Automating the Implementation of Kalman Filter Algorithms

JON WHITTLE
QSS Group/NASA Ames Research Center
and
JOHANN SCHUMANN
RIACS/NASA Ames Research Center

AUTOFILTER is a tool that generates implementations that solve state estimation problems using Kalman filters. From a high-level, mathematics-based description of a state estimation problem, AUTOFILTER automatically generates code that computes a statistically optimal estimate using one or more of a number of well-known variants of the Kalman filter algorithm. The problem description may be given in terms of continuous or discrete, linear or nonlinear process and measurement dynamics. From this description, AUTOFILTER automates many common solution methods (e.g., linearization, discretization) and generates C or Matlab code fully automatically. AUTOFILTER surpasses toolkit-based programming approaches for Kalman filters because it requires no low-level programming skills (e.g., to "glue" together library function calls). AUTOFILTER raises the level of discourse to the mathematics of the problem at hand rather than the details of what algorithms, data structures, optimizations and so on are required to implement it. An overview of AUTOFILTER is given along with an example of its practical application to deep space attitude estimation.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.13 [**Software Engineering**]: Reusable Software; G.4 [**Mathematical Software**]—*Algorithm design and analysis*

General Terms: Algorithm

Additional Key Words and Phrases: Code generation, Kalman filters, state estimation, automatic programming

1. INTRODUCTION

Once the mathematical models for a state estimation problem have been formulated, there is usually a significant amount of implementation work that has to be done before those models and their associated estimator can be tested. This is true even if existing libraries/toolkits are used to support implementation because such libraries still require "glue" code to be written to allow the library functions to work together. Testing usually suggests refinements

Authors' addresses: J. Whittle NASA Ames Research Center, MS 269-2, Moffett Field, CA 94035; email: {jonathw,schumann}@email.arc.nasa.gov.

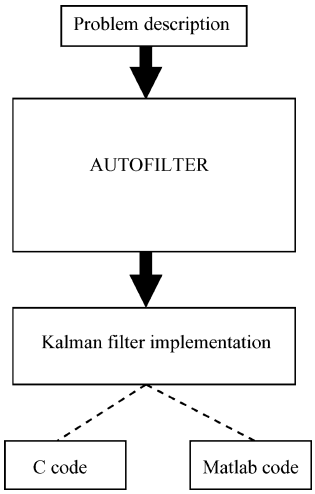

Fig. 1. An overview of AUTOFILTER.

to the models and/or the estimator that in turn results in additional coding effort. The goal of the AUTOFILTER project is to develop techniques and tools that substantially reduce the time and effort needed to develop reliable implementations of state estimators. AUTOFILTER is a knowledge-based tool that, given a high-level mathematical description of the process dynamics and measurements, can automatically generate a C or Matlab implementation that will compute a statistically optimal estimate of a specified state vector under the model assumptions. In particular, AUTOFILTER generates Kalman filter implementations and has so far been used on a number of applications and case studies concerning spacecraft attitude estimation and control. AUTOFILTER surpasses existing coding techniques (including the use of toolkits) for this class of problems because:

—analysts using AUTOFILTER need not be concerned with low-level implementation details;
—analysts need not be concerned with some problem solving methods (e.g., linearization) because AUTOFILTER carries them out automatically;
—changes in the model require no additional coding because AUTOFILTER just re-generates code for the updated model.

A Kalman filter [Brown and Hwang 1997] is a recursive algorithm for calculating the best estimate of a state vector, $\boldsymbol{x}$, based on noisy measurements, $\boldsymbol{z}$. The state vector contains variables of interest that will be estimated, for example, position and velocity. The Kalman filter estimate of this state vector incorporates knowledge given as a model of the process under analysis and a model of the relationship between the state vector and the measurements. AUTOFILTER generates Kalman filter implementations from a description of these models and a description of $\boldsymbol{x}$ and $\boldsymbol{z}$. Figure 1 gives an overview of the AUTOFILTER tool. The problem description defines the physical characteristics of the problem in terms of the process model and measurement model. The process

model can be defined in the usual way as a differential equation of the form

$$\frac{d}{dt}\boldsymbol{x}(t) = \boldsymbol{f}(\boldsymbol{x}(t), t) + \boldsymbol{g}(\boldsymbol{x}(t), t)\boldsymbol{w}(t) \tag{1}$$

where $\boldsymbol{f}, \boldsymbol{g}$ are functions, $\boldsymbol{x}(t)$ is the state vector, and $\boldsymbol{w}(t)$ is the Gaussian white process noise, with mean and covariance defined by

$$E[\boldsymbol{w}(t)] = \boldsymbol{0} \tag{2}$$

$$E[\boldsymbol{w}(t)\boldsymbol{w}^T(t')] = \boldsymbol{Q}(t)\delta(t - t') \tag{3}$$

where $\delta(t)$ is the Dirac delta function.

Assuming measurements arrive at discrete timepoints, the measurement equation at time t_k can be defined in the usual way

$$\boldsymbol{z}_k = \boldsymbol{h}(\boldsymbol{x}_k) + \boldsymbol{v}_k \tag{4}$$

where $\boldsymbol{v}_k$ is the measurement noise, a discrete Gaussian white noise process with mean and covariance given by

$$E[\boldsymbol{v}_k] = \boldsymbol{0} \tag{5}$$

$$E\left[\boldsymbol{v}_k \boldsymbol{v}_{k'}^T\right] = R_k \delta_{kk'} \tag{6}$$

where $\delta_{kk'}$ is the Kronecker delta.

In addition, the problem description contains further information needed for the filter such as initial process covariance and initial state estimates. Based on this problem description, AUTOFILTER chooses an appropriate Kalman filter (currently, the choice is between a standard filter, a linearized filter, an extended Kalman filter or a parallel bank of filters) and instantiates a generic representation of that filter. This instantiation may involve significant problem solving such as transformation from a continuous to a discrete problem formulation or linearization of a model. The resulting filter implementation can be generated by AUTOFILTER in an intermediate language that is readily translated into any sensible language (currently C, C++, and Modula II) for various target systems (e.g., Matlab or Octave). The process of generating the code is generally faster than compilation of the generated code and the code can easily be inserted into a test environment for immediate simulation/testing.

AUTOFILTER has a number of advantages, namely:

—*Coding effort is significantly reduced*. In order to generate a Kalman filter implementation, the user need only formulate the usual state and measurement equations. An implementation can then be generated fully automatically. This is in contrast to the use of toolkits where coding is still required to combine toolkit functions.

—*Rapid prototyping becomes easy*. Iterations on the models can be made quickly and easily. Based on simulation or testing, the user can modify the problem description and regenerate a new implementation which in turn can then be simulated/tested.

—*The design space can be explored quickly and thoroughly*. The rapid prototyping benefits mean that the analyst has more time to fully explore design alternatives and variations. Each variation can be generated easily once the models have been formulated.

—*The generated code is highly documented*. The generated code is well documented with the exact steps that were taken to derive the implementation and the simplifying assumptions that were made. Such documentation is useful for code reviews or communication between analysts and programmers.

—*The code can be made to adhere to existing code standards or architectures*. AUTOFILTER can be customized to generate code adhering to certain standards. An architecture description can be given that is then used to structure the generated code to fit into that architecture. The use of AUTOFILTER encourages reuse between projects because an enterprise's style can be fixed in advance.

The remainder of the article is structured as follows. Section 2 gives an overview of the design and architecture of the AUTOFILTER tool. Section 3 presents an application of AUTOFILTER to a deep space attitude estimation problem. Section 4 discusses related work and Section 5 concludes.

2. DESCRIPTION OF AUTOFILTER

Figure 2 gives an overview of the various components that make up AUTOFILTER. In this section, we will describe these elements in detail.

2.1 Input Language

The input language of AUTOFILTER allows the concise specification of the process model, the measurement model, and other important design information—a definition is given in the appendix. In order to illustrate the basics of AUTOFILTER's input language, let us consider the following simple example of a state estimation problem for a simple rover, taken from Roumeliotis et al. [1998]. The rover under consideration is a Pioneer AT rover with four wheels. The pair of wheels on the left side of the rover are mechanically coupled. Similarly, on the right side. Each side has a rotation encoding sensor, which returns the current speed of the wheels on that side of the vehicle. Furthermore, there is a gyro, which can be used to measure the yaw rate. In this example, we use a very simple process and measurement model that has three state variables $\boldsymbol{x} = (v^L, v^R, y)^T$ for the speed estimate of the left wheels, the right wheels, and the estimate of the yaw rate of the chassis, respectively. A straightforward discrete process model describing the dynamics of the rover can be defined as follows:

$$\begin{pmatrix} v^L_{k+1} \\ v^R_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} v^L_k \\ v^R_k \\ -v^L_k/l + v^R_k/l \end{pmatrix} + \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix}$$

with Gaussian white noise w_i, where l is the vehicle axis length. The sensors measure the state variables directly, so the measurement model is trivial and is given by $\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{v}$. Figure 3 gives the AUTOFILTER input specification for this

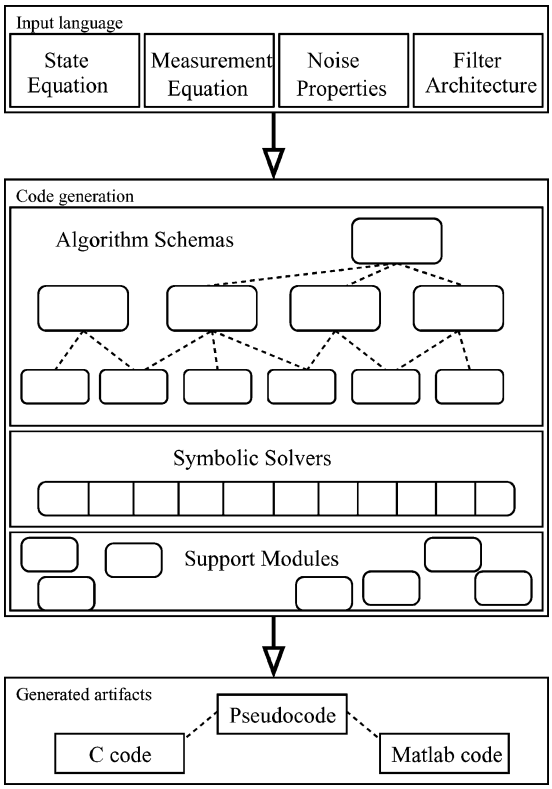

Fig. 2. AUTOFILTER architecture.

problem. In addition to defining the process and measurement models, the specification also defines variables, constants, data and estimator characteristics as described below.

The input language to AUTOFILTER allows the declaration of constants (using keyword `const`), input data (keyword `data`) and datatypes (`nat` for natural number, `double` etc.). Inline comments can be added using the `as` keyword. Vectors and matrices can be defined using an ellipsis—for example, the declaration `double x(1..3)` declares a 3-vector `x` whose elements are of type `double`. Distributions are declared using the ~ symbol. Index variables can be used to refer to all elements of a vector or matrix—for example, `w(I) ~ gauss(0,1)` declares all elements of `w` to be Gaussian distributed with zero mean and unit variance. Assignment is defined using the `:=` operator. The keyword `equation_set` defines a named set of equations delimited by `is` and `end`. The output is defined to be the estimate from a Kalman filter where this filter is defined by the keyword `estimator`.

The process equation for our example (Figure 3) is defined in lines 25–28 and the measurement equation is given in lines 36–39. The process model is given in discrete form (the notation `_k` is used to denote "at time k"). The initial estimate is given in lines 19–22. Lines 41–48 describe the Kalman filter that AUTOFILTER should generate. In particular, it is stated how many iterations of the

```
1  /*********************************************************************
2  * The Rover is the Pioneer AT which has four wheels (two on left, two
3  * on right). The wheels on the same side are mechanically
4  * coupled. There are two encoder sensors (one on each side) that
5  * return an estimate of the speeds of the wheel pairs. There is also a
6  * gyro that can be used to measure the yaw rate.
7  *
8  * This model uses 3 state variables:
9  * x(1) : speed estimate of the left wheels
10 * x(2) : speed estimate of the right wheels
11 * x(3) : yaw rate estimate of the chassis
12 *********************************************************************/
13  const nat n_statevars := 3 as 'Number of state variables'
14  double x(1..n_statevars) as 'state vector'
15  double w(1..n_statevars) as 'process noise vector'
16  double sigma(1..n_statevars) as 'variance of process noise'
17  w(I) ~ gauss(0, sigma(I))
18
19  double xinit(1..n_statevars) as 'initial state variable values'
20  data double xinit_noise(1..n_statevars) as 'initial state variance'
21  data double xinit_mean(1..n_statevars) as 'initial state mean'
22  xinit(I) ~ gauss(xinit_mean(I), xinit_noise(I))
23
24  const double l := 1 as 'vehicle axle length'
25  equation_set nominal_model is
26        x(1)_k+1 := x(1)_k + w(1)
27        x(2)_k+1 := x(2)_k + w(2)
28        x(3)_k+1 := -x(1)_k/l + x(2)_k/l + w(3) end
29
30  const nat m_measvars := 3 as 'Number of sensor measurements'
31  const n_steps := 50 as 'Number of filter iterations'
32  data double z(1..m_measvars, 1..n_steps) as 'measurement data'
33  double v(1..m_measvars) as 'measurement noise vector'
34  data double rho(1..m_measvars) as 'variance of measurement noise'
35  v(I) ~ gauss(0,rho(I))
36  equation_set measurement_model is
37        z(1,_) := x(1) + v(1)
38        z(2,_) := x(2) + v(2)
39        z(3,_) := x(3) + v(3) end
40
41  estimator nominal_filter
42  nominal_filter.update_interval := 1
43  nominal_filter.steps := n_steps
44  nominal_filter.process_eqs := nominal_model
45  nominal_filter.measurement_eqs := measurement_model
46  nominal_filter.initials := xinit(I)
47 /* mdiag(y) is the diagonal matrix with y along the diagonal */
48  nominal_filter.initial_covariance := mdiag(xinit_noise(I))
49
50  output nominal_filter
```

Fig. 3. Rover navigation specification.

filter should be executed, the time interval between iterations, and references to the process and measurement equations and initial conditions are given. Note that this specification defines a batch-mode processing of measurements (i.e., it assumes all observations for all time-steps are available on start-up). This is specified by defining an observation data matrix, z, in line 32, which contains all observations. The underscore in lines 37–39 denotes the fact the equations are independent of the second index of z. Online processing of measurements is also supported. The example specifies only one estimator. However, AUTOFILTER allows the specification of multiple estimators and the connections between them, for example, the definition of a parallel bank of Kalman filters.

2.2 Code Generation

The AUTOFILTER code generator is built on three levels. These levels correspond to three stages in solving a particular estimation problem. First, a particular algorithm (or combination of algorithms) that will solve the problem must be chosen. This is done at the *schema-level*. For example, AUTOFILTER may decide on an extended Kalman filter. The result is a high-level representation of the algorithms chosen.

Many state estimation algorithms involve an element of mathematical solving. This is done at the *solver-level*. For example, an extended Kalman filter requires that a linear approximation to a nonlinear process be derived. Finally, the *support-level* takes care of low-level code generation tasks that have not been taken care of elsewhere. These three levels are described in more detail below.

2.2.1 *Algorithm Schemas.* A schema is a generic representation of a well-known algorithm. Most generally, it is a high-level description of a program that captures the essential algorithmic steps but does not necessarily carry out the computations for each step. In AUTOFILTER, a schema has five parts:

—underlying *assumptions* of the algorithm;
—*applicability conditions* that must hold for the schema to be applied;
—a high-level description of the program in the form of a *template* that describes the key algorithmic steps;
—the *body* of the schema, which instantiates the template by calling routines at the schema-level, solver-level or support-level;
—a *blackboard* for storing intermediate results that may be needed by other schemas.

Assumptions are inherent limitations of the algorithm and appear as comments or run-time assertions in the generated code. Applicability conditions are preconditions that can be used to choose between alternative schemas. The key difference between assumptions and applicability conditions is that applicability conditions can be evaluated to see if they are true or false. Assumptions, on the other hand, are considered to be true but cannot be shown to be true given the body of knowledge in the specification. For example, an assumption for the standard Kalman filter schema is that the process noise is white. This fact cannot be validated from the specification but is a necessary

requirement for the use of a Kalman filter. An applicability condition for the standard Kalman filter would be that the process and measurement models are linear. The analogous condition for the extended Kalman filter would be that either the process or measurement model is nonlinear. The linearity condition can be explicitly checked on the specification. Note that in many applications, assumptions will in fact be violated: for example, the noise may be *almost* white. This is a consequence of the fact that the modeling process is approximate. Applicability conditions for the chosen schema, however, are never violated in an application.

Even in the presence of applicability conditions, it is possible that different schemas can apply to the same problem (e.g., a linearized Kalman filter and extended Kalman filter can both be applicable to the same problem). This leads to choice points. During code generation, these choices are explored in a depth-first manner. Whenever a dead-end is encountered (i.e., an incomplete code fragment has been generated but no schema is applicable), AUTOFILTER backtracks. This control regime allows AUTOFILTER to generate multiple program variants for the same problem.

The algorithm template is described using a simple template programming language that has the usual programming constructs such as if-statements, for and while loops, and a `series` construct that specifies the execution of a number of statements sequentially. The template may also contain variables, denoted with a prefix `$`. When the schema is executed during synthesis, these variables get instantiated by code fragments. The blackboard is included as a convenience because some schemas compute intermediate results that are used by other schemas. Storing these on the blackboard avoids recomputation.

Figure 4 shows an abstraction of the schema for an extended Kalman Filter. The calls in the body are parameterized over the schema variables so that the variables can be instantiated during the body calls. Note how the body describes the essential sequential steps of the algorithm and the template defines what the resulting generated code will look like.

The schema template sets up the code structure at a high level with variables that will be instantiated by the schema body. `$Local` will be instantiated to a series of local variable declarations needed by the extended Kalman filter (e.g., the declaration of the state transition matrix variable). `$Initial` will be instantiated to a series of initializations of these variables. The main filter loop comes next. The code in `$UpdateMeasurements` will get the next set of observations from the input stream. The state transition matrix and measurement matrix will have been initialized in `$Initial`. In `$UpdateStateTransition` and `$UpdateMeasurementMatrix`, any elements in these matrices that have changed since the last iteration are reassigned. Note that AUTOFILTER only reassigns elements that vary over time. Hence, in the case of a constant state transition matrix, there is no additional overhead in assignment. `$CalculateGain`, `$UpdateEstimate`, `$UpdateCovariance`, `$PropagateEstimate`, and `$PropagateCovariance` are the main parts of the Kalman Filter. For example, in a standard Kalman filter, `$CalculateGain` would implement the matrix equation $K_k = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1}$. Note that these variables can be instantiated in a number of different ways depending on the

```
%%% Assumptions

... white process noise ...
... white measurement noise ...
... process and measurement noise independent ...

%%% Applicability Conditions

... Gaussian process noise ...
... Gaussian measurement noise ...
... Nonlinear process or measurement model ...

%%% Body

... linearize process and/or measurement models ...
... discretize process model ...
... declare local variables ...
... initialize local variables ...
... update measurements ...
... update loop dependent quantities ...
... calculate Kalman gain ...
... update estimates ...
... update output vector ...
... propagate ...

%%% Template

Template = "
     series(
       $Local,
       $Initial,
       for(0,$NumberIterations,
        series(
               $UpdateMeasurements,
               $UpdateStateTransition,
               $UpdateMeasurementMatrix,
               $CalculateGain,
               $UpdateEstimate,
               $UpdateCovariance,
               $StoreOutput,
               $PropagateEstimate,
               $PropagateCovariance
               )
            )
       )"

return Template;
```

Fig. 4. Extended Kalman Filter schema.

type of filter that is being implemented. For example, in an information filter[1] the gain expression would instead be calculated using $K_k = P_k H_k^T R_k^{-1}$. In fact,

[1]An information filter is a Kalman filter variant often used in the case where very little is known about the process initially. In such cases, the standard formulation of the Kalman filter results in a division ∞/∞. The information filter avoids this by algebraically reformulating the Kalman filter equations.

any of the "slots" in the schema template may be instantiated in different ways by any of the function calls in the schema body. This highlights the fact that each schema corresponds not to a single algorithm but to a family of related algorithms. The particular choice of algorithm results from the characteristics of the problem at hand. The final slot in the template, `$StoreOutput` outputs the result of the estimator. By default, only the state estimate is retained and this is written to an output matrix, although this slot could easily be changed to, for example, write the process covariance at each iteration to standard output.

The advantage of using schemas to encode algorithmic knowledge rather than parameterized functions is that schemas encode a *family of algorithms* rather than a single algorithm. This allows a class of algorithms with significant variations within that class to be represented by a single schema. In the case of Kalman filters, for example, the basic filter, linearized and extended filter can all be represented by the same schema.

2.2.2 *Symbolic Solvers.* The solver-level of AUTOFILTER consists of a collection of symbolic solvers, written as a set of conditional rewrite rules. A conditional rewrite rule $C \Rightarrow L = R$ can be applied to an expression E with subexpression L' if there is a substitution ϕ for the variables in L such that $\phi(L) = L'$ and if $\phi(C)$ is true. In this case, L' is replaced by $\phi(R)$ to yield a new expression E'. By defining a set of such rewrite rules, common problem solving tasks can be accomplished by exhaustively applying rewrite rules to a given expression. We assume that for each set of rewrite rules, the order of application of the rules is irrelevant.[2]

Some examples of rewrite rule systems used in AUTOFILTER are rules for differentiating an expression, for matrix identities, for linearizing a set of equations (e.g., by calculating Jacobians), for carrying out various approximations, for evaluating trigonometric expressions, and so on. Rewrite rules are a good way of expressing certain kinds of domain knowledge because complex solvers can quickly and legibly be expressed as rewrites.

AUTOFILTER has a sophisticated rewrite rule engine for applying rewrite rules, which can, for example, apply rules under different assumptions. This rewrite engine was written by Bernd Fischer and is inherited from the AUTOBAYES data analysis code generation system [Fischer et al. 2000].

As a simple example of a rewrite rule, consider the task of approximating a matrix exponential by a truncated Taylor series. This problem arises in AUTOFILTER during discretization of a continuous process. The approximation is easy to write down as a rewrite rule:

$$is\_square\_matrix(A) \text{ and } rows(A) = n \Rightarrow \exp(A) = I_{n\times n} + A + \frac{1}{2}A^2 \qquad (7)$$

Maintenance of these approximations is also made easier through the use of rewrites—the above approximation can be replaced quickly with a third-order approximation or augmented with additional rewrites expressing alternative approximations such as approximation with an inverse Laplacian or numerical integration.

[2]We rely on the rule designer to enforce this constraint.

2.2.3 *Support Modules.* Schemas can be thought of as setting up the high-level definition of the generated code. Rewrites can then be seen as refining this definition down a level. The lowest level of definition, however, is typically given by various support modules. This is necessary in practice because, although it would be possible to instantiate the templates using only rewrite rules, it would be inconvenient and inefficient to do this for the many bookkeeping tasks (e.g., finding the name of the variable representing the state vector) that need to be carried out during code generation. The support modules can, in general, be written in any programming language. In AUTOFILTER, they are written in the logic programming language Prolog [Clocksin and Mellish 1984].

The example in Section 3 will illustrate how the three levels are applied to produce a faithful implementation of a model.

2.3 Generated Artifacts

2.3.1 *Translation to Programming Languages.* During the synthesis phase, a set of program schemas are instantiated and combined to form the structural skeleton of the filter implementation. Each of these instantiated schemas returns a code fragment in AUTOFILTER's intermediate language. This intermediate language is a simple procedural language with additional operators for compact handling of sums, vectors and matrices. This language is abstract and general enough to be used within the synthesis schemas, yet it is close enough to a standard procedural programming language to allow for a straight-forward generation of the final code. The AUTOFILTER code translators have been designed in a modular way such that the code generator can be easily adapted toward a specific target language or environment. We have developed intermediate language to code translators for several target systems and languages: C for Matlab (MEX interface), stand-alone C for embedded environments with various run-time libraries, C++ for the Octave [2003] system, and Modula II. We can also generate (interpreted) Matlab code.

Depending on the selected target system, the translator converts the high-level operators (like a matrix multiplication) into operations supported by the target system, which, for example can be a C++ method, a for-loop, or a call to a library.

2.3.2 *Correctness.* Any code generator should be concerned with the correctness of the code generated. This is particularly true for state estimation code, which is a safety critical component of flight software. Although outside the scope of this article, AUTOFILTER has been augmented with a number of techniques for verifying the correctness of the generated code. These techniques fall into two broad categories.

Firstly, we have developed methods for guaranteeing the correctness of the algorithm used in the generated code [Rosu and Whittle 2002a, 2002b]. This is important because AUTOFILTER may generate non-standard variants of the Kalman filter, and it may not be obvious that these variants are indeed optimal estimators. By providing machine-checked proofs of the optimality of the algorithm, AUTOFILTER provides assurance that this is the case.

Secondly, we are using *property verification*. In this approach, a set of safety properties is automatically checked for each statement of the code. We have developed a subsystem for AUTOFILTER, which can automatically check programming-language specific safety properties, like array-bounds safety, operator-definedness, or variable-initialization-before-use [Whalen et al. 2002].

3. APPLICATION TO DEEP SPACE ATTITUDE ESTIMATION

AUTOFILTER is currently at the research prototype stage—it has been used on a number of real-world applications that have shown that its use is feasible in practice. We describe one of those applications here. The application was an "after-the-fact" case study in the sense that code for the application had already been produced by the original code developers.

Deep Space I (DS1) is a deep space probe managed from the Jet Propulsion Laboratory (JPL) as a vehicle for testing a range of experimental NASA technologies under flight conditions. It was launched in October 1998 and retired in December 2001. Although the Deep Space I mission is now over, the spacecraft continues to operate in deep space.

In the summer of 2002, a case study was undertaken in which AUTOFILTER was used to recreate the Kalman filter portion of the DS1 attitude estimator implementation. AUTOFILTER was used to specify the mathematical models for the attitude estimator and around 400 lines of C code were automatically generated and then integrated and tested in the Autonomy Lab, a JPL testbed for deep space missions.

In this section, we present the mathematical models used to specify the DS1 attitude estimator. These models are the same as those used in the original DS1 estimator. This case study is limited to the parts of the code for which AUTOFILTER is suited, namely the core Kalman filter loop. Real-time issues, coordinate system transformations, and so on are assumed to be taken care of outside the filter and are thus not considered.

DS1 estimates attitude using a combination of an IMU (Inertial Measurement Unit) and a stellar reference unit[3] (SRU). The estimator can be in one of three modes—IMU only, SRU only or IMU and SRU. We will only consider the third mode in this article. In this mode, the gyro outputs from the IMU are augmented with readings from the SRU to provide a more accurate estimate of the attitude of the spacecraft. The SRU outputs a quaternion representation of the spacecraft attitude and this quaternion is used to augment the IMU readings. The design of the DS1 filter closely followed that given in Section XI of Lefferts et al. [1982]. In this example, quaternions were used to represent points and directions in space. Although computationally slightly more complex, this representation has the advantage that a transformation between coordinate systems does not exhibit any singularities (as can occur when Euler angles are used).

In the rest of this section, we show how the DS1 attitude estimator was modeled in AUTOFILTER and discuss the code generation process. Note that rather than estimating the full quaternion (representing the spacecraft attitude), the

[3]More commonly known as a star tracker.

Kalman filter estimates the error in a given base quaternion. This error is represented as an incremental quaternion, which must be composed with the base quaternion in order to obtain the true quaternion. When the filter starts, the initial base quaternion is given. The filter updates the base quaternion according to its error estimate on each iteration and the new base quaternion is used in the following iteration. As usual, the error quaternion is defined as

$$\bar{q} = \begin{pmatrix} \boldsymbol{q} \\ q_4 \end{pmatrix} \tag{8}$$

where $\boldsymbol{q} = \hat{\boldsymbol{n}} \sin(\delta\theta/2)$ and $q_4 = \cos(\delta\theta/2)$ and $\delta\theta$ is the incremental rotation about an axis $\hat{\boldsymbol{n}}$. Since $\delta\theta$ is small, $\boldsymbol{q} \approx \hat{\boldsymbol{n}}\,(\delta\theta/2)$ and $q_4 \approx 1$, so the fourth component of the error quaternion need not be estimated.

The complete specification in AUTOFILTER is given in Figure 5. In total, there are six state variables: $\delta\theta_i, 1 \leq i \leq 3$ and the three gyro biases $b_i, 1 \leq i \leq 3$. The IMU provides a 3-vector output, an attitude rate, $\boldsymbol{u}$. The process model is taken from Lefferts et al. [1982], equations (48), (51), and (135). Note that DS1 uses this model in the covariance propagation but state propagation instead uses the IMU data to approximate the model. From Lefferts et al. [1982], the vector equations for describing the process model are

$$\boldsymbol{\omega} = \boldsymbol{u} - \boldsymbol{b} - \boldsymbol{\eta_1} \tag{9}$$

$$\frac{d}{dt}\boldsymbol{b} = \boldsymbol{\eta_2} \tag{10}$$

$$\frac{d}{dt}\delta\boldsymbol{\theta} = -\hat{\boldsymbol{\omega}} \times \delta\boldsymbol{\theta} - (\boldsymbol{b} - \hat{\boldsymbol{b}} + \boldsymbol{\eta_1}). \tag{11}$$

$\boldsymbol{w}$ is the true angular rate of the spacecraft, which according to equation (9) is given by the gyro output minus the gyro biases and unbiased white noise $\boldsymbol{\eta_1}$. Equation (10) represents the gyro biases as a random walk where $\boldsymbol{\eta_2}$ is unbiased white noise. Equation (11) describes how the incremental angles $\delta\theta$ change over time—see Lefferts et al. [1982] for details—where $\hat{\boldsymbol{w}} = \boldsymbol{u} - \hat{\boldsymbol{b}}$ and $\hat{\boldsymbol{b}}$ is the estimated gyro bias. In the Kalman filter context, $\hat{\boldsymbol{b}}$ is taken to be the best estimate of the gyro bias from the previous iteration of the filter. Accordingly, in Figure 5, $\hat{\boldsymbol{b}}$ in lines 16–24 denotes the estimate of $\boldsymbol{b}$ obtained in the previous time step of the filter run. Lines 14–24, give the process model in AUTOFILTER syntax.

The SRU measurements are modeled as a 3-vector, $\boldsymbol{z}$, and measurement noise is given by $\boldsymbol{v}$. The actual DS1 SRU produces a full quaternion. In order to relate this quaternion to the error components of the incremental quaternion, $\delta\theta_i$, the estimated full quaternion is needed. Since this is not available in the AUTOFILTER DS1 specification, we assume the SRU full quaternion has been preprocessed to produce delta angles representing the error quaternion before being sent to the filter. In this case, as can be seen in Figure 5 (lines 33–36), the measurement matrix H turns out to be $[I_{3\times3} \quad 0_{3\times3}]$.

Note that in the original DS1 code, the "preprocessing" is taken care of within the filter loop by maintaining a current quaternion estimate of the attitude. The

```
 1   model ds1 as 'IMU + SRU attitude estimation for DS1'
 2   data nat n_steps as 'Number of iterations in filter'
 3   const double delta_t as 'delta time between iterations'
 4  /*****************************************************
 5  * Process model
 6  *****************************************************/
 7   double deltaTheta(1..3) as 'incremental quaternion elements'
 8   double b(1..3) as 'gyro biases'
 9   double eta(1..6) as 'process noise vector'
10   data double sigma(1..6)
11   eta(I) ~ gauss(0, delta_t * sigma(I))
12
13   double u(1..3) as 'gyro output'
14   equation_set process_model is
15   d/dt deltaTheta(1) :=  (u(3) - b̂(3)) * deltaTheta(2)
16                        - (u(2) - b̂(2)) * deltaTheta(3)
17                        - (b(1) - b̂(1)) - eta(1)
18   d/dt deltaTheta(2) :=  (u(1) -b̂(1)) * deltaTheta(3)
19                        - (u(3) - b̂(3)) * deltaTheta(1)
20                        - (b(2) - b̂(2)) - eta(2)
21   d/dt deltaTheta(3) :=  (u(2) - b̂(2)) * deltaTheta(1)
22                        - (u(1) - b̂(1)) * deltaTheta(2)
23                        - (b(3) - b̂(3)) - eta(3)
24   d/dt b(1) := eta(4)   d/dt b(2) := eta(5)   d/dt b(3) := eta(6) end
25  /*****************************************************
26  * Measurement model
27  *****************************************************/
28   data double z(1..3) as 'SRU adjusted measurements'
29   double v(1..3) as 'measurement noise vector'
30   data double rho(1..3)
31   v(I) ~ gauss(0,rho(I))
32
33   equation_set measurement_model is
34   z(1) := deltaTheta(1) + v(1)
35   z(2) := deltaTheta(2) + v(2)
36   z(3) := deltaTheta(3) + v(3) end
37
38  /*****************************************************
39  * Filter Specification
40  *****************************************************/
41   double xinit(1..6) as 'initial state variable values'
42   data double xinit_noise(1..6) as 'initial state variance'
43   data double xinit_mean(1..6) as 'initial state means'
44   xinit(I) ~ gauss(xinit_mean(I), xinit_noise(I))
45
46   estimator ds1_filter
47   ds1_filter.update_interval = delta_t
48   ds1_filter.steps = n_steps
49   ds1_filter.process = process_model
50   ds1_filter.measurements = measurement_model
51   ds1_filter.initials = xinit(I)
52  /* mdiag(y) is the diagonal matrix with y along the diagonal */
53   ds1_filter.initial_covariance = mdiag(xinit_noise(I))
```

Fig. 5. Deep space I attitude estimation specification.

measurement, $\boldsymbol{z}$, is then the correction in the attitude that should be applied to the quaternion estimate of the attitude assuming the measured quaternion from the SRU is the true value (see (118) in Lefferts et al. [1982]): $\boldsymbol{z} = \boldsymbol{q}_{\mathrm{SRU}} \otimes \boldsymbol{q}^{*}_{\mathrm{est}}$, where $\otimes$ is quaternion multiplication and * is quaternion inverse. Using AUTOFILTER, we take a slightly different approach in which the preprocessing is done outside the filter.

Given the specification in the previous section, AUTOFILTER generates code that implements the corresponding extended Kalman filter. AUTOFILTER detects that the problem is nonlinear and applies the appropriate algorithm schema. This involves linearizing the process model and since the process model is also continuous, a discretization step takes place. In essence, the process in Lefferts et al. [1982, pp. 425–6] takes place automatically within AUTOFILTER. Briefly, the process model is (automatically) linearized, resulting in

$$\frac{d}{dt}\Delta\boldsymbol{x} = \boldsymbol{F}\ \Delta\boldsymbol{x} + \boldsymbol{w} \tag{12}$$

where $\boldsymbol{x} = \begin{pmatrix} \delta\boldsymbol{q} \\ \boldsymbol{b} \end{pmatrix}$, $\boldsymbol{w} = \begin{pmatrix} \eta_1 \\ \eta_2 \end{pmatrix}$ and $\boldsymbol{F} = \begin{pmatrix} -\hat{\omega}^{\times} & -I_{3\times3} \\ 0_{3\times3} & 0_{3\times3} \end{pmatrix}$. $\boldsymbol{a}^{\times}$ is the standard skew-symmetric matrix generated from the 3-vector $\boldsymbol{a}$:

$$\boldsymbol{a}^{\times} = \begin{pmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{pmatrix} \tag{13}$$

As usual, the linearized process model for an extended Kalman filter considers state vector incrementals, $\Delta\boldsymbol{x}$, rather than the full state vector.

To obtain the state transition matrix, Φ, equation (12) must be discretized. This is done in AUTOFILTER by approximating[4] the solution of (12) by the truncated Taylor series expansion of $e^{\boldsymbol{F}\,\Delta t}$. The order of the expansion can be selected during synthesis time. In our case study we used 2nd and 5th order Taylor series. Truncated Taylor series are just one of several approximations that AUTOFILTER could have used.

From the specification in Figure 5, AUTOFILTER generates 400 lines of C code fully automatically with calls to JPL's own library for matrix operations. This library contains simple subroutines for matrix addition, subtraction, transposing, multiplication, and calculation of a 3×3 matrix inverse. AUTOFILTER can also generate its own matrix operation code, in which case the code generated amounts to 780 lines. The generated code was executed on a JPL testbed.

4. RELATED WORK AND DISCUSSION

A number of software packages exist for Kalman filtering—for example, the Matlab function `kalman()`, Murphy's Kalman filter toolbox [Murphy 2002], ReBEL [van der Merwe and Wan 2003], KALMTOOL [Noorgaard 2002], as well as libraries written in Fortran, C, Matlab and so on. It is important to realize, however, that AUTOFILTER is more than yet another toolbox. Toolkits provide concrete implementations of particular algorithms. The user must spend

[4]A standard approximation—see Grewal and Andrews [1993] for details.

a good deal of time understanding the functions in the toolkit and how they interoperate. Low-level coding is still required to implement a filter because the functions must be "glued" together. In addition, many toolkits only deal with simple versions of state estimation problems, for example, for linear systems only.

We consider the following as the main arguments as to why AUTOFILTER provides additional functionalities over Kalman filter toolboxes and pre-defined functions:

—Toolbox functions hard-code a specific implementation of a specific algorithm. AUTOFILTER, on the other hand, is designed to generate families of algorithms because the algorithms are encoded as schemas. This allows for much greater range and flexibility in the code generated.

—AUTOFILTER raises the level of discourse to that of the problem description rather than how to glue together toolbox functions.

—AUTOFILTER is more easily customizable. Toolboxes are written in a specific language and the source code for the functions may not be available. Since AUTOFILTER generates code in a generic intermediate programming language, it can be translated into many different programming languages, and the source code is available for hand modification if necessary.

—Toolboxes are purely interpretive systems and do not generate programs.

We believe the strongest argument in favor of AUTOFILTER over toolboxes is that AUTOFILTER provides so many variations of basic algorithms. This makes it very easy to rapidly prototype systems without ever thinking of implementation details. For example, the user may start off with a linear representation of a problem and use AUTOFILTER to generate code. Minor modifications to the problem description may make the problem nonlinear. The user can just rerun AUTOFILTER and have code returned that is significantly more complicated than the previous version of the code despite the fact that the specification changed only slightly. Using a toolkit-based approach, the original implementation would have to be modified significantly by hand to replace the original linear filter function by a nonlinear version and to introduce other functions such as a linearization routine. Another example is if the specification admits a constant state transition matrix. In the generated implementation, the matrix will be initialized once and never be recalculated. By changing the process model slightly, the state transition matrix may become time-varying. In this case, AUTOFILTER would generate code that updates the time-varying elements of the matrix on each iteration but leaves constant elements untouched. In this way, AUTOFILTER has optimized the calculation of the state transition matrix. Toolboxes would most likely just recalculate all of the matrix on each iteration.

AUTOFILTER is a close sister of the AUTOBAYES system [Fischer et al. 2000], which generates code for data analysis problems. In fact, the two systems share much of the same infrastructure, including the rewrite engine and schema-based approach. Where they differ is in the knowledge encoded in the schemas, the rewrite rules, and the support modules. Whereas AUTOFILTER has schemas for

Kalman filters and rewrite rules for linearization, AUTOBAYES has schemas for EM and clustering algorithms.

AUTOFILTER is related to other systems that generate code for a particular class of mathematical problems. In spirit it is similar to systems such as SciNapse [Akers et al. 1997], a problem solving environment for partial differential equations, or Ellman's systems for generating numerical simulation programs from differential equations [Ellman and Murata 1998] and physics-based animation programs from a specification of analytical dynamics [Ellman et al. 2002]. It differs mainly in its schema-based approach to code generation.

5. CONCLUSIONS AND FUTURE WORK

In this article, we have presented AUTOFILTER, a code generation system to automatically synthesize Matlab/C/C++ code for state estimation from a compact and concise specification. Using a schema based approach combined with a powerful symbolic mathematics module, AUTOFILTER can automatically convert the process and measurement model (given as a set of differential equations) into a Kalman filter algorithm. We have demonstrated the capabilities of AUTOFILTER on a case study within the realm of NASA: the state-estimation module for the Deep Space I spacecraft. AUTOFILTER can produce code that performs the appropriate state estimation task, and which easily can be incorporated into a target software architecture.

One extension area for AUTOFILTER would be to incorporate recent improvements to the Kalman filter algorithm, for example, an unscented Kalman filter [Wan and van der Merwe 2000]. Current specifications rely on the fact that process and measurement noise is a white Gaussian noise. This assumption is a direct prerequisite in order to instantiate a Kalman filter. We are planning to extend AUTOFILTER such that it can handle cases with non-Gaussian noises. In such cases, modern filter algorithms, for example, particle filter [Doucet et al. 2001], will automatically be generated.

Another area of extension will address the handling of sensor data. In most real applications, the state estimation module contains code to deal with sensor failures or to handle multiple sensor modes (e.g., used for different stages of a descent and landing process). A conservative extension of the AUTOFILTER specification language will allow us to specify sensor failures and sensor modes.

State estimation is a safety critical component of most flight software. Any error in specification, design, or implementation can lead to mission failures or even loss of life. Therefore, we are currently extending AUTOFILTER to support certification and review of the generated code. This means that it is not sufficient to *just* synthesize the code. Rather, we are developing a set of extensions for the synthesis system that can support a rigorous code review:

—the synthesized code is highly commented. In the current version, roughly one third of the lines of code are automatically generated comments. The structure and layout of the code is designed in such a way that the code reviewer can understand the code and easily relate it to the specification. An HTML version of the code can also be generated. It supports interactive navigation through the code.

—AUTOFILTER can generate a detailed design document. This design document combines all important pieces of information that show up during the synthesis process. Besides hyper-links to the specification and all generated artifacts (code, log files, etc), this document gives a detailed description of the interface and explicitly calls out detailed assumptions and design decisions.

We are confident that with an extended domain coverage and with certification support, the AUTOFILTER system will be a valuable tool for many state estimation problems in aircraft or spacecraft design or in robotics.

APPENDIX

A. AUTOFILTER SPECIFICATION LANGUAGE

The input language of AUTOFILTER has been designed to be close to the notations used by domain experts (i.e., vectors, matrices, and differential equations), yet it allows concise specifications of state estimation problems. An AUTOFILTER specification consists of 5 individual parts: *declarations*, specification of the *process* and *measurement* models, details about software *architecture*, and the *synthesis goal*. In the following, we give an abbreviated formal definition of the language.

A.1 Declarations

In this part of the AUTOFILTER specification, all constants and variables are declared. Each declared variable (`DECL`) is a scalar, vector, or matrix of the basic data types `nat`, `int`, or `double`. For vectors and matrices, the dimensions are specified with a lower and upper bound. These bounds can be arbitrary expressions, for example, `double m(0..10, 0..n-1)`. Variables and constants can be explicitly initialized. Some AUTOFILTER variables are statistical variables that have a distribution, for example, the process noise. The Gaussian distribution is given by a mean (usually 0) and the standard deviation as, for example `z ~ gauss(0, r)`. `IND` are all-quantified specification variables to denote generic indices. `x(I) ~ gauss(0, sigma(I)` means that each element of the vector `x` has a zero-mean Gaussian distribution with a standard deviation corresponding to the vector element `sigma(I)`.

```
DECL ::=    [ const | data ] TYPE VAR [ := EXPR ] [ as COMMENT ]
              | IVAR ~ gauss(EXPR, EXPR)
TYPE ::=    nat | int | double
VAR  ::=    NAME | NAME(EXPR .. EXPR) | NAME(EXPR .. EXPR, EXPR .. EXPR)
IVAR ::=    NAME | NAME(IND) | NAME(IND, IND)
COMMENT ::= STRING+
```

Expressions `EXPR` are scalar arithmetic expressions; vector and matrix accesses are done in a FORTRAN-style: `NAME(EXPR)`, and `NAME(EXPR, EXPR)`, respectively.

A.2 Process and Measurement Models

Both process and measurement models can be specified in various ways. The process model defines how the (noisy) plant, defined by its state vector, develops

over time; the measurement model how the measurements relate to the state vector. These models are thus given as a set of equations over the state vectors. For continuous models, differential equations ($\dot{\mathbf{x}} = \mathcal{F}(\mathbf{x})$) are used; a discrete model is given as a set of difference equations ($\mathbf{x}_{k+1} = \mathcal{F}(\mathbf{x}_k)$).

```
MODEL_DEF ::=  equation_set NAME is DISC_EQU+ | CONT_EQU+ end
DISC_EQU ::=   NAME(EXPR)_k+1 := EXPR
CONT_EQU ::=   d/dt  NAME(EXPR) := EXPR
```

A.3 Control, Interfaces, and Synthesis Goal

This section of an AUTOFILTER specification contains concise information on the architecture of the desired filter, on the interface, and additional information (e.g., on the filter initialization). The name of the Kalman filter is given after the `estimator` keyword. Then various properties of the filter are specified. Below is a list of the most important properties.

```
CONTROL    ::=  estimator NAME SET_ATTR+ SYNTH_GOAL
SET_ATTR   ::=
    NAME .steps := EXPR               % number of filter execution steps
  | NAME .process_eqs := NAME         % name of process equation set
  | NAME .measurement_eqs := NAME     % name of measurement equation set
  | NAME .initials := NAME(IND)       % initial values state vector
  | NAME .initial_covariance := EXPR  % initial covariance matrix

SYNTH_GOAL ::= output FNAME

FNAME ::= NAME | NAME par FNAME
```

The synthesis goal `output` finally triggers the synthesis of the entire Kalman filter with name `FNAME`. The desired output may be a single filter or a number of filters running in parallel (defined by `par`). The full language also supports the definition of operating modes in which different filters run in each mode, but this is not shown here. Also not shown are the language constructs for specifying how the output interfaces to the environment (which may involve parameter handling and separation of filter steps into separate functions).

ACKNOWLEDGMENTS

The authors would like to thank staff at JPL, Harry Balian and Abdullah Aljabri, who provided the case study in Section 3; Tom Pressburger who assisted with checking the AUTOFILTER generated code and in testing for the DS1 case study; and Ewen Denney, Pramod Gupta and Julian Richardson for comments on the article.

REFERENCES

AKERS, R., KANT, E., RANDALL, C., STEINBERG, S., AND YOUNG, R. 1997. SciNapse: A problem-solving environment for partial differential equations. *IEEE Comp. Sci. Eng. 4*, 3, 32–42.

BROWN, R. G. AND HWANG, P. 1997. *Introduction to Random Signals and Applied Kalman Filtering*. John Wiley & Sons.

CLOCKSIN, W. F. AND MELLISH, C. S. 1984. *Programming in Prolog*. Springer Verlag.

DOUCET, A., DE FREITAS, N., AND GORDON, N. 2001. *Sequential Monte Carlo Methods in Practice*. Springer Verlag.

ELLMAN, T., DEAK, R., AND FOTINATOS, J. 2002. Knowledge-based synthesis of numerical simulation programs for rigid-body in physics based animation. In *The 17th IEEE International Conference on Automated Software Engineering*. IEEE Computer Society.

ELLMAN, T. AND MURATA, T. 1998. Deductive synthesis of numerical simulation programs from algebraic and ordinary differential equations. In *The 13th IEEE International Conference on Automated Software Engineering*. IEEE Computer Society.

FISCHER, B., SCHUMANN, J., AND PRESSBURGER, T. 2000. Generating data analysis programs from statistical models. In *Workshop on Semantics, Applications, and Implementation of Program Generation*, W. Taha, Ed. Springer, 212–229.

GREWAL, M. AND ANDREWS, A. 1993. *Kalman filtering: Theory and Practice*. Prentice Hall.

LEFFERTS, E., MARKLEY, F., AND SHUSTER, M. 1982. Kalman filtering for spacecraft attitude estimation. *J. Guidance and Control 5*, 5, 417–429.

MURPHY, K. 2002. Kalman filter toolbox `http://www.ai.mit.edu/~murphyk/software/kalman/kalman.html`.

NOORGAARD, M. 2002. The KALMTOOL toolbox: `http://www.iau.dtu.dk/research/control/kalmtool.html`.

Octave 2003. GNU Octave `http://www.octave.org`.

ROSU, G. AND WHITTLE, J. 2002a. Towards certifying domain specific properties of synthesized code. In *Verification and Computational Logic (VCL02)*. Pittsburgh, PA.

ROSU, G. AND WHITTLE, J. 2002b. Towards certifying domain specific properties of synthesized code—extended abstract. In *Proceedings of the Conference on Automated Software Engineering (ASE02)*. Edinburgh, UK.

ROUMELIOTIS, S., SUKHATME, G., AND BEKEY, G. 1998. Fault detection and isolation in a mobile robot using multiple-model estimation. In *IEEE International Conference on Robotics and Automation*. IEEE Computer Society, 2223–2228.

VAN DER MERWE, R. AND WAN, E. 2003. ReBEL: Recursive Bayesian estimation library `http://choosh.ece.ogi.edu/rebel/index.html`.

WAN, E. AND VAN DER MERWE, R. 2000. The Unscented Kalman filter for nonlinear estimation. In *Proceedings of 2000 Symposium on Adaptive Systems for Signal Processing, Communication and Control (AS-SPCC)*.

WHALEN, M., SCHUMANN, J., AND FISCHER, B. 2002. Synthesizing certified code. In *Formal Methods Europe*. Springer, 431–450.

Received April 2003; revised March 2004; accepted June 2004