Source of Acquisition
NASA Ames Research Center

Explaining Verification Conditions

Ewen Denney[1] and Bernd Fischer[2]

[1] USRA/RIACS, NASA Ames Research Center, Moffett Field, CA 94035, USA
edenney@email.arc.nasa.gov
[2] DSSE Group, School of Electronics and Computer Science, University of Southampton, UK
B.Fischer@ecs.soton.ac.uk

Abstract. The Hoare approach to program verification relies on the construction and discharge of verification conditions (VCs) but offers no support to trace, analyze, and understand the VCs themselves. We describe a systematic extension of the Hoare rules by labels so that the calculus itself can be used to build up *explanations* of the VCs. The labels are maintained through the different processing steps and rendered as natural language explanations. The explanations can easily be customized and can capture different aspects of the VCs; here, we focus on their structure and purpose. The approach is fully declarative and the generated explanations are based only on an analysis of the labels rather than directly on the logical meaning of the underlying VCs or their proofs.
Keywords: program verification, Hoare calculus, traceability.

1 Introduction

Program verification is easy when everything is correct and automated tools do all the work: a verification condition generator (VCG) then takes a program that is annotated with "logical mark-up" (i.e., pre-/post-conditions and invariants) and produces a number of verification conditions (VCs) that are simplified, augmented with a domain theory, and finally discharged by an automated theorem prover (ATP). In practice, however, many things can—and typically do—go wrong: the program may be incorrect or unsafe, the annotations may be incorrect or incomplete, the simplifier may be too weak or counter-productive, the domain theory may be incomplete, and the ATP may run out of resources. In each of these cases, users are confronted only with failed VCs (i.e., the failure to prove them automatically), but typically receive no additional information about the causes of the failure. They must thus analyze the VCs by interpreting their constituent parts, and relating them through the applied Hoare rules and simplifications to the corresponding source code locations. Unfortunately, VCs are a very detailed and low-level representation of both the underlying information and the process used to derive it, so this is often difficult to achieve.

Here we describe an implemented technique that helps users to trace, analyze, and understand VCs. Our idea is to systematically extend the Hoare rules by "semantic mark-up" so that we can use the calculus itself to build up *explanations* of the VCs. This mark-up takes the form of *semantic labels* that are attached to the meta-variables used in the Hoare rules, so that the VCG then produces labeled versions of the VCs. The labels are maintained through the different processing steps, in particular the simplification, and are then extracted from the final VCs and rendered as natural language explanations.

Most verification systems based on Hoare logic offer some basic tracing support by emitting the current line number whenever a VC is constructed. However, this does not provide any information as to which other parts of the program have contributed to the VC, how it has been constructed, or what its purpose is, and is therefore insufficient as a basis for informative explanations. Some systems produce short captions for each VC (e.g., JACK [1] or PerfectDeveloper [2]). Other techniques focus on a detailed linking between source locations and VCs to support program debugging [10, 11]. Our approach, in contrast, serves as a customizable basis to explain different aspects of VCs. In this paper, we focus on explaining the *structure* (including source location information) and the *purpose* of VCs, helping users to understand what a VC means.

In our approach we only explain what has been explicitly declared to be significant using labels. Hence, the generated explanations are based only on an analysis of the labels but not of the structure or even logical meaning of the underlying VCs. For example, we would not try to infer that two formulas are the base and step case of an induction unless the formulas are specifically marked up with this information. This also lets us use a syntax-directed style for the labeling and to restrict the rendering to a local rather than a whole-program analysis; consequently, rendering is compositional and can be implemented using simple text templates. Finally, we restrict ourselves to explaining the construction of VCs (which is the essence of the Hoare approach) rather than their simplification and proof. Hence, we maintain, but do not introduce, labels during simplification, and strip them off before proving the VCs; existing techniques [8] could be applied to explain the detailed proofs, although it is unlikely that this would provide much additional insight since the key information is already expressed in the annotations and consequently in the VCs as well.

We developed our technique to support a certifiable code generation approach, where the generator provides Hoare-style safety proofs for the generated code. Here, human-readable explanations of the VCs are particularly important to gain confidence into the large and complex system. However, our technique is not tied to code generation or safety certification and can be used in any Hoare-style verification context.

In the next section, we briefly recall the necessary logical background; for details see [3, 4, 15]. In Section 3, we extend the basic calculus to provide structural explanations, and then describe our implementation. Section 5 extends the approach further to explain the purpose of VCs in more detail. Finally, Sections 6 and 7 compare related work and outline current and future work.

2 Logical Background

2.1 Hoare Logic and Program Verification

We follow the usual Hoare-style program verification approach (see [12] for more details), in which the verification problem is solved in two separate stages. In the first stage, a VCG applies the proof rules of the underlying Hoare calculus to the annotated program to produce a number of VCs. In the second stage, an ATP discharges the VCs. The main advantage of this two-stage approach is that it splits the decidable (given suitable annotations) construction of the VCs from their undecidable discharge. The main

$$(assign)\ \frac{}{Q[e/x]\ \{x := e\}\ Q} \qquad (update)\ \frac{}{Q[upd(x, e_1, e_2)/x]\ \{x\,[e_1]\ := e_2\}\ Q}$$

$$(skip)\ \frac{}{Q\ \{\mathbf{skip}\}\ Q} \qquad (if)\ \frac{P_1\ \{c_1\}\ Q \quad P_2\ \{c_2\}\ Q}{(b \Rightarrow P_1) \wedge (\neg b \Rightarrow P_2)\ \{\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\}\ Q}$$

$$(while)\ \frac{P\ \{c\}\ I \quad I \wedge b \Rightarrow P \quad I \wedge \neg b \Rightarrow Q}{I\ \{\mathbf{while}\ b\ \mathbf{inv}\ I\ \mathbf{do}\ c\}\ Q}$$

$$(for)\ \frac{P\ \{c\}\ I[i+1/i] \quad I \wedge e_1 \leq i \leq e_2 \Rightarrow P \quad I[e_2+1/i] \Rightarrow Q}{I[e_1/i]\ \{\mathbf{for}\ i := e_1\ \mathbf{to}\ e_2\ \mathbf{inv}\ I\ \mathbf{do}\ c\}\ Q}$$

$$(comp)\ \frac{P\ \{c_1\}\ R \quad R\ \{c_2\}\ Q}{P\ \{c_1\ ;\ c_2\}\ Q} \qquad (assert)\ \frac{P' \Rightarrow P \quad P\ \{c\}\ Q' \quad Q' \Rightarrow Q}{P'\ \{\mathbf{pre}\ P'\ c\ \mathbf{post}\ Q'\}\ Q}$$

Fig. 1. Basic Hoare rules for target language

disadvantage, however, is that the VCs become removed from the program context, which exacerbates the understanding problem.

Figure 1 shows the Hoare rules that are implemented by our VCG. They are formalized using the usual Hoare triples $P\ \{c\}\ Q$, i.e., if the condition P holds and the command c terminates, then Q holds afterwards. The *update*-rule uses McCarthy's select- and update-functions to handle array updates. For the completeness of the calculus, we also need the usual rule of consequence but since the VCG itself never applies it, we can ignore it here.

We restrict our attention to an imperative core language which is sufficient for the programs generated by NASA's certifiable code generators AUTOBAYES [9] and AUTOFILTER [16]. Extensions to other language constructs are straightforward, as long as the appropriate (unlabeled) Hoare rules have been formulated.

2.2 Source-Level Safety Certification

The purpose of safety certification is to demonstrate that a program does not violate certain conditions during its execution. A *safety property* is an exact characterization of these conditions based on the operational semantics of the language. A *safety policy* is a set of Hoare rules designed to show that safe programs satisfy the safety property of interest. Most approaches to safety certification, in particular proof-carrying code [13], operate on object code but since our goal is to explain VCs in relation to the original program, we follow a source-level approach. From this perspective, the important aspect of safety certification is that the formulas in the rules have more internal structure. This can be exploited by our approach to produce more detailed explanations.

For each notion of safety which is of interest a safety property and the corresponding safety policy must be formulated. This is usually straightforward; in particular, a safety policy can be constructed systematically by instantiating a generic rule set that is derived from the standard rules of the Hoare calculus [3]. The basic idea is to extend the standard environment of program variables with a "shadow" environment of safety

$$(assign)\quad \frac{}{Q[e/x, \text{INIT}/x_{\text{init}}] \wedge safe_{\text{init}}(e)\ \{x := e\}\ Q}$$

$$(update)\quad \frac{}{\begin{pmatrix} Q[upd(x, e_1, e_2)/x, upd(x_{\text{init}}, e_1, \text{INIT})/x_{\text{init}}] \\ \wedge\ safe_{\text{init}}(e_1) \wedge safe_{\text{init}}(e_2) \end{pmatrix}\ \{x\,[e_1]\ := e_2\}\ Q}$$

$$(if)\quad \frac{P_1\ \{c_1\}\ Q \quad P_2\ \{c_2\}\ Q}{(b \Rightarrow P_1) \wedge (\neg b \Rightarrow P_2) \wedge safe_{\text{init}}(b)\ \{\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\}\ Q}$$

$$(while)\quad \frac{P\ \{c\}\ I \quad I \wedge b \Rightarrow P \quad I \wedge \neg b \Rightarrow Q}{I \wedge safe_{\text{init}}(b)\ \{\textbf{while}\ b\ \textbf{inv}\ I\ \textbf{do}\ c\}\ Q}$$

$$(for)\quad \frac{P\ \{c\}\ I[i+1/i] \quad I[\text{INIT}/i_{\text{init}}] \wedge e_1 \le i \le e_2 \Rightarrow P \quad I[e_2+1/i] \Rightarrow Q}{I[e_1/i] \wedge safe_{\text{init}}(e_1) \wedge safe_{\text{init}}(e_2)\ \{\textbf{for}\ i := e_1\ \textbf{to}\ e_2\ \textbf{inv}\ I\ \textbf{do}\ c\}\ Q}$$

Fig. 2. Hoare rules modified for initialization safety

variables which record safety information related to the corresponding program variables. The rules are then responsible for maintaining this environment and producing the appropriate safety obligations. This can be achieved generically using a family of *safety substitutions* that are added to the normal substitutions, and a family of *safety predicates* that are added to the calculated weakest preconditions (WPCs). Safety certification then starts with the postcondition *true* and computes the weakest safety precondition (WSPC), i.e., the WPC together with all applied safety predicates and safety substitutions. If the program is safe then the WSPC will be provable without any assumptions, i.e., *true* $\{c\}$ *true* is derivable.

Figure 2 shows the modified rules for the initialization safety policy. This ensures that each variable or individual array element has been explicitly assigned a value before it is used. The safety environment consists of shadow variables x_{init} that contain the value INIT after the variable, x, has been assigned a value. Arrays are represented by shadow arrays to capture the status of the individual elements. All statements accessing lvars affect the value of a shadow variable (cf. the *assign*-, *update*-, and *for*-rules). However, all rules also need to produce the appropriate safety predicates $safe_{\text{init}}(e)$ for all immediate subexpressions e of the statements. Since the safety property defines an expression to be safe if all corresponding shadow variables have the value INIT, $safe_{\text{init}}(x\,[i]\,)$ for example simply translates to $i_{\text{init}} = \text{INIT} \wedge sel(x_{\text{init}}, i) = \text{INIT}$.

2.3 Annotations

A certifiable code generator [4, 15] derives not only code from high-level specifications but also the detailed annotations required to certify a given safety property. The basic idea is to make the annotations part of the templates used in the generator so that they can be instantiated and refined in parallel with the code fragments. The annotations focus on locally relevant information, without describing all the global information that may later be necessary for the proofs. It then relies on a separate annotation propagation phase after the code has been constructed.

The propagation algorithm can be seen as a crude approximation of a strongest postcondition predicate transformer. It pushes the generated local annotations forward

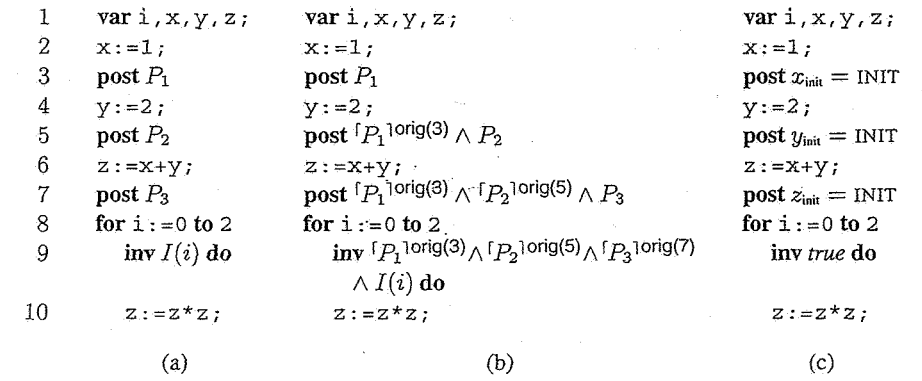

Fig. 3. (a) Code with annotation skeletons (b) Code with annotation skeletons after propagation (c) Code with actual annotations.

along the edges of the syntax tree as long as the information can be guaranteed to remain unchanged. The VCG then processes the code after propagation. The calculus assumes that all loops have an invariant; typically, they consist mainly of assertions which have been propagated from earlier in the program.

For us, human-readable explanations of the VCs are important in gaining confidence into the (large and complex) generator, the propagator, and the certifier. However, our approach is not tied to code generation; we only use the generator as a convenient source of the annotations that allow the construction of the VCs and thus the Hoare-style proofs. Moreover, even though there is no need to manually classify annotations (since all work can be done by the VCG), it is possible to add domain-specific information in the form of additional labels which are then processed in the usual way (cf. Section 5).

3 Explaining the Structure of VCs

The main aspect of VCs that we consider in this paper is their *structure*. Since VCs have the form $H_1 \wedge \ldots \wedge H_n \Rightarrow C$ after simplification, the structure is based on the top-level dichotomy between logical hypotheses and conclusions. However, for meaningful explanations we need a more detailed characterization of the sub-formulas. This information cannot be recovered from the obligations or the code but must be specified explicitly. A key insight of our approach is that the different sub-formulas stem from specific positions in the Hoare rules, and that the VCG can thus add the appropriate labels to the VCs. Note that locations alone are not even sufficient to distinguish between hypotheses and conclusions: due to backward edges in the control flow graph, the highest line number does not necessarily refer to the conclusion.

3.1 Example Explanations

Figure 3 shows three different versions of a small example program to illustrate the process. In Figure 3(a) and 3(b), the actual annotations are abstracted by meta-variables to

The purpose of this proof obligation is to show that the loop invariant at line 9 under the substitution originating from line 10 is still true after each loop iteration; it is also used to show the preservation of the loop invariant at line 9. Hence, given
- the postcondition at line 3 propagated into the invariant at line 9,
- the postcondition at line 5 propagated into the invariant at line 9,
- the postcondition at line 7 propagated into the invariant at line 9,
- the invariant at line 9,
- the loop bounds at line 10,

show that the loop invariant at line 9 under the substitution originating from line 11 is still true after each iteration to line 11.

Fig. 4. Explanation automatically generated for the VC $0 \leq i \leq 2 \wedge x_{\text{init}} = \text{INIT} \wedge y_{\text{init}} = \text{INIT} \wedge z_{\text{init}} = \text{INIT} \Rightarrow \text{INIT} = \text{INIT}$ derived from Figure 3(c).

simplify the presentation. Figure 3(a) shows the original annotations while Figure 3(b) shows the result of the propagation phase. Note that this step already introduces some labels; for example, in line 7 the sub-formulas P_1 and P_2 are labeled with their original locations (i.e., lines 3 and 5). Throughout the paper we use the notation $\lceil t \rceil^{\text{lab}}$ to denote a term t that is labeled with a label lab; the labels can also have internal structure (see Section 3.2 for details). Figure 3(c) shows the actual annotations required (before propagation) to certify the program as initialization safe.

The version in Figure 3(b) induces simple VCs whose structure still directly reflects their intended meaning. Each sub-formula is preserved and can be traced back to its origin, which allows a human to interpret them in text, e.g., (v) $P_1 \wedge P_2 \wedge P_3 \wedge I(i) \wedge 0 \leq i \leq 2$ corresponds to "Given the postcondition P_1 from line 3, the postcondition P_2 from line 5, and the postcondition P_3 from line 7, the loop bounds at line 8, and the loop invariant at line 9, show that the loop invariant at line 9 hold is still true after each loop iteration."The explanations become more complicated when the substitutions arising from the *assign-* and *update*-rules are taken into account because the non-local effects of substitution applications need explaining: the sub-formulas are no longer preserved intact and need to be traced to their different origins and for larger programs, the overall structure quickly becomes complex. Our technique thus mechanizes the textual interpretation, and Figure 4 shows the automatically generated structural explanation for the same VC as above, but now derived using the full annotations in Figure 3(c).[1]

3.2 Mark-Up Structures

Concept Hierarchy The basic information for explanation generation is the set of underlying concepts and their hierarchical relations. This hierarchy depends of course on the particular aspect of the VCs to be explained. In the case of the structural explanations, the concepts characterize either the origin or the purpose of a sub-formula. Figure 5 shows the concept hierarchy; the actual labels are shown in parentheses next to the corresponding concepts.

[1] Of course, aggressive simplification reduces the complexity of the VCs (and even reduces away the given example) but this does not solve the problem in general, and one could argue that it in fact compounds the problem by changing the formula structure.

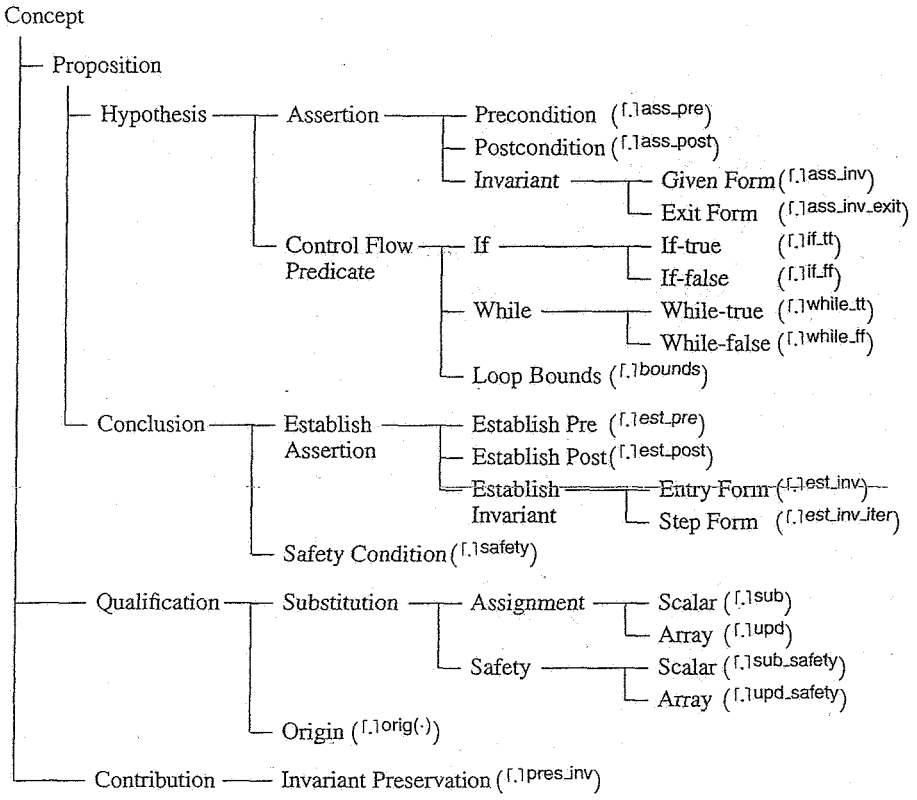

Fig. 5. Concept hierarchy for structural explanations

Propositions are either hypotheses or conclusions, reflecting their eventual position in the VC. *Hypotheses* are further divided into assertions and control flow predicates. *Assertions* refer to sub-formulas that occur as annotations in the program, either originally or after propagation (i.e., pre- and post-conditions and loop invariants). Since the (*for*)-rule uses the loop invariant as hypothesis in two different contexts, we distinguish the two concepts $\ulcorner.\urcorner^{\text{ass\_inv}}$ and $\ulcorner.\urcorner^{\text{ass\_inv\_exit}}$. *Control flow predicates* refer to sub-formulas that reflect the program's control flow. For both *if*-statements and *while*-loops, the control flow predicates occurring in the program are required by the rules in both their original and negated forms, so that we get four different concepts for explanations. For *for*-loops, the control flow predicate does not directly occur in the program but is derived from the given loop bounds.

Conclusions capture the primary purpose of a VC. For general verification, the different types of conclusions are given by the types of assertions, as each assertion has to be established (i.e., shown valid at its location). As in the case of the hypotheses, loop invariants are used in two different forms, the *entry form* (or base case) $\ulcorner.\urcorner^{\text{est\_inv}}$ and the *step form* $\ulcorner.\urcorner^{\text{est\_inv\_iter}}$. Note that an assertion can be used both as hypothesis and as conclusion, even in the same VC. Our approach allows the explanations to distinguish these two bits of information from the same source. For safety verification, we additionally have the safety conditions $\ulcorner.\urcorner^{\text{safety}}$ that have to be demonstrated.

Qualifications, which apply to both hypotheses and conclusions, further characterize the origin of a sub-formula. The different *substitution* concepts reflect the substitutions of the underlying Hoare calculus. Both *assignment* concepts capture the origin and effect of assignments on the form of the resulting VCs.For safety verification, we additionally get *safety substitutions*, again for the scalar ($^{\lceil.\rceil_{\text{sub\_safety}}}$) and array ($^{\lceil.\rceil_{\text{upd\_safety}}}$) cases. *Origin labels* ($^{\lceil.\rceil_{\text{orig}(\cdot)}}$) denote formulas that result from annotations that have been propagated from their original location; this location is the additional argument of the label. They are specific to the way the code generator produces the annotations.

Contributions capture the secondary purpose of a VC; this arises when a recursive call to the VCG produces VCs that are conceptually connected to the purpose of the larger structure. In general, contributions arise for nested program structures which result in "nested" VCs (e.g., loops within loops). For example, all VCs emerging from the premise $P\ \{c\}\ I$ of the *while*-rule (cf. Figure 2) contribute to showing the preservation of the invariant I over the loop body c, independent of their primary purpose. For the structural hierarchy, this preservation of invariants ($^{\lceil.\rceil_{\text{pres\_inv}}}$) is the only contribution concept.

Concept Classes In the concept hierarchy shown in Figure 5, only the minimal (i.e., leaf) concepts are associated with labels. However, the other concepts also capture important domain information, as they group together related labels. These concept classes are then used as filter predicates by the rendering phase (cf. Section 4.2) and also to show the consistency of the rule mark-up.

For the structural explanations, the important concept classes are the two proposition classes *hypothesis* and *conclusion*, the two qualification classes *substitution* and *origin*, and the *contribution* concept.

Label Structure The labels that are actually attached to the VCs are ground terms over the label functors Σ shown in Figure 5. Each label $c(o, n)$ thus comprises a concept $c \in \Sigma$ which describes the role the labeled term plays, the location o where it originated, and an optional list of nested labels n. The concept of the outermost label generally determines how the underlying term is rendered. Locations refer either to an individual position or to a range; in our implementation, we use plain line numbers for locations and ranges; to capture more details, we could instead use full term positions $\mathbb{N}^*$ without any substantial changes. The nested label list initially holds the qualifications for the concept of the top-level formula. After normalization and extraction, just before rendering, it also contains labels extracted from sub-formulas.

3.3 Rules

In general, it is not sufficient to just output explanations as the VCs are constructed. Instead, the VCG must add the right labels at the right positions; it must also pass mark-up back through the program by attaching it to the WSPC, so that information from one point in the program can be used at any other point. Modified Hoare rules concisely capture the semantic mark-up (i.e., label types and positions) required for any given explanation aspect. Labels can be added in three places: to the "incoming" postcondition of a recursive VCG call in the premise of an inference rule, to the WSPC, or to a generated VC. Figure 6 shows the rules for the initialization safety policy marked-up for

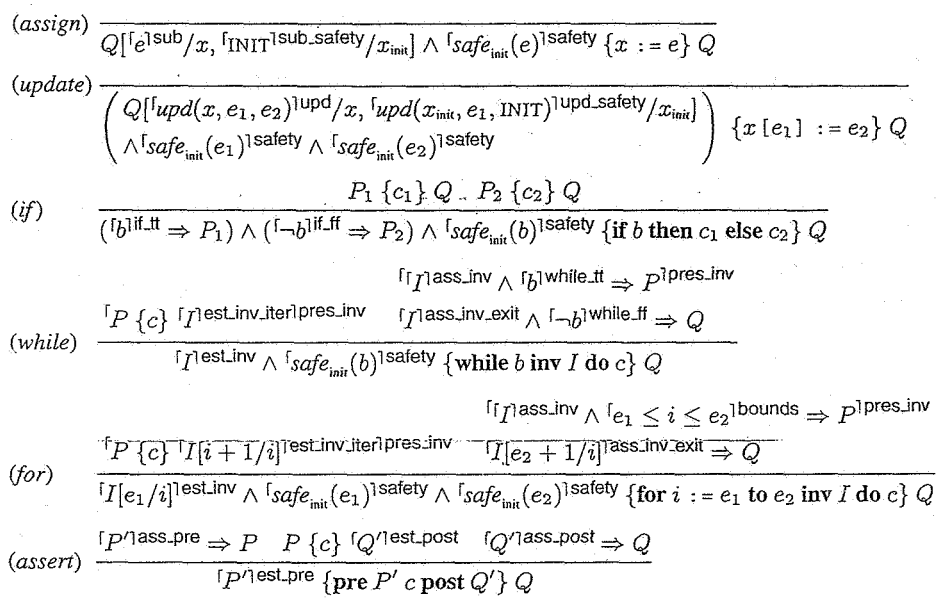

Fig. 6. Hoare rules for initialization safety with semantic markup

explaining the structural aspect of VCs; those rules not shown do not require any mark-up. The rules derive the usual triples, $P\ \{c\}\ Q$, but now all elements can be labeled. For clarity, we omit the location information in the rule formulation but assume that the VCG obtains it from the statements and annotations and appropriately incorporates it into the labels. Note that the incoming postconditions have no location information by themselves, although their sub-formulas do.

The *assign*- and *update*-rules only require mark-up in the WSPC. The safety predicate can be a complex sub-formula, depending on the property to be certified and the structure of the expression(s), but the mark-up is not dependent on the specific safety property—all we need to know for an explanation is that this is in fact the safety predicate. However, it could contain additional embedded labels for more detailed or property-specific explanations. The substitutions need mark-up to record their type and the origin of the substituted expressions. Note that by labeling only the expressions and not the variables we can use the normal substitution mechanisms.

The *if*-rule is also fairly simple. The safety predicate is labeled as above. The other two parts of the WSPC reflect the two branches of the *if*-statement. They are guarded by the appropriate version of b, and the labels just record which version has been used. Note that P_1 and P_2 are likely to be labeled, depending on the safety policy and the particular structure of the branches c_1 and c_2, but no additional labels are required for them in the context of the *if*-statement.

The loop rules are more complicated; we focus on the *while*-rule but the *for*-rule has the same overall structure. The WSPC comprises the safety predicate, which is labeled as before, and the invariant, which has to be established in the entry form and

Explanation ::= "`The purpose of this VC is to` " *Purpose%1* " `.\n` " *Body%1*
Purpose ::= *Conclusion* ["; `it is also used to` " *Contributions*] "."
Conclusion ::= ⊓est_pre | ⊓est_post | ⊓est_inv | ⊓est_inv_iter | ⊓safety
⊓est_pre ::= "`show the precondition at` " *Line%1.1* [*OriginsR%1*] [*Substitutions%1*]
...
Substitutions ::= *SubstBlock%1.2$Substitution*
SubstBlock ::= "`under the substitution from` " *Line* {", " *Line*}
Contributions ::= *Contribution* {"`which in turn is also used to` " *Contribution*}
Body ::= "`Given` " {"`\n` − " *Hypothesis* ", "} *Conclusion*
Hypothesis ::= ⊓ass_pre | ⊓ass_post | ⊓ass_inv | ⊓ass_inv_exit
| ⊓if_tt | ⊓if_ff | ⊓while_tt | ⊓while_ff | ⊓bounds
⊓ass_pre ::= [*OriginsL%1*] "`the precondition at` " *Line%1.1* [*Substitutions%1*]

Fig. 7. Explanation templates for structural explanations (excerpts)

is thus labeled with $\lceil.\rceil_{\text{est\_inv}}$. In the premise, individual sub-formulas of both the exit-condition $I \wedge \neg b \Rightarrow Q$ and the step-condition $I \wedge b \Rightarrow P$ are labeled appropriately; in addition, the entire step-condition is labeled with its secondary purpose, namely to contribute to showing the preservation of the invariant. In the triple $P\ \{c\}\ I$, the incoming postcondition I must be labeled with its purpose for the recursive call; moreover, all emerging VCs must be marked up with the secondary purpose $\lceil.\rceil_{\text{pres\_inv}}$. We indicate this by labeling the entire triple. Note how the same formula I is used in four different roles and consequently labeled in four different ways. This contextual knowledge is only available at the point of rule application and can not be easily recovered by a post-hoc analysis of the generated VCs. This is a strength of our approach.

Finally, the *assert*-rule is straightforward to mark up. The asserted pre- and post-conditions are labeled according to their use either as hypotheses (in the VCs) or as conclusions (in the WSPC and recursion).

3.4 Explanation Templates

Finally, we also need to define the underlying abstract syntax and actual textual representation of the explanations. Figure 7 shows parts of the BNF-grammar that specifies both. It is derived from the concept hierarchy: the leaf concepts (i.e., labels) double as non-terminals, and the concept classes correspond to alternative productions.

Since the purpose of the grammar is to generate rather than analyze text, the right-hand sides of the rules can be interpreted as functions of type *Label list* → *Text option*. These *explanation templates* are similar to C's format strings and can also contain special operators, which control parameter passing into other non-literals. The %-operator allows access to the incoming parameters, using a position notation so that %1 is the first parameter and %1.1 its first argument. The $-operator tests an argument against a pattern, so that for example *SubstBlock%1.2$Substitution* takes the second argument of the incoming label (i.e., the nested labels), filters out those that match the pattern *Substitution* (i.e., are concepts in that class) and passes the result on.

$$
\begin{array}{llllll}
\ulcorner \mathit{true} \urcorner^{\mathsf{l}} & \to \mathit{true} & P \Rightarrow P' & \to \mathit{true} & \text{if } |P| = |P'| & \left.\begin{array}{l}\\\\\end{array}\right\} \\
\neg \ulcorner \mathit{false} \urcorner^{\mathsf{l}} & \to \mathit{true} & t = t' & \to \mathit{true} & \text{if } |t| = |t'| & \left.\begin{array}{l}\\\\\end{array}\right\} (i) \\
\ulcorner \mathit{false} \urcorner^{\mathsf{l}} \Rightarrow P & \to \mathit{true} & & & & \\
\ulcorner \mathit{false} \urcorner^{\mathsf{l}} \vee P & \to P & P \Rightarrow \ulcorner \mathit{false} \urcorner^{\mathsf{l}} & \to \neg P & & (ii) \\
\ulcorner \mathit{false} \urcorner^{\mathsf{l}} \wedge P & \to \ulcorner \mathit{false} \urcorner^{\mathsf{l}} & P \wedge \neg P' & \to \ulcorner \mathit{false} \urcorner^{[\![P]\!] \oplus [\![P']\!]} & \text{if } |P| = |P'| & (iii) \\
\ulcorner P \wedge Q \urcorner^{\mathsf{l}} & \to \ulcorner P \urcorner^{\mathsf{l}} \wedge \ulcorner Q \urcorner^{\mathsf{l}} & P \Rightarrow \ulcorner Q \Rightarrow R \urcorner^{\mathsf{l}} & \to P \wedge \ulcorner Q \urcorner^{\mathsf{l}} \Rightarrow \ulcorner R \urcorner^{\mathsf{l}} & & (iv) \\
\neg \ulcorner \neg P \urcorner^{\mathsf{l}} & \to \ulcorner P \urcorner^{\mathsf{l}} & \ulcorner \ulcorner t \urcorner^{\mathsf{m}} \urcorner^{\mathsf{n}} & \to \ulcorner t \urcorner^{\mathsf{n} \otimes \mathsf{m}} & & \left.\begin{array}{l}\\\end{array}\right\} \\
\multicolumn{5}{l}{\mathit{sel}(\ulcorner \mathit{upd}(x, i_1, t) \urcorner^{\mathsf{l}}, i_2) \to \ulcorner i_1 = i_2 \urcorner^{\mathsf{l}} \,?\, \ulcorner t \urcorner^{\mathsf{l}} : \mathit{sel}(x, i_2)} & (v)
\end{array}
$$

Fig. 8. Labeled rewrite rules

The templates allow an easy customization and fine-grained control of the textual explanations. For example, we render origins in hypotheses differently than in conclusions (cf. *OriginL* and *OriginR* in Figure 7).

4 Explanation Generation

The generation of the actual textual explanations is independent of the particular aspect which is to be explained and can thus be reused. It proceeds in two phases, a rewrite-based normalization of the VCs and corresponding labels followed by a rendering phase that extracts and further normalizes the final label structure and, using the specific explanation templates, turns it into natural language text.

4.1 Labeled Rewriting

The VCs (whether labeled or unlabeled) become quite complex and need to be simplified aggressively before they can be submitted to an ATP with any hope of success (cf. [6] for experimental evidence). In the unlabeled case, the simplification can easily be implemented by a term rewriting system. Unfortunately, the unlabeled rules cannot be reused "as is" for the labeled case because (*i*) the labeling changes the term structure and thus the applicability of the rules and (*ii*) the labels need careful handling—on the one hand, they cannot simply be distributed over all operators because this can destroy their proper scope, while on the other, they cannot just be pushed to the top of the VC because this would result in redundant and imprecise explanations.

Figure 8 shows the labeled rewrite rules that are used together with additional unlabeled rules to simplify the labeled VCs. Their formulation uses the auxiliary functions $|\cdot|$ to remove labels from terms, and $[\![\cdot]\!]$ to extract the labels of a term. $[\![\cdot]\!]$ is defined by

$$
\begin{array}{ll}
[\![\ulcorner f(t_1, \ldots, t_n) \urcorner^{\mathsf{lab}}]\!] & = \mathsf{lab} \otimes ([\![t_1]\!] \oplus \ldots \oplus [\![t_n]\!]) \\
[\![f(t_1, \ldots, t_n)]\!] & = [\![t_1]\!] \oplus \ldots \oplus [\![t_n]\!]
\end{array}
$$

where $\oplus$ is list concatenation and the label composition operator appends the inner labels l to the list of labels nested in the outer label $c(o, n)$, i.e., $c(o, n) \otimes l = c(o, n \oplus l)$.

The rules are based on the assumption that the majority of the VCs can be rewritten to *true*. Their purpose is thus (*i*) to remove redundant labels, (*ii*) to minimize the scope of the remaining labels, and (*iii*) to keep enough labels to explain any unexpected failures. The rules fall into five different categories. The first category removes labels from trivially true (sub-) formulas because these require no explanations. The next category *selectively* removes labels from trivially false sub-formulas so that the equivalent unlabeled versions can apply and eliminate the sub-formulas. The remaining context then provides the information for the explanations. However, the labels obviously need to be retained if the unlabeled versions rewrite the *entire* formula into *false*, since there is no remaining context to explain the failure. Note that the second rule only extracts the labels from the two contradictory literals but does not introduce additional mark-up to explain the contradiction. The rules in the fourth distribute labels over conjunction and (nested) implication, respectively, so that the label scopes are minimized in the final simplified VCs. The final category encodes knowledge about how the labels will be interpreted in the underlying domain. For example, the second rule flattens nested labels using the label composition operator, thus enabling other labeled and unlabeled rules to apply. The last rule specifies the effect of selecting into an updated array. In order to explain the resulting term we need to know that the disappearing *upd*-functor is conceptually reflected in the guard and the success-branch of the conditional, but not in the failure-branch, and that the label must thus be attached to these two only.

The rewrite system is not confluent modulo labels, in the sense that terms such as $\ulcorner \mathit{false} \urcorner^{\mathsf{l}} \wedge \neg \ulcorner \mathit{false} \urcorner^{\mathsf{m}}$ can be rewritten into differently labeled normal forms (in this case $\ulcorner \mathit{false} \urcorner^{\mathsf{l}}$, $\ulcorner \mathit{false} \urcorner^{\mathsf{m}}$ (using commutativity of $\wedge$), and $\ulcorner \mathit{false} \urcorner^{[\mathsf{l},\mathsf{m}]}$). However, the system is confluent after label stripping $|\cdot|$, and since the rules are labeled versions of rules in the underlying unlabeled rewrite system, labeling does not interfere with the underlying unlabeled normalization.

4.2 Rendering

The core rendering routine, which turns the (labeled) VCs into human-readable text, is independent of the actual aspect that is explained. It relies on the building blocks described so far and comprises four steps: (*i*) VC normalization, using the labeled rewrite system; (*ii*) label extraction, using $[\![\cdot]\!]$; (*iii*) label normalization, to fit explanation templates; (*iv*) text generation, using the explanation templates.

The third step flattens nested substitution- and contribution-labels, so that for example sub(p, sub(q, sub(r))) is rewritten into the list [sub(p), sub(q), sub(r)]. It also merges back together conclusions from the same line which have been split over different literals during the first step. This is realized by an additional rewrite system (omitted here) that is defined together with the explanation templates.

The renderer contains code to interpret the BNF meta-symbols (i.e., lists and alternatives) as well as some glue code (e.g., sorting label lists by line numbers) that is spliced in to support the text generation. It also provides default templates for concepts that are useful for different explanation aspects, for example lines and ranges.

```
1  var i,j,x,m[0:9];
2  i:=0;x:=0;
3  while i<100
4     inv i > 0 ⇒ ∀k · (0≤k≤9 ⇒ sel(m_init, k) = INIT) do
5     i:=i+1;
6     if i>0 then
7        for j:=0 to 9
8           inv ∀k · (0≤k≤9 ⇒ sel(m_init, k) = INIT) do
9           x:=x+m[i];
       else
10       for j:=0 to 9
11          inv ∀k · (0≤k≤j−1 ⇒ sel(m_init, k) = INIT) do
12          m[i]:=i;
13       post ⌜∀k · (0≤k≤9 ⇒ sel(m_init, k) = INIT)⌝^init(10:12,m)
```

Fig. 9. Code with domain-specific label

5 Explaining the Purpose of VCs

The explanations constructed by the rules in Figures 6 relate primarily to the structure of the VCs. However, this structural information also has some semantic information since it distinguishes the multiple roles a single annotation can take. We can go further and use domain-specific information to give a more semantic explanation of VCs which complements the purely structural logical view.

For example, usually one code block initializes a variable while other parts of the program use it and so rely on its initialization. For safety certification, numerous VCs will be generated both within the *definition* of the variable and for each of its uses. However, the structural explanations do not relate any of the VCs to their role either as a constituent obligation within a definition for a given safety policy or when used later in another part of the program. We call this role (informally) the *purpose* of a VC.

In order to reflect this, we need to explain the appropriate VCs in terms of establishing a definition, while all VCs produced within the definition block should be labeled as contributing to that definition, similar to the structural case of nested loops. Likewise, whenever the final postcondition is used as a hypothesis, e.g., to show the safety of a later use, it must be labeled as originating from the definition.

Of course, the situation can quickly become complex. There might be irrelevant code within the the definition block, so that the VCs are not obviously tied to the structure of the definition. Similarly, as shown in the example in Figure 9, the path from definition to use might be convoluted and via intermediate annotations. To deal with all this, we need to extend the structural explanations with definitional information.

We do this with a simple extension to the existing calculus. The basic idea is to add labels to the formulas in the program's annotations that express *domain-specific* refinements of the purely structural concepts shown in Figure 5. For example, in Figure 9 the postcondition $\forall i \cdot 0 \leq i \leq 9 \Rightarrow sel(m_{init}, i) = \text{INIT}$ in line 13 is labeled with init(m, 10:12) to represent the fact that at this program point the array m is fully initialized, due to the statements in lines 10–12. Note that the fact that this code block represents

The purpose of this proof obligation is to establish the postcondition at line 13 (i.e., line 10 to line 12 define the variable m); it is also used to show the definition at line 13, which in turn is used to show the preservation of the loop invariant at line 4. Hence, given
- the guard at line 3,
- the invariant at line 4,
- the negation of the condition at line 6 under the substitution originating from line 5,
- the invariant at line 11 in its exit form after line 12,

show the postcondition at line 13 (i.e., line 10 to line 12 define the variable m).

Fig. 10. Explanation automatically generated for the VC $(i < 100 \wedge i + 1 \leq 1 \wedge (i > 0 \Rightarrow (\forall k \cdot (0 \leq k \leq 9) \Rightarrow sel(m_{\text{init}}, k) = \text{INIT}))) \Rightarrow ((\forall k \cdot (0 \leq k \leq 9) \Rightarrow sel(m_{\text{init}}, k) = \text{INIT}) \Rightarrow (\forall k \cdot (0 \leq k \leq 9) \Rightarrow sel(m_{\text{init}}, k) = \text{INIT}))$ derived from Figure 9.

an initialization cannot be inferred by the VCG but must be specified externally, e.g., by the code generator, so the explanations are still based on explicitly specified labels.

Domain-specific labels are only introduced at annotations; we can restrict them even further to pre- and postconditions since the enclosed statements correspond to blocks with a logical structure which is reflected in the label and used in the explanation. Hence, we only need to introduce one additional rule

$$(\textit{label})\ \frac{\ulcorner P \urcorner_{\text{ass\_pre(l)}} \Rightarrow \ulcorner P \urcorner_{\text{contrib(l)}} \quad \ulcorner P \ \{c\}\ Q' \urcorner_{\text{est\_post(l)}\urcorner\text{contrib(l)}} \quad \ulcorner Q' \urcorner_{\text{ass\_post(l)}} \Rightarrow Q}{\ulcorner P \urcorner_{\text{est\_pre(l)}}\ \{\textbf{pre}\ P'\ c\ \textbf{post}\ \ulcorner Q' \urcorner\}\ Q}$$

that generalizes the *assert*-rule. The *label*-rule "plucks" the label off the postcondition and passes it into the appropriate positions. Labels in positions that are already labeled by the *assert*-rule need to be modified to take the domain-specific labels as an additional argument. For example, ass_post(lab) then refers to an asserted postcondition (i.e., a postcondition used as a hypothesis) for a lab-block. In addition, we also introduce a new contribution label contrib(lab), similar to the invariant preservation in the structural concept hierarchy. This is added to the WSPC recursively computed for the block, and to all VCs emerging during that process. These more refined labels let the renderer determine when a hypothesis is actually the postcondition of a domain-specific block, or when a VC is just a contributor to the block. Note that this modification does not affect the other rules of the calculus since the "backwards" formulation of the rules means that the labels plucked from the formulas will be silently passed through into the final VCs. It is the responsibility of the renderer to make use of this additional information as it appears in the final labeled VCs. Figure 10 shows an example for this.

6 Related Work

Most VCGs link VCs to source locations, i.e., the actual position in the code where the respective rule was applied and hence where the VC originated. Usually, the systems only deal with line numbers but Fraer [10] describes a system that supports a "deep linking" to detailed term positions. JACK [1] and PerfectDeveloper [2] classify the VCs on the top-level and produce short captions like "precondition satisfied", "return value satisfies specification", etc. In general, however, none of these approaches maintain more non-local information (e.g., substitution applications).

Denney and Venkatesan [7] present a system for generating safety documents. It also turns VCs into text in order to explain the safety of the code but differs from the current work by being more policy- and code-driven. It only uses the VCs which comprise safety conditions, and combines rendering of the VCs with further code analysis to give a global explanation, whereas we concentrate on explaining all VCs directly.

Leino et al [11] also have a system for explaining VCs. In particular, their explanations are of traces to safety conditions. This is sufficient for debugging programs, which is their main motivation. However, they only label the "counterexample context" whereas we label the entire VC and can also explain the more global purpose of a proof obligation. Both [11] and the current work are based on extending an underlying logic with labels to represent semantic explanatory information. However, they differ in how these labels are used by the verification architectures. Leino's system introduces the labels by first desugaring the language into a lower-level form. Labels are treated as uninterpreted predicate symbols and labeled formulas are therefore just ordinary formulas. This labeled language is then processed by a standard VCG which is "label-blind". In contrast, we do not have a desugaring stage, and use the VCG and propagator to insert the labels. Consequently, our simplifier needs to be label-aware, but since we strip labels off the final VCs after the explanation has been constructed, we do not suffer any performance problems with the ATP, nor do we place special requirements on the prover like they do.

7 Conclusions and Future Work

The explanation mechanism which we have described here has been successfully implemented and incorporated into our certification browser [5]. This tool is used to navigate the artifacts produced during certifiable code generation, and it uses the system described in this paper to successfully explain all the VCs produced by both AUTOFILTER and AUTOBAYES for all safety policies.

We are currently redesigning the annotation generation mechanism for our code generators and the explanation mechanism has helped us debug several problems that arose there. In addition to its use in debugging, the explainer can also be used as a means of gaining assurance that the verification is itself trustworthy. This complements our previous work on proof checking [14]: there a machine checks one formal artifact (the proof), here we support human checking of another (the VCs). With this role in mind, we are currently extending the tool to be useful for code reviews.

Much more work can be done to improve and extend the actual explanations themselves. Similar to frame axioms, VCs often contain complex subformulas that remain unchanged because they are just pushed through parts of the program, and the current explanations do not explain that. Similarly, they do not make any use of the particular safety policy which is being certified but this could be used to give more informative explanations, particularly in combination with inductive steps. For example, for the initialization safety which is used as an example in this paper we would like to generate phrases of the form "initialized up to i-1". More generally, we would like to allow explanations to be based on entirely different explanation structures or *ontologies*. Our approach can, for example, also be used to explain the *provenance* of a VC (i.e., the

tools and people involved in its construction) or to link it together with supporting information such as code reviews, test suites, or off-line proofs. Moreover, although we have restricted our attention here to text generation, the underlying mechanisms are more general and could for example also be used to generate prover hints or even entire proof scripts for the VCs. The techniques could also be applied to other program logics, such as the refinement calculus.

Finally, there are also interesting theoretical issues. The renderer relies on the existence of an *Explanation Normal Form*, which states intuitively that each VC is labeled with a unique conclusion. This is essentially a rudimentary soundness result, which can be shown in two steps, first by induction over the marked-up Hoare rules in Figure 6 and then by induction over the labeled rewrite rules in Figure 8. We are currently developing a theoretical basis for the explanation of VCs that is generic in the aspect that is explained, with appropriate notions of soundness and completeness.

References

1. L. Burdy and A. Requet. Jack: Java applet correctness kit. In *Proc. 4th Gemplus Developer Conference*, Singapore, Nov. 2002.
2. D. Crocker. Perfect developer: a tool for object-oriented formal specification and refinement. In *FM 2003 Tool Exhibition Notes*, pp. 37–41, Pisa, Italy, 2003.
3. E. Denney and B. Fischer. Correctness of source-level safety policies. In *FM 2003*, *LNCS 2805*, pp. 894–913. Springer, 2003.
4. E. Denney and B. Fischer. Certifiable program generation. In *GPCE 2005*, *LNCS 3676*, pp. 17–28. Springer, 2005.
5. E. Denney and B. Fischer. A program certification assistant based on fully automated theorem provers. In *Proc. Intl. Workshop on User Interfaces for Theorem Provers, (UITP'05)*, Edinburgh, 2005.
6. E. Denney, B. Fischer, and J. Schumann. An empirical evaluation of automated theorem provers in software certification. *Intl. J. of AI Tools*, 15(1):81–107, 2006.
7. E. Denney and R. P. Venkatesan. A generic software safety document generator. In *10th AMAST*, *LNCS 3097*, pp. 102–116. Springer, 2004.
8. A. Fiedler. Natural Language Proof Explanation. In *Mechnizing Mathematical Reasoning*, *LNAI 2605*, pp. 342–363. Springer, 2005.
9. B. Fischer and J. Schumann. AutoBayes: A system for generating data analysis programs from statistical models. *J. Functional Programming*, 13(3):483–508, 2003.
10. R. Fraer. Tracing the origins of verification conditions. In *5th AMAST*, *LNCS 1101*, pp. 241–255. Springer, 1996.
11. K. R. M. Leino, T. Millstein, and J. B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 55(1–3):209–226, 2005.
12. J. C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.
13. G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *PLDI'98*, pp. 333–344. ACM Press, 1998.
14. G. Sutcliffe, E. Denney, and B. Fischer. Practical proof checking for program certification. In *Proc. CADE-20 Workshop on Empirically Successful Classical Automated Reasoning (ESCAR'05)*, Tallinn, Estonia, July 2005.
15. M. Whalen, J. Schumann, and B. Fischer. Synthesizing certified code. In *FME'02*, *LNCS 2391*, pp. 431–450. Springer, 2002.
16. J. Whittle and J. Schumann. Automating the implementation of Kalman filter algorithms. *ACM Trans. Mathematical Software*, 30(4):434–453, 2004.