Adding Assurance to Automatically Generated Code

Ewen Denney[†], Bernd Fischer[‡], Johann Schumann[‡]
[†]QSS / [‡]RIACS, NASA Ames Research Center, {edenney, fisch, schumann}@email.arc.nasa.gov

***Abstract*— Code to estimate position and attitude of a spacecraft or aircraft belongs to the most safety-critical parts of flight software. The complex underlying mathematics and abundance of design details make it error-prone and reliable implementations costly. AutoFilter is a program synthesis tool for the automatic generation of state estimation code from compact specifications. It can automatically produce additional safety certificates which formally guarantee that each generated program individually satisfies a set of important safety policies. These safety policies (e.g., array-bounds, variable initialization) form a core of properties which are essential for high-assurance software. Here we describe the AutoFilter system and its certificate generator and compare our approach to the static analysis tool PolySpace.**

I. Introduction

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 for a spacecraft, or wheel rotation sensors for a planetary rover. State estimation is 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.

AutoFilter [1] 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. From a user's point of view, the system can be seen as an intelligent compiler and, as is the case with compilers, the correctness of the generated code depends on the correctness of the generator itself. However, even though AutoFilter has a formal basis, a full verification is not feasible due to the size, complexity, and dynamic nature of the system.

We have thus developed and implemented a *product-oriented certification* approach in which checks are performed on each and every generated program rather than on the generator (i.e., AutoFilter) itself. We focus on *safety properties*, which are generally accepted as important for quality assurance and are used in code reviews of high-assurance software.

Our tool uses program verification techniques based on Hoare logic and processes logical pre- and post-conditions statement by statement to produce proof obligations. These are then processed further by an automatic theorem prover. However, such techniques require additional program annotations (usually loop invariants) which makes their application very hard in practice. We overcome this obstacle by extending AutoFilter to synthesize *simultaneously* the code *and* all required annotations. This enables a fully automatic certification which is transparent to the user and produces machine-readable certificates showing that the generated code does not violate the given safety policies.

II. Auto-generation of State Estimation 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, code is derived by repeated application of *schemas*. A schema can be seen as a high-level macro or axiom which can be applied to (sub-)problems of a certain structure, e.g., linear process models. AutoFilter performs substantial 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; currently, AutoFilter supports C (both stand-alone and with the Matlab and Octave libraries) and Modula-2. Depending on the specific platform, the necessary matrix operations are mapped to library calls or to nested loops. Typically, the final code is between 300 and 800 lines of C or C++ code including auto-generated comments.

III. Product-oriented Certification

The safety policies checked by our system describe either language-specific or domain-specific properties which a safe program must satisfy. A typical example of a language-specific property (C/C++) is array-bounds safety; violations can lead to serious flaws, as many buffer-overrun attacks have shown.

Checks for consistency of physical units or symmetry of matrices are specifically tailored to the application domain and provide additional assurance.

Our system currently handles array-bounds (i.e., array indices must be within bounds), variable-initialization (i.e., variables must be initialized before use), variable-usage (i.e., all input/output variables are used), and matrix symmetry. This last property is specific to the AUTOFILTER domain, and ensures that the code does not result in skewed covariance matrices. However, it does not yet take numerical round-off errors into account. All of these safety properties have been identified as important by a recent study within NASA and the aerospace industry [2].

The properties are checked using a standard approach based on Hoare rules. Hoare rules use triples of the form $P\ \{C\}\ Q$, meaning "if pre-condition P holds before execution of statement C, then post-condition Q holds after". For each kind of statement and for each safety property, such a Hoare rule is given. Starting with the final postcondition *true*, a verification condition generator (VCG) applies these rules backwards and computes, statement by statement, first-order logic formulae (verification conditions, VCs) which describe the safety obligations. The VCG needs auxiliary annotations in the code (mostly loop invariants) to perform this step automatically. However, since we know at *synthesis time* (i) what form the code will take and (ii) which safety policy is used, AUTOFILTER can generate the appropriate annotations. The annotation generation is interleaved with the code generation, and annotation skeletons, which are part of the schema, are instantiated in parallel with the algorithm skeleton. The VCs are then simplified and fed into an automated theorem prover, in our case E-Setheo. If and only if all VCs can be shown to be true, then the property holds for the entire program. Finally, the proofs can be double-checked by an independent proof checker tool to yield a tamper-proof certificate. Figure 1 shows the overall architecture of the system; for more details see [3], [4].

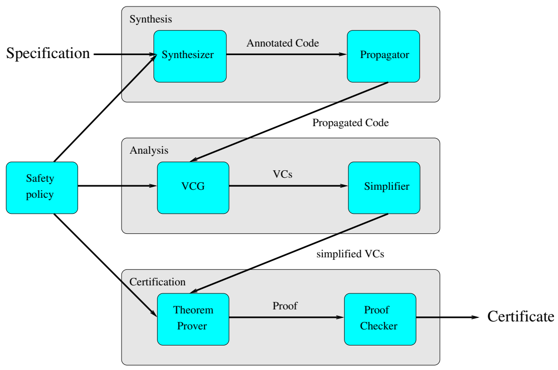

Fig. 1. Architecture of the AUTOFILTER program synthesis system with automatic certificate generation.

IV. Experimental Results

Table I shows the results for two specifications and the different policies supported by AUTOFILTER. The first example is taken from the attitude control system of NASA's Deep Space One mission. The second example specifies a component in a simulation for the Space Shuttle docking procedure at the International Space Station. The number of generated VCs depends on the safety policy and the synthesized code but for both examples E-Setheo was able to prove all tasks. All times have been measured on a 2GHz/2GB standard PC.

TABLE I

CERTIFICATION RESULTS AND TIMES

Spec.	Policy	LoC	T_{synth}	#VCs	T_{proof}	T_{Poly}
ds1	array	431 + 0	5.5s	1	0.1s	
	init	431 + 86	11.4s	74	70.6s	22m 28s (array, init, in-use)
	in-use	431 + 60	8.1s	21	29.5s	
	symm	431 + 83	70.8s	865	756.7s	N/A
thruster	array	755 + 0	24.7s	4	2.9s	
	init	755 + 87	39.7s	71	64.9s	32m 6s (array, init, in-use)
	in-use	755 + 59	33.3s	28	32.4s	
	symm	755 + 87	66.2s	480	472.8s	N/A

The numbers for the program size (LoC) show the size of code itself separately from the size of the auto-generated annotations; note that the latter varies significantly with the safety policy and can make up a substantial fraction of the overall code size. The synthesis times T_{synth} include the time spent on generating and simplifying the VCs, with the latter being the dominating factor, but not the proof time T_{proof}. Overall, the runtimes demonstrate that our approach to automatic certification of safety properties is feasible.

We also compared our approach with the state-of-the-art static analysis tool PolySpace; its runtimes are shown in the last column of the table. The results cannot be compared directly since PolySpace has a fixed built-in safety policy which is more comprehensive than any single policy in our framework. However, the combination of the three language-specific properties already provides a good approximation, but requires less time. Moreover, static analysis tools raise a large number of "false alarms", especially in cases with complicated array accesses, which is common in our domain.

V. Conclusions

We have developed an extension to the AUTOFILTER code generator that can automatically check important safety properties for the generated code. Of course, this is not equivalent to full functional verification so the approach does not obviate the need for testing. However, in principle, it is complete for any given safety policy. In practice, the prover can fail to prove some provable VCs and thus raise false alarms but their number is much lower than typically achieved with state-of-the-art static analysis tools. Our current efforts focus on integrating additional safety properties and extending the approach to synthesized code that has been modified manually.

References

[1] J. Whittle and J. Schumann, "Automating the implementation of Kalman-filter algorithms," 2003, in review.

[2] S. Nelson and J. Schumann, "What makes a code review trustworthy?," in *Proc. HICSS-37*. IEEE, 2004, to appear.

[3] M. Whalen, J. Schumann, and B. Fischer, "Synthesizing certified code," in *Proc. FME 2002*, LNCS 2391, pp. 431–450. Springer, 2002.

[4] E. Denney and B. Fischer, "Correctness of source-level safety policies," in *Proc. FM 2003*, LNCS 2805, pp. 894–913. Springer, 2003.