Formal Safety Certification of Auto-Generated Aerospace Software

Ewen Denney and Bernd Fischer
{edenney,fisch}@email.arc.nasa.gov
USRA/RIACS, NASA Ames Research Center, Moffett Field, CA 94035, USA

Code generators can address many of the increasing demands placed on software in the aerospace industry, yet trust in the code produced by commercial generators is notoriously difficult to achieve and traditionally relies on qualification of the generator. We describe an alternative approach that directly ensures trust in each individual generated program, using a combination of three fully automated formal techniques: *i.*) the generation of safety proofs, *ii.*) the generation of documentation that explains the generated code, and *iii.*) the generation of hyperlinks between all the elements of the code generation and certification process. Our approach is integrated with the AutoFilter generator for state estimation code, but it could, in principle, also be integrated with commercial code generators such as RealTime Workshop.

I. Introduction

In principle, formal methods offer many advantages for aerospace software development: they can help to achieve very high reliability, and they can be used to provide evidence of the reliability claims which can then be subjected to external scrutiny. However, formal methods are not much used in practice, and despite many research advances, three major shortcomings remain. First, their application is still expensive because they are labor- and knowledge-intensive. Second, they are difficult to scale up to complex systems because they are based on deep mathematical insights about the behavior of the systems. Third, they are based on proofs that can be difficult to interpret, and typically stand in isolation from the original code.

In this paper, we describe an approach and tool for formally demonstrating safety properties of aerospace software, which largely circumvents these problems. We focus on safety-relevant aspects because it has been observed[1] that safety violations such as out-of-bounds memory accesses or use of uninitialized variables constitute the majority of the coding errors found in the aerospace domain. In our approach, safety means that the program will not violate a set of rules (similar to coding standards) that can range from simple memory access rules to high-level flight rules. The different safety properties are formalized by different *safety policies* in Hoare logic,[2] which are then used by a verification condition generator (VCG) along with the code and logical annotations in order to derive formal safety conditions; these are then sent to an automated theorem prover (ATP). If all derived safety conditions are proven, the program is guaranteed to satisfy the safety property. Our certification system is currently integrated into a model-based code generation toolset for state estimation code that generates the annotations together with the code. However, this *automated formal certification* technology is not exclusively constrained to our code generator and could, in principle, also be integrated with other code generators such as RealTime Workshop or even be applied to legacy code.

Our approach circumvents the historical problems with formal methods by increasing the degree of automation on all levels. The restriction to safety properties results in simpler proof problems that can generally be solved by fully automatic theorem provers;[3,4] restricted classes of functional behavior could be handled similarly. An automated explanation mechanism uses semantic markup added by the VCG to produce natural-language explanations of the safety conditions and thus supports their interpretation in relation to the code. Currently, the explanations reflect the logical structure of the safety obligation but the mechanism can in principle be customized using different sets of domain concepts. An automated linking mechanism between the safety conditions and the code provides some of the traceability mandated by process standards such as DO-178B.[5] An automatically generated *certification browser* lets users inspect the generated code along with the safety conditions (including textual explanations), and uses hyperlinks to automate tracing between the two levels.

Our long-term goal is a seamless integration of code generation with verification, tracing, and documentation that results in a "certification pipeline" in which specifications are automatically transformed into executable code, together with the supporting artifacts necessary for achieving and demonstrating the high levels of assurance needed in the aerospace domain.

II. Automatic Generation of State Estimation Code

State estimation is the task of determining with the best possible accuracy the position, attitude, and speed of a moving vehicle from potentially noisy sensor measurements. Typical sensors are gyros, accelerometers, and star trackers. It is at the core of most guidance, navigation, and control (GN&C) tasks; the state estimation code is thus one of the most safety-critical, high-assurance components of any GN&C system. However, as many missions (e.g., Mars Climate Orbiter) have shown, such code is error-prone and difficult to develop. Automated code generation techniques can help overcoming these problems. AutoFilter[6] is an automated code-generator which takes as input a compact, high-level description of a state estimation task in the form of differential equations and produces highly documented C or C++ code.

A state estimation problem is defined by (*i*) the system state, which is given in the form of a vector of state variables, (*ii*) the process model, which describes how the system state evolves over time, and (*iii*) the measurement model, which relates the sensor readings to the system state. For example, a very simple planetary rover might be modeled in terms of the speed v_L and v_R of its left and right wheels, respectively, and the yaw y of the chassis. The system state is thus described adequately by the state vector $x = \langle v_L, v_R, y \rangle$. The discrete process model is then given as a linear function $x_{t+1} = Hx_t + w$ where H is a state transition matrix, and w is Gaussian noise. If the rover has sensors which measure the speed of the wheels directly, and a gyro to measure the yaw, the measurement model is given in similar terms, i.e., $x = z + v$ for measurements z and Gaussian noise v.

An AutoFilter specification allows a concise formulation of such models; it also includes details on the desired software architecture. From such specifications, AutoFilter derives code by repeated application of *schemas*. A schema can be seen as a high-level macro or template which can be applied to (sub-) problems of a certain structure, e.g., linear process models. AutoFilter performs symbolic calculations (e.g., linearization, discretization, Taylor series expansion) to make schemas applicable. When a schema is applied, code is generated by instantiating an algorithm skeleton which represents, e.g., an appropriate variant of a Kalman filter algorithm. The code fragments from the individual schema applications are assembled and the entire code is optimized and then translated into a target platform; depending on the specific platform, the necessary matrix operations are mapped to library calls or to nested loops. Currently, AutoFilter supports C/C++ (both stand-alone and with the Matlab and Octave libraries) and the CLARAty[7] and MDS architectures. Typically, the final code is between 300 and 800 lines of C/C++ code, including auto-generated comments.

III. Formal Safety Certification

Formal software certification is based on the idea that a mathematical proof of some aspect of a software system can be regarded as a certificate of correctness which can be subjected to external scrutiny. It uses

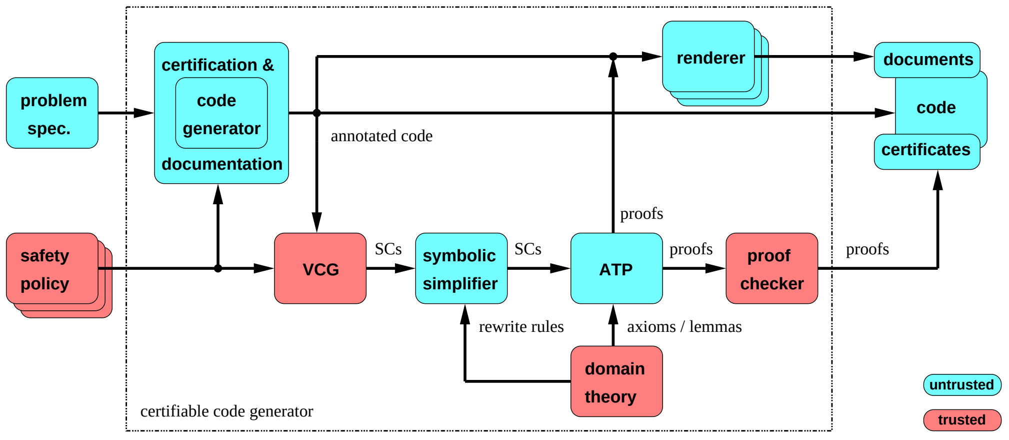


Figure 1. Certifiable code generation: System architecture

the same underlying technology as full program verification but only addresses individual safety properties and not the complete program behavior, which makes it more tractable. A *certifiable code generator* derives and formally certifies code. It consists of the original code generator that is extended for certification and documentation purposes and further complemented by a number of separate certification components that generate and process the safety conditions. Its architecture distinguishes between trusted and untrusted components, shown in Figure 1 in red (dark grey) and blue (light grey), respectively.[2–4] Components are called *trusted*—and must thus be correct—if any errors in them can compromise the assurance provided by the overall system. *Untrusted* components, in contrast, are not crucial to the assurance because their results are double-checked by at least one trusted component. In particular, the correctness of the entire system does not depend on the correctness of the two largest components: the code generator and the ATP; instead, only the safety policy, the VCG, and the proof checker need to be trusted.

Our certification approach works on the source code level but the complete certification chain should properly go down to the object code level. This can be achieved by coupling our system with a *certifying compiler*[8,9] to ensure that the compilation step does not compromise the demonstrated safety policy.

A. Safety Policies

The certifiable code generator generator guarantees that each generated program complies with a given *safety policy*. This is a formal characterization that the program does not "go wrong" (i.e., does not violate certain safety conditions) and consists of a set of logical rules and auxiliary definitions.[2] A key feature of our approach is that policies for different safety aspects are kept distinct, which enables a separation of concerns: different policies can be mixed and matched as appropriate to the certification goal at hand. Since the safety policies are defined in an explicit and declarative way, the system is also *extensible*: users can define new policies, or modify existing ones.

Safety policies exist at two levels of granularity. *Language-specific* policies are expressed in terms of the constructs of the underlying programming language itself, e.g., array accesses. They are sensible for any given program written in the language, regardless of the application domain. Various coding standards (e.g., restrictions on the use of loop indices) also fall into this category. *Domain-specific* properties are, in contrast, specific to a particular application domain and not applicable to all programs. These typically relate to high-level concepts outside the language (e.g., matrix multiplication). In principle, they are independent of the target programming language but in practice they tend to be be expressed in terms of program fragments.

We have integrated four different safety policies with AutoFilter so far. Array-bounds safety requires each access to an array element to be within the specified upper and lower bounds of the array. Variable initialization-before-use ensures that each variable or individual array element has been assigned a defined value before it is used. Both are typical examples of language-specific properties. Matrix symmetry requires certain two-dimensional arrays to be symmetric. Sensor input usage is a variation of the general initialization-property which guarantees that each sensor reading passed as an input to the Kalman filter is actually used during the computation of the output estimate. These two examples are specific to the state estimation domain.

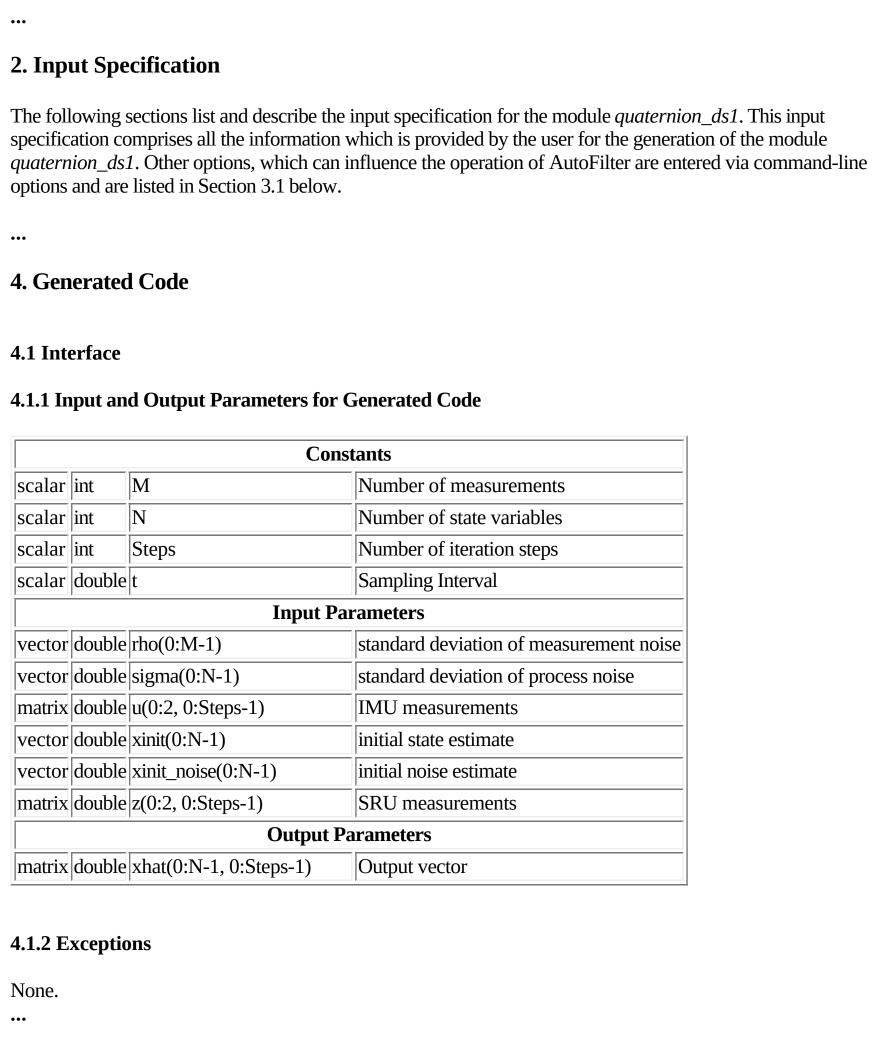
B. Generating Safety Obligations

For certification purposes, the code generator *annotates* the code with mark-up information relevant to the selected safety policy. The annotations contain information in the form of logical pre- and post-conditions and loop invariants, which are part of the schemas and thus are instantiated in parallel with the code fragments. The annotated code is then processed by the VCG, which works backwards through the code, applies the rules of the safety policy and generates safety conditions at each statement. The VCG has been designed to be “correct-by-inspection”, i.e., to be sufficiently simple that it is straightforward to see that it correctly implements the rules of the logic. Hence, it does not implement any simplifications and, consequently, the generated safety conditions tend to be large and must be simplified separately. This is done by a symbolic simplifier that applies a set of rewrite rules specified in the domain theory. The simplified conditions are then augmented by the axioms and lemmas of the domain theory and processed by an ATP and the resulting proofs can be sent to a separate proof checker to ensure their validity.[10]

C. Summary of Experimental Results

We have tested our certification approach in two different “after-the-fact” case studies, where we generated and certified Kalman filters from the requirements of existing applications. In the first case study, we extracted the mathematical model of the state estimator from the requirements of the attitude control system of NASA’s Deep Space I mission and reformulated it as an AutoFilter specification. It combines inputs from an inertial measurement unit (IMU) and a star tracker to obtain a more accurate estimate of the attitude of the spacecraft. The filter has three state variables representing change in spacecraft attitude since the last measurement, and three state variables representing the IMU gyro drift. For this specification, AutoFilter generates code based on an extended Kalman filter. The second case study was taken from a simulation of the Space Shuttle docking procedure at the International Space Station, for which a different configuration of an extended Kalman filter is generated.

In both cases, the proportion of annotations to code and the number of safety conditions vary widely with the safety policy. In general, language-specific policies tend to be simpler than domain-specific ones, which require more detailed annotations and produce more conditions. The simplest policy is array bounds, which requires no annotations at all and produces only a few, easy conditions that are discharged immediately. The other policies require annotations that amount to five to ten percent of the overall code size and produce up to 865 conditions. However, the complexity of these conditions is well within the capabilities of current ATPs. For two of the policies (i.e., initialization, and symmetry) the ATP was able to discharge all safety conditions with average proof times of approximately one second. For the input usage policy, the system produces one unprovable condition for each of the programs; these conditions take much longer to detect and thus distort the average and total proof times. However, it is important to notice that unprovable conditions do not necessarily imply that the programs will fail but rather indicate problems that require more detailed human scrutiny. Here, the unprovable obligations are a consequence of the conservative way the safety policy is formulated.



...

2. Input Specification

The following sections list and describe the input specification for the module *quaternion_ds1*. This input specification comprises all the information which is provided by the user for the generation of the module *quaternion_ds1*. Other options, which can influence the operation of AutoFilter are entered via command-line options and are listed in Section 3.1 below.

...

4. Generated Code

4.1 Interface

4.1.1 Input and Output Parameters for Generated Code

Constants			
scalar	int	M	Number of measurements
scalar	int	N	Number of state variables
scalar	int	Steps	Number of iteration steps
scalar	double	t	Sampling Interval
Input Parameters			
vector	double	rho(0:M-1)	standard deviation of measurement noise
vector	double	sigma(0:N-1)	standard deviation of process noise
matrix	double	u(0:2, 0:Steps-1)	IMU measurements
vector	double	xinit(0:N-1)	initial state estimate
vector	double	xinit_noise(0:N-1)	initial noise estimate
matrix	double	z(0:2, 0:Steps-1)	SRU measurements
Output Parameters			
matrix	double	xhat(0:N-1, 0:Steps-1)	Output vector

4.1.2 Exceptions

None.
...

Figure 2. Generated software design document (excerpts)

IV. Automatic Documentation Generation

In our formal safety certification approach, all evidence is given as logical proofs. However, these proofs are unlikely to be recognized as such, even by mathematicians, since they consist primarily of the low-level steps carried out by the applied ATPs. Moreover, proofs themselves—even if they were on a higher level of abstraction—are of little interest to engineers if they do not explicitly refer back into the program. We address these issues by combining formal certification with automatic documentation generation techniques.

A. Design Documentation

In addition to the code, we can also generate a standardized software design document (see Figure 2 for excerpts) that contains administrative information (names of files, versions, etc.), interface descriptions, *The assignment* `a[2*d-1-i]=i` *at line* `12` *is safe (if the condition* `i<5` *at line* `10` *is false); the term* `d` *is initialized from* `d=b*b+c*c` *at line* `8`*; the term* `b` *is initialized from* `b=1` *at line* `6`*; the term* `c` *is initialized from* `c=2` *at line* `7`*; the loop index* `i` *ranges from* `0` *to* `9` *and is initialized at line* `9`*.*

The access `a[a[5]]` *at line* `13` *is safe; using the invariant for the loop at line* `9` *and the postcondition* `i=9+1` *after the loop;* `a[5]` *is within* `0` *and* `9`*; and hence the access is within the bounds of the array* `a` *declared at line* `1`*.*

Figure 3. Generated explanations for initialization (top) and array bounds (bottom) safety policies (excerpts)

specific input and output constraints, and synthesis and compiler warnings. The document is hyperlinked to the input specification, the code, and any other intermediate artifacts generated by the system. Since it is produced at generation time, it can refer back to the original specification and also include design details which are difficult to infer from the generated code alone.

B. Safety Documentation

We have also developed a generic framework for generating detailed textual justification of compliance with the given safety policy.[11] We use the information obtained during the logical analysis of the software to produce explanations for *why* the different parts of a program are safe. The system is based on customizable explanation templates which convert logical entities that appear in the safety conditions such as loop invariants into text. It can be instantiated with a range of different safety policies, and new policies can easily be added to the system. It uses labels which are propagated through the VCG and into the safety conditions so that the explanations can explicitly refer back to program locations. Figure 3 shows excerpts from an automatically generated safety document for the initialization and array-bounds safety of a synthesized program.

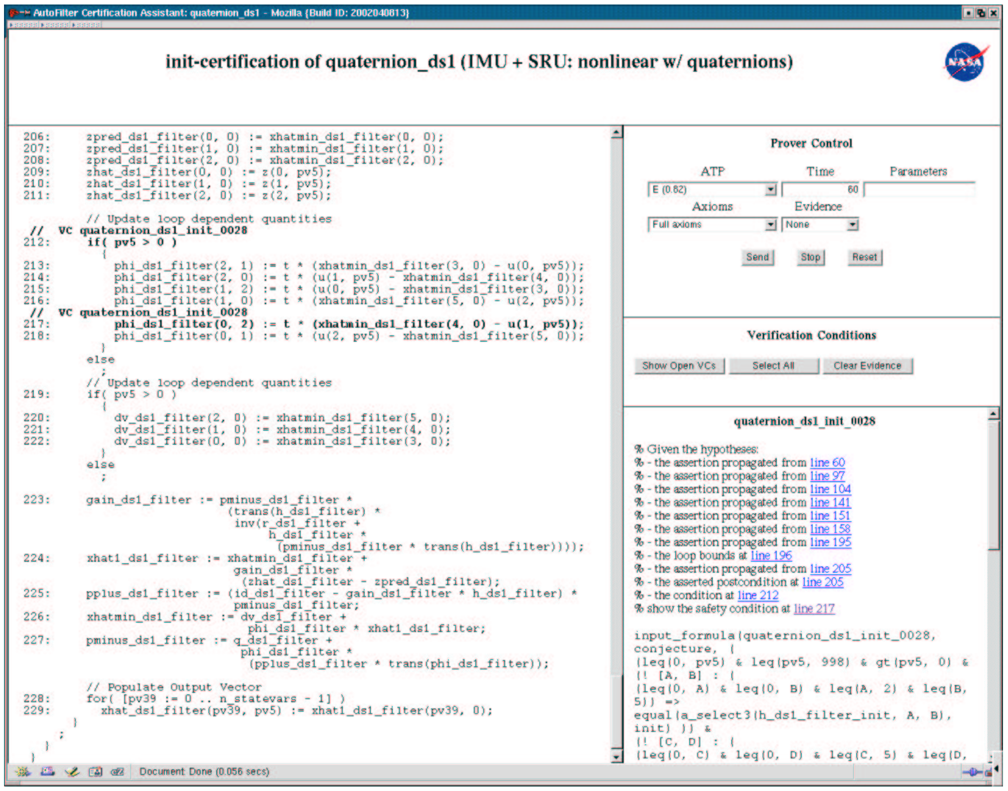
Since safety documents can potentially contain a huge amount of information, the system allows two ways of focusing attention. First, users can restrict attention either to specific program variables or to certain lines of code so that, in effect, the system *slices* the program. Second, the system has a heuristic ordering of the importance of various classes of information (e.g., information about loop invariants is more important than that from assignments) so that users can set the level of importance they care about.

C. Linking Safety Conditions

A safety condition can fail to be proven for a number of different reasons: the (generated) annotations may be insufficient or wrong, the theorem prover may time-out, either due to the size and complexity of the condition, or due to an incomplete domain theory, and there may of course be an actual safety violation in the code. For certification purposes, however, it is important to distinguish between unsafe programs and any other reasons for failure, and in the case of genuine safety violations, to locate the unsafe parts of the program.

However, manually tracing the safety conditions back to their source is quite difficult as the certification process is inherently complex. The conditions can become very large and go through substantial structural simplifications, and any single condition can depend on a variety of information distributed throughout the program. In order to support tracing between the conditions and the code, the VCG adds appropriate location information to the formulas it constructs as it processes a statement at a given source code location. The locations information is then threaded through all stages of our certification architecture (cf. Figure 1).

Figure 4 shows how the tracing information can be used to support the certification process. A click on the source link associated with each condition prompts the certification browser to highlight in boldface all



AutoFilter Certification Assistant: quaternion_ds1 - Mozilla {Build ID: 2002040813}

init-certification of quaternion_ds1 (IMU + SRU: nonlinear w/ quaternions)

```
206:     zpred_ds1_filter(0, 0) := xhatmin_ds1_filter(0, 0);
207:     zpred_ds1_filter(1, 0) := xhatmin_ds1_filter(1, 0);
208:     zpred_ds1_filter(2, 0) := xhatmin_ds1_filter(2, 0);
209:     zhat_ds1_filter(0, 0) := z(0, pv5);
210:     zhat_ds1_filter(1, 0) := z(1, pv5);
211:     zhat_ds1_filter(2, 0) := z(2, pv5);

         // Update loop dependent quantities
 //   VC quaternion_ds1_init_0028
212:     if( pv5 > 0 )
           {
213:         phi_ds1_filter(2, 1) := t * (xhatmin_ds1_filter(3, 0) - u(0, pv5));
214:         phi_ds1_filter(2, 0) := t * (u(1, pv5) - xhatmin_ds1_filter(4, 0));
215:         phi_ds1_filter(1, 2) := t * (u(0, pv5) - xhatmin_ds1_filter(3, 0));
216:         phi_ds1_filter(1, 0) := t * (xhatmin_ds1_filter(5, 0) - u(2, pv5));
 //   VC quaternion_ds1_init_0028
217:         phi_ds1_filter(0, 2) := t * (xhatmin_ds1_filter(4, 0) - u(1, pv5));
218:         phi_ds1_filter(0, 1) := t * (u(2, pv5) - xhatmin_ds1_filter(5, 0));
           }
         else
           ;
         // Update loop dependent quantities
219:     if( pv5 > 0 )
           {
220:         dv_ds1_filter(2, 0) := xhatmin_ds1_filter(5, 0);
221:         dv_ds1_filter(1, 0) := xhatmin_ds1_filter(4, 0);
222:         dv_ds1_filter(0, 0) := xhatmin_ds1_filter(3, 0);
           }
         else
           ;

223:     gain_ds1_filter := pminus_ds1_filter *
                             (trans(h_ds1_filter) *
                              inv(r_ds1_filter +
                                   h_ds1_filter *
                                    (pminus_ds1_filter * trans(h_ds1_filter))));
224:     xhat1_ds1_filter := xhatmin_ds1_filter +
                              gain_ds1_filter *
                               (zhat_ds1_filter - zpred_ds1_filter);
225:     pplus_ds1_filter := (id_ds1_filter - gain_ds1_filter * h_ds1_filter) *
                              pminus_ds1_filter;
226:     xhatmin_ds1_filter := dv_ds1_filter +
                                phi_ds1_filter * xhat1_ds1_filter;
227:     pminus_ds1_filter := q_ds1_filter +
                               phi_ds1_filter *
                                (pplus_ds1_filter * trans(phi_ds1_filter));

         // Populate Output Vector
228:     for( [pv39 := 0 .. n_statevars - 1] )
229:       xhat_ds1_filter(pv39, pv5) := xhat1_ds1_filter(pv39, 0);
     }
   ;
  }
 }
```

Document Done (0.056 secs)

Prover Control

ATP: E (0.82)
Time: 60
Parameters:
Axioms: Full axioms
Evidence: None

Verification Conditions

quaternion_ds1_init_0028

% Given the hypotheses:
% - the assertion propagated from line 60
% - the assertion propagated from line 97
% - the assertion propagated from line 104
% - the assertion propagated from line 141
% - the assertion propagated from line 151
% - the assertion propagated from line 158
% - the assertion propagated from line 195
% - the loop bounds at line 196
% - the assertion propagated from line 205
% - the asserted postcondition at line 205
% - the condition at line 212
% show the safety condition at line 217

```
input_formula(quaternion_ds1_init_0028,
conjecture, (
(leq(0, pv5) & leq(pv5, 998) & gt(pv5, 0) &
(! [A, B] : (
(leq(0, A) & leq(0, B) & leq(A, 2) & leq(B,
5)) =>
equal(a_select3(h_ds1_filter_init, A, B),
init) )) &
(! [C, D] : (
(leq(0, C) & leq(0, D) & leq(C, 5) & leq(D,
```

Figure 4. Certification browser

affected lines of the code. A further click on the condition link itself displays the formula, which can then be interpreted in the context of the relevant program fragments. This helps domain experts assess whether the safety policy is actually violated, which parts of the program are affected, and eventually how the violation can be resolved. This traceability is also mandated by relevant standards such as DO-178B.[5]

In practice, safety checks are often carried out during code reviews,[12] where reviewers look in detail at each line of the code and check the individual safety properties statement by statement. To support this, linking works in both directions: clicking on a statement or annotation displays all conditions to which it contributes.

V. Conclusions

Code generators can address many of the increasing demands placed on software in the aerospace industry, but currently they cannot be used to generate flight-ready code without substantial post-hoc verification and adaptation. We have described an alternative approach to tool qualification that can be used to gain trust in automatically generated code. Our approach is based on the application of mathematical logic, and formally certifies the compliance of the generated code with a given safety policy.

We are currently applying these techniques in several projects, to both our own AutoFilter generator and to commercial tools. These applications include the automated certification of reconfigured rover software, the certification of auto-generated code for the RASCAL rotorcraft testbed, and using code generation to support the adaptation of legacy Shuttle software.

We anticipate that our framework of safety polices can also be extended to restricted classes of functional behaviors. For example, proving the optimality of Kalman filters has been automated.[13] The key to extending our current technology to such properties is having explicit domain models which directly represent concepts such as measurements, controls, integrators, gains, and filters. This knowledge is already implicit in our code generator, but it remains to integrate it with the certification framework. This will also allow us to provide more meaningful and domain-specific explanations and to trace fragments of the code to corresponding elements of the model.

References

[1]Kandt, R., "Software Defect Avoidance and Detection: Practices and Techniques," Tech. rep., JPL, 2003, Document D-24993.

[2]Denney, E. and Fischer, B., "Correctness of Source-Level Safety Policies," *Proceedings of theFM 2003: Formal Methods*, edited by K. Araki, S. Gnesi, and D. Mandrioli, Vol. 2805 of *Lecture Notes in Computer Science*, Springer, Pisa, Italy, Sept. 2003, pp. 894–913.

[3]Denney, E., Fischer, B., and Schumann, J., "Using Automated Theorem Provers to Certify Auto-Generated Aerospace Software," *Proceedings of the Second International Joint Conference Automated Reasoning*, edited by D. Basin and M. Rusinowitch, Vol. 3097 of *Lecture Notes in Artificial Intelligence*, Springer, Cork, Ireland, 2004, pp. 198–212.

[4]Denney, E., Fischer, B., and Schumann, J., "An Empirical Evaluation of Automated Theorem Provers in Software Certification," *International Journal of AI Tools*, 2005, To appear.

[5]RTCA Special Committee 167, "Software Considerations in Airborne Systems and Equipment Certification," Tech. rep., RTCA, Inc., Dec. 1992.

[6]Whittle, J. and Schumann, J., "Automating the Implementation of Kalman Filter Algorithms," *ACM Transactions on Mathematical Software*, Vol. 30, No. 4, Dec. 2004, pp. 434–453.

[7]Nesnas, I., Wright, A., Bajracharya, M., Simmons, R., Estlin, T., and Kim, W. S., "CLARAty: An Architecture for Reusable Robotic Software," *SPIE Aerosense Conference*, Orlando, Florida, April 2003.

[8]Necula, G. C. and Lee, P., "The Design and Implementation of a Certifying Compiler," *Proceedings of the ACM Conference on Programming Language Design and Implementation 1998*, edited by K. D. Cooper, ACM Press, Montreal, Canada, June 17–19 1998, pp. 333–344, Published as SIGPLAN Notices 33(5).

[9]Santhanam, V., "The anatomy of an FAA-qualifiable Ada subset compiler," *Proceedings of the 2002 annual ACM SIGAda international conference on Ada*, edited by J. McCormick, ACM Press, Houston, Texas, USA, 2002, pp. 40–43.

[10]Sutcliffe, G., Denney, E., and Fischer, B., "Practical Proof Checking for Program Certification," *Proceedings of the CADE-20 Workshop on Empirically Successful Classical Automated Reasoning (ESCAR'05)*, July 2005.

[11]Denney, E. and Venkatesan, R. P., "A Generic Software Safety Document Generator," *Proceedings of the 10th International Conference on Algebraic Methodology and Software Technology*, edited by C. Rattray, S. Maharaj, and C. Shankland, Vol. 3097 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 102–116.

[12]Nelson, S. and Schumann, J., "What makes a Code Review Trustworthy?" *Proceedings of the Thirty-Seventh Annual Hawaii International Conference on System Sciences (HICSS-37)*, IEEE, 2004.

[13]Lowry, M., Pressburger, T., and Rosu, G., "Certifying Domain-Specific Policies," *Proceedings of the 16th International Conference on Automated Software Engineering*, edited by M. S. Feather and M. Goedicke, IEEE Computer Society Press, San Diego, CA, Nov. 26–29 2001, pp. 118–125.