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. 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 labelings that explain 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, software certification, traceability, code generation, Matlab.

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, 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, the domain theory may be incomplete, and 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 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 structured *labels* that are attached to the meta-variables used in the Hoare rules, so that the VCG 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, although this is still not common with other static analysis techniques. 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 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 using labels to be significant. 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 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. Consequently, explanation generation 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 proof. Hence, we maintain, and can also introduce, labels during simplification, but strip them off before proving the VCs. Although there are techniques for explaining proofs (e.g., [9]), this would not provide additional insight, and in fact would be less useful for our purposes since the key information is expressed in the annotations and VCs.

We developed our technique to support a certifiable code generator, which 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, the core of our technique is not tied to either code generation or safety certification and can be used in any Hoare-style verification context.

2 Logical Background

Hoare Logic and Program Verification We follow the usual Hoare-style program verification approach (see [13] for more details), in which the verification problem is solved in two separate stages: 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 (given suitable annotations) construction of the VCs from their undecidable discharge, but it has the disadvantage that 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 [18] 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.

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

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

$$(\textit{if})\ \frac{P_1\ \{c_1\}\ Q \quad P_2\ \{c_2\}\ Q}{(b \Rightarrow P_1) \wedge (\neg b \Rightarrow P_2) \wedge \textit{safe}_{\textsc{init}}(b)\ \{\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\}\ Q}$$

$$(\textit{while})\ \frac{P\ \{c\}\ I \quad I \wedge b \Rightarrow P \quad I \wedge \neg b \Rightarrow Q}{I \wedge \textit{safe}_{\textsc{init}}(b)\ \{\textbf{while}\ b\ \textbf{inv}\ I\ \textbf{do}\ c\}\ Q}$$

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

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

Fig. 1. Core Hoare rules for initialization safety

Source-Level Safety Certification The purpose of safety certification is to demonstrate that a program does not violate during its execution 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. Most approaches to safety certification, in particular proof-carrying code [14], 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.

Figure 1 shows the initialization safety policy, which we will use as our main example here; we omit the rules for functions and procedures, which are not required for the examples. The rules 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. 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 which records safety information related to the program variables. Here, each shadow variable $x_{\textsc{init}}$ contains the value INIT after the corresponding 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, and each corresponding rule (the *assign*-, *update*-, and *for* rules) is responsible for updating the shadow environment accordingly. In addition, all rules also add the appropriate safety predicates $\textit{safe}_{\textsc{init}}(e)$ for all immediate subexpressions e of the statements. Here, the safety property defines an expression to be safe if all corresponding shadow variables have the value INIT, so that $\textit{safe}_{\textsc{init}}(x[i])$ for example translates to $i_{\textsc{init}} = \textsc{init} \wedge x_{\textsc{init}}[i] = \textsc{init}$. Safety certification then starts with the safety requirements on the output variables and computes the weakest safety precondition (WSPC), which contains all applied safety predicates and safety

substitutions. If the program is safe then the WSPC 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 A certifiable code generator [3, 17] derives not only code from high-level specifications but also the detailed annotations required to certify a given safety property. The annotations can either be made part of the templates used in the generator and then instantiated and refined in parallel with the code fragments, or they can be constructed in a post-generation inference phase that exploits the idiomatic structure of automatically generated code [5]. The annotations focus on locally relevant information, without describing all the global information that may later be necessary for the proofs. An annotation propagation step pushes the local annotations along the edges of the control flow graph. The VCG then processes the code after propagation. This ensures that all loops have the required invariant; typically, however, they consist mainly of assertions which have been propagated from elsewhere in the program.

Human-readable explanations provide and communicate 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. 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.

3.1 Explanation Examples

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 double-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

```
...
5 const M=6, N=12;
...
  <init h>
183 post ∀ 0≤i<M, 0≤j<N · h_init[i,j]=INIT
...
  <init r>
525 post ∀ 0≤i,j<M · r_init[i,j]=INIT
    ...
683 while t<Tmax
      inv ∀ 0≤i<M, 0≤j<N · h_init[i,j]=INIT ∧...∧ ∀ 0≤i,j<M · r_init[i,j]=INIT ∧... do
      ...
728   for k:=0 to N-1
        inv ∀ 0≤i<M, 0≤j<N · h_init[i,j]=INIT ∧...∧ ∀ 0≤i,j<M · r_init[i,j]=INIT ∧...
          ∧ ∀ 0≤i,j<N · i < k ⇒ u_init[i,j]=INIT ∧ d_init[i,j]=INIT do
729     for l:=0 to N-1
          inv ∀ 0≤i<M, 0≤j<N · h_init[i,j]=INIT ∧...∧ ∀ 0≤i,j<M · r_init[i,j]=INIT ∧...
            ∧ ∀ 0≤i,j<N · (i < k ∨ i = k ∧ j < l) ⇒ u_init[i,j]=INIT ∧ d_init[i,j]=INIT do
730       u[k,l]:=0;
731       d[k,l]:=0;
        post ∀ 0≤i<M, 0≤j<N · h_init[i,j]=INIT ∧...∧ ∀ 0≤i,j<M · r_init[i,j]=INIT ∧...
          ∧ ∀ 0≤i,j<N · i ≤ k ⇒ u_init[i,j]=INIT ∧ d_init[i,j]=INIT
      post ∀ 0≤i<M, 0≤j<N · h_init[i,j]=INIT ∧...∧ ∀ 0≤i,j<M · r_init[i,j]=INIT ∧...
        ∧ ∀ 0≤i,j<N · u_init[i,j]=INIT ∧ d_init[i,j]=INIT
      ...
      <use u, d>
      ...
      <use h,..., r>
      ...
  end;
```

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

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.

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

$$\begin{array}{l} 0\le k,l\le 11 \wedge \forall\, 0\le i,j<12 \cdot h_{\text{init}}[i,j]=\text{INIT} \wedge \ldots \wedge \forall\, 0\le i<6, 0\le j<12 \cdot r_{\text{init}}[i,j]=\text{INIT} \\ \wedge \forall\, 0\le i,j<12 \cdot i<k \Rightarrow d_{\text{init}}[i,j]=\text{INIT} \wedge \forall\, 0\le i,j<12 \cdot i=k \wedge j<l \Rightarrow d_{\text{init}}[i,j]=\text{INIT} \\ \wedge \forall\, 0\le i,j<12 \cdot i<k \Rightarrow u_{\text{init}}[i,j]=\text{INIT} \wedge \forall\, 0\le i,j<12 \cdot i=k \wedge j<l \Rightarrow u_{\text{init}}[i,j]=\text{INIT} \\ \Rightarrow \forall\, 0\le i,j<12 \cdot (i=k \wedge j\le l \wedge j\ne l) \Rightarrow u_{\text{init}}[i,j]=\text{INIT} \end{array} \quad (1)$$

that emerges from showing that the invariant is preserved through one iteration of the inner loop. Note that the full VC is substantially larger and contains many irrelevant hypotheses, which makes it actually easier to prove than to understand. However, we can see that its 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. The VC is marked up with labels that represent this information in order to generate the explanation shown below. Note that the 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 729, which in turn is 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.

Nested Propositions and Simultaneous Conclusions If the VCs contain existential quantifiers, which can be introduced by the annotations or by the rule for procedure calls, they might no longer have a unique conclusion that denotes their primary purpose. Hence, we must modify our notion of conclusion to allow multiple conclusions that must be satisfied simultaneously for an existentially quantified witness, and explicitly represent and render conclusions from local assumptions (i.e., nested implications). Consider, for example, the VC

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

that arises in certifying frame safety (i.e., consistent use of coordinate frames [16]) in navigation software generated from a Simulink model. The purpose of this VC is to show the correctness of a procedure call, i.e., to show that all its preconditions are satisfied and that the function postcondition implies the required postcondition. Hence, we need to show existence of a quaternion q (represented by the four scalars q_0, $v[0]$, $v[1]$, and $v[2]$ and equivalent to a previously given direction cosine matrix d), such that if the term M (which does not matter here) is updated first with the scalar q_0 and then with the vector v, this gives a term in the frame ECI to NED. Our system explains this VC 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-value that will simultaneously
> - establish the function precondition at line 799 (#1),
> - establish the function precondition at line 799 (#2),
> - establish the function precondition at line 799 (#3) under the substitution originating in line 794,
> - establish the postcondition at line 798 (#1) assuming the function postcondition at line 799 (#1).

Note that the explanation only reflects the VC's structure, not the detailed interpretation we have given above; for that, we would need to mark it up with additional policy-specific detail (see Section 4.3).

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. *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: 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. 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 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 *assignment* concept sub captures the origin and effect of assignments and array updates on the form of the resulting VCs; for the shadow environment, we additionally get *safety substitutions* safety_sub. *Origin* labels (origin) denote formulae that result from annotations that have been propagated from their original location. They are specific to the way the code generator produces the annotations.

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.

Label Structure and Labeled Terms We use labeled terms $\ulcorner t \urcorner^{\mathsf{l}}$, where each term t can be adorned with a label l. Labels of the form $c(o, n)$ are called *plain*. Here the concept c 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. The optional list of labels n nested inside contains further qualifying information, which applies either directly to the top-level term, or has been extracted from sub-terms during normalization and extraction. To represent nested implications and and simultaneous goals, we use the *meta-labels* nested and sim, respectively. This gives us the labeled version of the VC (2):

$$\begin{array}{l}
\ldots \wedge \ulcorner T[0]+T[4]+T[8] > 0 \urcorner^{\text{if_tt}(\langle \text{sub}(794) \rangle)} \wedge \ulcorner \text{frame}(T, \text{dcm}(\text{eci}, \text{ned})) \urcorner^{\text{ass_pre}(794)} \\
\Rightarrow \forall q_0 : \text{real}, v : \text{vec} \cdot \exists d : \text{dcm} \cdot \\
\quad \ulcorner \ulcorner \text{tr(d)} = T[0]+T[4]+T[8] \urcorner^{\text{est_fpre}(799,\ \langle \text{sub}(794) \rangle)} \wedge \ulcorner \text{tr(d)} > 0 \urcorner^{\text{est_fpre}(799)} \\
\quad \wedge \ulcorner \text{rep_dcm}(d, T[5], T[7], T[2], T[6], T[1], T[3]) \urcorner^{\text{est_fpre}(799)} \\
\quad \wedge \ulcorner (\exists q : \text{quat} \cdot \ulcorner \text{eq_dcm_quat}(d, q) \wedge \text{rep_quat}(q, q_0, v[0], v[1], v[2]) \urcorner^{\text{ass_post}(799)} \\
\qquad \Rightarrow \ulcorner \text{frame}(\text{vupd}(\text{upd}(M, 0, q_0), 1, 3, v), \text{quat}(\text{eci}, \text{ned}))) \urcorner^{\text{est_post}(798) \urcorner \text{nested} \urcorner \text{sim}}
\end{array}$$

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 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 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 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. 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.

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 WSPC 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

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

$$\textit{(update)}\ \frac{}{\left(\begin{array}{l} Q[\ulcorner \mathit{upd}(x, e_1, e_2)\urcorner^{\mathsf{upd}}/x, \ulcorner \mathit{upd}(x_{\mathrm{init}}, e_1, \mathrm{INIT})\urcorner^{\mathsf{upd\_safety}}/x_{\mathrm{init}}] \\ \wedge \ulcorner \mathit{safe}_{\mathrm{init}}(e_1)\urcorner^{\mathsf{safety}} \wedge \ulcorner \mathit{safe}_{\mathrm{init}}(e_2)\urcorner^{\mathsf{safety}} \end{array}\right)\ \{x[e_1] := e_2\}\ Q}$$

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

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

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

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

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

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 post-conditions are labeled according to their use either as hypotheses (in the VCs) or as conclusions (in the WSPC 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 (see [6] for experimental evidence). Unfortunately, the 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 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*. For the formulation of the rules, we use the auxiliary functions $|\cdot|$ to remove labels from terms, and $[\![\cdot]\!]$ to extract the

labels of a term. $[\![\cdot]\!]$ is defined by

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

where $\oplus$ is list concatenation and the label composition operator $\otimes$ 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 themselves then fall into five different groups. The first group contains rules such as $\ulcorner \mathit{true} \urcorner^{\mathsf{l}} \to \mathit{true}$ or $P \Rightarrow P' \to \mathit{true}$ if $|P| = |P'|$ that remove labels from trivially true (sub-) formulas because these require no explanations. The next group consists of rules such as $\ulcorner \mathit{false} \urcorner^{\mathsf{l}} \vee P \to P$ 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., $\ulcorner \mathit{false} \urcorner^{\mathsf{l}} \wedge P \to \ulcorner \mathit{false} \urcorner^{\mathsf{l}}$. The rules $\ulcorner P \wedge Q \urcorner^{\mathsf{l}} \to \ulcorner P \urcorner^{\mathsf{l}} \wedge \ulcorner Q \urcorner^{\mathsf{l}}$ and $P \Rightarrow \ulcorner Q \Rightarrow R \urcorner^{\mathsf{l}} \to P \wedge \ulcorner Q \urcorner^{\mathsf{l}} \Rightarrow \ulcorner R \urcorner^{\mathsf{l}}$ 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 $\mathit{sel}(\ulcorner \mathit{upd}(x, i_1, t) \urcorner^{\mathsf{l}}, i_2) \to \ulcorner i_1 = i_2 \urcorner^{\mathsf{l}}$ **?** $\ulcorner t \urcorner^{\mathsf{l}}$ **:** $\mathit{sel}(x, i_2)$ 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 $\ulcorner \ulcorner t \urcorner^{\mathsf{m}} \urcorner^{\mathsf{n}} \to \ulcorner t \urcorner^{\mathsf{n} \otimes \mathsf{m}}$ 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.

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.

3.5 Rendering

We define the underlying structure and actual textual representation of the explanations via a BNF-grammar, 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 final generation of the actual explanations, i.e., turning the (labeled) VCs into human-readable text, is called *rendering*. It is independent of the actual aspect that is explained, and can thus be reused. 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 the labels to the explanation templates; (iv) text generation, using the explanation templates.

The third step flattens nested qualifiers, so that for example 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. 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 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 meta-labels.

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 semantic 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 semantic concept 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 or only for specific program constructs, and makes it easier to treat the qualifiers uniformly in different structural contexts.

4.1 Adding Index Information to Loop Explanations

We can tie the explanations of VCs emerging from for-loops more closely to the program, and thus make them easier to understand, if we add more detailed information about the index variables and bounds as qualifiers to the loop invariants. These qualifiers are then used to augment the text generated for the qualified structural label. In our running example we thus get a refined explanation of the VC's purpose, while the rest of the explanation remains unchanged:

> 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 729, which in turn ...

Note that the way the qualifier is rendered depends on the enclosing label, to properly reflect the different substitutions that are applied to the invariant in the different cases (see Figure 1); in particular, the qualifier is ignored when the invariant is used as asserted hypothesis (i.e., for the ass_inv-label).

Since the complete index information required for the qualifier is contained in the for-loop itself, the VCG can easily extract it and add the qualifiers in the respective positions of the *for*-rule. This information is almost impossible to recreate with a post hoc analysis of the formula.

4.2 Adding Relative Positions to Loop Explanations

We can further improve the explanations for VCs emerging from nested loops by referring to the underlying loops not only via their absolute source locations (which are often very close to each other, and thus easily confused), but also by their relative position, distinguishing, for example, the inner from the outer invariant:

> 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 729, which in turn ...

However, the VCG has no built-in notion of "outer" and "inner" loops, so it cannot add the respective qualifiers automatically. Rather than extending the VCG, these qualifiers can be added directly to the annotations by the annotation generator.

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: [1]

> 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 729, which in turn is 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) (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).

Note that the labels also include origin information, explaining where the semantic concepts have originally been established. Again, we can encapsulate the necessary extensions in the annotation generator.

[1] 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 $\forall 0 \leq i, j < N \cdot (i < k \vee i = k \wedge j < l) \Rightarrow A_{\text{init}}[i, j] = \text{INIT}$.

We can go further and 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. 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.

Domain-specific labels are only introduced at annotations. For pre- and postconditions we need to generalize the *assert* rule to use contributor labels since the enclosed statements correspond to blocks which are used in the explanation:

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

The *label* rule "plucks" the label off the 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.

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).

Our work grew out of the earlier work by Denney and Venkatesan [8] which used information from a particular a 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 complies with a safety policy. 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, Leino does not represent the origin of substitutions nor the secondary purpose of the VCs. Moreover, both 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. 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.

6 Conclusions and Future Work

The explanation mechanism which we have described here has been successfully implemented and incorporated into our certification browser [4, 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 all safety policies.

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 [15]: 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. 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.

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.

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 *Tool Exhibition Notes, FM 2003: 12th International FME Symposium*, pp. 37–41, Pisa, Italy, 2003.
3. E. Denney and B. Fischer. Certifiable program generation. In *GPCE 2005, LNCS 3676*, pp. 17–28, Springer 2005.
4. 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)*, pp. 98–116. Edinburgh, 2005.
5. E. Denney and B. Fischer. A generic annotation inference algorithm for the safety certification of automatically generated code. In *GPCE 2006*, pp. 121–130. ACM Press, 2006.
6. E. Denney, B. Fischer, and J. Schumann. An empirical evaluation of automated theorem provers in software certification. *Intl. J. AI Tools*, 15(1):81–107, 2006.
7. E. Denney and S. Trac. A software safety certification tool for automatically generated guidance, navigation and control code. In *Proc. IEEE Aerospace Conference*, IEEE, 2008.
8. E. Denney and R. P. Venkatesan. A generic software safety document generator. In *10th AMAST, LNCS 3097*, pp. 102–116. Springer, 2004.
9. A. Fiedler. Natural language proof explanation. In *Mechanizing Mathematical Reasoning, LNCS 2605*, pp. 342–363. Springer, 2005.
10. B. Fischer and J. Schumann. AutoBayes: A system for generating data analysis programs from statistical models. *J. Functional Programming*, 13(3):483–508, May 2003.
11. R. Fraer. Tracing the origins of verification conditions. In *5th AMAST, LNCS 1101*, pp. 241–255. Springer, 1996.
12. 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.
13. J. C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.
14. G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *PLDI'98* pp. 333–344, ACM Press, 1998. Published as SIGPLAN Notices 33(5).
15. 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, July 2005.
16. D. A. Vallado. *Fundamentals of Astrodynamics and Applications*. Space Technology Library. Microcosm Press and Kluwer Academic Publishers, second edition, 2001.
17. M. Whalen, J. Schumann, and B. Fischer. Synthesizing certified code. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *FME'02, LNCS 2391* pp. 431–450. Springer, 2002.
18. J. Whittle and J. Schumann. Automating the implementation of Kalman filter algorithms. *ACM Trans. Mathematical Software*, 30(4):434–453, 2004.