C, \mathsf{wprec}(S, Q))
\end{aligned}$$

$$\begin{aligned}
\mathsf{wvc}(\mathbf{skip}, Q) &= \emptyset \\
\mathsf{wvc}(x := e, Q) &= \emptyset \\
\mathsf{wvc}(\mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f, Q) &= \mathsf{wvc}(S_t, Q) \ \cup\ \mathsf{wvc}(S_f, Q) \\
\mathsf{wvc}(\mathbf{while}\ b\ \mathbf{do}\ \{I\}\ S, Q) &= \{I \wedge b \to \mathsf{wprec}(S, I), I \wedge \neg b \to Q\} \cup \quad \mathsf{wvc}(S, I) \\
\mathsf{wvc}(C; S, Q) &= \mathsf{wvc}(C, \mathsf{wprec}(S, Q)) \ \cup\ \mathsf{wvc}(S, Q)
\end{aligned}$$

FORWARD PROPAGATION:

$$\mathsf{VCG}^s(P,\ S,\ Q) = \mathsf{svc}(S,Q) \ \cup\ \{\mathsf{spost}(S,P) \to Q\}$$

$$\begin{aligned}
\mathsf{spost}(\mathbf{skip}, P) &= P \\
\mathsf{spost}(x := e, P) &= \exists\, v.\ P[v/x] \wedge x = e[v/x] \\
\mathsf{spost}(\mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f, P) &= \mathsf{spost}(S_t, P \wedge b) \vee\ \mathsf{spost}(S_f, P \wedge \neg b) \\
\mathsf{spost}(\mathbf{while}\ b\ \mathbf{do}\ \{I\}\ S, P) &= I \wedge \neg b \\
\mathsf{spost}(\mathbf{call}\,\mathbf{p}, P) &= \exists\, \overline{x_f}.\ P[\overline{x_f}/\overline{x}] \wedge (\forall\, \overline{y_f}.\ \mathbf{pre}(\mathbf{p})[\overline{y_f}/\overline{y}, \overline{x_f}/\overline{x}] \to \mathbf{post}(\mathbf{p})[\overline{y_f}/\overline{y}]) \\
\mathsf{spost}(C; S, P) &= \mathsf{spost}(S, \mathsf{spost}(C, P))
\end{aligned}$$

$$\begin{aligned}
\mathsf{svc}(\mathbf{skip}, P) &= \emptyset \\
\mathsf{svc}(x := e, P) &= \emptyset \\
\mathsf{svc}(\mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f, P) &= \mathsf{svc}(S_t, P \wedge b) \ \cup\ \mathsf{svc}(S_f, P \wedge \neg b) \\
\mathsf{svc}(\mathbf{while}\ b\ \mathbf{do}\ \{I\}\ S, P) &= \{P \to I, \mathsf{spost}(S, I \wedge b) \to I\} \cup \quad \mathsf{svc}(S, I \wedge b) \\
\mathsf{svc}(\mathbf{call}\,\mathbf{p}, P) &= \emptyset \\
\mathsf{svc}(C; S, P) &= \mathsf{svc}(C, P) \ \cup\ \mathsf{svc}(S, \mathsf{spost}(C, P))
\end{aligned}$$

$\overline{y}$ is a sequence of the auxiliary variables of $\mathbf{p}$, $\overline{x}$ is a sequence of the program variables occurring in $\mathbf{body}(\mathbf{p})$
$\overline{x_f}$ and $\overline{y_f}$ are sequences of fresh variables, $t[\overline{e}/\overline{x}]$denotes the parallel substitution $t[e_1/x_1, \ldots, e_n/x_n]$

$$\begin{aligned}
\mathsf{wprec}^k(S,Q) &= \mathsf{wprec}(C_k\ ;\ \ldots\ ;\ C_n, Q) & \qquad \mathsf{spost}^0(S,P) &= P \\
\mathsf{wprec}^{n+1}(S,Q) &= Q & \mathsf{spost}^k(S,P) &= \mathsf{spost}(C_1\ ;\ \ldots\ ;\ C_k, P) \\
\mathsf{wvc}^k(S,Q) &= \mathsf{wvc}(C_k\ ;\ \ldots\ ;\ C_n, Q) & \mathsf{svc}^0(S,P) &= \emptyset \\
\mathsf{wvc}^{n+1}(S,Q) &= \emptyset & \mathsf{svc}^k(S,P) &= \mathsf{svc}(C_1\ ;\ \ldots\ ;\ C_k, P)
\end{aligned}$$

VERIFICATION CONDITIONS OF A PROGRAM:

$$\mathsf{Verif}(\Pi) \ = \bigcup_{\mathbf{p} \in \mathcal{P}(\Pi)} \mathsf{VCG}^w(\mathbf{pre}(\mathbf{p}),\ \mathbf{body}(\mathbf{p}),\ \mathbf{post}(\mathbf{p}))$$

or

$$\mathsf{Verif}(\Pi) \ = \bigcup_{\mathbf{p} \in \mathcal{P}(\Pi)} \mathsf{VCG}^s(\mathbf{pre}(\mathbf{p}),\ \mathbf{body}(\mathbf{p}),\ \mathbf{post}(\mathbf{p}))$$

Figure 1. Generation of Verification Conditions

$$
\begin{array}{llll}
\textbf{Assert} & \ni A & ::= & b \mid \mathsf{true} \mid \mathsf{false} \mid A \wedge A \mid A \vee A \mid \neg A \mid A \to A \mid \forall x.\, A \mid \exists x.\, A \\
\textbf{Comm} & \ni C & ::= & \mathbf{skip} \mid x := e \mid \mathbf{if}\ b\ \mathbf{then}\ S\ \mathbf{else}\ S \mid \mathbf{while}\ b\ \mathbf{do}\,\{A, e_v\}\, S \mid \mathbf{call}\ \mathbf{p} \\
\textbf{Block} & \ni S & ::= & C \mid C\,;\, S \\
\textbf{Proc} & \ni \Phi & ::= & \mathbf{pre}\ A\ \mathbf{post}\ A\ \mathbf{proc}\ \mathbf{p} = S \\
\textbf{Prog} & \ni \Pi & ::= & \Phi \mid \Pi\ \Phi
\end{array}
$$

Figure 2. Programming language syntax

branch corresponding to C, with postcondition $\mathsf{wprec}(S, Q)$. Thus the function wprec guides the strategy by selecting an intermediate assertion propagated backwards from Q.

A similar reasoning is used to a VCGen that employees a *forward propagation strategy* and it is also shown in Figure 1.

III. Labeled Control Flow Graphs

This section introduces the notion of control flow graph annotated with pairs of assertions and properties of these Labeled Control Flow Graphs (LCFG) are studied. It will be explained how they can be used as a basis for the interactive generation of verification conditions.

Definition 1 (Labeled Control Flow Graph): Given a program S, precondition P and postcondition Q such that $S = C_1\,;\ldots;\,C_n$, the **labeled control flow graph** $LCFG(S, P, Q)$ of S with respect to (P, Q) is a labeled directed acyclic graph (DAG) whose edge labels are pairs of logical assertions on program states. To each command C in the program S we associate an input node $IN(C)$ and an ouput node $OUT(C)$.

The graph is constructed as follows:

1) Each command C_i in S will be represented by one (in the case of **skip** and assignment commands) or two nodes (for conditional and loop commands):
   - If C_i is **skip** or an assignment command, there is a new node C_i in the graph.
     We set $IN(C_i) = OUT(C_i) = C_i$.
   - If $C_i = \mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f$, there are two new nodes $if(b)$ and fi in the graph.
     We set $IN(C_i) = if(b)$ and $OUT(C_i) = fi$.
   - If $C_i = \mathbf{while}\ b\ \mathbf{do}\,\{I\}\, S$ or $C_i = \mathbf{while}\ b\ \mathbf{do}\,\{I, e_v\}\, S$, there is two new nodes $do(b)$ and od in the graph.
     We set $IN(C_i) = do(b)$ and $OUT(C_i) = od$.
2) Let $LCFG(S, P, Q)$ also contain two additional nodes $START$ and END.
3) Let $LCFG(S, P, Q)$ contain an edge $(OUT(C_i), IN(C_{i+1}))$ for $i \in \{1, \ldots, n-1\}$, and two additional edges $(START, IN(C_1))$ and $(OUT(C_n), END)$. The labels of these edges are set as follows:
$$
\begin{aligned}
lb\,(START, IN(C_1)) &= (\mathsf{spost}^0(S, P), \mathsf{wprec}^1(S, Q)) \\
&= (P, \mathsf{wprec}^1(S, Q)); \\
lb\,(OUT(C_i), IN(C_{i+1})) &= (\mathsf{spost}^i(S, P), \mathsf{wprec}^{i+1}(S, Q)); \\
lb\,(OUT(C_n), END) &= (\mathsf{spost}^n(S, P), \mathsf{wprec}^{n+1}(S, Q)) \\
&= (\mathsf{spost}^n(S, P), Q).
\end{aligned}
$$
4) For $i \in \{1, \ldots, n\}$, if $C_i = \mathbf{if}\ b\ \mathbf{then}\ S_t\ \mathbf{else}\ S_f$, we recursively construct the graphs
$$LCFG(S_t, b \wedge \mathsf{spost}^{i-1}(S, P), \mathsf{wprec}^{i+1}(S, Q))$$
and
$$LCFG(S_f, \neg b \wedge \mathsf{spost}^{i-1}(S, P), \mathsf{wprec}^{i+1}(S, Q))$$
These graphs are grafted into the present graph by removing their $START$ nodes and setting the origin of the dangling edges to be in both cases the node $IN(C_i)$, and similarly removing their END nodes and setting the destination of the dangling edges to be the node $OUT(C_i)$.
5) For $i \in \{1, \ldots, n\}$, if $C_i = \mathbf{while}\ b\ \mathbf{do}\,\{I\}\, S$, we recursively construct the graph
$$LCFG(S, I \wedge b, I)$$
or
$$LCFG(S, I \wedge b \wedge e_v = x_0, I \wedge e_v < x_0)$$
if a loop variant is present, i.e. $C_i = \mathbf{while}\ b\ \mathbf{do}\,\{I, e_v\}\, S$ (x_0 is a fresh variable).
This graph is grafted into the present graph by removing its $START$ node and setting the origin of the dangling edge to be the node $IN(C_i)$, and similarly removing its END node and setting the destination of the dangling edge to be the node $OUT(C_i)$.

Clearly every subprogram $\hat{S}$ of S is represented by a subgraph of $LCFG(S, P, Q)$ delimited by a pair of nodes $START$ / END, if/fi, or do/od. The basic intuition of labeled CFGs is that for every pair of consecutive commands $\hat{C}_i$, $\hat{C}_{i+1}$ in $\hat{S}$, there exists an edge $(\hat{C}_i, \hat{C}_{i+1})$ in $LCFG(S, P, Q)$ whose label consists of the strongest postcondition of the prefix of $\hat{S}$ ending with $\hat{C}_i$, and the weakest precondition of the suffix of $\hat{S}$ beginning with

$\hat{C}_{i+1}$, with respect to the local specification $(\hat{P}, \hat{Q})$ of $\hat{S}$ propagated from (P, Q).

If loops are annotated with variants, this is taken into account when constructing the subgraph corresponding to the loop's body (point 5 of the definition). So we now have that $\models \mathsf{VCG}_t^w(P, S, Q)$ implies $\models \phi \rightarrow \psi$ for every label (ϕ, ψ) in the graph.

IV. Verification Graphs

In this section the notion of Verification Graph and its properties are studied. In particular, it is shown how the LCFG previously introduced can be used for interactive verification.

Definition 2 (Verification Graph): Let Π be a program and $\mathbf{p} \in \mathcal{P}(\Pi)$ a procedure of Π. Then the *verification graph* of $\mathbf{p}$, $\mathsf{VG}(\mathbf{p})$, is the graph $LCFG(\mathbf{body}(\mathbf{p}), \mathbf{pre}(\mathbf{p}), \mathbf{post}(\mathbf{p}))$. A formula $\phi \rightarrow \psi$ such that (ϕ, ψ) is the label of an edge in the verification graph of $\mathbf{p}$ will be called an **edge condition** of $\mathbf{p}$. $\mathsf{EC}(\mathbf{p})$ will denote the set of Edge Conditions of procedure $\mathbf{p}$.

Naturally, we may also speak of the set of verification graphs of a program Π, which is the set of verification graphs of all its procedures.

After defining the concept of Verification Graph, it is possible to prove the following lemmas [3]:

Lemma 1: Let S be a block of commands and P, Q assertions.

Then $\mathsf{VCG}^w(P, S, Q) \subseteq \mathsf{EC}(LCFG(S, P, Q))$.

This lemma implies that the verification graph of a procedure contains a set of verification conditions for it (generated using weakest preconditions exclusively).

What is obtained is a control flow graph annotated with assertions (the edge conditions) from which different sets can be picked whose validity is sufficient to guarantee the correctness of the procedure. Each particular set corresponds to one particular verification strategy, mixing the use of strongest postconditions and weakest preconditions. The reason why there are different choices is that many edge conditions in the same graph are equivalent.

Every block of code in a procedure is represented as a set of paths in its verification graph (a single path if the block contains no branching), and the label of each edge in the path consists of the strongest postcondition of a prefix and the weakest precondition of a suffix of the block, with respect to its local specification. Now it is easy to see that in the case of atomic commands, the labels of adjacent edges correspond to equivalent assertions, and in the case of conditional commands it is the conjunction of labels of the branching edges that is equivalent to the adjacent edge.

In a block not containing loops and conditionals, it is equivalent to check the validity of any edge condition in the block's path in the verification graph. If the block contains conditionals, it is indifferent to verify (i) an edge condition in the path before the conditional, or (ii) two edge conditions, one for each branch path, or (iii) an edge condition in the path after the conditional.

One way to formalize this is to assign a status to each edge. Colors green and black will be used for this. The idea is that the edges whose conditions are known to be valid (because they have been checked with a prover) are set to green, and we let this "checked" status propagate along the edges of the verification graph. If all edges become green then the procedure has been successfully verified.

Definition 3 (Edge Color in a Verification Graph): Given a procedure $\mathbf{p}$ let E be the set of edges of its verification graph and consider a subset $\mathcal{A} \subseteq \mathsf{EC}(\mathbf{p})$ of its edge conditions. The function $\mathsf{color}_{\mathcal{A}} : E \rightarrow \{\mathsf{black}, \mathsf{green}\}$ is defined as follows, where we write

$$O \overset{(\phi,\psi)}{\rightarrow} D$$

for the edge with source O, destination D, and label (ϕ, ψ) (the label may be omitted when not relevant).

- If $\phi \rightarrow \psi \in \mathcal{A}$, then $\mathsf{colour}_{\mathcal{A}}(O \overset{(\phi,\psi)}{\rightarrow} D) = \mathsf{green}$
- If O corresponds to an atomic command and $\mathsf{colour}_{\mathcal{A}}(N \rightarrow O) = \mathsf{green}$ for some node N, then $\mathsf{colour}_{\mathcal{A}}(O \rightarrow D) = \mathsf{green}$
- If D corresponds to an atomic command and $\mathsf{colour}_{\mathcal{A}}(D \rightarrow N) = \mathsf{green}$ for some node N, then $\mathsf{colour}_{\mathcal{A}}(O \rightarrow D) = \mathsf{green}$
- If O is an "*if*" node and $\mathsf{colour}_{\mathcal{A}}(N \rightarrow O) = \mathsf{green}$ for some node N, then $\mathsf{colour}_{\mathcal{A}}(O \rightarrow D) = \mathsf{green}$ (note there are two such D for each node)
- If D is an "*if*" node and $\mathsf{colour}_{\mathcal{A}}(D \rightarrow N_1) = \mathsf{green}$ and $\mathsf{colour}_{\mathcal{A}}(D \rightarrow N_2) = \mathsf{green}$ for some nodes N_1, N_2 ($N_1 \neq N_2$), then $\mathsf{colour}_{\mathcal{A}}(O \rightarrow D) = \mathsf{green}$
- If O is an "*fi*" node and $\mathsf{colour}_{\mathcal{A}}(N_1 \rightarrow O) = \mathsf{green}$ and $\mathsf{colour}_{\mathcal{A}}(N_2 \rightarrow O) = \mathsf{green}$ for some nodes N_1, N_2 ($N_1 \neq N_2$), then $\mathsf{colour}_{\mathcal{A}}(O \rightarrow D) = \mathsf{green}$
- If D is an "*fi*" node and $\mathsf{colour}_{\mathcal{A}}(D \rightarrow N) = \mathsf{green}$ for some node N, then $\mathsf{colour}_{\mathcal{A}}(O \rightarrow D) = \mathsf{green}$ (note there are two such O for each node)
- Otherwise $\mathsf{colour}_{\mathcal{A}}(O \rightarrow D) = \mathsf{black}$

The green color propagates freely through atomic command nodes in either direction. In branching nodes, it propagates from outside the conditional command into both branches; in the reverse direction, it will only propagate outwards when both branch edges are green.

A third color red could be additionally considered, for edges whose conditions have been proved to be invalid. For interactive verification, this would have the advantage of identifying the segments of the code where problems exist. red would also propagate freely across atomic command nodes; it would not propagate into conditional branches (since only one branch is known to have an invalid VC), but it would propagate outwards from any conditional branch (without requiring the other branch to also be red).

While loops have not yet been discussed. Each loop is represented by a pair of nodes *do*, *od*, and a path or set of paths from the former to the latter, corresponding to the loop's body. In *do* and *od* nodes there exists no equivalence between the edge conditions of the incoming and outgoing edges. Consider the subgraph

$$A \overset{(\phi_1,\psi_1)}{\rightarrow} do \overset{(\phi_2,\psi_2)}{\rightarrow} \ldots \overset{(\phi_3,\psi_3)}{\rightarrow} od \overset{(\phi_4,\psi_4)}{\rightarrow} B$$

Then $\phi_1 \rightarrow \psi_1$ is the loop initialization VC, $\phi_2 \rightarrow \psi_2, \ldots, \phi_3 \rightarrow \psi_3$ are invariant preservation VCs, and $\phi_4 \rightarrow \psi_4$ is the VC ensuring that the postcondition is granted by the invariant. Unlike the case of atomic command or branching nodes, in the presence of *do* / *od* nodes it is necessary to establish independently the validity of these VCs. In terms of coloring, these nodes *block* the propagation of the green color.

A consequence of this is that in the presence of an arbitrary path $A \overset{(\phi_1,\psi_1)}{\rightarrow} B \rightarrow \ldots \rightarrow C \overset{(\phi_2,\psi_2)}{\rightarrow} D$ if the path contains no loop nodes, then $\phi_1 \rightarrow \psi_1 \equiv \phi_2 \rightarrow \psi_2$, otherwise the equivalence does not hold: each loop introduces the need to prove three independent VCs.

V. Interactive Verification using GamaSlicer

To illustrate that all the concepts introduced in this paper work in practice we developed GamaSlicer [8]. GamaSlicer includes both traditional and interactive verification functionality and also a highly parameterizable semantic slicer [9] for Java programs annotated in JML [10].

Since the underlying logic of slicing algorithms as well as the VCGen is first-order logic, the tool outputs proof obligations written in the SMT-Lib (Satisfiability Modulo Theories library) language. It was chosen SMT-Lib since it is nowadays the language employed by most provers used in program verification, including, among many others, Z3 [11] and Alt-Ergo [12].

To understand how interactive verification works, consider the fragment of a program used to calculate the income taxes in the United Kingdom[1] (method `TaxesCalculation` in Listing 1). Consider as precondition $P = age \geq 18$ and as postcondition $Q = personal > 5750$. A careful observation of the method allows one to detect problems, as in some circumstances the result value of *personal* can in fact be less than 5750. If one decides to check whether the program is correct or not using a standard VCGen algorithm like the one presented in Figure 1, the result is a single VC, too big and complex to be understandable (the resulting verification condition is shown in Figure 3). It could be hard to understand what is causing the incorrectness of the program just by looking at the VC and the output of the prover.

[1]The complete source code of this program can be found in [13] and has been used as a benchmark test for slicing algorithms based on assertions.

```
1  if (age >= 75) then
2      personal := 5980
3  else
4      if (age >= 65) then
5          personal := 5720
6      else
7          personal := 4335
8
9  if ((age >= 65) and (income > 16800)) then
10     t := personal - ((income - 16800)/2)
11 else
12     if (t > 4335) then
13         personal := t + 2000
14     else
15         personal := 4335
```

Listing 1. Program TaxesCalculation

The software engineer responsible for finding the bugs would start by visualizing the entire verification graph for the program under consideration and then would select a part of the graph to start checking it. Zooming-in on the subgraph correspondent to the second conditional statement in the method, suppose that the user select the last edge inside the *then* branch. The edge condition sent to the prover is:

$$\exists v0 : age \geq 65 \;\&\&\; income > 16800 \;\&\&\; t > 4335 \;\&\&\; personal == t + 2000 \rightarrow personal > 5750$$

which the prover identifies as being valid.[2] Thus, the edges inside the branch and the assignment statement are shown in green and the *if*/*fi* nodes remain black (Figure 4), according to the coloring rules in Definition 3. This means that the statements inside the *then* branch are not the cause of failure. If we now pick the last edge inside the *else* branch, the following condition is sent to the prover:

$$\exists v0 : age \geq 65 \;\&\&\; income > 16800 \;\&\&\; t > 4335 \;\&\&\; personal \neq t + 2000 \rightarrow personal > 5750$$

The result returned by the prover is now that the condition is not valid. Thus, the edges inside the branch, the assignment node, and the *if*/*fi* nodes all become red (Figure 5). At this step, we learn that the statement in this path is causing the procedure to be incorrect.

In fact this is not the only problem in this procedure. Repeating this process for the inner conditional inside the first one, we can observe that the statement $personal = 5720$ is also preventing the program from being correct.

In order to assist the software engineer, during the interactive verification of a program, several features are available in order to make this process more intuitive and user friendly. Once an edge is chosen (by clicking on it), the condition

[2]In fact it is the *negation* of this formula that is sent to the SMT solver, which returns *unsat*, meaning that the original formula is valid.

$((age \geq 18) \rightarrow (((age \geq 75) \rightarrow ((((age \geq 65) \wedge (income > 16800)) \rightarrow ((((5980 - ((income - 16800)/2)) > 4335) \rightarrow (((5980 - ((income - 16800)/2)) + 2000) > 5750)) \wedge ((!((5980 - ((income - 16800)/2)) > 4335)) \rightarrow (4335 > 5750)))) \wedge ((!((age \geq 65) \wedge (income > 16800))) \rightarrow true))) \wedge ((!(age \geq 75)) \rightarrow (((age \geq 65) \rightarrow ((((age \geq 65) \wedge (income > 16800)) \rightarrow ((((5720 - ((income - 16800)/2)) > 4335) \rightarrow (((5720 - ((income - 16800)/2)) + 2000) > 5750)) \wedge ((!((5720 - ((income - 16800)/2)) > 4335)) \rightarrow (4335 > 5750)))) \wedge ((!((age \geq 65) \wedge (income > 16800))) \rightarrow true))) \wedge ((!(age \geq 65)) \rightarrow ((((age \geq 65) \wedge (income > 16800)) \rightarrow ((((4335 - ((income - 16800)/2)) > 4335) \rightarrow (((4335 - ((income - 16800)/2)) + 2000) > 5750)) \wedge ((!((4335 - ((income - 16800)/2)) > 4335)) \rightarrow (4335 > 5750)))) \wedge ((!((age \geq 65) \wedge (income > 16800))) \rightarrow true)))))))$

Figure 3. Verification condition for the TaxesCalculation program

sent to the prover is shown in a Verification list, at the bottom of the window (Figures 4 and 5). The user is free to expand/collapse this list. When clicking on a list item, the edges/nodes related with the condition are momentarily highlighted. Also, in order to keep track of the edge conditions previously considered, complementing the Verification list, an history of images (also expandable/collapsable) is shown on the right (Figures 4 and 5). Once the selected condition has been processed, and according to the result returned by the prover, the current edge is colored: *red* if the prover returned false, *green* in case of true and *yellow* when the returned value is unknown.

Nodes in the graph will also be displayed in color, as follows: a node is shown in green if all its incoming and outgoing edges are green; it is shown in red if at least one of its incoming and outgoing edges is red; it is shown in black otherwise (i.e. no red edges and at least one black edge). For instance when an if node is shown in green, this means that the edge condition of the incoming edge is valid, i.e. the precondition of the corresponding conditional statement poses no problems.

In the traditional approach, to understand which statements are leading to the incorrectness the software engineer must debug the program manually. However, this is not the desirable way. Although the considered program in this section is of small size, when dealing with larger programs this process becomes hard, boring and error prone. An interactive verification may help us to find the statements that prevent the program from being correct.

VI. Applications

Intermediate Conditions: A feature that can increase even more the advantages of interactivity is the possibility to check conditions inserted at arbitrary points of the verification graph. An intermediate assertion must hold at the point where it is inserted, and provides information that can be used subsequently. Introducing assertions in a verification graph can be a great help when automation fails; they act as lemmas that are easy to prove but may then allow for more difficult VCs to be discharged automatically.

Automatic Error Path Discovery: An alternative approach to generating verification conditions is based on *symbolic execution* [14]. For programs in single-assignment form, symbolic execution is very closely related to strongest postcondition calculations. But a defining characteristic of symbolic execution is that it generates one formula *for each execution path* of the program.

Symbolic execution VCs with respect to a specification (P, Q) can be generated in a straightforward manner for single-assignment programs without loops: it suffices to traverse the CFG from $START$ to END, constructing a conjunctive formula as follows: let P initially be the given precondition. For each node with label $x := e$ crossed, let P become $P \wedge x = e$; for each node with label $\mathbf{if}(b)$ crossed towards the *then* (resp. *else*) branch, let P become $P \wedge b$ (resp. $P \wedge \neg b$). When END is reached, generate the verification condition $P \rightarrow Q$. Repeat until all paths have been traversed (using a depth-first strategy).

Of course, this method may generate an exponential number of VCs on the length of the block (as many as there are execution paths). But it does have one significant positive aspect, which is the fact that there exists a direct association between VCs and error traces: if one of the above conditions were shown to be invalid, it would immediately be known which execution path of the program produced an error; moreover, the number of erroneous paths would also be known.

State of the art VCGens solve this problem by instrumenting the VCs with additional labels and reading back the counter-examples generated by the automatic provers, which can then be mapped to execution traces. Verification graphs offer an alternative solution to this problem, which can be used with any prover, even when counter-examples are not available. Recall that the validity of the condition of an edge e implies that there is no error in any of the execution paths containing e; if the condition is invalid this means that *at least one* of the execution paths containing e is erroneous. Thus it is straightforward to conceive an algorithm for identifying error traces.

The algorithm identifies error paths by performing a depth-first traversal of the verification graph and invoking an external proof tool. Whenever a branching node with label $\mathbf{if}(b)$ is met, the algorithm first follows the *then* branch, and will later follow the *else* branch. For each branch explored, the condition of the first edge is checked. If it is valid, the

traversal backtracks, since no execution path containing the edge produces errors. If not, the traversal is continued in order to identify the error paths containing the present edge. An error path is found when END is reached (or when no more **if** nodes can be met from the current node to END).

Note that even though the number of execution paths is exponential, this technique is feasible if the number of error paths is small, since each condition checked as valid will reduce by half the number of paths left to explore. In particular, if a single error path exists, finding it requires checking a linear number of conditions on the length of the block.

Verification Condition Splitting: For the case of VCs based on weakest preconditions, it has been shown [2] that splitting VCs (i.e. having a bigger number of smaller conditions) can in certain circumstances lead to substantial improvements with respect to the performance of an SMT solver, and also with respect to the quality of the error messages produced when verification fails. The authors reach the conclusion that some splitting may dramatically improve the running time of the prover (bringing it to reasonable values when it was initially unfeasible), but too much splitting may have a negative effect.

Verification graphs offer a method for identifying split versions of a procedure's VCs, when forward propagation and backward propagation are combined, which is compatible with the optimizations allowed by passive / single-assignment programs. In particular the *dynamic splitting* strategy suggested in [2] prescribes that one should first try to prove a single VC for the entire block. The prover is given a time-out limit, after which the VC should be split and the two resulting VCs sent to the prover (again with a time-out limit), and so on. This can be readily formulated as a forward or a backward splitting strategy on the verification graph: one starts with the edge condition of the edge with origin $START$, or of the edge with destination END, and follows the branching structure of the CFG in the forward or backward direction.

But the graph would also allow one to identify the points of the program where the level of splitting is maximal.

VII. Conclusion

In this paper, we propose an alternative approach to the traditional verification of programs based on labeled control flow graphs. The proposed technique is then to show the whole or a single part of the LCFG to help the software engineer, in a interactive way, to exclude the correct statements and focus on the wrong ones in order to fix the existing bugs.

Graph-based generation of VCs has already been used in the context of the verification of reactive systems: Manna and colleagues introduced the notion of *temporal verification diagram* [15] to represent a proof that a system enjoys a given property, expressed as a temporal logic formula. The idea is that such a diagram, whose edges are labelled by sets of transitions, corresponds to an approximation of the computations of a transition system. A set of (first order) verification conditions is produced from the diagram such that, if all VCs are valid, the system is guaranteed to satisfy the property under consideration.

Although this approach could seem inefficient due to the propagation of both precondition and postcondition along the edges, it is not computationally costly when compared with the cost of automatic proof, and although verification graphs contain redundancy (since many equivalent edge conditions may be present), this is not reflected in the proof process itself, as a precise criterion can be used to select sets of conditions not containing redundancy. We can optimize the construction of verification graphs to reduce redundancy: for a given verification strategy based on graphs, assertions can be propagated lazily. For instance, with a forward propagation strategy, the spost labeling needs not be propagated at all since only the first edge condition in each path is required. With a user-directed strategy, in which the user manually selects the edges in the graph whose conditions will be checked (for instance with the help of a visual front-end), the propagation can be directed by the user's selection on demand. The result is that only one of the labelings is calculated for each edge, with the exception of edges whose conditions will be checked.

This interactive approach is thus useful when a program is incorrect and we want to detect which part of the program is causing such problems. Using this approach, step-by-step the software engineer will identify the statements that are violating a given contract. The software engineer can change iteratively the contracts and perform again the same steps until he gets a fully verified program (all nodes in graph colored in green).

As future work, we intend to automatize the discovery of the error paths. This way, the work of Software Engineers will be even more reduced. A possible way to do this, would be to start by the more internal edges in a given program and color the graph according to the rules presented in Definition 3.

Acknowledgements: This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundac ao para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-020486.

References

[1] C. Flanagan and J. B. Saxe, "Avoiding exponential explosion: generating compact verification conditions," in *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 2001, pp. 193–205.

[2] K. R. M. Leino, M. Moskal, and W. Schulte, "Verification condition splitting," Microsfot Research, 2008.

[3] D. da Cruz, P. R. Henriques, , and J. S. Pinto, "Verification graphs for programs with contracts," Universidade do Minho, Tech. Rep. TR-HASLab:01:2012, 2012, available from http://haslab.di.uminho.pt/TechnicalReports.

[4] B. Meyer, "Design by contract," Interactive Software Engineering Inc., Technical Report TR-EI-12/CO, 1986.

[5] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 3, pp. 212–232, 2005.

[6] M. Barnett, K. Rustan, M. Leino, and W. Schulte, "The Spec# programming system: An overview." in *CASSIS : construction and analysis of safe, secure, and interoperable smart devices*, vol. 3362. Springer, Berlin, March 2004, pp. 49–69.

[7] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*, 1st ed. Addison Wesley, March 2003.

[8] D. da Cruz, P. R. Henriques, and J. S. Pinto, "Gamaslicer: an Online Laboratory for Program Verification and Analysis," in *proceedings of the 10th. Workshop on Language Descriptions Tools and Applications (LDTA'10)*, 2010, pp. 3:1–3:8, to appear.

[9] J. B. Barros, D. da Cruz, P. R. Henriques, and J. S. Pinto, "Assertion-based slicing and slice graphs," *Software Engineering and Formal Methods, IEEE International Conference on*, vol. 0, pp. 93–102, 2010.

[10] G. Leavens, E. Poll, C. Clifton, Y. Cheon, and C. Ruby, "Jml reference manual," 2002. [Online]. Available: citeseer.ist.psu.edu/leavens04jml.html

[11] L. M. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[12] S. Conchon, E. Contejean, and J. Kanig, "Ergo : a theorem prover for polymorphic first-order logic modulo theories," 2006. [Online]. Available: http://ergo.lri.fr/papers/ergo.ps

[13] R. M. Hierons, M. Harman, C. Fox, L. Ouarbya, and M. Daoudi, "Conditioned slicing supports partition testing." *Softw. Test., Verif. Reliab.*, pp. 23–28, 2002.

[14] C. S. Pasareanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, pp. 339–353, October 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1667865.1667869

[15] I. A. Browne, Z. Manna, and H. Sipma, "Generalized temporal verification diagrams," in *Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science*. London, UK: Springer-Verlag, 1995, pp. 484–498. [Online]. Available: http://dl.acm.org/citation.cfm?id=646833.708045

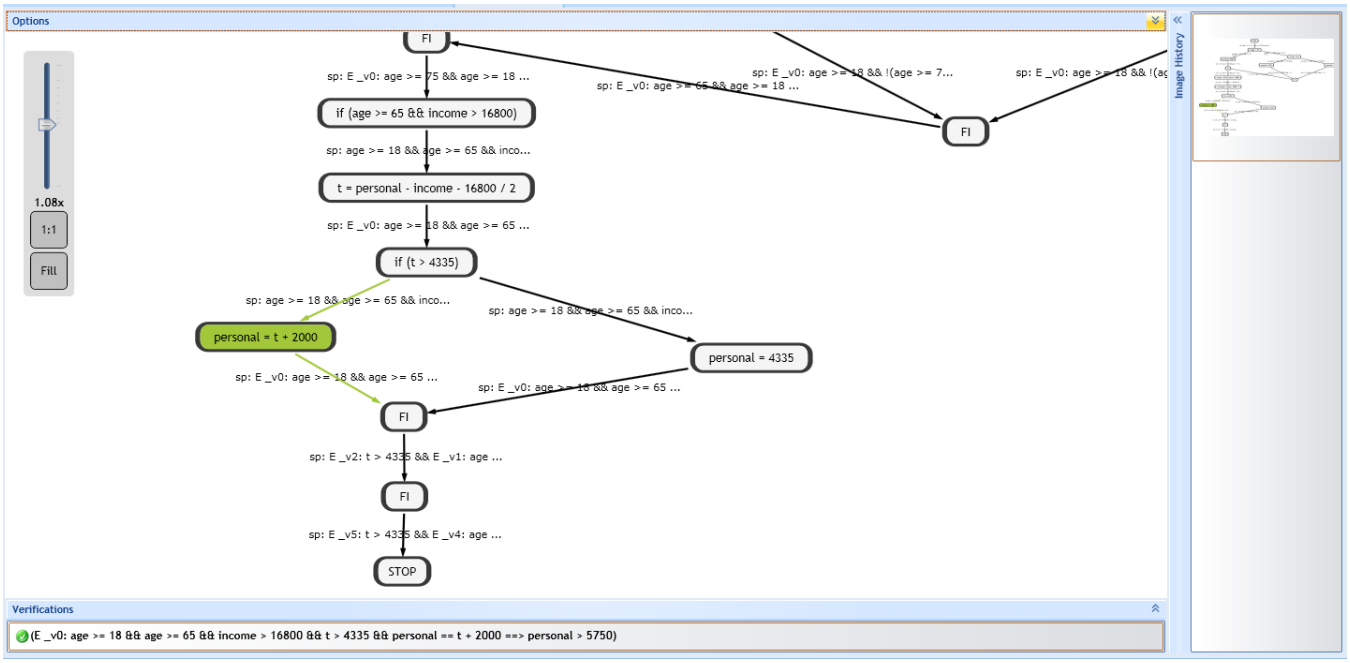

Figure 4. Verifying the Edge Conditions of Verif. Graph for TaxesCalculation program (1)

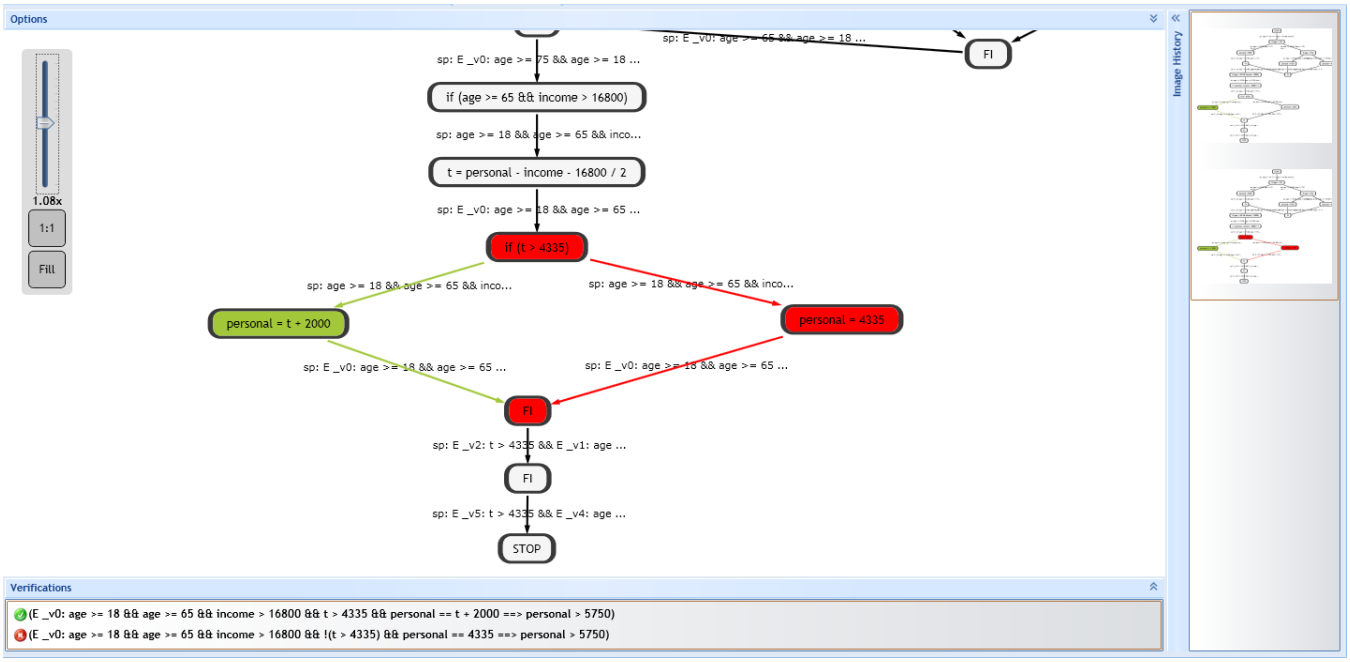

Figure 5. Verifying the Edge Conditions of Verif. Graph for TaxesCalculation program (2)