# Explaining Verification Conditions

Ewen Denney[1] and Bernd Fischer[2]

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

**Abstract.** Hoare-style 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 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. The explanations can be customized to capture different aspects of the VCs; here, we focus on labelings that explain their structure and purpose.

## 1 Introduction

Program verification is easy when automated tools do all the work: a verification condition generator (VCG) takes a program that is "marked-up" with logical annotations (i.e., pre-/post-conditions and invariants) and produces a number of verification conditions (VCs) that are simplified, completed by 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, the domain theory may be incomplete, or the ATP may run out of resources. In each of these cases, users are typically confronted only with failed VCs (i.e., the failure to prove them automatically) but receive no additional information about the causes of the failure. They must thus analyze the VCs, interpret their constituent parts, and relate them through the applied Hoare rules and simplifications to the corresponding source code fragments. Even if all VCs can be proven automatically, there is often still a need to understand their intent, for example if the formal verification is being used to support a code review. Unfortunately, VCs are a detailed, low-level representation of both the underlying information and the process used to derive it, so understanding them is often difficult.

Here we describe an technique that helps users to trace 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 structured *labels* that are attached to the meta-variables used in the Hoare rules (or to the annotations in the program), so that the VCG produces labeled versions of the VCs. The labels are maintained through the different processing steps, 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, these line numbers on their own are insufficient to understand a VC since they do 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 are thus insufficient as a basis for informative explanations. Some systems produce short captions for each VC (e.g., JACK [1] or Perfect Developer [2]). Other techniques focus on a detailed linking between source locations and VCs to support program debugging [11,12]. Our approach, in contrast, serves as a customizable basis to explain different aspects of VCs. Here, we focus on explaining the *structure* and *purpose* of VCs, helping users to understand what a VC means and how it contributes to the overall certification of a program.

In our approach we only explain what has been explicitly declared to be significant using labels. The generated explanations are based on an analysis of the labels and not of the structure or logical meaning of the underlying VCs. For example, we do not try to infer that two formulas are the base and step case of an induction and hence would not generate an explanation to that end unless the formulas are specifically marked up with this information. Finally, we restrict ourselves to explaining the construction of VCs (which is the essence of the Hoare approach) rather than their proof. Hence, we maintain, and can also introduce, labels during simplification, but strip them off before proving the VCs. Techniques for explaining proofs (e.g., [9]) provide no additional insight, and are in fact less useful for our purposes since the key information is expressed in the annotations and VCs.

We developed our technique as part of an autocode certification system [4,6], and we will use the safety verification of automatically generated code as an application example. Here, human-readable explanations of the VCs are particularly important to gain confidence into the generated code. However, our technique is not tied to either code generation or safety certification and can be used in any Hoare-style verification context. We first briefly describe the core calculus, and then its labeled extension. We also describe several refinements to the labels, which give rise to richer explanations. Some of these refinements are specific to our application domain (i.e., safety verification) while others are specific to our verification method (i.e., automated annotation).

## 2 Logical Background

**Hoare Logic and Program Verification.** We follow the usual Hoare-style program verification approach: first, a VCG applies the rules of the underlying Hoare calculus to the annotated program to produce a number of VCs, then an ATP discharges the VCs. This splits the decidable construction of the VCs from their undecidable discharge, but in return the VCs become removed from the program context, which exacerbates the understanding problem.

Here, we restrict our attention to an imperative core language which is sufficient for the programs generated by NASA's certifiable code generators, AUTOBAYES [10] and AUTOFILTER [15], and by Real-Time Workshop (RTW), a commercial code generator for Matlab. Extensions to other language constructs are straightforward, as long as the appropriate (unlabeled) Hoare rules have been formulated.

$$(\mathit{assign})\ \frac{}{Q[e/x, \textsc{init}/x_{\text{init}}] \land \mathit{safe}_{\text{init}}(e)\ \{x\ := e\}\ Q}$$

$$(\mathit{update})\ \frac{}{\begin{pmatrix} Q[\mathit{upd}(x, e_1, e_2)/x, \mathit{upd}(x_{\text{init}}, e_1, \textsc{init})/x_{\text{init}}] \\ \land\ \mathit{safe}_{\text{init}}(e_1) \land \mathit{safe}_{\text{init}}(e_2) \end{pmatrix}\ \{x\,[\,e_1\,]\ := e_2\}\ Q}$$

$$(\mathit{if})\ \frac{P_1\ \{c_1\}\ Q \quad P_2\ \{c_2\}\ Q}{(b \Rightarrow P_1) \land (\neg b \Rightarrow P_2) \land \mathit{safe}_{\text{init}}(b)\ \{\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\}\ Q}$$

$$(\mathit{while})\ \frac{P\ \{c\}\ I \quad I \land b \Rightarrow P \quad I \land \neg b \Rightarrow Q}{I \land \mathit{safe}_{\text{init}}(b)\ \{\textbf{while}\ b\ \textbf{inv}\ I\ \textbf{do}\ c\}\ Q}$$

$$(\mathit{for})\ \frac{P\ \{c\}\ I[i+1/i] \quad I[\textsc{init}/i_{\text{init}}] \land e_1 \leq i \leq e_2 \Rightarrow P \quad I[e_2+1/i, \textsc{init}/i_{\text{init}}] \Rightarrow Q}{e_1 \leq e2 \land I[e_1/i] \land \mathit{safe}_{\text{init}}(e_1) \land \mathit{safe}_{\text{init}}(e_2)\ \{\textbf{for}\ i\ := e_1\ \textbf{to}\ e_2\ \textbf{inv}\ I\ \textbf{do}\ c\}\ Q}$$

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

**Fig. 1.** Core Hoare rules for initialization safety

**Source-Level Safety Certification.** Safety certification demonstrates that the execution of a program does not violate certain conditions, which are formalized as a *safety property*. A *safety policy* is a set of Hoare rules designed to show that safe programs satisfy the safety property of interest [3]. Here, 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.

Figure 1 shows the initialization safety policy, which we will use as our main example here; we omit the rules for functions and procedures. The rules use the usual Hoare triples $P\ \{c\}\ Q$, i.e., if the condition $P$ holds and the command $c$ terminates, then $Q$ holds afterwards. For example, the *assert* rule says that we must first prove that the asserted postcondition $Q'$ implies the arbitrary incoming postcondition $Q$. We then compute the $P$ as *weakest precondition* (WPC) of $c$ for $Q'$ and show that the asserted precondition $P'$ implies $P$. The asserted precondition $P'$ is then passed on as WPC; note that $P$ is only WPC of the "plain" statement $c$, but not of the annotated statement.

Initialization safety ensures that each variable or individual array element has been explicitly assigned a value before it is used. It uses a "shadow" environment where each shadow variable $x_{\text{init}}$ contains the value $\textsc{init}$ after the corresponding variable $x$ has been assigned a value; shadow arrays capture the status of the individual array elements. All statements accessing lvars affect the value of a shadow variable, and each corresponding rule (the *assign*-, *update*-, and *for* rules) is responsible for updating the shadow environment accordingly. The rules also add the appropriate *safety predicates* $\mathit{safe}_{\text{init}}(e)$ for all immediate subexpressions $e$ of the statements. Since an expression is defined to be safe if all corresponding shadow variables have the value $\textsc{init}$, $\mathit{safe}_{\text{init}}(x\,[\,i\,]\,)$ for example translates to $i_{\text{init}} = \textsc{init} \land x_{\text{init}}[i] = \textsc{init}$. Safety certification then computes the WPC for the safety requirements on the output variables. The WPC contains all applied safety predicates and safety substitutions. If the program is safe then the WPC will follow from the assumptions, and all VCs will be provable. Rules for other policies can be given by modifying the shadow variables and safety predicate.

**Annotation Construction.** Hoare-style program verification requires logical annotations, in particular loop invariants. In our application, we use the code generator to provide them together with the code [4,6]. The generator first produces core annotations that focus on on locally relevant aspects, without describing all the global information that may later be necessary for the proofs. A propagation step then pushes the core annotations along the edges of the control flow graph. This ensures that all loops have the required invariant; typically, however, they consist mainly of assertions propagated from elsewhere in the program. Figure 2 shows an example code fragment with annotations. The VCG then processes the code after propagation.

Human-readable explanations provide insight into the VCs. For us, this is particularly important because the underlying annotations have been derived automatically: the explanations help us to gain confidence into the (large and complex) generator and the certifier, and thus into the generated code. 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.

## 3 Explaining the Purpose and Structure of VCs

After simplification, the VCs usually have a form that is reminiscent of Horn clauses (i.e., $H_1 \wedge \ldots \wedge H_n \Rightarrow C$). Here, the unique conclusion $C$ of the VC can be considered its *purpose*. However, for a meaningful explanation of the *structure*, we need a more detailed characterization of the sub-formulas. This information cannot be recovered from the VCs or the code but must be specified explicitly. The 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. Here we first show generated example explanations, and then explain the underlying machinery. Section 4 shows more refined explanations for our running example.

### 3.1 Simple Structural Explanations

Figure 2 shows a fragment of a Kalman filter algorithm with Bierman updates that has been generated by AUTOFILTER from a simplified model of the Crew Exploration Vehicle (CEV) dynamics; the entire program comprises about 800 lines of code. The program initializes some of the vectors and matrices (such as `h` and `r`) with model-specific values before they are used and potentially updated in the main while-loop. It also uses two additional matrices `u` and `d` that are repeatedly zeroed out and then partially recomputed before they are used in each iteration of the main loop (lines 728–731). We will focus on these nested for-loops.

For initialization safety the annotations need to formalize that each of the vectors and matrices is fully initialized after the respective code blocks. For the loops initializing `u` and `d`, invariants formalizing their partial initialization are required to prove that the postcondition holds. However, since these loops precede the use of vectors and matrices initialized outside the main loop, the invariants become cluttered with propagated annotations that are required to discharge the safety conditions from the later uses.

```
    ...
  5 const M=6, N=12;
    ...
    <init h>
183 post $\forall\, 0 \le i < M, 0 \le j < N \cdot h_{\text{init}}[i,j] = \text{INIT}$
    ...
    <init r>
525 post $\forall\, 0 \le i, j < M \cdot r_{\text{init}}[i,j] = \text{INIT}$
      ...
683 while t<Tmax
      inv $\forall\, 0 \le i < M, 0 \le j < N \cdot h_{\text{init}}[i,j] = \text{INIT} \wedge \ldots \wedge \forall\, 0 \le i, j < M \cdot r_{\text{init}}[i,j] = \text{INIT} \wedge \ldots$ do
      ...
728   for k:=0 to N-1
        inv $\forall\, 0 \le i < M, 0 \le j < N \cdot h_{\text{init}}[i,j] = \text{INIT} \wedge \ldots \wedge \forall\, 0 \le i, j < M \cdot r_{\text{init}}[i,j] = \text{INIT} \wedge \ldots$
          $\wedge \forall\, 0 \le i, j < N \cdot i < k \Rightarrow u_{\text{init}}[i,j] = \text{INIT} \wedge d_{\text{init}}[i,j] = \text{INIT}$ do
729     for l:=0 to N-1
          inv $\forall\, 0 \le i < M, 0 \le j < N \cdot h_{\text{init}}[i,j] = \text{INIT} \wedge \ldots \wedge \forall\, 0 \le i, j < M \cdot r_{\text{init}}[i,j] = \text{INIT} \wedge \ldots$
            $\wedge \forall\, 0 \le i, j < N \cdot (i < k \vee i = k \wedge j < l) \Rightarrow u_{\text{init}}[i,j] = \text{INIT} \wedge d_{\text{init}}[i,j] = \text{INIT}$ do
730       u[k,l]:=0;
731       d[k,l]:=0;
        post $\forall\, 0 \le i < M, 0 \le j < N \cdot h_{\text{init}}[i,j] = \text{INIT} \wedge \ldots \wedge \forall\, 0 \le i, j < M \cdot r_{\text{init}}[i,j] = \text{INIT} \wedge \ldots$
          $\wedge \forall\, 0 \le i, j < N \cdot i \le k \Rightarrow u_{\text{init}}[i,j] = \text{INIT} \wedge d_{\text{init}}[i,j] = \text{INIT}$
      post $\forall\, 0 \le i < M, 0 \le j < N \cdot h_{\text{init}}[i,j] = \text{INIT} \wedge \ldots \wedge \forall\, 0 \le i, j < M \cdot r_{\text{init}}[i,j] = \text{INIT} \wedge \ldots$
        $\wedge \forall\, 0 \le i, j < N \cdot u_{\text{init}}[i,j] = \text{INIT} \wedge d_{\text{init}}[i,j] = \text{INIT}$
      ...
      <use u, d>
      ...
      <use h,..., r>
      ...
    end;
```

**Fig. 2.** Example code fragment and annotations generated by AutoFilter

The certification of the program generates 71 VCs; 12 of these are related to the loop at lines 728–731. This shows that location information alone is insufficient as a basis for explaining VCs. Here, we focus on one VC

$$
\begin{array}{l}
0 \le k \le 11 \wedge 0 \le l \le 11 \wedge \forall\, 0 \le i,j < 12 \cdot h_{\text{init}}[i,j] = \text{INIT} \\
\;\wedge \ldots \wedge \\
\forall\, 0 \le i < 6, 0 \le j < 12 \cdot r_{\text{init}}[i,j] = \text{INIT} \wedge \\
\forall\, 0 \le i,j < 12 \cdot i < k \Rightarrow d_{\text{init}}[i,j] = \text{INIT} \;\wedge\; \forall\, 0 \le i,j < 12 \cdot i = k \wedge j < l \Rightarrow d_{\text{init}}[i,j] = \text{INIT} \;\wedge \\
\forall\, 0 \le i,j < 12 \cdot i < k \Rightarrow u_{\text{init}}[i,j] = \text{INIT} \;\wedge\; \forall\, 0 \le i,j < 12 \cdot i = k \wedge j < l \Rightarrow u_{\text{init}}[i,j] = \text{INIT} \\
\Rightarrow \forall\, 0 \le i,j < 12 \cdot (i = k \wedge j \le l \wedge j \ne l) \Rightarrow u_{\text{init}}[i,j] = \text{INIT}
\end{array}
$$

that emerges from showing that the invariant is preserved through one iteration of the inner loop. The full VC is substantially larger (approx. 180 lines) and contains many irrelevant hypotheses, which make it hard for a human to grasp. In fact, the sheer amount of irrelevant information is often the biggest hurdle to understand automatically generated VCs. However, if we (manually) interpret the formula, we can see that the hypotheses are either constraints that originate from the loop bounds ($0 \le k, l \le 11$), post-conditions that have originally been established before the loop and then been propagated into the invariant (e.g., $\forall\, 0 \le i,j < 12 \cdot h_{\text{init}}[i,j] = \text{INIT}$), or the actual "local" invariant as hypotheses. The conclusion comprises parts of the invariant (where $l$ has been replaced by $l + 1$), but due to simplification this is difficult to see. In addition, all constants have been replaced by their values. This post hoc analysis of the VC, however, is not possible automatically. Simplification can change the VC structure arbitrarily, and even without simplification two subformulas can look the same but have different meaning (cf. the different occurrences of the loop invariant in the *while* rule.) In our approach, the VC is marked up with labels that represent this information in order to generate the explanation shown below; multiple annotations at the same source line are marked with #-signs. Note that the generated explanation also spells out the verification context, which is the VCs "secondary" purpose.

> The purpose of this VC is to show that the loop invariant at line 729 (#1) under the substitutions originating in line 5 and line 730 is still true after each iteration to line 731; it is also used to show the preservation of the loop invariants at line 728, which in turn is used to show the preservation of the loop invariants at line 683. Hence, given
> - the loop bounds at line 728 under the substitution originating in line 5,
> - the invariant at line 729 (#1) under the substitution originating in line 5,
> - the invariant at line 729 (#2) under the substitution originating in line 5,
>
>   ...
> - the invariant at line 729 (#15) under the substitution originating in line 5,
> - the loop bounds at line 729 under the substitution originating in line 5,
>
> show that the loop invariant at line 729 (#1) under the substitutions originating in line 5 and line 730 is still true after each iteration to line 731.

## 3.2 Mark-Up Structure

**Concepts.** The basic information for explanation generation is a set of underlying concepts, which depends of course on the particular aspect of the VCs to be explained. In the case of the structural explanations, most concepts characterize a proposition either as hypotheses or conclusions, reflecting their eventual position in the VC. Other concepts capture information about origin and secondary purpose of the propositions.

*Hypotheses* consist of assertions and control flow predicates. *Assertions* refer to subformulas that occur as annotations in the program, either originally or after propagation. They include asserted pre- and post-conditions (labels ass_pre and ass_post), function pre- and post-conditions (ass_fpre and ass_fpost), and loop invariants. Since the loop rules use the loop invariant as hypothesis in two different positions and instantiations, we distinguish ass_inv and ass_inv_exit (Figure 3). *Control flow predicates* refer to subformulas that reflect the program's control flow. For both *if*-statements and *while*-loops, the control flow predicates occur in positive and negated forms, giving four different concepts: if_tt, if_ff, while_tt, and while_ff. 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, which includes establishing (i.e., showing to hold at the given location) the different types of assertions. As in the case of the hypotheses, invariants are used in two different forms, the *entry form* (or base case) est_inv and the *step form* est_inv_iter. Note that an assertion can be used both as hypothesis and as conclusion, even in the same VC. The explanations distinguish these two bits of information from the same source. For safety verification, we additionally have the safety conditions safety that have to be demonstrated.

*Qualifiers* further characterize both hypotheses and conclusions by recording the origin of a sub-formula. The different *substitution* concepts reflect the substitutions of the underlying Hoare calculus. The concepts sub and upd capture the origin and effect of assignments and array updates on the form of the resulting VCs; for the shadow environment (Section 2), we additionally get *safety substitutions* sub_safety and upd_safety.

*Contributors* 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, contributors 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 1) contribute to showing the preservation of the invariant $I$ over the loop body $c$, independent of their primary purpose, and are thus labeled with pres_inv.

**Label Structure and Labeled Terms.** We use labeled terms $\ulcorner t \urcorner^{\mathsf{l}}$, where each term $t$ can be adorned with a label $\mathsf{l}$, or, by abuse of notation, a list of labels. Labels have the form $c(o, n)$. Here $c$ is one of the concepts introduced above; it describes the role the labeled term plays and thus determines how it is rendered. The location $o$ records where it originated; it refers either to an individual position or to a range. We use file names and line numbers for locations. $n$ is a list of labels that contains further qualifying information. Initially, $n$ is empty; after normalization, it holds labels that have been "bubbled-up" from subterms. In our running example, the loop bounds are originally represented as $\ulcorner 0 \leq k \leq \ulcorner 11 \urcorner^{\mathsf{sub}(5,\langle\rangle)} \urcorner^{\mathsf{bounds}(728,\langle\rangle)}$, i.e., with the label on the upper bound reflecting the source of the substitution, and the label on the sub-formula reflecting its role. After simplification, the sub-label is nested inside the bounds-label, reflecting the original nesting in the term: $\ulcorner 0 \leq k \leq 11 \urcorner^{\mathsf{bounds}(728,\langle \mathsf{sub}(5,\langle\rangle)\rangle)}$.

## 3.3 Modified Hoare 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 WPC, 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 are added in three places: to the "incoming" postcondition of a recursive VCG call in the premise of an inference rule, to the WPC, or to a generated VC. Figure 3 shows the core rules of the initialization safety policy marked-up for explaining the structural aspect of VCs. 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.

The *assign* and *update* rules only require mark-up in the WPC. 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. The substitutions need mark-up to record their type and the origin of the substituted expressions. By labeling only the expressions and not the variables we can use the normal substitution mechanisms.

$$(\mathit{assign})\quad \frac{}{Q[\ulcorner e\urcorner^{\mathsf{sub}}/x,\ \ulcorner \text{INIT}\urcorner^{\mathsf{sub\_safety}}/x_{\text{init}}] \wedge \ulcorner \mathit{safe}_{\text{init}}(e)\urcorner^{\mathsf{safety}}\ \{x\ := e\}\ Q}$$

$$(\mathit{update})\quad \frac{}{\begin{pmatrix} Q[\ulcorner \mathit{upd}(x, e_1, e_2)\urcorner^{\mathsf{upd}}/x,\ \ulcorner \mathit{upd}(x_{\text{init}}, e_1, \text{INIT})\urcorner^{\mathsf{upd\_safety}}/x_{\text{init}}]\wedge \\ \ulcorner \mathit{safe}_{\text{init}}(e_1)\urcorner^{\mathsf{safety}} \wedge \ulcorner \mathit{safe}_{\text{init}}(e_2)\urcorner^{\mathsf{safety}} \end{pmatrix}\ \{x\,[e_1]\ := e_2\}\ Q}$$

$$(\mathit{if})\quad \frac{P_1\ \{c_1\}\ Q \quad P_2\ \{c_2\}\ Q}{(\ulcorner b\urcorner^{\mathsf{if\_tt}} \Rightarrow P_1) \wedge (\ulcorner \neg b\urcorner^{\mathsf{if\_ff}} \Rightarrow P_2) \wedge \ulcorner \mathit{safe}_{\text{init}}(b)\urcorner^{\mathsf{safety}}\ \{\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\}\ Q}$$

$$(\mathit{while})\quad \frac{\ulcorner P\ \{c\}\ \ulcorner I\urcorner^{\mathsf{est\_inv\_iter}}\urcorner^{\mathsf{pres\_inv}} \quad \begin{matrix} \ulcorner\ulcorner I\urcorner^{\mathsf{ass\_inv}} \wedge \ulcorner b\urcorner^{\mathsf{while\_tt}} \Rightarrow P\urcorner^{\mathsf{pres\_inv}} \\ \ulcorner I\urcorner^{\mathsf{ass\_inv\_exit}} \wedge \ulcorner \neg b\urcorner^{\mathsf{while\_ff}} \Rightarrow Q \end{matrix}}{\ulcorner I\urcorner^{\mathsf{est\_inv}} \wedge \ulcorner \mathit{safe}_{\text{init}}(b)\urcorner^{\mathsf{safety}}\ \{\textbf{while}\ b\ \textbf{inv}\ I\ \textbf{do}\ c\}\ Q}$$

$$(\mathit{for})\quad \frac{\ulcorner P\ \{c\}\ \ulcorner I[i+1/i]\urcorner^{\mathsf{est\_inv\_iter}}\urcorner^{\mathsf{pres\_inv}} \quad \begin{matrix} \ulcorner\ulcorner I\urcorner^{\mathsf{ass\_inv}} \wedge \ulcorner e_1 \le i \le e_2\urcorner^{\mathsf{bounds}} \Rightarrow P\urcorner^{\mathsf{pres\_inv}} \\ \ulcorner I[e_2+1/i,\ \text{INIT}/i_{\text{init}}]\urcorner^{\mathsf{ass\_inv\_exit}} \Rightarrow Q \end{matrix}}{\begin{pmatrix} e_1 \le e_2 \wedge \ulcorner I[e_1/i]\urcorner^{\mathsf{est\_inv}}\wedge \\ \ulcorner \mathit{safe}_{\text{init}}(e_1)\urcorner^{\mathsf{safety}} \wedge \ulcorner \mathit{safe}_{\text{init}}(e_2)\urcorner^{\mathsf{safety}} \end{pmatrix}\ \{\textbf{for}\ i\ := e_1\ \textbf{to}\ e_2\ \textbf{inv}\ I\ \textbf{do}\ c\}\ Q}$$

$$(\mathit{assert})\quad \frac{\ulcorner P'\urcorner^{\mathsf{ass\_pre}} \Rightarrow P \quad P\ \{c\}\ \ulcorner Q'\urcorner^{\mathsf{est\_post}} \quad \ulcorner Q'\urcorner^{\mathsf{ass\_post}} \Rightarrow Q}{\ulcorner P'\urcorner^{\mathsf{est\_pre}}\ \{\textbf{pre}\ P'\ c\ \textbf{post}\ Q'\}\ Q}$$

**Fig. 3.** Hoare rules for initialization safety with semantic markup

While labeling the *if* rule is straightforward, the loop rules are more complicated; we focus on the *while* rule but the *for* rule has a similar structure. The WPC comprises the safety predicate, which is labeled as before, and the invariant, which has to be established for loop entry and is thus labeled with 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 (i.e., re-establish the invariant after one loop iteration) for the recursive call; moreover, all emerging VCs must be marked up with the secondary purpose 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.

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

## 3.4 Labeled Rewriting

The VCs (whether labeled or unlabeled) become quite complex and need to be simplified aggressively before they can be proven by an ATP. Unfortunately, unlabeled simplification 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. The purpose of the labeled simplification rules 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, based on the assumption that the majority of the VCs can be rewritten to *true*.

The rules themselves fall into five different groups. The first group contains rules such as $\ulcorner \mathit{true} \urcorner^{\mathsf{l}} \rightarrow \mathit{true}$ or $P \Rightarrow P' \rightarrow \mathit{true}$ if $\mid P \mid = \mid P' \mid$ that remove labels from trivially true (sub-) formulas because these require no explanations;[1] The next group consists of rules such as $\ulcorner \mathit{false} \urcorner^{\mathsf{l}} \vee P \rightarrow P$ that *selectively* remove trivially false labeled sub-formulas. The remaining context then provides the information for the explanations. However, the labels obviously need to be retained if the underlying unlabeled rule version rewrite the *entire* formula into *false*, since there is no remaining context to explain the failure, e.g., $\ulcorner \mathit{false} \urcorner^{\mathsf{l}} \wedge P \rightarrow \ulcorner \mathit{false} \urcorner^{\mathsf{l}}$. The rules $\ulcorner P \wedge Q \urcorner^{\mathsf{l}} \rightarrow \ulcorner P \urcorner^{\mathsf{l}} \wedge \ulcorner Q \urcorner^{\mathsf{l}}$ and $P \Rightarrow \ulcorner Q \Rightarrow R \urcorner^{\mathsf{l}} \rightarrow P \wedge \ulcorner Q \urcorner^{\mathsf{l}} \Rightarrow \ulcorner R \urcorner^{\mathsf{l}}$ comprise the fourth group; they distribute labels over conjunction and (nested) implication, respectively, so that the label scopes are minimized in the final simplified VCs. The last group encodes knowledge about how the labels will be interpreted in the underlying domain. For example, the rule $\mathit{sel}(\ulcorner \mathit{upd}(x, i_1, t) \urcorner^{\mathsf{l}}, i_2) \rightarrow \ulcorner i_1 = i_2 \urcorner^{\mathsf{l}} \,?\, \ulcorner t \urcorner^{\mathsf{l}} : \mathit{sel}(x, i_2)$ 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. This group also contains an unnesting rule $\ulcorner \ulcorner t \urcorner^{\mathsf{m}} \urcorner^{\mathsf{n}} \rightarrow \ulcorner t \urcorner^{\mathsf{n} \otimes \mathsf{m}}$ that "bubbles" nested labels to the top term, and so enables other labeled and unlabeled rules to apply, but keeps the nesting structure on the labels itself. This ensures that qualifiers remain nested properly, and apply to the originally qualified term.

## 3.5 Rendering

The final stage is generation of the actual explanations, i.e., turning the (labeled) VCs into human-readable text, is called *rendering*. The underlying structure and actual textual representation of the explanations can be specified as a grammar (omitted here), where the right-hand side of each rule is an *explanation template* that is similar to a format string in C. These templates allow an easy customization and fine-grained control of the textual explanations. The renderer contains code to interpret the templates 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 substitutions and the sim- and nested-labels. Rendering comprises four steps: $(i)$ VC normalization, using the labeled rewrite system; $(ii)$ label extraction, using the unnesting rule; $(iii)$ label normalization,

[1] We use an auxiliary function $\mid \cdot \mid$ to remove labels from terms and the label composition operator $\otimes$ to append a list of inner labels to the list of labels nested in the outer label, i.e., $c(o, l) \otimes m = c(o, l \bullet m)$, where $\bullet$ is list concatenation.

to fit the labels to the explanation templates; $(iv)$ text generation, using the explanation templates. The third step flattens nested qualifiers, so that for example the label sub($p$, sub($q$, sub($r$))) is rewritten into the list $\langle$sub($p$), sub($q$), sub($r$)$\rangle$. It also merges back together conclusions from the same line which have been split over different literals during the first step.

## 3.6 Putting It All Together

Our example VC emerges from the first hypothesis (i.e., $I \wedge e_1 \leq i \leq e_2 \Rightarrow P$) of the *for* rule. $P$ is computed as the WPC of the two assignments in lines 730 and 731 with respect to the appropriately labeled step form of the invariant at line 729:

$$\ulcorner(\forall\, 0 \leq i,j < N \cdot i = k \wedge j \leq l+1 \Rightarrow (\ulcorner upd(u_{\text{init}}, [k,l], \text{INIT})\urcorner^{\mathsf{upd\_safety(728)}})[i,j] = \text{INIT}) \wedge \ldots \urcorner^{\mathsf{est\_inv\_iter(729\text{–}731)}}$$

Here, the *update* rule added the upd_safety-label, while the substitution of $N$ by 12 will eventually introduce a sub-label. Since all this happens as part of handling the enclosing for- and while-loops, $P$ will be wrapped into two corresponding pres_inv-labels.

Simplification splits the implication into several independent VCs, including the example, and "bubbles" all labels to the top. The renderer then strips away the enclosing contributors (i.e., the pres_inv-labels) and uses the user-defined templates to convert them into the text shown in Section 3.1. It will then search the remaining label list for the unique conclusion (here est_inv_iter) to produce a caption from the corresponding template and the contributor text, before it renders the assumptions.

## 3.7 Local Assumptions and Simultaneous Conclusions

All VCs generated in the example above have a unique conclusion that denotes their primary purpose. However, for VCs that contain existential quantifiers (introduced by the annotations or by the rule for procedure calls), this is not necessarily the case any longer. Hence, we must explicitly represent and render multiple conclusions that have to be satisfied simultaneously for an existentially quantified witness, and conclusions from local assumptions. Consider, for example, the following VC, that arises in certifying frame safety (i.e., consistent use of coordinate frames [14]) in navigation software generated by Real-Time Workshop from a Simulink model:

$$\begin{aligned}
&\ldots \wedge \mathrm{lo}(T) = 0 \wedge \mathrm{hi}(T) = 8 \wedge T[0]+T[4]+T[8] > 0 \wedge \mathrm{frame}(T, \mathrm{dcm}(\mathrm{eci}, \mathrm{ned}))\\
&\Rightarrow \forall q_0 : \mathrm{real}, v : \mathrm{vec} \cdot \exists d : \mathrm{DCM} \cdot\\
&\qquad \mathrm{tr(d)} = T[0]+T[4]+T[8] \wedge \mathrm{tr(d)} > 0 \wedge \mathrm{rep\_dcm}(d, T[5], T[7], T[2], T[6], T[1], T[3])\\
&\qquad \wedge(\exists q : \mathrm{quat} \cdot \mathrm{eq\_dcm\_quat}(d, q) \wedge \mathrm{rep\_quat}(q, q_0, v[0], v[1], v[2])\\
&\qquad\qquad \Rightarrow \mathrm{frame}(\mathrm{vupd}(\mathrm{upd}(M, 0, q_0), 1, 3, v), \mathrm{quat}(\mathrm{eci}, \mathrm{ned})))
\end{aligned}$$

The purpose of this VC is to show the correctness of a procedure call. Hence, we need to show for each argument (i.e., $q_0$ and $v$) the existence of a direction cosine matrix $d$ such that the function's three preconditions are satisfied and that the function postcondition implies the required postcondition. Our system explains this as follows:

. . . Hence, given
- the precondition at line 794 (#1),
- the condition at line 798 under the substitution originating in line 794,

show that there exists a DCM that will simultaneously
- establish the function precondition for the call at line 799 (#1),
- establish the function precondition for the call at line 799 (#2),
- establish the function precondition for the call at line 799 (#3) under the substitution originating in line 794,
- establish the postcondition at line 815 (#1) assuming the function postcondition for the call at line 799 (#1).

Note that the structure of the explanation reflects the VC's logical structure, and shows which goals have to be established simultaneously, and that the function postcondition can only be used as assumption to establish the call-time postcondition, but of course not the function's preconditions. These labels do not give a detailed explanation of the VC's individual parts (e.g., the function postcondition); for that, we would need to mark up the annotations with additional policy-specific details (see Section 4.3).

We only need to introduce two additional conclusion labels local and sim to represent local assumptions and simultaneous conclusions, as outlined above. In addition, we need simplification rules that introduce these labels to properly maintain the VC structure in the explanations, e.g., $\exists x : t \cdot \ulcorner P \urcorner^{\mathsf{l}} \Rightarrow \ulcorner Q \urcorner^{\mathsf{m}} \rightarrow \ulcorner \exists x : t \cdot P \Rightarrow Q \urcorner^{\mathsf{local}(\langle \mathsf{l}, \mathsf{m} \rangle)}$.

## 4 Refined Explanations

Even though the explanations constructed so far relate primarily to the structure of the VCs, they already provide some "semantic flavor", since they distinguish the multiple roles a single annotation can take. However, for structurally complex programs, the labels do not yet convey enough information to allow users to understand the VCs in detail. For example, a double-nested for-loop can produce a variety of VCs that will all refer to "the invariant", without further explaining whether it is the invariant of the inner or the outer loop, leaving the user to trace through the exact program locations to resolve this ambiguity. We can produce refined explanations that verbalize such conceptual distinctions by introducing additional qualification labels that are wrapped inside the existing structural labels. We chose this solution over extending the structural labels because it allows us to handle orthogonal aspects independently, and makes it easier to treat the extensions uniformly in different contexts.

### 4.1 Adding Index Information to Loop Explanations

Explanations of VCs emerging from for-loops are easier to understand if they are tied closer to the program by adding more detailed information about the index variables and bounds; our running example then becomes (cf. Section 3.1; emphasis added manually):

The purpose of this VC is to show that the loop invariant at line 729 (#1) under the substitutions originating in line 5 and line 730 is still true after each iteration to line 731 *(i.e., in the form with l+1 replacing l)*; it is also used to show the preservation of the loop invariants at line 728, which in turn . . .

Note that the way the qualifier is rendered depends on the particular loop concept that it qualifies, to properly reflect the different substitutions that are applied to the invariant in the different cases (see Figure 3); for example, in the step case (i.e., for the est_inv_iter-label), the variable is used, while in the base case, the lower bound is used; the qualifier is ignored when the invariant is used as asserted hypothesis (i.e., for the ass_inv-label).

The information required for all different cases (i.e., variable name, lower and upper bounds) is almost impossible to recreate with a post hoc analysis of the formula. The VCG can easily extract this from the index of the for-loop itself and add it as qualifier to the different labels used in the *for*-rule, changing for example the label added to the invariant in the base case to est_inv($\langle i := e_1$ **to** $e_2\rangle$) (cf. Figure 3). Since the labeled simplification rules ensure that the qualifiers are never moved outside their base label, the explanation templates for the qualifiers simply need to take the base label as an additional argument to produce the right text, for example `render`(est_inv_iter, $i := e_1$ **to** $e_2$) = "in the form with " • i • "+1 replacing " • i.

## 4.2 Adding Relative Positions to Loop Explanations

VCs emerging from nested loops refer to the underlying loops via their absolute source locations but since these are often very close to each other, they can easily be confused. We can thus further improve the explanations by adding information about the relative loop ordering, distinguishing, for example, the inner from the outer invariant. In conjunction with the the syntactic index information described above, our running example then becomes (emphasis added manually):

> The purpose of this VC is to show that the loop invariant at line 729 (#1) *(i.e., the inner invariant)* under the substitutions originating in line 5 and line 730 is still true after each iteration to line 731 (i.e., in the form with *l*+1 replacing *l*); it is also used to show the preservation of the loop invariants at line 728, which in turn ...

Since the VCG has no built-in notion of "outer" and "inner" loops, it cannot add the respective qualifiers automatically. Instead, the annotations in the program must be labeled accordingly, either by the programmer, or, in our case, the annotation generator. No further changes are required to the machinery: the VCG simply processes the labeled annotations, and the outer- and inner-qualifiers are rendered by parameterized templates as before.

## 4.3 Adding Domain-Specific Semantic Explanations

We can construct semantically "richer" explanations if we further expand the idea outlined in the previous section, and add more *semantic labels* to the annotations, which represent domain-specific interpretations of the labeled sub-formulae. For example, in initialization safety the VCs usually contain sub-formulae of the form $\forall 0 \leq i, j < N \cdot A_{\text{init}}[i, j] = \text{INIT}$, which expresses the fact that the array $A$ is fully initialized (e.g, most postconditions in Figure 2). By labeling this formula, or more precisely, the annotation from which it is taken, we can produce an appropriate explanation without any need to analyze the formula structure: [2]

[2] Note that the formulae expressing the domain-specific concepts can become arbitrarily complex, and make any post hoc analysis practically infeasible. For example, to express the row-major, partial initialization of an array up to position $(k, l)$, we would already need to identify a formula equivalent to $\forall 0 \leq i, j < N \cdot (i < k \vee i = k \wedge j < l) \Rightarrow A_{\text{init}}[i, j] = \text{INIT}$.

. . . Hence, given
- the loop bounds at line 728 under the substitution originating in line 5,
- the invariant at line 729 (#1) (i.e., the array h is fully initialized, which is established at line 183) under the substitution originating in line 5,

  . . .
- the invariant at line 729 (#11) (i.e., the array r is fully initialized, which is established at line 525) under the substitution originating in line 5,

  . . .
- the invariant at line 729 (#15) under the substitution originating in line 5,
- the loop bounds at line 729 under the substitution originating in line 5,

show that the loop invariant at line 729 (#1) under the substitutions originating in line 5 and line 730 is still true after each iteration to line 731 (i.e., the array u is initialized up to position (k,l).

For this extension, we need two different qualifiers, init(a, o) which states that the array $a$ is fully initialized after line $o$, and init_upto(a, k, l) which states that $a$ is initialized in a row-major fashion up to position $(k, l)$. Again, the annotation generator can add these labels to the annotations in the program.

We can use the domain-specific information to give a semantic explanation of the hierarchical relations between the VCs which complements the purely structural view provided by the pres_inv labels. We thus generalize the *assert* rule to use the domain-specific labels as contributors:

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

The *label* rule "plucks" the label off the post-condition and passes it into the appropriate positions. The labels need to be modified to take the domain-specific labels as an additional argument. For example, ass_post(init(183, h)) refers to the postcondition asserted after the statements that initialize the array $h$. In addition, we also introduce a new contribution label (e.g., contrib(init(183, h)), similar to the invariant preservation in the structural concept hierarchy. This is added to the WPC that is recursively computed for the annotated statement, and to all VCs emerging during that process (e.g., if the initialization uses a nested loop and thus generates multiple VCs). These more refined labels let the renderer determine whether a VC actually establishes the asserted post-condition of a domain-specific block, or whether it is just an individual contributor to this.

## 5 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 [11] describes a system that supports a "deep linking" to detailed term positions. JACK [1] and Perfect Developer [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) or secondary purpose.

Our work grew out of the earlier work by Denney and Venkatesan [8] which used information from a particular subset of VCs (in the current terminology: where the purpose is to establish a safety condition) in order to give a textual account for why the code is safe. It soon became clear, however, that a full understanding of the certification process requires the VCs themselves to be explained (as does any debugging of failed VCs). The current work extends the explanations to arbitrarily constructed formulas, that is, VCs where the labels on constituent parts come from different sources. This allows formulas to be interpreted in different ways.

Leino et al. [12] use explanations for traces to safety conditions. This is sufficient for debugging programs, which is their main motivation. Like our work, Leino's approach is based on extending an underlying logic with labels to represent explanatory semantic information. Both approaches use essentially the same types of structural labels, and Leino's use of two different polarities (lblpos and lblneg) corresponds to our distinction between asserting and establishing an annotation. However, their system does not represent the origin of substitutions nor the secondary purpose of the VCs. Similarly, it does not incorporate refined explanations with additional information. Moreover, the approaches 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 mainly use the VCG to insert the labels, which allows us to take advantage of domain-specific labels. While our simplifier needs to be label-aware, we strip labels off the final VCs after the explanation has been constructed, and thus do not place special requirements on the ATP like they do. This allows us to use off-the-shelf high-performance ATPs.

## 6 Conclusions and Future Work

The explanation mechanism which we have described here has been successfully implemented and incorporated into our certification browser [5,7]. 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 AUTOFILTER, AUTOBAYES, and Real-Time Workshop for a range of safety policies. Complexity of VCs is largely independent of the size of the program, but we have applied our tool up to the subsystem level (around 1000 lines of code), where the largest VCs are typically around 200 lines of formula.

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 [13]: 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. 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. More generally, we would like to allow explanations to be based on entirely different explanation structures or *ontologies*.

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 3 and then by induction over the labeled rewrite rules. 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.

**Acknowledgments.** This material is based upon work supported by NASA under awards NCC2-1426 and NNA07BB97C.

## References

1. Barthe, G., et al.: JACK – a tool for validation of security and behaviour of Java applications. In: Formal Methods for Components and Objects. LNCS, vol. 4709, pp. 152–174. Springer, Heidelberg (2007)
2. Crocker, D.: Perfect Developer: a tool for object-oriented formal specification and refinement. In: Tool Exhibition Notes, FM 2003, pp. 37–41 (2003)
3. Denney, E., Fischer, B.: Correctness of source-level safety policies. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 894–913. Springer, Heidelberg (2003)
4. Denney, E., Fischer, B.: Certifiable program generation. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 17–28. Springer, Heidelberg (2005)
5. Denney, E., Fischer, B.: A program certification assistant based on fully automated theorem provers. In: Proc. Intl. Workshop on User Interfaces for Theorem Provers (UITP 2005), Edinburgh, pp. 98–116 (2005)
6. Denney, E., Fischer, B.: A generic annotation inference algorithm for the safety certification of automatically generated code. In: GPCE 2006, pp. 121–130. ACM Press, New York (2006)
7. Denney, E., Trac, S.: A software safety certification tool for automatically generated guidance, navigation and control code. In: IEEE Aerospace Conference Electronic Proceedings, IEEE, Big Sky (2008)
8. Denney, E., Venkatesan, R.P.: A generic software safety document generator. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 102–116. Springer, Heidelberg (2004)
9. Fiedler, A.: Natural language proof explanation. In: Hutter, D., Stephan, W. (eds.) Mechanizing Mathematical Reasoning. LNCS (LNAI), vol. 2605, pp. 342–363. Springer, Heidelberg (2005)
10. Fischer, B., Schumann, J.: AutoBayes: A system for generating data analysis programs from statistical models. J. Functional Programming 13(3), 483–508 (2003)
11. Fraer, R.: Tracing the origins of verification conditions. In: Nivat, M., Wirsing, M. (eds.) AMAST 1996. LNCS, vol. 1101, pp. 241–255. Springer, Heidelberg (1996)
12. Leino, K.R.M., Millstein, T., Saxe, J.B.: Generating error traces from verification-condition counterexamples. Science of Computer Programming 55(1–3), 209–226 (2005)
13. Sutcliffe, G., Denney, E., Fischer, B.: Practical proof checking for program certification. In: Proc. CADE-20 Workshop on Empirically Successful Classical Automated Reasoning (ESCAR 2005), Tallinn (July 2005)
14. Vallado, D.A.: Fundamentals of Astrodynamics and Applications, 2nd edn. Space Technology Library. Microcosm Press and Kluwer Academic Publishers (2001)
15. Whittle, J., Schumann, J.: Automating the implementation of Kalman filter algorithms. ACM Trans. Mathematical Software 30(4), 434–453 (2004)