Rapid Exploration of the Design Space During Automatic Generation of Kalman Filter Code

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

Abstract— State estimation is a core capability for autonomous systems such as satellites and planetary rovers. Kalman filters provide a computationally efficient way to determine the values of state variables (e.g. position, velocity) from noisy measurements.

The Automated Software Engineering group at NASA Ames Research Center has previously developed a number of systems for generating program code in NASA-relevant domains, including AMPHION/NAIF [12] for generating code for mission planning and AUTOBAYES [6] for generating data analysis code.

In this paper[1][2], we outline a program generation system, AUTOFILTER [13], which has been developed at NASA Ames Research Center. AUTOFILTER synthesizes (i.e. generates) Kalman filter code. It takes as input a textual specification — a description of the mathematical model underlying the Kalman filter — and automatically generates code suitable for compilation using MATLAB libraries (for prototyping and testing the filter) or standalone C code (for deploying the filter).

We describe how AUTOFILTER assists the iterative development of Kalman filters in various ways: permitting changes in the mathematical model underlying the filter to be rapidly realized as code and tested, different Kalman filters to be synthesized from the same model, code automatically assessed for computational performance, and approximating assumptions applied to the code in order to improve efficiency. AUTOFILTER provides assistance for ensuring that the generated code is correct by generating program documentation and correctness certificates in addition to the code itself.

TABLE OF CONTENTS

[1]IEEEAC Paper Number 1205
[2]0-7803-8870-4/05/$20.00(C) 2005 IEEE

1. INTRODUCTION

State estimation is a core capability for autonomous systems such as satellites and planetary rovers. Kalman filters provide a computationally efficient way to determine the values of state variables (e.g. position, velocity) from noisy measurements. In a Kalman filter, the relationship between the measurements and the state variables which are to be estimated is encoded by a set of *measurement equations*. The behavior of the system is encoded by a set of *process equations*, which express how the state variables are expected to evolve, typically with time. The other main parts of the Kalman filter model are matrices representing the covariance structure of the measurement and state variables, and the initial values of these and the state variables.

Implementing a Kalman filter involves a number of trade offs: a simple model with a small number of state variables may not be able to accurately model the behavior of the system — it may be necessary to increase the number of state variables in order to achieve good estimates. Due to the heavy use of matrix operations (including multiplication and inverse), increasing the number of state variables in a Kalman filter very significantly increases the computational complexity of the filter. Code deployed in radiation hard environments must satisfy tough timing constraints: compare Mars Pathfinder's 20 MHz RAD6000 processor running at 2 MIPS with a 1 GHz desktop Pentium III running at 3000 MIPS. In addition, problems may arise in which some state variables become unobservable and their values no longer adequately constrained by the input measurements.

Implementation of Kalman filters for new aerospace applications is an iterative process. Typically, the filter is prototyped in a simulation environment, such as MATLAB, until a satisfactory mathematical model has been found. This is then recoded for deployment on a target platform in a low-level language such as C. This code is tested and adjusted until its computational characteristics (e.g. runtimes or numerical accuracy) meet the requirements. This may lead to a redesign of the mathematical model. AUTOFILTER can simplify this process in several ways: It can rapidly turn around model changes, which allows the exploration of the filter design space, it can generate multiple alternative implementations for the same specification, which allows the exploration of the implementation design space, it can generate both MATLAB code (for prototyping) and target platform code (for deployment) from the same specification, it can apply simplifying assumptions to the generated code in order to reduce its

worst case execution time, it can generate documentation at the same time as code, and it can automatically demonstrate that the code it generated is free from various classes of defects.

In this paper we briefly describe our system, AUTOFILTER, which automates the synthesis (i.e. generation) of Kalman filter code from succinct specifications. We outline the various ways in which AUTOFILTER aids the iterative development of Kalman filters.

The paper is structured as follows: in §2, we give a high level overview of AUTOFILTER. We then (§3) describe how Kalman filters are developed iteratively, and summarize the capabilities provided by AUTOFILTER to assist steps in the iterative process. We provide more detail on some of these capabilities, including the benefits of code derivation from a high level specification, generation of multiple implementations from the same specification, estimation of worst case execution time, application of approximations, use of different back ends to generate both prototype and deployed code, and automatic certification and documentation of the generated code in order to ensure correctness. In §5 we discuss directions for further work, and conclude the paper in §6.

2. OVERVIEW OF AUTOFILTER

In this section we outline how AUTOFILTER implements program synthesis, and provide an example.

Anatomy of AUTOFILTER

AUTOFILTER takes as input a textual specification of a Kalman filter, and automatically outputs a program (by default in C using the OCTAVE matrix libraries — a different target language or library can be requested using command line flags) implementing a Kalman filter meeting that specification. Figure 3 (see the next subsection for details) shows part of a specification describing the core of the Deep Space 1 attitude control system.

Figure 1 depicts the structure of the AUTOFILTER system. Synthesis proceeds as follows:

- The specification is parsed and stored internally.
- Code is generated in a simplified imperative language called the *intermediate language*.
- Approximations, if specified, are applied to simplify the intermediate code.
- The intermediate code is checked to ensure it meets runtime requirements.
- Code in the requested target language is generated by the back end.

In more detail, first, the specification is read and stored internally in what we call the *model*. The core of the synthesis system (middle large box) implements *schema-guided synthesis* [7]. A synthesis schema essentially consists of a code template, plus preconditions which restrict its applicability and logic which instantiates the code template to yield a well-formed code fragment. Schema-guided synthesis is very naturally implemented in Prolog, a logic programming language which searches for alternative solutions to a problem using a process called *backtracking*.

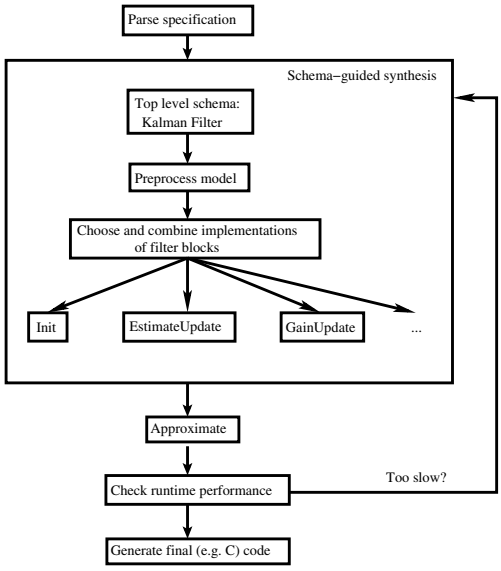

Figure 1. Structure of AUTOFILTER. The main part of the synthesis takes place in the middle large box.

A schema implementing a part of a Kalman filter effectively encodes the knowledge which we would expect to find in a text book description of that implementation technique: when engineering a schema-guided synthesis system, a good book is extremely useful, e.g. [2], [8]. Each schema maps a program synthesis task to a code fragment which performs that task. Each schema has preconditions, for example the schema outlined in Figure 2 applies only to discrete Kalman filters or continuous Extended Kalman filters. If a schema's preconditions hold, then it first generates names for program variables it will use, then generates and returns a code fragment. If the preconditions do not hold, the schema fails and backtracking causes the system to try an alternative schema.

In AUTOFILTER, there is currently a single top level schema, which carries out some preprocessing of the model — in particular linearization of the process equations when they are non-linear, for example in the case of an extended Kalman filter — and determines how the synthesis of a filter should be broken down into the synthesis of a number of simpler components. Each of these simpler components, which in our current implementation correspond closely to different functional blocks of a Kalman filter, may have a number of schemas which can generate alternative implementations.

If code fragments for all the components can be synthesized,

schema propagate(state)
preconditions:
continuous extended KF or discrete KF
program variables:
input: posterior state estimate, $\hat{x}$,
state transition matrix, Φ,
time between updates, δ, driving function, u
output: prior state estimate, $\hat{x}^-$
local: none
...generation of code fragment...

Figure 2. Outline of a schema for synthesizing code for state propagation.

they are composed by the top level schema into code implementing a Kalman filter. By default, the synthesized code is a function which loops over all measurements (provided as a matrix input to the program), returning as output a matrix of all state estimates. The code is expressed in a simplified imperative language called the *intermediate language*. The intermediate language contains constructs for loops, variable and function declarations and calls etc, but avoids complex features of real programming languages like tests with side effects. The intermediate code is translated by the back end into the desired target language, e.g. C. The economy of the intermediate language facilitates its analysis and manipulation, and simplifies the back-end translation.

The modularity of schema-guided synthesis makes it relatively easy to extend the system to generate alternative implementations of the basic Kalman filter blocks, for example using a different equation for the Kalman gain. More profound extensions may require new top level schemas. For example, we plan to implement a new top level schema to generate code for unscented Kalman filters [11].

An Example

The Deep Space I (DS1) probe was deployed as a testbed for a range of experimental NASA technologies. In the Summer of 2002, we performed a case study to reverse engineer the DS1 attitude estimation code. DS1's attitude was estimated by combining measurements from an IMU (Inertial Measurement Unit) with those from and a stellar reference unit (SRU, i.e. a star tracker). The Kalman filter is based on [9, Section XI], and has three state variables representing change in spacecraft attitude since the last measurement, and three state variables representing gyro drift. It is standard when essentially the same quantities are read from different sensors to incorporate some sensor readings into the process model as driving functions, considering others as the measurements in the measurement model. In this filter, readings from the IMU are modeled as a driving function in the process model, while readings from the SRU are the measurements.

```
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 architecture
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 3. Excerpt from a specification of the DS1 attitude control system.

An extended Kalman filter is automatically synthesized from the specification. The synthesized code was compared to the original DS1 attitude estimation code. Parts of the code from the deployed DS1 state estimator were manually replaced by code synthesized from the above specification, and both versions run in the DS1 simulator. The estimates produced by both versions were found to be essentially identical.

3. ITERATIVE DEVELOPMENT OF KFS

Implementation of Kalman filters for new aerospace applications is an iterative process. Typically, the filter is prototyped in a simulation environment such as MATLAB until a satisfactory mathematical model has been found. At this stage,

decisions are made as to whether and how to simplify or otherwise manipulate the process and measurement equations, what kind of Kalman filter to use (an extended Kalman filter, or a linearized one, continuous or discrete etc), what kind of updates to use (simultaneous updates to cope with possibly correlated measurement noise, Bierman, Carlson square root etc). Once the prototype filter has been tested, it is then recoded for deployment on a target platform in a low-level language such as C. The code is tested and adjusted until its computational characteristics (e.g. runtimes or numerical accuracy) meet the requirements. In the worst case, this can lead to a redesign of the mathematical model. AUTOFILTER can simplify this process in several ways: It can rapidly turn around model changes, which allows the exploration of the filter design space, it can generate multiple alternative implementations for the same specification, which allows the exploration of the implementation design space, it can generate both MATLAB code (for prototyping) and target platform code (for deployment) from the same specification, it can generate documentation at the same time as code, and it can automatically demonstrate that the code it generates is free from various classes of defects. In more detail, this process may involve:[3]

1. Develop model: dynamics (process equations and state variables), relationship of sensor inputs to state (measurement equations), noise characteristics and filter initial conditions. Development is iterative:
   (a) Mathematical modeling.
   (b) Choice of implementation, e.g. square root filtering to improve numerical stability.
   (c) Prototype, e.g. implement filter as a MATLAB function.
   (d) Simulate and test filter and assess performance.

Synthesis helps by: Automatic derivation of prototype code from model (specification) — changes in model are quickly reflected in changes in the prototype. Manually or automatically choosing implementation from a library of schemas — multiple implementations can be derived from the same specification. The process and measurement models, together with initial values of the state variables and declared noise variables, can be used to derive simulated input data for testing purposes.

2. Improve computational performance of filter if necessary:
   (a) Code level optimizations.
   (b) Use of approximations.
   (c) Reduction in number of state variables.

Synthesis helps by: automating optimizations, including common subexpression elimination, loop unfolding, and replacement of $1 \times n$ *and* 1×1 *matrices by vectors and scalars. Simplification of code using approximations stated in the specification, with statements optionally incorporated to check that the approximation is within required error bounds.*

3. Interface to target system:
   (a) Use of appropriate data types.
   (b) Use of correct function/method prototypes.
   (c) Use of appropriate library functions.

Synthesis helps by: automatic derivation of code for various implementation languages and target architectures from the same specification used for prototyping. High level 'specification of how to divide Kalman filter implementation into functional blocks.

4. Refine data handling:
   (a) Buffering.
   (b) Windowing.
   (c) Definition of what constitutes bad data, and what to do with it.
5. Validate and ensure correctness of implementation.
   (a) Testing.
   (b) Code review.
   (c) Use of verification tools, e.g. static analysis.

Synthesis helps by: certification automatically inserts annotations into the synthesized code which are later used in static analysis to formally guarantee safety properties such as non-violation of array bounds, and initialization of variables before use. Testing is assisted by automatic generation of test data. The generated code is designed to be human-readable, and is commented. Design documents can be generated with the code. These help code review and ascertaining fitness for purpose.

4. AIDING DESIGN SPACE EXPLORATION

In this section, we present in more detail some of the ways in which synthesis assists exploration of the implementation design space.

High Level Specification

AUTOFILTER's specification language is designed to allow the succinct expression of the governing equations for a Kalman filter. Where possible, the specification is declarative, i.e. it separates the filter's model from how it is implemented. Significant changes in the filter's implementation can be achieved with small changes in the specification.

For example, the following specifies a process model in which differential readings `omega(0,t)..omega(2,t)` from a gyroscope are integrated into three state variables `x(0) .. x(2)`, with process noise `eta(0)..eta(2)`.

```
equation_set process_model has
 [
 dot x(0) := omega(0,tvar) + eta(0),
 dot x(1) := omega(1,tvar) + eta(1),
 dot x(2) := omega(2,tvar) + eta(2)
 ].
```

[3] this is not meant to be an exhaustive list

The model above not take account of gyro drift, which is conventionally modeled by the addition of variables representing gyro offsets which vary along a random walk. This is easily added to the AUTOFILTER specification:

```
equation_set process_model has
 [
 dot x(0) := x(3) + omega(0,tvar) + eta(0),
 dot x(1) := x(4) + omega(1,tvar) + eta(1),
 dot x(2) := x(5) + omega(2,tvar) + eta(2),
 dot x(3) := eta(3),
 dot x(4) := eta(4),
 dot x(5) := eta(5)
 ].
```

Without carrying out large-scale user studies, it is difficult to provide an accurate measure of the savings obtained by using synthesis from a high-level specification, compared to manual coding.[4] One measure which we expect to be correlated with effort saving is provided by comparing the effort necessary to alter a filter following changes in the underlying mathematical model, using synthesis compared to manual coding. For this, we can count the number of changes made in the specification to the number of changes in the induced program. Not counting changes in temporary variable names or formatting, adding gyro drift to the model as described above requires changing 9 lines in the specification (addition of the new process equations, change to the number of state variables, initialization for the new covariance matrix entries). The corresponding synthesized programs have 135 differing lines (change in the number of state variables — appears in several places, new initialization statements for most internal matrices, some loops are unfolded which were previously not unfolded). Thus, for even this fairly simple change, significant leverage is obtained by updating the specification rather than a program which implements the specification.

A more complex change is to use a Bierman measurement update rather than a standard measurement update. This is achieved without changing the specification (rather, a command line flag instructs the synthesis system to use the Bierman update), and results in a synthesized program with 170 changed lines compared to the standard update.

It seems reasonable to conclude, therefore, that modifications to the Kalman filter's model which fit well with the specification language can be much more economically made at the specification level than at the code level. This raises the question of the adequacy of the specification language: can it express succinctly the kinds of Kalman filters which are usually required in aerospace applications? We have some evidence in favor of its adequacy: we have successfully applied AUTOFILTER to a number of case studies and text book examples.

[4]Note that we should assess cost savings achieved across the entire software lifecycle, including code debugging, maintenance, review etc.

Different Kinds of KF

Synthesis of Kalman filters in AUTOFILTER is performed in a modular way: the top level Kalman filter schema performs some preprocessing on the model (obtained from the input specification), employs a number of subschemas to synthesize code fragments for the different phases of the Kalman filter processing, and knits these code fragments together to obtain a complete implementation of the Kalman filter. For a discrete Kalman filter, these subschemas correspond roughly to the blocks of [2, Figure 5.8]: computing the Kalman gain, updating the state with the measurements, computing the error covariance for the updated estimate, and projecting the state and error covariance ahead to the next time step.

Different implementations of a Kalman filter specification are obtained by varying the schemas used to implement its component blocks. There are schemas for several kinds of square root update, for example Bierman measurement update. There are also schemas for propagating estimates and covariances by integration to allow for degenerate Kalman filters in which there are no measurements, as for example used in [1].

Given a number of alternative schemas implementing a given part of a filter, AUTOFILTER can either be allowed to try them all, possibly returning alternate synthesized programs on backtracking, or can be directed to use a particular one using command-line flags.

Estimation of Worst Case Execution Time

Calculating the complexity of a program is a well-established technique for obtaining estimates (usually, upper bounds) on the amount of time (alternatively, space) which will be necessary for a program to execute. The complexity is usually presented as a function of the sizes of the input arguments of the program. For example, the complexity of Quicksort is $O(n\,log(n))$. The "big O" form of complexity analysis is not very appropriate for judging the real-time performance of programs — depending on the constants involved, an $O(n^3)$ algorithm may be much better in practice than an $O(n^2 log(n))$ algorithm.

Accurate analysis of worst case execution time is in general complex, since it must take into account processor architectures including the use of processor and memory cache [5]. In AUTOFILTER, we simplify the analysis by calculating the worst case execution time for intermediate code (i.e. irrespective of implementation language or platform). This analysis serves to provide a basis for guiding synthesis towards the generation of efficient code. Worst case execution time is calculated by abstract interpretation of the synthesized intermediate code. Each program construct has an associated rule expressing an upper bound on its worst case execution time as a function of the worst case execution time of its component parts. For example, if $|\,\mathtt{E}\,|$ denotes an upper bound on

the execution time of a statement or expression E, equations (1,2) express upper bounds on the execution time of an `if` statement and a `for` loop in terms of their parts:

$$|if\ (testexpr)\ then\ stmt1;\ else\ stmt2;| \leq 1 + |testexpr| + max(|stmt1|, |stmt2|) \quad (1)$$

$$|for\ I := Lo\ to\ Hi\ \{stmt\}| \ \leq\ (Hi - Lo + 1).|stmt| \quad (2)$$

This analysis is much simpler than for arbitrary code written in C or any comparable imperative language, since the intermediate language avoids difficult language features — for example, the intermediate language does not allow side effects in loop tests (enabling analysis of the *if* statement to be decomposed into independent analyses of the loop test, *then* and *else* parts) — and we can generally ensure that the schemas do not synthesize code which is hard to analyze. The second rule (2) assumes that the loop bounds are known at analysis time. For general programs, this assumption is easily violated, but for the intermediate code we analyze, quantities such as array sizes, which would often be parameters of a hand-coded program can be numeric constants in the specification and the analyzed program — these quantities can be changed when necessary by changing the specification and resynthesizing. Rule (2) also assumes that an upper bound on the execution time of the loop body can be computed which is independent of the value of the loop index I. Again, for general programs this will often not be possible, but for our synthesized code it often is, since we can ensure that the schemas avoid synthesizing problematic constructs such as nested loops.

Using such rules, an upper bound on the execution time of a program is computed as a function of the execution time of basic arithmetic expressions $sin, cos, +, exp$ etc. A performance model gives the execution time (in cycles) of these basic expressions. The upper bound expression can therefore be evaluated to an integer, permitting synthesized programs to be assessed for worst case execution time in a meaningful way. The performance model used in testing[5] specifies that basic arithmetic operations, for example +, *, complete in a single cycle, whereas complex floating operations, for example sin, cos, exp, require 200 cycles.

AUTOFILTER's specification language has been extended with a construct for specifying the maximum permissible execution time. If analysis of the synthesized program gives a worst case execution time greater than this maximum, the synthesized program is rejected, and AUTOFILTER automatically tries to synthesize an alternative program which may be able to meet this requirement.

The upper bound on execution time is expressed as the maximum number of cycles allowed for a single complete run of the synthesized program. For example, the following is from a specification which tested estimation of worst case execution time on the specification of a Kalman filter for a docking simulation example:

```
% 10Hz state update, 1MHz processor @ 10% load.
% Run 589 iters must take < 589*0.1 seconds =
% 589*0.1*1000000*10% cycles:
thruster_filter complexityof time
    withbound 589/10*1000000/10.
```

This mechanism combines well with the approximation capability. The next section describes how approximation can be used in conjunction with analysis of worst case execution time in order to synthesize (sufficiently) efficient programs.

Approximations

Approximations can be used to simplify code and reduce its worst case execution time. For example, a common approximation is to assume that an angle remains close to a given constant angle. When applied manually, however, approximations complicate code and introduce assumptions about the circumstances in which it will run which may not be properly documented.

AUTOFILTER's specification language permits approximations to be stated in the specification, allowing efficient code to be derived while making assumptions explicit and keeping the simplicity of the original specification. For example, the small angle approximation was added to the specification of a Kalman filter used in the docking simulation example (i.e. the assumption was added that docking keeps the spacecraft at a nearly constant attitude):

```
const double q as 'Attitude [radians]'.
const double sth:= sin(q).
const double cth:= cos(q).
sin(j) ~~ sth + cth*(j - q) witherror 0.05.
cos(j) ~~ cth + sth*(j - q) witherror 0.05.
```

Synthesis produces program variants, both using the approximation and not using the approximation. When the approximation is applied, statements can be inserted automatically into the synthesized code to verify that the approximation remains valid within the specified error bounds, e.g. that whenever the angle j above changes, the absolute value of the difference between the approximated (LHS of ~~) and approximating (RHS of ~~) expressions is less than the error bound (which follows the `witherror` keyword).

In the docking example, the approximation was combined with analysis of the worst case execution time of the synthesized program. In the synthesized code, the approximation is applied to eliminate calculation of sines and cosines in the Kalman filter loop. Equations (3,4) show the complexity calculations for a statement from the synthesized code before (3) and after (4) the approximation has been applied.

$$|\hat{z}^-(0,0) = lxA * cos(\hat{x}^-(0,0)) + lyA * sin(\hat{x}^-(0,0)) + \hat{x}^-(1,0);| = 1.|cos| + 1.|sin| + 2.|*| + 2.|+| = 404 \quad (3)$$

$$|\hat{z}^-(0,0) = lxA * (cth - sth * (\hat{x}^-(0,0) - q)) + lyA * (sth + cth * (\hat{x}^-(0,0) - q)) + \hat{x}^-(1,0);| = 4.|*| + 3.|+| + 3.|-| = 10 \quad (4)$$

[5]The figures used are for testing only and not intended to reflect any real processor/runtime system.

The synthesis system generates two programs, the second one using the approximation. The following diagnostic output (edited for brevity) from the system shows that automatic complexity analysis determines that the first program fails the timing requirements declared in the specification, while the second meets them:

```
[Schema selection:]
 discretization...declarations...
 initialization...update measurement...
 update loop dependents...propagate estimates...
[Trying approximations:
    1 approximation can be applied.]
Synthesizing pseudocode - 2 programs generated
Postprocessing intermediate code for thruster
Checking code meets timing requirements...
...code FAILS timing requirements of
    1.21e+07=<5890000.
Generating/compiling code for thruster
...
Checking code meets timing requirements.
...code meets timing requirements of
    2.83e+06=<5890000.
```

Back End

The core of the synthesis system (which uses the synthesis schemas) generates code in a simplified intermediate language. This code is then translated by the back end into code of the desired target language/software platform. Our aim is to be able to generate code which is easily tested, and code which can be deployed (unchanged) in a target software architecture, from the same specification. Currently, we have implemented back ends targeting OCTAVE and MATLAB for generating code which can be easily tested, and standalone C for efficient deployment. Prototype back end code generators exist for Modula2 and C++ using the GREASE state estimation library.

The back end also generates a software design document (SDD) for the synthesized code that adheres to NASA and IEEE software standards. This SDD gives a detailed description of the synthesis process, defines the interface and calling conventions, and provides a hyperlinked version of the specification and the generated code. Because the SDD is generated automatically, the SDD, the code and the specification are always consistent, eliminating a major source of errors.

Correctness

AUTOFILTER is based upon formal methods and mathematical logic, so correctness can be guaranteed in principle ("correct by construction"). However, AUTOFILTER is a large and complicated piece of software for which a formal verification is not feasible. In order to guarantee a high quality of the code, we are therefore not certifying the tool, but rather provide a mechanism for *product-oriented certification*. This means that each piece of generated code comes with a certificate, an externally checkable document, that the code obeys important safety properties. The AUTOFILTER system uses a Hoare-style logic and automated theorem provers to automatically process the required proofs [4], [3]. The safety properties that are checked automatically include:

array bounds: All accesses to vectors and matrices are checked for correct range of indices. Thus we are able to guarantee the absence of buffer overflow errors.
uninitialized variables: Variables which are not initially set to a specific value are a major source of severe errors. Although simple cases can be detected by a modern compiler, our AUTOFILTER system is able to automatically cope with complicated cases (e.g., the check that all elements of a vector are initialized).
unused variables: Input/output variables which are not used in the code usually indicate an error or a bad interface specification. This property can eliminate an important class of errors.
matrix symmetry: Some of the matrices in the Kalman filter algorithm must be symmetric by construction. A detailed check (not regarding round-off errors) provides additional assurance on the correctness of the numerical algorithms.

The certification system can be customized to handle specific safety properties and is tied in closely with the document generation, With a detailed link between the certificates and the generated code, the software certification process can be supported effectively, ultimately leading to a much cheaper, but equally concise software certification.

5. FURTHER WORK

AUTOFILTER can generate code implementing a range of Kalman filter specifications using a variety of implementation techniques, and generate documentation and correctness certificates for the generated code. There are several important directions for further work:

- Implementation of synthesis schemas for other kinds of Kalman filters including information filters, unscented Kalman filters and particle filters.
- Implementation of synthesis schemas to generate code to handle bad or missing sensor inputs and windowing.
- Assembling a comprehensive library of Kalman filter specifications and synthesized code by systematic application to text book examples. This endeavor will provide sample specifications to guide users in applying the system to new problems, and help to further assess the adequacy of the specification language.
- Generation of code for a new target software architecture currently involves extending the back end synthesis. We plan to develop mechanisms for specifying the architecture of a target software platform, thereby making it easier to drop synthesized code directly into a flight system.

6. CONCLUSIONS

In this paper we have described the AUTOFILTER program synthesis system, which automatically generates Kalman filter code from succinct specifications of their underlying mathematical models. AUTOFILTER has been applied successfully to text book examples and a number of case studies.

We have outlined how AUTOFILTER assists the iterative development of Kalman filters in various ways: permitting changes in the mathematical model underlying the filter to be rapidly realized as code and tested, different Kalman filters to be synthesized from the same model, code automatically assessed for runtime performance, and approximating assumptions applied to the code in order to improve efficiency. AUTOFILTER provides assistance for ensuring that the generated code is correct by generating program documentation and correctness certificates in addition to the code.

REFERENCES

[1] E. Baumgartner, H. Aghazarian, and A. Trebi-Ollennu. Rover localization results for the FIDO rover. In *Proc. SPIE Conf. Sensor Fusion and Decentralized Control in Autonomous Robotic Systems*, 2001.

[2] R. Brown and P. Hwang. *Introduction to Random Signals and Applied Kalman Filtering*. John Wiley & Sons, 3rd edition, 1997.

[3] E. Denney, B. Fischer, and J. Schumann. Adding assurance to automatically generated code. In Ramamoorthy [10], pages 297–299.

[4] E. Denney, B. Fischer, and J. Schumann. Using automated theorem provers to certify auto-generated aerospace software. In Ramamoorthy [10].

[5] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Embedded Software Workshop*, Lake Tahoe, USA, October 2001. Springer LNCS vol. 2211.

[6] B. Fischer and J. Schumann. AutoBayes: A system for generating data analysis programs from statistical models. *J. Functional Programming*, 13(3):483–508, 2003.

[7] P. Flener, K.-K. Lau, M. Ornaghi, and J. Richardson. An abstract formalisation of correct schemas for program synthesis. *Journal of Symbolic Computation*, 30(1):93–127, 2000.

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

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

[10] C. V. Ramamoorthy, editor. Tampa, FL, 2004. IEEE Comp. Soc. Press.

[11] E. Wan and R. van der Merwe. The unscented kalman filter for nonlinear estimation. In *Proceedings of Symposium 2000 on Adaptive Systems for Signal Processing, Communication and Control (AS-SPCC)*, 2000.

[12] J. Whittle, J. V. Baalen, J. Schumann, P. Robinson, T. Pressburger, J. Penix, P. Oh, M. Lowry, and G. Brat. Amphion/NAV: Deductive synthesis of state estimation software. In *Proc IEEE Conference on Automated Software Engineering*, 2001.

[13] J. Whittle and J. Schumann. Automating the implementation of Kalman filter algorithms. *ACM Transactions on Mathematical Software*, 2005. To appear.

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.

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 certification techniques for aerospace software. He has published more than 40 papers in areas including component retrieval, program synthesis, and 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 on 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. 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.