Certifying Optimality of State Estimation Programs

Grigore Roşu[1], Ram Prasad Venkatesan[1], Jon Whittle[2], and Laurenţiu Leuştean[3]

[1] Department of Computer Science, University of Illinois at Urbana-Champaign
[2] NASA Ames Research Center, California
[3] National Institute for Research and Development in Informatics, Bucharest

{grosu,rpvenkat}@cs.uiuc.edu, jonathw@email.arc.nasa.gov, leo@ici.ro

Abstract. The theme of this paper is certifying software for state estimation of dynamic systems, which is an important problem found in spacecraft, aircraft, geophysical, and in many other applications. The common way to solve state estimation problems is to use *Kalman filters*, i.e., stochastic, recursive algorithms providing statistically optimal state estimates based on noisy sensor measurements. We present an optimality certifier for Kalman filter programs, which is a system taking a program claiming to implement a given formally specified Kalman filter, as well as a formal certificate in the form of assertions and proof scripts merged within the program via annotations, and tells whether the code correctly implements the specified state estimation problem. Kalman filter specifications and certificates can be either produced manually by expert users or can be generated automatically: we also present our first steps in merging our certifying technology with AutoFilter, a NASA Ames state estimation program synthesis system, the idea being that AutoFilter synthesizes proof certificates together with the code.

1 Introduction

A common software development task in the spacecraft navigation domain is to design a system that can estimate the attitude of a spacecraft. This is typically mission-critical software because an accurate attitude estimate is necessary for the spacecraft controller to tilt the craft's solar panels towards the sun. Attitude estimators for different spacecraft, as well as a spectrum of other estimators for dynamic systems in general, are typically variations on a theme, solving a *state estimation problem*. The common way to solve state estimation problems is to use *Kalman filters*. A Kalman filter is essentially a set of mathematical equations implementing a predictor-corrector type estimator that is *optimal* in the sense that it minimizes the estimated error between the real and the predicted states. Since the time of their introduction [8] in 1960, Kalman filters have been the subject of extensive research and application, particularly in the area of autonomous or assisted navigation. This is likely due in large part not only to advances in digital computing that made their use practical, but also to the relative simplicity and robust nature of the filter itself.

Despite their apparent simplicity, implementations of state estimation problems are quite hard to prove correct. Correctness in this context means that the state calculated by a given implementation, called the state estimate, is mathematically optimal when one considers all the hypotheses given in the description of the state estimation problem. Descriptions of such problems are given as sets of stochastic difference equations, and optimality is rigorously, statistically formulated using matrix differentiation. Since this domain is not common in computer-aided verification, we find it appropriate to dedicate an entire section, Section 2, to Kalman filters, their subtleties (especially the assumptions under which they can be proved optimal), as well as to their variations.

Due to the need of accurate state estimates in critical applications, it is important to be able to provide optimality guarantees whenever possible. In this paper we present work in progress on a verification environment for the nontrivial class of domain-specific programs solving state estimation problems. Our long term goal is to provide a domain formalization, including axiomatization of several segments of mathematics together with significant lemmas, and a friendly tool making use of it to formally certify optimality. Up to this moment we were able to develop a common framework and a prototype tool with which we were able to certify three of the most important Kalman filters, the simple Kalman filter, the information filter and the extended Kalman filter. One would, of course, like to certify software as automatically as possible, but this is very rarely feasible due to intractability arguments and clearly close to impossible for the complex domain presented in this paper. Therefore, user intervention is often needed to insert domain-specific knowledge into the programs to be certified, usually under the form of code annotations. The certifier in this paper needs annotations for model specifications, assertions, and proof scripts. It is mostly implemented in Maude [5], a freely distributed high-performance executable specification system in the OBJ [7] family, supporting both rewriting logic [11] and membership equational logic [12]. Because of its efficient rewriting engine and because of its metalanguage and modularization features, Maude is an excellent tool to develop executable environments for various logics, models of computation, theorem provers, and even programming languages. The work in this paper falls under what was called *domain-specific certification* in [10].

A growth area in the last couple of decades has been code generation. Although commercial code generators are mostly limited to generating stub codes from high level models (e.g., in UML), program synthesis systems that can generate fully executable code from high level behavioral specifications are rapidly maturing (see, for example, [20,18]), in some cases to the point of commercialization (e.g., SciNapse [1]). In program synthesis, there is potential for automatically verifying nontrivial properties because additional background information – from the specification and the synthesis knowledge base – is available. Following the ideas in [16,15], we show how we coupled together AutoFilter, a NASA Ames synthesis system for the state estimation domain, and our prototype certifier.

The main idea here is to modify AutoFilter to synthesize not only the code, but also the appropriate formal annotations needed by the certifier.

Due to space limitation, we only give a high level overview of our work in optimality certification of state estimates. The reader is referred to a 50 page report [9] presenting in detail a previous version of this work. Important related work includes proof-carrying code [14] and extended static checking [3,17].

2 Kalman Filters

A Kalman filter is essentially a set of mathematical equations implementing a predictor-corrector type estimator that is *optimal* in the sense that it minimizes the estimated *error* covariance - when some assumptions are met. Since the time of their introduction [8], Kalman filters have been the subject of extensive research and application, particularly in the area of autonomous or assisted navigation. This is likely due in large part to advances in digital computing that made their use practical, but also to the relative simplicity and robust nature of the filter itself. We have tested our state estimation certification technique on three Kalman filters in current use, the simple Kalman filter, the information filter and the extended Kalman filter, which are briefly discussed next.

The *simple Kalman filter* addresses the general problem of estimating the state $x \in \mathbb{R}^n$ of a discrete-time controlled system that is governed by the linear stochastic difference equation for x with measurement $z \in \mathbb{R}^m$:

$$x_{k+1} = \Phi_k x_k + w_k \quad (1) \qquad\qquad z_k = H_k x_k + v_k. \quad (2)$$

x_k is the process state vector at time k. For example, the state vector x_k might contain three variables representing the rotation angles of a spacecraft. Equation (1) is the *process model*, describing the state dynamics over time – the state at time $k+1$ is obtained by multiplying the state transition matrix Φ_k by the previous state x_k. The model is imperfect, however, as represented by the addition of the process noise vector w_k. Equation (2) is the *measurement model* and models the relationship between the measurements and the state. This is necessary because the state usually cannot be measured directly. The measurement vector, z_k, is related to the state by matrix H_k. The random vectors w_k and v_k represent the process and measurement noise, respectively, and they are assumed to be independent of each other, white, and with normal distribution:

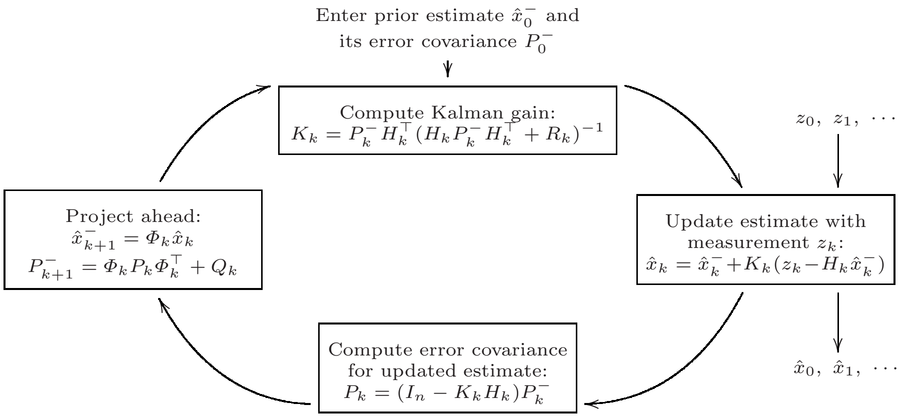
$$p(w_k) \sim N(0, Q_k) \quad (3) \qquad E[w_k w_i^\top] = \begin{cases} Q_k, & \text{if } i = k \\ 0, & \text{if } i \neq k \end{cases} \quad (6)$$

$$p(v_k) \sim N(0, R_k) \quad (4)$$

$$E[w_k v_i^\top] = 0 \quad (5) \qquad E[v_k v_i^\top] = \begin{cases} R_k, & \text{if } i = k \\ 0, & \text{if } i \neq k. \end{cases} \quad (7)$$

As an example of how the simple Kalman filter works in practice, consider a simple spacecraft attitude estimation problem. Attitude is usually measured using gyroscopes, but the performance of gyroscopes degrades over time so the error in the gyroscopes is corrected using other measurements, e.g., from a star tracker. In this formulation, the process equation (1) would model how the gyroscopes degrade and the equation (2) would model the relationship between the star tracker measurements and the three rotation angles that form the state (in this case, H_k would be the identity matrix because star trackers measure rotation angles directly). From these models, a Kalman filter implementation would produce an optimal estimate of the current attitude, where the uncertainties in the problem (gyro degradation, star tracker noise, etc.) have been minimized.

Before we present the implementation of the simple Kalman filter, we need several important notions. Let us define $\hat{x}_k^- \in \mathbb{R}^n$ to be the *a priori state estimate* at step k given knowledge of the process prior to step k, and $\hat{x}_k \in \mathbb{R}^n$ be the *a posteriori state estimate* at step k given measurement z_k, with their *a priori* and *a posteriori estimate errors* $e_k^- = x_k - \hat{x}_k^-$ and $e_k = x_k - \hat{x}_k$, respectively. The *a priori estimate error covariance* is then $P_k^- = E[e_k^-(e_k^-)^\top] = E[(x_k - \hat{x}_k^-)(x_k - \hat{x}_k^-)^\top]$ and the *a posteriori estimate error covariance* is $P_k = E[e_k {e_k}^\top] = E[(x_k - \hat{x}_k)(x_k - \hat{x}_k)^\top]$. We define also the *measurement prediction* as $z_k^- = H_k \hat{x}_k^-$.


Enter prior estimate $\hat{x}_0^-$ and its error covariance P_0^-
Compute Kalman gain: $K_k = P_k^- H_k^\top (H_k P_k^- H_k^\top + R_k)^{-1}$
$z_0,\ z_1,\ \cdots$
Update estimate with measurement z_k: $\hat{x}_k = \hat{x}_k^- + K_k(z_k - H_k\hat{x}_k^-)$
$\hat{x}_0,\ \hat{x}_1,\ \cdots$
Compute error covariance for updated estimate: $P_k = (I_n - K_k H_k)P_k^-$
Project ahead: $\hat{x}_{k+1}^- = \Phi_k \hat{x}_k$ $P_{k+1}^- = \Phi_k P_k \Phi_k^\top + Q_k$

```
input xhatmin(0), Pminus(0);
for(k, 0, n) {
  zminus(k) := H(k) * xhatmin(k);
  gain(k) := (Pminus(k)*trans(H(k)))*minv(((H(k)*Pminus(k))*trans(H(k)))+R(k));
  xhat(k) := xhatmin(k) + (gain(k) * (z(k) - zminus(k)));
  P(k) := (id(n) - (gain(k) * H(k))) * Pminus(k);
  xhatmin(k + 1) := Phi(k) * xhat(k);
  Pminus(k + 1)  := ((Phi(k) * P(k)) * trans(Phi(k))) + Q(k);}
```

Fig. 1. Simple Kalman filter loop and intermediate code.

In deriving the equations for the Kalman filter program, one needs to first find an equation that computes an a posteriori estimate $\hat{x}_k$ as a linear combination of an a priori estimate $\hat{x}_k^-$ and a weighted difference between the actual measurement z_k and the measurement prediction z_k^- as shown below:

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - z_k^-). \tag{8}$$

where $(z_k - z_k^-)$ is called the *measurement innovation*, or the *residual*, and reflects the discrepancy between the predicted and the actual measurements. The $n \times m$

matrix K_k is chosen to be the *gain*, or *blending factor*, that minimizes the a posteriori estimate error covariance P_k. One form of the Kalman gain is

$$K_k = P_k^- H_k^\top (H_k P_k^- H_k^\top + R_k)^{-1}, \tag{9}$$

which, after hundreds of basic equational steps, yields the covariance matrix $P_k = (I_n - K_k H_k)P_k^-$. Figure 1 gives a complete picture of the simple Kalman filter, both as a diagram and as intermediate code that AutoFilter generates.

AutoFilter takes as input a mathematical specification including equations (1) - (7) and also descriptions of the noise characteristics and filter parameters, and first generates code like in Figure 1 and then translates it into C++, Matlab or Octave. In this paper we only consider code in the intermediate language. Despite its apparent simplicity, the proof of optimality for the simple Kalman filter is quite complex. The main proof task is to show that the vector $\hat{x}_k$ is the best estimate of the state vector x_k at time k, under appropriate simplifying assumptions, and is usually presented in books informally on several pages (see [2], for example). In the sequel, we sketch this proof, emphasizing those aspects which are particularly relevant for its mechanization, especially the *assumptions.*

The very first assumption is that $\hat{x}_0^-$ and P_0^- are the best initial prior estimate and its error covariance matrix. Another assumption is that the measurement prediction at any given time k, z_k^-, is the most probable measurement. The most important assumption says that the best estimate $\hat{x}_k$ is a *linear* blending of the residual and the prior estimate (8). The justification for this assumption is rooted in the probability of the prior estimate $\hat{x}_k^-$ conditioned on all the prior measurements z_k (see [2,19] for more details). Formally, this says that the best estimate $\hat{x}_k$ is somewhere in the image of the function $\hat{x}_k(y) := \lambda y.(\hat{x}_k^- + y(z_k - z_k^-))$, where the blending factor y is an $n \times m$ matrix. If $P_k(y) := E[(x_k - \hat{x}_k(y))(x_k - \hat{x}_k(y))^\top]$ is the a posteriori error covariance matrix regarded as a function of y, then we wish to find the particular y that minimizes the individual terms along the major diagonal of $P_k(y)$, because these terms represent the estimation error covariances for the elements of the state vector being estimated. Using another assumption, that the individual mean-square errors are also minimized when the total is minimized, our problem reduces to finding the y that minimizes the trace, $\mathit{trace}(P_k(y))$, of $P_k(y)$, where the trace of a square matrix is the sum of the elements on its major diagonal. This optimization is done using a differential calculus approach. Differentiation of matrix functions is a complex field that we partially formalized and which we cannot cover here, but it is worth mentioning, in order for the reader to anticipate the non-triviality of this proof, that the y we are looking for is the solution of the equation $d(\mathit{trace}(P_k(y)))/dy = 0$, where for a standard function $f(y_{11}, y_{12}, \ldots)$ on the elements of the matrix y, such as $\mathit{trace}(P_k(y))$, its derivative df/dy is the matrix $(df/dy_{ij})_{ij}$ having the same dimension $n \times m$ as y. Using two important differentiation lemmas, namely "$d(\mathit{trace}(yA))/dy = A^\top$ if yA is a square matrix" and "$d(\mathit{trace}(yAy^\top))/dy = 2yA$ if A is a symmetric matrix", after several thousands of basic proof steps one gets the desired solution, the so called Kalman gain (9). The a posteriori best estimate, $\hat{x}_k := \hat{x}_k(K_k)$, and the covariance matrix associated with the optimal estimate, $P_k(K_k)$, can now also be calculated by equational reasoning, and the updated estimate $\hat{x}_k$ is "projected ahead" via the transition matrix, $\hat{x}^-_{k+1} = \Phi_k \hat{x}_k$. The fact $\hat{x}^-_{k+1}$ is the best prior estimate at time $k+1$ follows by another important assumption, saying that the best prior estimate at the next step follows the state equation (1) using the best estimate at the current state, but where the contribution of the process noise w_k is ignored (we are justified in doing this because the noise w_k has zero mean and is not correlated with any one of the previous w's). By equational reasoning it follows now that P^-_{k+1}, the error covariance matrix of $\hat{x}^-_{k+1}$, is $\Phi_k P_k \Phi_k^\top + Q_k$.

The flow of calculations above is for simple Kalman filters, but its equations can be algebraically manipulated into a variety of forms. An alternative form is known as the *information filter* [2], which additionally assumes that the matrices P^-_k, P_k, and R_k admit inverses. $(P^-_k)^{-1}$ is thought of as a measure of the information content of the a priori estimate. Then $P_k = \infty$, i.e., $(P^-_k)^{-1} = 0$, corresponds to infinite uncertainty, or zero information. This leads to the indeterminate form $\frac{\infty}{\infty}$ in the Kalman gain expression (9), so one can not apply the simple Kalman filter due to rounding errors. The information filter accommodates this situation and is based on the equations (which can be obtained as above):

$$P_k^{-1} = (P^-_k)^{-1} + H_k^\top R_k^{-1} H_k \qquad (10) \qquad\qquad K_k = P_k H_k^\top R_k^{-1}. \quad (11)$$

The Kalman filters discussed so far address the general problem of estimating the state $x \in \mathbb{R}^n$ of a discrete-time controlled process that is governed by a *linear* stochastic difference equation. However, some of the most interesting and successful applications of Kalman filters are *non-linear*, i.e., the process and measurement models are given by equations of the form

$$x_{k+1} = f(x_k, u_k) + w_k \qquad (12) \qquad\qquad z_k = h(x_k) + v_k, \quad (13)$$

where f and h are non-linear functions, u_k is a deterministic forcing function (regard it as an input), and the random vectors w_k and v_k again represent the process and the measurement noise and satisfy the same conditions as for the simple Kalman filter. To simplify computations and to make the problem implementable, one can linearize it about a trajectory that is continually updated with the state estimates resulting from the measurements. The new filter obtained this way is called *extended Kalman filter* (or simply *EKF*). Since the noises w_k and v_k have mean 0 and since $\hat{x}_k$ and $\hat{x}^-_k$ are estimates anyway, one can approximate the a priori state estimate and measurement prediction vectors as $x^-_{k+1} = f(\hat{x}_k, u_k)$, and $z^-_k = h(\hat{x}^-_k)$, respectively. Taking Φ_k and H_k the Jacobian matrices

$$\Phi_k = (\frac{\delta f_i}{\delta x_j}(\hat{x}_k, u_k))_{i,j}, \qquad\qquad H_k = (\frac{\delta h_i}{\delta x_j}(\hat{x}^-_k))_{i,j},$$

one can approximate f and h with their first order Taylor series expansions

$$f(x_k, u_k) = f(\hat{x}_k, u_k) + \Phi_k(x_k - \hat{x}_k), \qquad h(x_k) = h(\hat{x}^-_k) + H_k(x_k - \hat{x}^-_k).$$

One can get the new governing equations that linearize an estimate

$$x_{k+1} = \hat{x}^-_{k+1} + \Phi_k(x_k - \hat{x}_k) + w_k, \qquad z_k = z^-_k + H_k(x_k - \hat{x}^-_k) + v_k,$$

which can be solved and implemented following closely the simple Kalman filter.

3 Specifying and Annotating Kalman Filters

In order to perform computer-aided optimality certification of programs implementing sophisticated state estimation problems like the ones above, one first needs to rigorously specify the statistical problem together with all its needed *assumptions*. In fact, an initially unexpected important benefit of our technique, whose value we discovered progressively during experiments, is that it makes our user aware of all the assumptions under which a Kalman filter program indeed calculates the expected optimum state estimate, which usually include strictly those mentioned by authors of theorems in textbooks[1]. These specifications are defined on top of an axiomatically formalized abstract domain knowledge theory, including matrices, differentiation and probability theory (presented in the next section), and use Maude membership equational logic notation [6,5], which is natural but we do not explain here. The simple Kalman filter, for example, has 45 axioms (properties and assumptions); we comment on a few of these next.

Operations or constants declaring the matrices and vectors involved together with their dimensions are defined first:

```
ops x z v w : Nat -> RandomMatrix .          ops R Q Phi H : Nat -> Matrix .
ops n m p : -> Nat .                         var K : Nat .
eq row(x(K)) = n .   eq column(x(K)) = 1 .    eq row(w(K)) = n .  eq column(w(K)) = 1 .
eq row(Phi(K)) = n . eq column(Phi(K)) = n .  eq row(z(K)) = m .  eq column(z(K)) = 1 .
eq row(v(K)) = m .   eq column(v(K)) = 1 .    eq row(H(K)) = m .  eq column(H(K)) = n .
...
```

The sort `Nat` comes from a Maude builtin module, while the sorts `Matrix` and `RandomMatrix` are defined within the abstract domain, presented in the next section, and stay for random matrix variables and for matrices, respectively. Other axioms specify model equations (which are labeled for further use), such as

```
[ax*> | KF-next]          eq x(K + 1) = Phi(K) * x(K) + w(K) .
[ax*> | KF-measurement]   eq z(K) = H(K) * x(K) + v(K) .
[ax*> | RK]               eq R(K) = E| v(K) * trans(v(K)) | .
[ax*> | QK]               eq Q(K) = E| w(K) * trans(w(K)) | .
```

Operations `_*_`, `_+_`, `trans` (matrix transpose) and `E|_|` (error covariance) on matrices or random matrices are all axiomatized in the abstract domain. Other assumptions that we do not formalize here due to space limitation include independence of noise, and the fact that the best prior estimate at time `k + 1` is the product between `Phi(k)` and the best estimate calculated previously at step `k`. One major problem that we encountered while developing our proofs following textbook proofs was that these assumptions, and many others not mentioned here, are so well and easily accepted by experts that they don't even make their

[1] Which is understandable, otherwise theorems would look too heavy for humans.

use explicit in proofs. One cannot do this in formal proving, so one has the nontrivial task to detect and then declare them explicitly as special axioms.

In order to machine check proofs of optimality, they must be properly decomposed and linked to the actual code. This can be done in many different ways. For simplicity, we prefer to keep everything in one file. This can be done by adding the specification of the statistical model at the beginning of the code, and then by adding appropriate formal statements, or assertions, as annotations between instructions, so that one can prove the next assertion from the previous ones and the previous code. Formal proofs are also added as annotations where needed. Notice that by "proof" we here mean a series of *hints* that allows our proof assistant and theorem prover, Maude's Inductive Theorem Prover (ITP) [4], to generate and then check the detailed proof. The next shows how the 8 line simple Kalman filter code in Figure 1 is annotated in order to be automatically verifiable by our optimality certifier. In order to keep the notation simple, we either removed or replaced by plain English descriptions the formal specification, assertions and proofs occurring in the code as comments. It may be worth mentioning that the entire annotated simple Kalman filter code has about 300 lines, that it compresses more than 100,000 basic proof steps as generated by ITP, and that it takes about 1 minute on a 2.4GHz standard PC to generate all the proofs from their ITP hints and then check them:

```
/* Specification of the state estimation problem ... about 45 axioms/assumptions */
1.  input xhatmin(0), Pminus(0);
/* Proof assertion 1: ... */
/* Assertion 1: xhatmin(0) and Pminus(0) are best prior estimate and its error covar. */
2.  for(k,0,n) {
/* Assertion 2: xhatmin(k) and Pminus(k) are best prior estimate and its error covar. */
3.    zminus(k) := H(k) * xhatmin(k);
4.    gain(k) := Pminus(k) * trans(H(k)) * minv(H(k) * Pminus(k) * trans(H(k)) + R(k));
/* Proof assertion 3: ... */
/* Assertion 3: gain(k) minimizes the error covariance matrix */
5.    xhat(k) := xhatmin(k) + (gain(k) * (z(k) - zminus(k)));
/* Proof assertion 4: ... */
/* Assertion 4: (the main goal) xhat(k) is the best estimate */
6.    P(k) := (id(n) - (gain(k) * H(k))) * Pminus(k);
/* Proof assertion 5: ... */
/* Assertion 5: P(k) is the error covariance matrix of xhat(k) */
7.    xhatmin(k + 1) := Phi(k) * xhat(k);
8.    P(k + 1)  := ((Phi(k) * P(k)) * trans(Phi(k))) + Q(k);
/* Proof assertion 2 at time k + 1: ... */
}
```

The proof assertions and proofs above should be read as follows: proof assertion n is a proof of assertion n in its current environment. The best we can assert between instructions 1 and 2 is that `xhatmin(0)` and `Pminus(0)` are initially the best prior estimate and error covariance matrix, respectively. This assertion is an assumption in the theory of Kalman filters, axiom in our specification, so it can be immediately checked. Between 2 and 3 we assert that `xhatmin(k)` and `Pminus(k)` are the best prior estimate and its error covariance matrix, respectively. This is obvious for the first iteration of the loop, but needs to be proved for the other iterations. Therefore, our certifier performs an implicit proof by induction. The assertion after line labeled 4 is that `gain(k)` minimizes the a posteriori error covariance matrix. This was the part of the proof that was the most difficult to formalize. We show it below together with its corresponding ITP proof script:

```
[proof assertion-3]
(set (instruction-1-1 instruction-2 assertion-1-1 assertion-1-2 Estimate KF-measurement
      zerro-cross_v_x-xhatmin_1 zerro-cross_v_x-xhatmin_2 RK id*left id*right id-trans
      distr*trans minus-def1 misc-is E- E+ E*left E*right trans-E fun-scalar-mult
      fun-mult fun-trans fun-def1 fun-def2 minimizes trace+ trace- tracem-def1
      tracem-def2 trace-lemma1 trace-lemma1* trace-lemma2 lemma-1 lemma-2 lemma-3
      lemma-4 lemma-5 lemma-6 lemma-7 lemma-8 lemma-9 lemma-10 lemma-11) in (1) .)
(rwr (1) .)
(idt (1) .)
---------------------------------------------------------------------- */
/* ----------------------------------------------------------------------
[assertion-3]
  eq gain(K) minimizes /\ y . (E|(x(K) - Estimate(K,y)) * trans(x(K) - Estimate(K,y))|)
     = (true) .
---------------------------------------------------------------------- */
```

Hence, we first "set" 44 axioms, lemmas, and/or previous assertions or instructions, all referred to by their labels, and then simplify the proof task by rewriting with the ITP command `rwr`. This particular proof can be done automatically (`idt` just checks for identity). The axioms are either part of the particular Kalman filter specification under consideration (such as those on the second row) or part of the axiomatization of the abstract domain (such as those on the third and fourth rows), which is general to all Kalman filters. The lemmas are essentially properties of the state estimation domain, so they belong to the abstract domain. As explained in the next section, these lemmas have been devised by analyzing several concrete state estimation problems and extracting their common features.

4 Specifying the Abstract Domain Knowledge

To generate and automatically certify the optimality proofs discussed so far, one needs to first formalize the state estimation *domain knowledge*, which includes matrices, random matrices, functions on matrices, and matrix differentiation. Formalizing the abstract domain was by far the most tedious part of this project, because it suffered a long series of refinements and changes as new and more sophisticated state estimation problems were considered. Since most of the operations on matrices are partial, since domain specifications are supposed to be validated by experts and since our work is highly experimental at this stage, we decided to use Maude [5] and its ITP tool [4] to specify and prove properties of our current domain knowledge, because these systems provide implicit strong support for partiality[2] (via memberships), their specifications are human readable due to the mix-fix notation, and can be easily adapted or modified to fulfill our continuously changing technical needs. Our current domain passed the criterion telling if a total theory can be safely regarded as partial [13]. However, we think that essentially any specification language could be used instead, if sufficient precautions are taken to deal properly with partial operations.

Matrices and Random Matrices. Matrices are extensively used in all state estimation problems and their optimality proofs, so we present their formalization first. Four sorts, in the subsort lattice relationship (using Maude notation)

[2] We are not aware of any systems providing explicit support for partiality.

"`subsorts Matrix < MatrixExp RandomMatrix < RandomMatrixExp`", have been introduced. Random matrices are matrices whose elements can be random variables, such as the state to be estimated, the measurements and/or the noise, and (random) matrix expressions can be formed using matrix operators. Most of the operations and axioms/lemmas in matrix theory are *partial*. For example, multiplication is defined iff the number of columns of the first matrix equals the number of rows of the second. It is a big benefit, if not the biggest, that Maude provides support for partiality, thus allowing us to compactly specify matrix theory and do partial proofs. The resulting sort (or rather "kind") of a partial operator is declared between brackets in Maude; for example, the partial operation of multiplication is defined as "`op _+_ : MatrixExp MatrixExp -> [MatrixExp]`". Transpose of a matrix is total, so it is defined as "`op trans : MatrixExp -> MatrixExp`". Membership assertions, stating when terms have "proper" sorts, can now be used to say when a partial operation on matrices is defined (two total operators, "`ops row column : MatrixExp -> Nat`", are needed). For example, the following axioms define multiplication together with its dimensions:

```
vars P Q R : MatrixExp .                        cmb P*Q : MatrixExp if column(P) == row(Q).
ceq column(P*Q) = column(Q) if P*Q : MatrixExp .  ceq row(P*Q) = row(P) if P*Q : MatrixExp.
```

Most of the matrix operators are *overloaded*, i.e., defined on both matrices and random matrices. Operators relating the two, such as the error covariance operator "`op E|_| : RandomMatrixExp -> MatrixExp`", together with plenty of axioms relating the various operators on matrices, such as distributivity, transpose of multiplications, etc., are part of the abstract domain theory.

Functions on Matrices. An important step in state estimation optimality proofs (see Section 2) is that the best estimate is a linear combination of the best prior estimate and the residual (8). The coefficient of this linear dependency is calculated such that the error covariance P_k is minimized. Therefore, before the optimal coefficient is calculated, and in order to calculate it, the best estimate vector is regarded as a *function* of the form $\lambda y.(\langle prior \rangle + y * \langle residual \rangle)$. In order for this function to be well defined, y must be a matrix having proper dimensions. We formally define functions on matrices and their properties by declaring new sorts, `MatrixVar` and `MatrixFun`, together with operations for defining functions and for applying them, respectively:

```
    op /\_._ : MatrixVar MatrixExp -> MatrixFun .
    op __ : MatrixFun MatrixExp -> [MatrixExp] .
    cmb (/\ X . P)(R) : MatrixExp if X + R : MatrixExp .
```

Several axioms on functions are defined, such as

```
    ceq (/\X.X)(R) = R if X + R : MatrixExp .
    ceq (/\X.(P+Q))(R) = (/\X.P)(R) + (/\X.Q)(R) if X + R : MatrixExp and P + Q : MatrixExp .
```

Matrix Differentiation. As shown in Section 2, in order to prove that $\hat{x}_k$ is the best estimate of the state vector x_k, a differential calculus approach is used. Axiomatization matrix differentiation can be arbitrarily complicated; our approach is top-down, i.e., we first define properties *by need*, use them, and then prove them from more basic properties as appropriate. For example, the only property used so far linking optimality to differentiation is that K minimizes $\lambda y.P$ iff $(d(trace(\lambda y.P))/dy)(K) = 0$. For that reason, to avoid deep axiomatizability of mathematics, we only defined a "derivative" operation "`op d|trace_|/d_ : MatrixFun MatrixVar -> MatrixFun`" with axioms like:

```
ceq (d|trace(/\X.X)|/d(X))(R) = id(row(X)) if X + R : MatrixExp .
ceq (d|trace(/\X.(P+Q))|/d(X))(R) = (d|trace(/\X.P)|/d(X))(R) + (d|trace(/\X.Q)|/d(X))(R)
   if X + R : MatrixExp and P + Q : MatrixExp .
```

The two important lemmas used in Section 2 are also added as axioms:

```
ceq d|trace(/\X.(X*P))|/d(X) = /\X.trans(P) if X * P : MatrixExp and not(X in P)) .
ceq d|trace(/\X.(X*P*trans(X)))|/d(X) = /\X.(2*X*P)
   if X * P : MatrixExp and P * trans(X) : MatrixExp and trans(P) == P and not(X in P) .
```

Domain-Specific Lemmas. One could, of course, prove the properties above from more basic properties of reals, traces, functions and differentiations, but one would need to add a significant body of mathematical knowledge to the system. It actually became clear at an early stage in the project that a database of domain-specific lemmas was needed. On the one hand, lemmas allow one to device more compact, modular and efficiently checkable proofs. On the other hand, by using a large amount of lemmas, the amount of knowledge on which our certification system is based can be reduced to just a few axioms of real numbers, so the entire system can be more easily validated and therefore accepted by domain experts. We currently have only 34 domain-specific lemmas, but their number is growing fast, as we certify more state estimation problems and refine the axiomatization of the abstract domain. These lemmas were proved using ITP [4], and were stored together with their proofs in a special directory. The user refers to a lemma by its unique label, just as to any axiom in the domain, and the certifier uses their proofs to synthesize and check the optimality proof.

5 Certifying Annotated Kalman Filters

There are several types and levels of certification, including testing and human code review. In this paper we address certification of programs for conformance with domain-specific properties. We certify that the computation flow of a state estimation program leads to an optimum solution. The reader should not get trapped by thinking that the program is thus "100% correct". There can be round-off or overflow errors, or even violations of basic safety policies, e.g., when a matrix uses the metric measurement unit system and another uses the English system, which our certifier cannot catch. What we guarantee is that, under the given hypotheses (i.e., the specification at the beginning of the code), the mathematics underlying the given code provably calculates the best state estimate of the specified dynamic system. The certifier presented next uses a combination of theorem proving and proof checking; the interested reader is encouraged to download the certifier and its manual from `http://fsl.cs.uiuc.edu`.

Cleaning the Code. The certifier first removes the insignificant details from the code, including empty blocks, comments that bear no relevance to the domain and statements that initialize the variables. We are justified in doing this because we are interested in the correctness of the mathematical flow of computation and not in the concrete values of the variables or matrices.

Generating Proof Tasks. The cleaned annotated Kalman filter, besides code and specification, contains assertions and proof scripts. By analyzing their labels and positions, the certifier generates a set of proof tasks. This is a technical process whose details will appear elsewhere, but the idea is that a task "$Domain{+}Spec{+}Before_A \models A$" is generated for each assertion A, where $Before_A$ is the knowledge accumulated before A is reached in the execution flow. Each generated task is placed in a separate file, together with an *expanded proof*, in ITP [4] notation. The expanded proofs are generated from the original proof scripts (which refer to lemmas via their names), by replacing each use of a lemma by an instance of its proof[3], which is taken from the database of domain-specific lemmas. The reason for doing so is based on the belief that certifying authorities have no reason to trust complex systems like ITP, but rather use their own simple proof checkers; in this case we would use ITP as a *proof synthesizer*.

Proof Checking. At this moment, however, we use ITP both as a proof generator and as a proof checker. More precisely, the certifier sends each proof task to ITP for validation. ITP executes the commands in the provided (expanded) proof script, and logs its execution trace in another file, so a skeptical user can double check it, potentially using a different checker.

The Tool. The certifier is invoked with the command `certifier [-cgv] pgm`. By default, it cleans, generates proof tasks and then verifies the annotated Kalman filter input. The cleaned code and the generated proof tasks by default are saved in appropriate files for potential further interest. Each of the options disable a corresponding action: `-c` disables saving the cleaned version, `-g` the generated proof tasks, and `-v` the verification step. Thus, `certifier -cv rover.code` would only generate the proof tasks associated to the state estimation code `rover.code`.

6 Synthesizing Annotated Kalman Filters

Like in proof-carrying code [14], the burden of producing the assertions and their proof scripts falls entirely on code producer's shoulders. Despite the relative support provided by proof-assistants like ITP, this can still be quite inconvenient if the producer of the code is a human. Not the same can be said if the producer of the code is another computer program. In this section we present our efforts in merging the discussed certification technology with a NASA Ames state estimation program synthesis system, called AutoFilter, thus underlying the foundations of what we called *certifiable program synthesis* in [16,15].

[3] This is similar in spirit to what is also known as "cut elimination".

AutoFilter takes a detailed specification of a state estimation problem as input, and generates intermediate code like the one in Figure 1 which is further transformed into C++, Matlab or Octave. It is built on an *algorithm schema* idea. A schema is a generic representation of a well-known algorithm. Most generally, it is a high-level description of a program which captures the essential algorithmic steps but does not necessarily carry out the computations for each step. In AutoFilter, a schema includes assumptions, applicability conditions, a *template* that describes the key steps of the algorithm, and the *body* of the schema which instantiates the template. Assumptions are inherent limitations of the algorithm and appear as comments in the generated code. Applicability conditions can be used to choose between alternative schemas. Note, however, that different schemas can apply to the same problem, possibly in different ways. This leads to choice points which 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, thus allowing it to generate multiple program variants for the same problem.

The main idea underlying the concept of certifiable program synthesis is to make a synthesis system generate not only code, but also *checkable correctness certificates*. In principle, this should be possible because any synthesis system worth its salt generates code from a logical specification of a problem, by performing formal reasoning, so it must be able to answer the question "why?" rigorously when it generates a certain block of code; otherwise, there is no legitimate reason to trust such a system. We are currently modifying AutoFilter to generate annotations together with its state estimation programs, in a form which is certifiable by the tool presented in the previous section. We are currently able to automatically certify any synthesized program which is an instance of a simple Kalman filter, and provide a general mechanism to extend it to all algorithm schemas. The annotations are stored with the program schemas at the *template* or *body* level. The template contains annotations that are global to the algorithm represented by that schema, e.g., the specification of the filter. The body contains annotations local to a particular piece of intermediate code used to instantiate the template, e.g., an assertion that `gain(k)` minimizes the covariance matrix *for a particular instantiation of the gain matrix and the covariance matrix.* Since AutoFilter can generate multiple implementations of each schema, by attaching annotations to a schema, it can generate annotations for each variation. The annotations of each schema are added by experts, who essentially formally prove each schema correct with respect to its hypotheses. The advantage of synthesis in this framework is that these hard proofs are done *only once* and then instantiated by the synthesis engine whenever needed.

References

1. R. Akers, E. Kant, C.Randall, S. Steinberg, and R.Young. Scinapse: A problem-solving environment for partial differential equations. *IEEE Computational Science and Engineering*, 4(3):32–42, 1997.

2. R.G. Brown and P. Hwang. *Introduction to Random Signals and Applied Kalman Filtering*. John Wiley & Son, 3rd edition, 1997.
3. Compaq : Extended Static Checking. `http://www.research.compaq.com/SRC/esc`.
4. M Clavel. ITP tool. Department of Philosophy, University of Navarre, `http://sophia.unav.es/ clavel/itp/`.
5. M. Clavel, F.J. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 285:187–243, 2002.
6. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In *Proceedings of WRLA'06*, volume 4 of *ENTCS*. Elsevier, 1996.
7. J.Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action*, pages 3–167. Kluwer, 2000.
8. R.E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME-Journal of Basic Engineering*, 82:35–45, 1960.
9. Laurenţiu Leuştean and Grigore Roşu. Certifying Kalman Filters. Technical Report TR 03-02, RIACS, 2003.
10. M. Lowry, T. Pressburger, and G.Roşu. Certifying domain-specific policies. In *Proceedings of ASE'01*, pages 81–90. IEEE, 2001. Coronado Island, California.
11. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, pages 73–155, 1992.
12. J. Meseguer. Membership algebra as a logical framework for equational specification. In *Proceedings of WADT'97*, volume 1376 of *LNCS*, pages 18–61, 1998.
13. J. Meseguer and G. Roşu. A total approach to partial algebraic specification. In *Proceedings of ICALP'02*, volume 2380 of *LNCS*, pages 572–584, 2002.
14. G.C. Necula. Proof-carrying code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.
15. G. Roşu and J. Whittle. Towards certifying domain-specific properties of synthesized code. In *Proceedings, Verification and Computational Logic (VCL'02)*, 2002. Pittsburgh, PA, 5 October 2002.
16. G. Roşu and J. Whittle. Towards certifying domain-specific properties of synthesized code (extended abstract). In *Proceedings of ASE'02*. IEEE, 2002.
17. K. Rustan, M. Leino, and Greg Nelson. An extended static checker for modula-3. In K. Koskimies, editor, *Compiler Construction: 7th International Conference, CC'98*, volume 1383 of *LNCS*, pages 302–305. Springer, April 1998.
18. Y. V. Srinivas and R. Jüllig. Specware: Formal support for composing software. In Bernhard Moller, editor, *Mathematics of Program Construction: third international conference, MPC '95*, volume 947 of *LNCS*, Kloster Irsee, Germany, 1995. Springer.
19. G. Welch and G. Bishop. An Introduction to the Kalman Filter. Course, SIGGRAPH 2001.
20. J. Whittle, J. van 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 *Proceedings of ASE'01*, San Diego, CA, USA, 2001.