Automatic Certification of Kalman Filters for Reliable Code Generation

Ewen Denney, Bernd Fischer, Johann Schumann
Julian Richardson
USRA/RIACS, NASA Ames Research Center, {edenney,fisch,schumann,julianr}@email.arc.nasa.gov

Abstract—AUTOFILTER is a tool for automatically deriving Kalman filter code from high-level declarative specifications of state estimation problems. It can generate code with a range of algorithmic characteristics and for several target platforms. The tool has been designed with reliability of the generated code in mind and is able to automatically certify that the code it generates is free from various error classes. Since documentation is an important part of software assurance, AUTOFILTER can also automatically generate various human-readable documents, containing both design and safety related information. We discuss how these features address software assurance standards such as DO-178B.

TABLE OF CONTENTS

1. INTRODUCTION

Code to estimate the position and attitude of an aircraft or spacecraft is one of the most safety-critical parts of flight software. Moreover, the complex underlying mathematics and abundance of design details make it error-prone and reliable implementations costly. Automatic code generation or *program synthesis* techniques can help solve this predicament by completely automating the coding phase. A code generator takes as input a domain-specific high-level description of a task (e.g., a set of differential equations) and produces optimized and documented low-level code (e.g., C or C++) that is based on algorithms appropriate for the task (e.g., the extended Kalman filter). This automation increases developer productivity and—in principle—prevents the introduction of coding errors. Ultimately, however, the correctness of the generated code depends on the correctness of the generator itself. This dependency has led the FAA to require that development tools be qualified to the same level of criticality as the developed flight software [14]. Unfortunately, due to their size, complexity, and dynamic nature, the qualification of advanced code generators is difficult and expensive, which has severely limited their application in flight software development [1].

We have thus developed an alternative product-oriented certification approach that breaks the dependency between generator correctness and code correctness by checking each and every generated program individually, rather than the generator itself. Our approach uses program verification techniques based on standard Hoare-style program logic. It applies rules of the logic backwards and computes, statement by statement, logical formulae or *safety obligations* which are then processed further by an automatic theorem prover. To perform this step automatically, however, auxiliary annotations are required throughout the code. We thus extend the code generator to *simultaneously* synthesize the code and all required annotations: since the code generator has full knowledge about the form the code will take and which property is being checked, it can generate the appropriate 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 required safety properties. For DO-178B compliance, this formal certification must be complemented by various forms of (human-readable) documentation. We have thus extended our system to automatically generate these documents, eliminating a laborious and error-prone step.

Our approach focuses on safety properties, which are generally accepted as important for quality assurance and have been identified as important by a recent study within NASA and the aerospace industry [11]. We currently handle array bounds, variable initialization, proper sensor input usage (i.e., all input variables are used in the computation of the filter output), and matrix symmetry (i.e., covariance matrices are not skewed). The last two properties are specific to the state estimation domain, and are difficult to check with other tools. The approach can readily be extended to other properties required by DO-178B, e.g., numeric underflow/overflow.

In this paper we describe the extension of our AUTOFILTER [17] state estimation code generator by these certification techniques. We have applied the extended sys-

Paper No. 1207: 0-7803-8870-4/05/$20.00

tem to a model taken from the attitude control system of NASA's Deep Space I (DS1) mission and to a simulation for the space shuttle docking procedure at the International Space Station. These case studies have demonstrated that the certification approach is feasible and produces an order of magnitude fewer false alarms than commercial analysis tools. In Sections 2 and 3 we give a short overview of the AutoFilter system and different software assurance techniques, respectively. In Section 4, we describe our overall approach to synthesizing certifiable code, while we focus in Section 5 on the automatic generation of human-readable support documentation. In Section 6 we describe the two certification case studies in more detail, before we conclude in Section 7.

2. AutoFilter

AutoFilter [17] is a domain-specific program synthesis system that generates customized Kalman filters for state estimation tasks specified in a high-level notation. It is implemented in SWI-Prolog [18] and currently comprises about 75,000 lines of code. AutoFilter's specification language uses differential equations for the process and measurement models and statistical distributions to describe the noise characteristics. It also allows some details of the desired software configuration to be included, such as update intervals and the number of time steps. Figure 4 in Section 6 contains as an example the specification of the DS1 attitude estimator. Note that the specifications are *fully declarative*: they only describe properties of the problem and configuration constraints on the solution but do not contain any low-level implementation details.

From such specifications, AutoFilter derives code implementing the specified task by repeated application of schemas. A *program schema* consists of a parameterized code fragment or template and a set of constraints formalizing the template's applicability to a given task. Schemas represent the different algorithm families of the domain such as information filter, Kalman filter, or particle filter as well as the algorithm alternatives within each family such as standard, sequentialized, and Bierman measurement update. The code fragments are formulated in an intermediate language that is essentially a "sanitized" variant of C (e.g., neither pointers nor side-effects in expressions) but also contains a number of higher-level domain-specific constructs (e.g., vector/matrix operations, finite sums, and convergence-loops). The constraints are formulated in the underlying implementation language (i.e., Prolog) but can use a number of predefined operations (e.g., to construct variable declarations) that keep the formulations compact. Schemas can thus be seen as high-level macros that can be applied to subproblems of a certain structure. Schema application then roughly corresponds to macro expansion: when a schema is applied, code is generated by expanding the template (i.e., instantiating its parameters) over the problem. However, there are three major differences. First, the expansion is conditional and controlled by the constraints; moreover, checking the constraints can further instantiate the template parameters. Second, the expansion can have multiple solutions: when different schemas are applicable to the same problem as for example a linearized Kalman filter and an extended Kalman filter, AutoFilter explores these choices and generates alternative solutions. Third, AutoFilter can perform substantial symbolic calculations (e.g., linearization, discretization, Taylor series expansion) during schema application.

The code fragments resulting from the individual schema applications are assembled and the resulting code is optimized and then translated into a chosen target platform (i.e., language and libraries); currently, AutoFilter supports C/C++ (both stand-alone and with the Matlab and Octave libraries), Ada, 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.

3. Software Assurance

Software assurance approaches can be characterized as *process-oriented* or as *product-oriented*. Process-oriented approaches, which are dominant in safety-critical domains, focus on organizational aspects; they typically restrict the entire software development process and require support documentation to show adherence to the prescribed process model. Product-oriented approaches focus on the produced artifacts, in particular the source code. They cover a variety of techniques, ranging from informal to rigorously formal but typically require the artifacts to satisfy certain criteria. The two approaches are not exclusive of each other, and software assurance processes typically include certain product-oriented steps. Here, we briefly summarize the most common approaches and techniques.

Process Standards

The most widely used standard in AutoFilter's domain is laid out by the FAA-mandated DO-178B [14], which covers all phases of the software development process. DO-178B specifies that the software must be accompanied by various documents (e.g., requirements specifications, system designs, certification plans, etc.) to achieve flight certification. The format of these documents must meet various guidelines; among others, they must be unambiguous, complete, modifiable, and in particular *traceable*. Similarly, system architecture and source code should be "traceable, verifiable, and consistent". The notion of traceability thus links the different phases and abstraction levels together: system re-

quirements to software requirements, high-level requirements to low-level requirements (i.e., system architecture) to source code, and verification of requirements to implementation of software requirements (cf. Section 5). DO-178B does not prescribe how traceability is to be achieved but requires the design description to address how the software satisfies the high-level requirements (including algorithms and data structures, architecture, I/O, data/control flow, and resource limitations), and link the rationale for design decisions to safety considerations. DO-178B does not specify detailed checklists for coding standards, although many common error classes such as memory corruption, out of bounds errors for arrays and loops, and overflows are highlighted, and it stipulates that unnecessary complexity should be avoided.

Tool Qualification— In contrast to the earlier DO-178A, DO-178B recognizes code generation and verification technologies. For tool qualification, DO-178B makes a distinction between tools which can potentially introduce errors into flight code (i.e., development tools) and those which cannot (i.e., analysis tools). Development tools need to be qualified to the same level as the flight software while analysis tools can be qualified more easily.

Software Assurance Techniques

Testing and Simulation— Testing and simulation are the two most common (and basic) software assurance techniques. Testing uncovers both implementation and design errors by executing the software and comparing its outputs to the expected results. This provides only limited assurance because it only demonstrates the presence of errors but not their absence, and because it can require the code to be modified with instrumentation. Simulation executes a *model* of the software and can thus uncover only design errors. In the state estimation domain, filters are typically prototyped and then simulated in a high-level language like Matlab, in order to check that the filter converges with results that are within the desired tolerance, something which is difficult to check statically.

Code Reviews—Detailed manual code reviews are able to detect roughly half the defects which are ultimately discovered [12], making them an effective and important step in software development, despite the fact that they are very cost intensive and require experienced reviewers (cf. IEEE 12207, MIL STD 498, or DO-178B). Code reviews are usually carried out with the help of checklists, which, however, are not standardized and vary widely between different application areas, institutions, and reviewers. A study in the aerospace industry [12] has resulted in a number of important code properties and estimates of how difficult manual checks of each property are.

Program Analysis—Control-flow and data-flow based program analysis techniques were originally developed for application in compilers but can also be used to identify (potential) errors in software by automating some aspects of code reviews, e.g., detecting the use of uninitialized variables. More advanced techniques are usually based on the ideas of *symbolic execution* and *abstract interpretation* and use safe and efficient compile-time approximations to compute the set of values or behaviors that can occur at run time. One of the most advanced static program analysis tools is PolySpace [13]. It analyzes programs for compliance with a fixed notion of safety that includes array bound violations and *nil*-pointer dereferences, and marks unsafe and potentially unsafe program locations in a marked-up browsable format, thus providing a limited form of documentation.

Model Checking— Model checking exhaustively explores the space of transitions between different program states and checks whether this state space is a logical model for a given formula. It is particularly well-suited to finding concurrency errors caused by unexpected interleavings of different program threads, such as deadlocks, because it allows a concise formulation and efficient checking of concurrency properties using temporal logics. Early approaches such as Spin [8] used separate modeling languages but modern software model checkers like JPF [16] work directly on the program code and can thus be considered as an advanced program analysis technique.

Program Verification— In principle, axiomatic program verification techniques offer the highest level of assurance because they construct a detailed mathematical proof of the functional correctness of the software. They apply the inference rules of a program logic to the program and construct a number of logical formulae or *proof obligations*, whose validity imply the correctness of the program. In practice, however, program verification has a number of severe limitations. First, it requires detailed specifications of the program behavior which need to be supplied the developers. Second, the correctness can only be shown with respect to the specification. If the specification contains errors or is incomplete, the assurance provided by the proof can be treacherous. Third, it can produce a large number of proof obligations. Checking their validity requires the application of theorem proving tools which are knowledge- and labor-intensive.

4. SYNTHESIZING CERTIFIABLE CODE

Since a code generator is a software development tool, DO-178B requires its qualification to flight-critical levels before the generated code can be deployed in an aircraft. The goal of this qualification step is to demonstrate that the generator cannot introduce errors into the generated software, which is usually done by showing that produces correct code for all possible inputs. This is still a process-oriented assurance approach—it certifies

the (now fully automated) software development process. However, this is impractical for a large code generator like AUTOFILTER that accepts a wide variety of specifications and can produce a wide range of programs. A more feasible alternative is to employ a product-oriented approach and to certify the generated programs individually.

Certifiable Synthesis Architecture

Formal 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 is a limited variant of full program verification because it only proves individual properties and not the complete behavior, but it uses the same underlying technology. A *certifiable program synthesis system* generates and formally certifies code. Our system comprises a number of different components, as shown in Figure 1. At its core is the original synthesis system. This is extended with a verification condition generator (VCG), simplifier (for the generated verification conditions), and an automated theorem prover (ATP). These components are described below and in more detail in [3], [4], [5].

As in standard in proof carrying code [10], the architecture distinguishes between trusted and untrusted components, shown in Figure 1 in red (dark grey) and blue (light grey), respectively. 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, on the other hand, are not crucial to the assurance because their results are double-checked by at least one trusted component. In particular, the correctness of the certification system does not depend on the correctness of the two largest components: the synthesizer, and the theorem prover; instead, we need only trust the safety policy, the VCG, and the proof checker.

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* [10], [15] to ensure that the compilation step does not compromise the demonstrated safety policy.

Safety Policies

The certification tool automatically certifies that a 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 conditions. These conditions are defined by a set of logical rules and auxiliary definitions, the formal basis of which is explored in [3].

A key feature of our approach is that policies for different areas of concern are kept distinct, rather than amalgamated into a single logical analysis. This is a consequence of the logical framework, but, more importantly, it enables a separation of concerns: different policies can be mixed and matched as appropriate to the certification effort 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 can be 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 although, 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 (*array*) requires each access to an array element to be within the specified upper and lower bounds of the array. Variable initialization-before-use (*init*) 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 (*symm*) requires certain two-dimensional arrays to be symmetric. Sensor input usage (*in-use*) is a variation of the general *init*-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.

Generating Safety Obligations

For certification purposes, the synthesis system *annotates* the code with mark-up information relevant to the selected safety policy. These annotations are part of the schema and thus are instantiated in parallel with the code fragments. The annotations contain local information in the form of logical pre- and post-conditions and loop invariants, which is then propagated through the code. The fully annotated code is then processed by the VCG, which applies the rules of the safety policy to the annotated code in order to generate the safety conditions. As usual, the VCG works backwards through the code and safety conditions are generated at each line. 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, the VCG does not implement

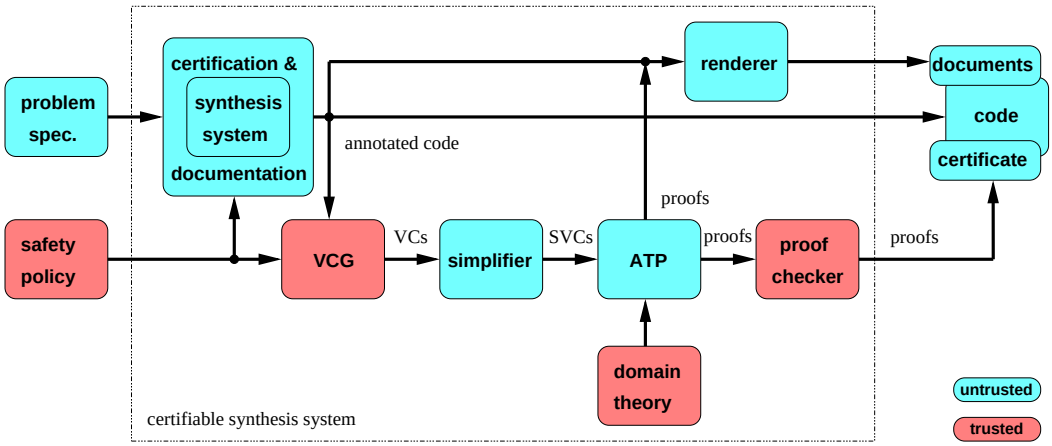


Figure 1. Certifiable program synthesis: System architecture

any optimizations, such as structure sharing on verification conditions (VCs) or even apply any simplifications. Consequently, the generated VCs tend to be large and must be simplified separately; the more manageable simplified verification conditions (SVCs) which are produced are then processed by an automated theorem prover (ATP). The resulting proofs can be sent to a proof checker.

Proof Search and Checking

Essentially, an ATP is a search procedure: it applies the inference rules of its calculus until it either finds a proof or fails because none of the rules are applicable. In order to handle extra-logical operations as for example the arithmetic functions, the ATP needs an additional *domain theory* that specifies their intended meaning as axioms. The domain theory is a trusted component because inconsistencies and errors in the axioms can lead to wrong proofs.

Basic ATPs can be implemented in a few lines of code [2] and can easily shown to be correct, following the same "correct-by-inspection" argument as the VCG. However, the state-of-the-art high performance ATPs in our system use complicated calculi, elaborate data structures, and optimized implementations to increase their power and obtain fast results. This makes a formal verification of their correctness impossible in practice. Although they have been extensively validated by the theorem proving community, the ATPs thus remain a weak link in the certification chain. As an alternative to formal verification, ATPs can be extended to generate sufficiently detailed proofs which can then be independently checked by a small and thus verifiable algorithm. This is the same approach we have taken in extending the synthesis system to generate annotated code, rather than directly verifying the synthesizer. Further discussion of this issue can be found in [4].

Customizability

One of the main advantages of our automated code generator over commercial tools is the degree to which it can be customized. Each of the main subsystems can be tailored and guided in various ways to suit specific application requirements. Users can control the code generator by specifying that particular algorithm schemas should be used, and that the code be implemented using one of the target backends. It is also possible to limit the complexity of the generated code, as required by DO-178B, either by turning off various optimizations (common subexpression elimination etc.), or by specifying that loops should be unfolded where possible. This is similar to the approach taken by the certifying compiler described in [15], which avoids optimizations altogether. Users can also specify which of the various intermediate artifacts generated during synthesis are to appear in the generated documentation. Finally, the safety documentation (which is described in Section 5) can be customized in different ways, including the customization of the safety policies.

5. Documentation Generation

A major drawback of most commercial code generators becomes apparent when their output needs to be inspected or modified: the generated code contains little documentation. In particular, there is no explanation of how each of the statements has been generated, what exactly they do, and how they relate to the original input specification. In AutoFilter, the schema-based synthesis approach is used to automatically generate customized comments together with the generated code. Typically, there is about one line of auto-generated comment to every two to three lines of actual code. These comments describe selected parts of the algorithm, give detailed derivations of mathematical formulas, and relate program constructs and variable names back to the spec-

ification. For this task, AUTOFILTER uses text templates which are instantiated and composed together with the code templates.

In addition to the code, AUTOFILTER also generates a separate documentation suite, which is described below. It contains a standardized software design document, safety certification documentation, and hyperlinked documents to visualize the links between the VCs and the generated code.

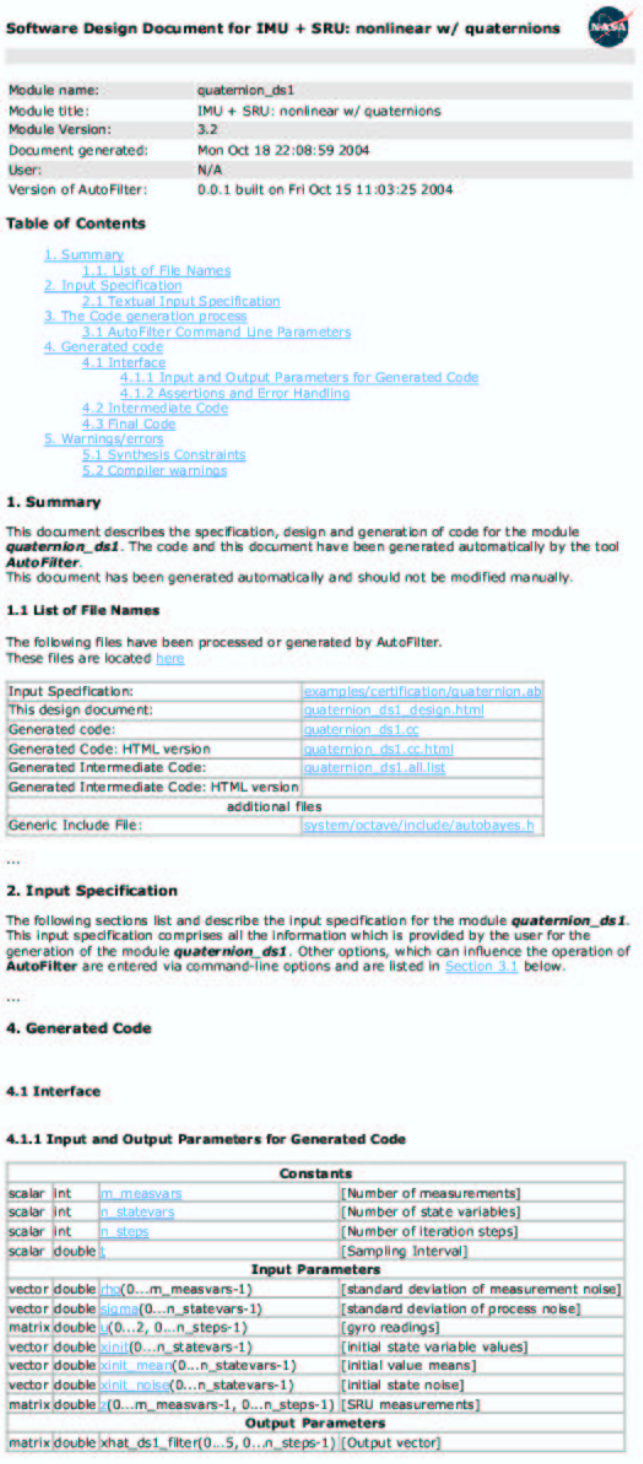
Design Documentation

Most software processes require that a detailed software design document (SDD) is produced for the software. The SDD should contain a detailed description of the software, its interface, special calling conventions, and so on. Since keeping this document synchronized with the actual software is a time-consuming and error-prone task in practice, AUTOFILTER has been extended to automatically generate SDDs together with the code from a single high-level specification. This eliminates version problems and inconsistencies between code and documentation. The format of the SDD follows different NASA and ANSI standards. It contains an interface description, administrative information (names of files, versions, etc.), specific input and output constraints, and synthesis and compiler warnings. The document is hyperlinked to the input specification, the generated code, and any intermediate artifacts generated during synthesis. Figure 2 shows some excerpts from an example SDD. Since the design document is generated at synthesis time, it is able to include design details which would be difficult to infer after the code has been generated.

Safety Documentation

In our formal certification approach evidence for the freedom from certain error classes is given as a mathematical proof. However, these "proofs" are unlikely to be recognized as such by even a mathematician, since they consist primarily of the low-level steps carried out by an automated theorem prover. Moreover, standalone proofs themselves—even if they are on a higher level of mathematical abstraction—are unlikely to be of much interest to engineers because they do not explicitly refer back into the program. What is missing is a trace between the verification conditions and the program being certified.

A more general point is that sophisticated analysis techniques need to be balanced with more user-friendly overviews. The increasing use of theorem provers in both software and hardware verification presents a problem for the applicability of formal methods: how can such specialized tools be combined with traditional process-oriented development methods?



Software Design Document for IMU + SRU: nonlinear w/ quaternions

Module name:	quaternion_ds1
Module title:	IMU + SRU: nonlinear w/ quaternions
Module Version:	3.2
Document generated:	Mon Oct 18 22:08:59 2004
User:	N/A
Version of AutoFilter:	0.0.1 built on Fri Oct 15 11:03:25 2004

Table of Contents

1. Summary

This document describes the specification, design and generation of code for the module ***quaternion_ds1***. The code and this document have been generated automatically by the tool ***AutoFilter***.
This document has been generated automatically and should not be modified manually.

1.1 List of File Names

The following files have been processed or generated by AutoFilter.
These files are located here

Input Specification:	examples/certification/quaternion.ab
This design document:	quaternion_ds1_design.html
Generated code:	quaternion_ds1.cc
Generated Code: HTML version	quaternion_ds1.cc.html
Generated Intermediate Code:	quaternion_ds1.all.list
Generated Intermediate Code: HTML version	
additional files	
Generic Include File:	system/octave/include/autobayes.h

...

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_measvars	[Number of measurements]
scalar	int	n_statevars	[Number of state variables]
scalar	int	n_steps	[Number of iteration steps]
scalar	double	t	[Sampling Interval]
Input Parameters			
vector	double	rho(0...m_measvars-1)	[standard deviation of measurement noise]
vector	double	sigma(0...n_statevars-1)	[standard deviation of process noise]
matrix	double	u(0...2, 0...n_steps-1)	[gyro readings]
vector	double	xinit(0...n_statevars-1)	[initial state variable values]
vector	double	xinit_mean(0...n_statevars-1)	[initial value means]
vector	double	xinit_noise(0...n_statevars-1)	[initial state noise]
matrix	double	z(0...m_measvars-1, 0...n_steps-1)	[SRU measurements]
Output Parameters			
matrix	double	xhat_ds1_filter(0...5, 0...n_steps-1)	[Output vector]

Figure 2. Generated Software Design Document (excerpts).

We address these issues by combining our documentation generation with the formal certification. We have developed a generic framework [6] for generating textual explanations for *why* a program is safe. We use the information obtained from the mathematical analysis of the software to produce a detailed textual justification

of compliance with the given safety policy, and trace these proofs back to the corresponding parts of the program. The system is based on customizable explanation templates which convert logical entities that appear in the verification conditions, such as safety conditions and loop invariants, into text. It uses labels which are propagated through the VCG and into the VCs so that the proofs of safety can explicitly refer back to program components. Our framework is generic in the sense that we can instantiate the system with a range of different safety policies, and can easily add new policies to the system. Figure 3 shows an excerpt from an automatically generated safety document for the array-bounds safety of a synthesized program.

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 explanation for *array* safety policy

One of the problems is that safety documents can potentially contain a huge amount of information, so it is essential to focus attention where it is needed. There are two ways of doing this. First, the user can restrict attention either to specific program variables or to certain lines of code so that, in effect, the system does a *slice* of the program. Second, the system has a heuristic ordering of the importance of various pieces of information (for example, information which depends on a loop invariant is more important that that which comes from an assignment, say) so that the user can set the level of importance they care about.

Tracing VCs to Code

Our formal certification approach works by applying a VCG to the generated annotated code, generating a number of VCs, and then sending them to an automated theorem prover. A VC can fail to be proven for a number of reasons. First, there may be an actual safety violation in the code. Second, the generated annotations may be insufficient. The schema only contains annotation templates that are specific to a given safety policy. Third, the theorem prover may time-out, either due to the size and complexity of the VC, or due to an incomplete domain theory.

The annotations, which consist of pre- and post-conditions and loop invariants, need to be propagated throughout the code. Errors can come from any part of the template, or from the propagation phase: an annotation might not be propagated far enough, or it might be propagated out of scope. Since the annotations are automatically generated and propagated, it can thus be difficult to determine whether and where they must be modified. If any of these cases happens, debugging an attempted certification by manually tracing the VCs back to their source is quite difficult. The verification process is inherently logically complex, and the VCs can be very large. The VCs go through substantial simplifications, after which they are typically of the form (see [4]):

$$hyp_1 \wedge \cdots \wedge hyp_n \Rightarrow conc$$

where a hypothesis *hyp* is either a loop invariant, an index bound, or a propagated assumption, and the conclusion *conc* is either an annotated assertion or a generated safety condition. Hence, a single VC can depend on a variety of information distributed throughout the program.

Because of these difficulties, we have implemented an automated linking feature. The synthesized program is displayed in a web browser along with its VCs in a separate frame. Both are hyperlinked so that if the user clicks on a VC, the lines of program which correspond to it will be highlighted; similarly, if the user clicks on a line of the program, the browser displays all the related VCs. This is achieved with the same approach described in the previous section: the generated code is marked up with labels, which are passed through the different steps of the certification system so that the VCs can refer back to the code. The crucial aspect is to maintain these labels during the simplification step.

Linking VCs to code is also useful for safe programs. Indeed, such traceability is an important part of the assurance provided by the formal certification process, and directly addresses the tracing from the *verification* of requirements to their *implementation*, as mandated by DO-178B.

6. Case Studies

We have tested our certifiable synthesis approach in two different "after-the-fact" case studies, where we recreated and certified Kalman filters from the requirements of existing applications. In the first case study (`ds1`), 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, which is shown in Figure 4. It combines inputs from an inertial measurement unit (IMU) and a star tracker or stellar reference unit (SRU) to obtain a more accurate estimate of the attitude of the spacecraft. The estimator has three state variables representing change in spacecraft attitude since the last measurement, and three state variables representing the IMU gyro drift. It models the IMU-readings `f` as a driving function in the process model, and only considers the SRU-readings `z` as measurements. This is a standard technique when essentially the same quantities are read from different sensors [9, Section XI]. For this specification, AutoFilter generates code based on an extended Kalman filter. A more detailed description

```
model ds1.

% Process model: ẋ = F_t x + u
% Process noise: u ~ N(0, q · I) ⇔ u_i ~ N(0, q_i)

const nat n := 6 as '# state variables'.
data double f(1..3, time) as 'IMU readings'.
double x(1..n) as 'state variable vector'.
double u(1..n) as 'process noise vector'.
double q(1..n) as 'variance of process noise'.
u(I) ~ gauss(0, q(I)).

equations process_eqs are [
  dot x(1) := (hat x(4) - x(4)) - u(1)
              + x(2) * (f(3,t) - hat x(6))
              - x(3) * (f(2,t) - hat x(5)),
  dot x(2) := ...
  dot x(3) := (hat x(6) - x(6)) - u(3)
              + x(1) * (f(2,t) - hat x(5))
              - x(2) * (f(1,t) - hat x(4)),
  dot x(4) := u(4),
  dot x(5) := u(5),
  dot x(6) := u(6)
].

% Measurement model: z = x + v
% Measurement noise: v ~ N(0, r · I) ⇔ v_i ~ N(0, r_i)

const nat m := 3 as '# measurement variables'.
data double z(1..m, time) as 'SRU readings'.
double v(1..m) as 'measurement noise vector'.
double r(1..m) as 'variance of measurement noise'.
v(I) ~ gauss(0, r(I)).

equations measurement_eqs are [
  z(1,t) := x(1) + v(1),
  z(2,t) := x(2) + v(2),
  z(3,t) := x(3) + v(3)
].

% Filter configuration
const double delta := 1/400 as 'Interval'.
units delta in seconds.
...

estimator ds1_filter.
  ds1_filter::process_model     ::= process_eqs.
  ds1_filter::measurement_model ::= measurement_eqs.
  ds1_filter::steps             ::= 24000.
  ds1_filter::time              ::= t.
  ds1_filter::update_interval   ::= delta.
  ds1_filter::initials          ::= xinit(_).
output ds1_filter.
```

Figure 4. Example specification for DS1

of the DS1 attitude estimator and the code derivation process can be found in [17]; note, however, that AUTOFILTER's specification language has evolved and the specifications thus differ slightly. The second case study (`iss`) was taken from a simulation from the space shuttle docking procedure at the International Space Station, for which a different configuration of an extended Kalman filter is generated.

Table 1 summarizes the number of verification conditions and the proof efforts involved in certifying the two extended Kalman filters generated from the `ds1`- and `iss`-specifications for the four different safety policies described in Section 4. The lines of code (LoC) are broken down into lines of (executable) code and lines of annotations. The time T_{synth} to synthesize the code and the time T_{proof} to prove the VCs using the automated theorem prover (e-setheo) are measured on a 2.4GHz standard PC.

Table 1. Certification results and times (same algorithm, different policies)

Spec.	Policy	LoC	T_{synth}	#VC	T_{proof}
`ds1`	*array*	438 + 0	6.7	1	0.8
	init	438 + 84	10.4	74	79.1
	in-use	438 + 58	7.6	21	254.5
	symm	444 + 77	68.1	865	838.7
`iss`	*array*	788 + 0	31.2	4	3.5
	init	788 + 88	39.1	71	85.7
	in-use	788 + 62	32.3	1	34.2
	symm	799 + 79	65.8	480	528.6

Table 2. Certification results and times (same policy, different algorithms)

Spec.	Alg.	LoC	T_{synth}	#VC	T_{proof}
`ds1`	`par`	438 + 0	6.7	1	0.8
	`seq`	484 + 0	8.4	1	0.7
	`bier`	589 + 0	29.0	6	5.0
`iss`	`par`	788 + 0	31.2	4	3.5
	`seq`	808 + 0	32.8	3	2.6
	`bier`	891 + 0	66.4	9	7.9

It can be seen that the proportion of annotations to executable code and consequently the number of VCs, varies widely depending on the safety policy. Language-specific policies tend to be easier than domain-specific ones, which require more detailed annotations. The complexity of the generated VCs is well within the capabilities of current ATPs. For three of the four policies (*array*, *init*, and *symm*) e-setheo was able to discharge all obligations with average proof times of approximately one second. For the *in-use* policy, the system produces one unprovable obligation for each of the programs, which take much longer to detect and thus distort the average and total proof times. However, it is important to notice that unprovable obligations 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 *in-use* safety policy is formulated.

In Table 2, in contrast, we compare the effort in certifying three algorithmic variants of the extended Kalman filters generated from these two specifications, for a single safety policy (*array*). These variants differ in how the measurement update is implemented. `par` refers to the default algorithm generated by our system, where the updates of the filter loop are expressed in terms of matrix operations and, in effect, take place in parallel. In `seq`, however, the measurements are processed se-

quentially, one at a time, which is possible when the measurements are not correlated. This is more efficient since matrix inverses can then be avoided. `bier` also processes the measurements sequentially, but uses the Bierman update [7]. This is an example of a square-root filter, where UD decomposition of the covariance matrix is used to minimize error propagation. The Bierman update is more algorithmically complex and so produces a greater number of proof obligations. Again, e-setheo was able to discharge all VCs easily.

7. CONCLUSIONS

We have described our state-of-the-art AUTOFILTER program synthesis system. It uses a novel combination of synthesis, verification and documentation for ultra-reliability and has been designed so that the certification subsystem is an integral part of the entire synthesis system. We believe that documentation and certification capabilities such as this are essential for formal techniques to gain acceptance, and provide an approach to merging automated certification with traditional certification procedures.

For future work, in addition to continually increasing the system's synthesis power (with more algorithmic schemas, more specification features, and allowing more control over the derivation), we plan to extend it in two main areas.

First, we are developing a more declarative and explicit modeling style. Much of the domain knowledge used by the system in deriving code is currently implicit; by making it explicit this can be used to (among other things) facilitate traceability between the code and its derivation in the generated documentation.

Second, we continue to extend the certification power of the system (with more policies, more automation, and a more integrated approach to the documentation generation subsystems). We are also investigating a software certificate management system, which will keep track of the information used during certification, and will enable and then record audits of the software certificates.

REFERENCES

[1] I. Bate, N. Audsley, and S. Crook-Dawkins. Automatic code generation for airborne systems – the next generation of software productivity tools. In *Proceedings of IEEE Aerospace Conference*, pages 11–19. IEEE, 2003.

[2] B. Beckert and J. Posegga. leanT^AP: Lean tableau-based deduction. *J. Automated Reasoning*, 15(3):339–358, 1995.

[3] E. Denney and B. Fischer. Correctness of source-level safety policies. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proc. FM 2003: Formal Methods*, volume 2805 of *Lect. Notes Comp. Sci.*, pages 894–913, Pisa, Italy, Sept. 2003. Springer.

[4] E. Denney, B. Fischer, and J. Schumann. An empirical evaluation of automated theorem provers in software certification. In *Proceedings of the IJCAR 2004 Workshop on Empirically Successful First Order Reasoning (ESFOR)*, 2004.

[5] E. Denney, B. Fischer, and J. Schumann. Using automated theorem provers to certify auto-generated aerospace software. In *Proceedings of the 2nd International Joint Conference on Automated Reasoning (IJCAR'04)*, volume 3097 of *Lect. Notes Artificial Intelligence*, pages 198–212, Cork, Ireland, 2004.

[6] E. Denney and R. P. Venkatesan. A generic software safety document generator. In *Proceedings of the 10th International Conference on Algebraic Methodology and Software Technology, AMAST'04*, pages 102–116, Stirling, Scotland, 2004.

[7] M. S. Grewal and A. P. Andrews. *Kalman Filtering: Theory and Practice Using MATLAB*. Wiley Interscience, 2001. 2nd edition.

[8] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[9] E. Lefferts, F. Markley, and M. Shuster. Kalman filtering for spacecraft attitude estimation. *Journal of Guidance and Control*, 5:417–429, 1982.

[10] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In K. D. Cooper, editor, *Proc. ACM Conf. Programming Language Design and Implementation 1998*, pages 333–344, Montreal, Canada, June 17–19 1998. ACM Press. Published as SIGPLAN Notices 33(5).

[11] S. Nelson, E. Denney, B. Fischer, J. Richardson, J. Schumann, and P. Oh. NASA/CR Product-oriented Software Certification Process for Software Synthesis. Technical report, NASA Ames, 2003.

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

[13] PolySpace Technologies, 2003. `http://www.polyspace.com`.

[14] RTCA Special Committee 167. Software considerations in airborne systems and equipment certification. Technical report, RTCA, Inc., Dec. 1992.

[15] V. Santhanam. The anatomy of an FAA-qualifiable Ada subset compiler. In *Proceedings of the 2002 annual ACM SIGAda international conference on Ada*, pages 40–43, Houston, Texas, USA, 2002. ACM Press.

[16] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2), 2003.

[17] J. Whittle and J. Schumann. Automating the implemen-

tation of Kalman filter algorithms. *ACM Transactions on Mathematical Software*, 2005. To appear.

[18] J. Wielemaker. *SWI-Prolog 5.2.9 Reference Manual*. Amsterdam, 2003.

Dr. Ewen Denney *(PhD University of Edinburgh, 1999) has published over 20 papers in the areas of automated code generation, software modeling, software certification, and the foundations of computer science. He has been at NASA Ames for two years, where he has been mainly involved in developing the certification subsystem for* AUTOFILTER.

Dr. Bernd Fischer *(PhD University of Passau, 2001) is a Research Scientist in the Automated Software Engineering Group, NASA Ames. He is engaged in research on automatic program generation and on certification techniques for aerospace software. He has published more than 40 papers in areas including component retrieval, program synthesis, and program transformation.*

Dr. Johann Schumann *(PhD 1991, habilitation degree 2000) is a Senior Scientist in the Automated Software Engineering Group, NASA Ames. He is engaged in research in automatic program generation and on verification and validation of adaptive controllers and learning software. Dr. Schumann is author of a book on theorem proving in software engineering and has published more than 60 articles on automated deduction and its applications, automatic program generation, and neural network oriented topics.*

Dr. Julian Richardson *(PhD University of Edinburgh, 1995) is a Research Scientist in the Automated Software Engineering Group, NASA Ames. He is engaged in research on automatic program generation, in particular of Kalman filters, and in research on the effectiveness of verification and validation techniques for aerospace software. He has published more than 25 papers in areas including automated theorem proving, program synthesis and transformation.*