Formal Safety Certification of Aerospace Software

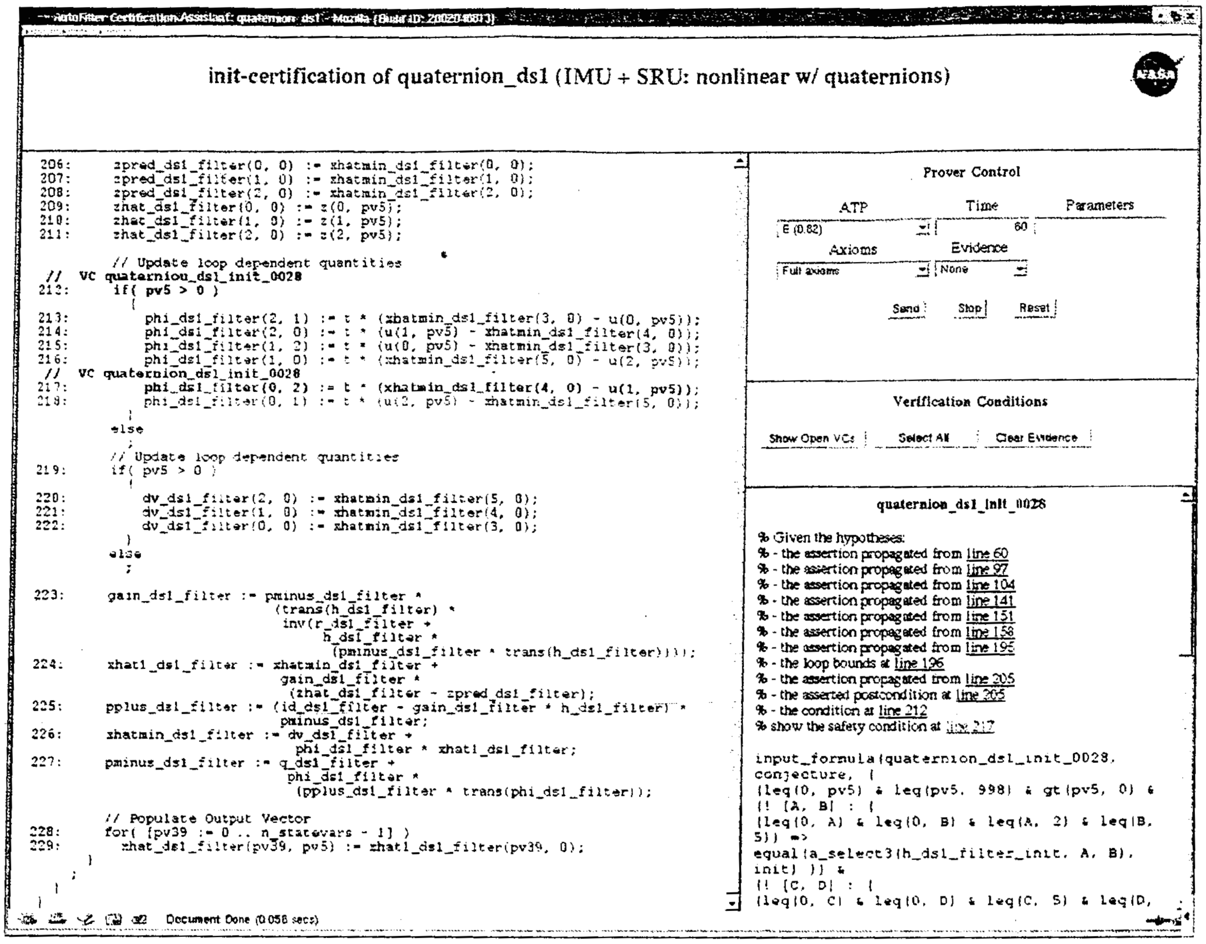
Ewen Denney and Bernd Fischer
{edenney,fisch}@email.arc.nasa.gov
USRA/RIACS, NASA Ames Research Center, Moffett Field, CA 94035, USA

In principle, formal methods offer many advantages for aerospace software development: they can help to achieve ultra-high reliability, and they can be used to provide evidence of the reliability claims which can then be subjected to external scrutiny. However, despite years of research and many advances in the underlying formalisms of specification, semantics, and logic, formal methods are not much used in practice. In our opinion this is related to three major shortcomings. First, the application of formal methods 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 (i.e., they rely on the "heroic proof"). Third, the proofs can be difficult to interpret, and typically stand in isolation from the original code.

In this paper, we describe a tool for formally demonstrating safety-relevant aspects of aerospace software, which largely circumvents these problems. We focus on safety properties 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 errors found in the aerospace domain. In our approach, safety means that the program will not violate a set of rules that can range for the simple memory access rules to high-level flight rules. These different safety properties are formalized as different *safety policies* in Hoare logic, which are then used by a verification condition generator along with the code and logical annotations in order to derive formal safety conditions; these are then proven using an automated theorem prover. Our certification system is currently integrated into a model-based code generation toolset 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 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 policies (as opposed to arbitrary functional behavior) results in simpler proof problems that can generally be solved by fully automatic theorem provers.[2] An automated linking mechanism between the safety conditions and the code provides some of the traceability mandated by process standards such as DO-178B.[3] An automated explanation mechanism uses semantic markup added by the verification condition generator to produce natural-language explanations of the safety conditions and thus supports their interpretation in relation to the code. Figure 1 shows an automatically generated *certification browser* that lets users inspect the (generated) code along with the safety conditions (including textual explanations), and uses hyperlinks to automate tracing between the two levels. Here, the explanations reflect the logical structure of the safety obligation but the mechanism can in principle be customized using different sets of domain concepts. The interface also provides some limited control over the certification process itself.

Our long-term goal is a seamless integration of certification, code generation, and manual coding that results in a "certified 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.



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

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

```
206:      zpred_dsl_filter(0, 0) := xhatmin_dsl_filter(0, 0);
207:      zpred_dsl_filter(1, 0) := xhatmin_dsl_filter(1, 0);
208:      zpred_dsl_filter(2, 0) := xhatmin_dsl_filter(2, 0);
209:      zhat_dsl_filter(0, 0) := z(0, pv5);
210:      zhat_dsl_filter(1, 0) := z(1, pv5);
211:      zhat_dsl_filter(2, 0) := z(2, pv5);

          // Update loop dependent quantities
 //   VC quaterniou_dsl_init_0028
212:      if( pv5 > 0 )
            {
213:          phi_dsl_filter(2, 1) := t * (xbatmin_dsl_filter(3, 0) - u(0, pv5));
214:          phi_dsl_filter(2, 0) := t * (u(1, pv5) - xhatmin_dsl_filter(4, 0));
215:          phi_dsl_filter(1, 2) := t * (u(0, pv5) - xhatmin_dsl_filter(3, 0));
216:          phi_dsl_filter(1, 0) := t * (xhatmin_dsl_filter(5, 0) - u(2, pv5));
 //   VC quaternion_dsl_init_0028
217:          phi_dsl_filter(0, 2) := t * (xhatmin_dsl_filter(4, 0) - u(1, pv5));
218:          phi_dsl_filter(0, 1) := t * (u(2, pv5) - xhatmin_dsl_filter(5, 0));
            }
          else
            ;
          // Update loop dependent quantities
219:      if( pv5 > 0 )
            {
220:          dv_dsl_filter(2, 0) := xhatmin_dsl_filter(5, 0);
221:          dv_dsl_filter(1, 0) := xhatmin_dsl_filter(4, 0);
222:          dv_dsl_filter(0, 0) := xhatmin_dsl_filter(3, 0);
            }
          else
            ;

223:      gain_dsl_filter := pminus_dsl_filter *
                              (trans(h_dsl_filter) *
                               inv(r_dsl_filter +
                                   h_dsl_filter *
                                    (pminus_dsl_filter * trans(h_dsl_filter))));
224:      xhat1_dsl_filter := xhatmin_dsl_filter +
                               gain_dsl_filter *
                                (zhat_dsl_filter - zpred_dsl_filter);
225:      pplus_dsl_filter := (id_dsl_filter - gain_dsl_filter * h_dsl_filter) *
                               pminus_dsl_filter;
226:      xhatmin_dsl_filter := dv_dsl_filter +
                                 phi_dsl_filter * xhat1_dsl_filter;
227:      pminus_dsl_filter := q_dsl_filter +
                                phi_dsl_filter *
                                 (pplus_dsl_filter * trans(phi_dsl_filter));

          // Populate Output Vector
228:      for( [pv39 := 0 .. n_statevars - 1] )
229:        zhat_dsl_filter(pv39, pv5) := zhat1_dsl_filter(pv39, 0);
        }
      ;
    }
  }
```

Document Done (0.058 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_dsl_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_dsl_filter_init, A, B),
init) )) &
(! [C, D] : (
(leq(0, C) & leq(0, D) & leq(C, 5) & leq(D,
```

Figure 1. Certification browser

References

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

[2]Denney, E., Fischer, B., and Schumann, J., "Using Automated Theorem Provers to Certify Auto-Generated Aerospace Software," *Proceedings of the 2nd International Joint Conference on Automated Reasoning (IJCAR'04)*, Vol. 3097 of *Lecture Notes in Artificial Intelligence*, Cork, Ireland, 2004, pp. 198–212.

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