Certifying Kalman Filters

Laurenţiu Leuştean
Grigore Roşu

RIACS Technical Report 03.02
January 2003

Certifying Kalman Filters[1]

Laurențiu Leuștean, National Institute for Research and Development in Informatics, Bucharest

Grigore Roșu, Department of Computer Science, University of Illinois at Urbana-Champagne

RIACS Technical Report 03.02
January 2003

Code certification is a lightweight approach to demonstrate software quality on a formal level. Its basic idea is to require code producers to provide formal proofs that their code satisfies certain quality properties. In this paper, we present a certifier for three Kalman Filters: Simple Kalman Filter, Information Filter and Extended Kalman Filter. Kalman Filters are stochastic, recursive algorithms which provide statistically optimal estimates of the state of a system based on noisy sensor measurements. They are the most common way of solving a state estimation problem, an important problem found in spacecraft, aircraft and geophysical applications.

[1]This work was supported in part by the National Aeronautics and Space Administration (NASA) under Cooperative Agreement NCC 2-1006 with the Universities Space Research Association (USRA).

Certifying Kalman Filters

Laurenţiu Leuştean[†]
National Institute for Research and Development in Informatics, Bucharest.
E-mail: leo@ici.ro

Grigore Roşu
Department of Computer Science, University of Illinois at Urbana-Champaign.
E-mail: grosu@cs.uiuc.edu

Abstract

Formal code certification is a rigorous approach to demonstrate software quality. Its basic idea is to require that code producers provide formal certificates, or proofs, that their code satisfies certain quality properties. In this paper, we focus on certifying software developed for state estimation of dynamic systems, which is an important problem found in spacecraft, aircraft, geophysical, and in many other applications. The most common way of solving state estimation problems consists of using *Kalman filters*, which are stochastic, recursive algorithms providing statistically optimal estimates of the state of a system based on noisy sensor measurements. We present a certifier for Kalman filter programs, which is a program that takes as input a program claiming to implement a Kalman filter together with a specification of that Kalman filter, as well as a certificate under the form of assertions and proof scripts merged within the program via annotations, and tells whether the code correctly implements the state estimation problem specified. We tested our certifier on three Kalman filters: simple Kalman filter, information filter and extended Kalman filter. So far we have played both the role of the producer of these programs and the role of the consumer, but our next step is to merge the presented technology with AutoFilter, a state estimation program sysnthesis system developed by researchers at NASA Ames, the idea being to have AutoFilter synthesize the correctness certificates together with the code.

1 Introduction

Guaranteeing properties of code has long been a major goal of the software analysis community and has been approached using static analysis as well as formal methods, such as model checking and theorem proving. Of these, static analysis is perhaps the most practical; commercial systems, usually based on abstract interpretation such as PolySpace [13], are now emerging. Unfortunately, practical applications of static analysis techniques are limited to checking programming language level properties such as illegal type conversions, invalid arithmetic operations (e.g., division by zero) and overflow/underflow, i.e., properties which could turn into runtime errors that would stop the normal execution of the program. Whilst important, such efforts do not address the issue of

[†]This research was done in Summer 2002, when the first author participated in the Summer Student Research Program (SSRP) sponsored by RIACS/USRA (see `http://www.riacs.edu/ssrp/SSRP2002.html`).

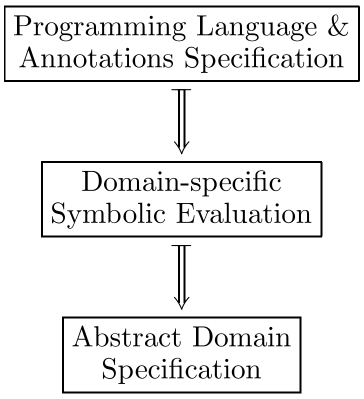

Programming Language & Annotations Specification
Domain-specific Symbolic Evaluation
Abstract Domain Specification

Figure 1: Domain-specific certifier.

how to guarantee more complex properties. Traditional formal methods techniques, such as model checking and/or theorem proving, do not always scale up well due either to their high computational complexity or to the necessary user intervention (to add and prove lemmas, etc.).

In this paper we adopt a theorem proving style and develop a verification environment for a *class* of programs and a *class* of properties, belonging to a specific domain of interest and respecting a specific set of constraints. We generically called our approach *domain-specific certification.* A domain-specific safety policy certifier was presented in [8] for the domain of coordinate frames, and one for safety with respect to units of measurement (e.g., one does not add "meters" and "seconds") was prototyped by the second author when he was a researcher at NASA Ames; these are both of crucial importance to many applications including astronomical navigation. Conceptually, a domain-specific certifier consists of three main components, as shown in Figure 1. The programming language and the abstract domain are linked via domain-specific abstract symbolic evaluation. Many software products are developed for domains that are quite complex and involve a significant body of mathematical knowledge, which is not the case for the semantics of standard programming languages in which these programs are usually written. Consequently, appropriate domain-specific *abstractions* of programming language constructs into domains of interest are required in order to perform domain-specific analysis of software. Unlike in standard static analysis of programs, where abstraction consists of grouping various values of the concrete domain into an abstract one (such as, each real number larger than 1 is "large positive"), in domain-specific analysis there might be no relationship between the concrete and the abstract domains; for example, 2.7 can be "meter", or "second", or "meter$*$Kg$*$second^{-2}", or any other measurement unit. 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.

In this paper, we focus on certifying software developed for state estimation of dynamic systems, which is an important problem found in spacecraft, aircraft, geophysical, and in many other applications. The most common way of solving state estimation problems consists of using *Kalman filters*, which are stochastic, recursive algorithms providing statistically optimal estimates of the state of a system based on noisy sensor measurements. We present a certifier for Kalman filter programs, which is a program that takes as input a program claiming to implement a Kalman filter together with a specification of that Kalman filter, as well as a certificate under the form of assertions and proof scripts merged within the program via annotations, and tells whether the code correctly implements the state estimation problem specified or not. We tested our certifier on three Kalman filters: simple Kalman filter, information filter and extended Kalman filter. So far we have played both the role of the producer of these programs and the role of the consumer, but we next intend to merge the presented technology with AutoFilter, a state estimation program synthesis system developed by researchers at NASA Ames, which is described below. In fact, our certifier is designed to work on any properly annotated code, but, as explained below, we believe that the full power of domain-specific certification comes from its combination with program synthesis.

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, [19, 17]), in some cases to the point of commercialization (e.g., SciNapse [1]). In program synthesis, there is potential for automatically verifying more interesting properties because additional background information — from the specification and the synthesis knowledge base — is available. The claim made in [15, 14] was that by coupling together program synthesis and property verification, it was possible to automatically certify that a piece of generated code satisfies certain complex properties.

The motivation for this coupling is best illustrated by example. 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 are generally variations on a theme, and yet, currently, there is very little software reuse between projects. Program synthesis offers the potential to reduce development costs through rapid prototyping and a rapid turnaround cycle. However, for these kinds of applications, only 20% of effort is spent in software development, the other 80% being spent on validation of the code[1], including code walkthroughs, formal and informal testing. To reduce the 80% development costs, it is necessary to provide techniques that can avoid code inspections or reduce testing. We believe that property verification can reduce testing time and, moreover, that many properties cannot be verified automatically without the application of program synthesis. Verifying that a state estimator implementation *actually* produces a mathematically optimal estimate cannot be done automatically using the code alone, because the code does not reflect all the information needed, such as the statistical model. In [15, 14] we investigated a possible combination of a previous work of the one presented in this paper with AutoFilter, a state estimation program synthesis engine. AutoFilter takes as input a specification of a state estimation problem and it generates a C/C++ program which implements that specification. We have modified AutoFilter to generate not only the code, but also code annotations certifying the correctness of that code with respect to its specification. The use of additional domain knowledge and information from the synthesis process allows such verification tasks to be not only possible, but also automated. This paper explores in greater detail the certification aspect of the work started in [15, 14], as well as aspects related to the domain-specific proof engineering of the field of Kalman filters and its afferent mathematical knowledge.

Related work. Our domain-specific certification approach requires more sophisticated reasoning

[1]Personal communication from the Jet Propulsion Laboratory (JPL).

than in approaches to date for *proof-carrying code (PCC)* [12]. The abstract domain specification is much richer than memory safety, and verifying the safety of each line of code can require tens of thousands of inference steps. The two specification levels, for the programming language and for the abstract domain, are independently reusable; e.g., once an abstract domain has been formulated it can be used to certify programs written in various programming languages, and conversely, programs can be certified for various domain-specific properties. *Extended Static Checker (ESC)* [5, 16] is a tool that finds programming errors at compile time, such as array index bounds errors, nil dereferences, deadlocks and race conditions. The user of ESC annotates the programs with specifications in a precondition-postcondition style (similar to ours) which are checked statically using a theorem prover for untyped predicate calculus with equality. The type system of the target programming language is implemented in untyped first-order logic. The use of ESC is therefore limited to programming language definable types and to properties that can be proved automatically using ESC's internal theorem prover. By allowing proof scripts as annotations in the programs to certify, we practically extend the usability of our certifiers to whatever properties that can be proved. However, some domain-specific proofs can be very complex, so, even if possible in theory, we do not anticipate that our domain-specific certifier will be used independently from the synthesis engine.

2 Kalman Filters

A Kalman filter is essentially a set of mathematical equations that implements a predictor-corrector type estimator that is *optimal* in the sense that it minimizes the estimated *error* covariance - when some presumed conditions are met. Since the time of its introduction [7], the simple Kalman filter has 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 use of the filter practical, but also to the relative simplicity and robust nature of the filter itself. Rarely do the conditions necessary for optimality exist, and yet the filter apparently works well for many applications in spite of this situation.

2.1 Simple Kalman Filter

The *simple Kalman filter* addresses the general problem of trying to estimate the state $x \in \mathbb{R}^n$ of a discrete-time controlled system that is governed by the linear stochastic difference equation

$$x_{k+1} = \Phi_k x_k + w_k \tag{1}$$

with a measurement $z \in \mathbb{R}^m$ that is

$$z_k = H_k x_k + v_k. \tag{2}$$

In these equations, x_k is the process state vector at time k. In a typical attitude estimation problem, for example, the state vector, x_k, might contain three variables representing the rotation angles of a spacecraft. Equation (1) is the *process model* which describes the dynamics of the state 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 probability distribution:

$$\begin{aligned} p(w_k) &\sim N(0, Q_k) && (3)\\ p(v_k) &\sim N(0, R_k) && (4)\\ E[w_k v_i^\top] &= 0 && (5)\\ E[w_k w_i^\top] &= \begin{cases} Q_k, & \text{if } i = k \\ 0, & \text{if } i \neq k \end{cases} && (6)\\ E[v_k v_i^\top] &= \begin{cases} R_k, & \text{if } i = k \\ 0, & \text{if } i \neq k. \end{cases} && (7) \end{aligned}$$

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.

We 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 $\hat{x}_k \in \mathbb{R}^n$ to be the *a posteriori state estimate* at step k given measurement z_k. We can then define *a priori* and *a posteriori estimate errors* as

$$\begin{aligned} e_k^- &= x_k - \hat{x}_k^-, && (8)\\ e_k &= x_k - \hat{x}_k. && (9) \end{aligned}$$

The *a priori estimate error covariance* is then

$$P_k^- = E[e_k^- {e_k^-}^\top] = E[(x_k - \hat{x}_k^-)(x_k - \hat{x}_k^-)^\top] \qquad (10)$$

and the *a posteriori estimate error covariance* is

$$P_k = E[e_k {e_k}^\top] = E[(x_k - \hat{x}_k)(x_k - \hat{x}_k)^\top]. \qquad (11)$$

We define also the *measurement prediction* as

$$z_k^- = H_k \hat{x}_k^-. \qquad (12)$$

In deriving the equations for the Kalman filter program, we begin with the goal of finding 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:

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - z_k^-). \qquad (13)$$

The difference $(z_k - z_k^-)$ is called the *measurement innovation*, or the *residual.* The residual reflects the discrepancy between the predicted measurement, z_k^-, and the actual measurement, z_k. A residual of zero means that the two are in complete agreement.

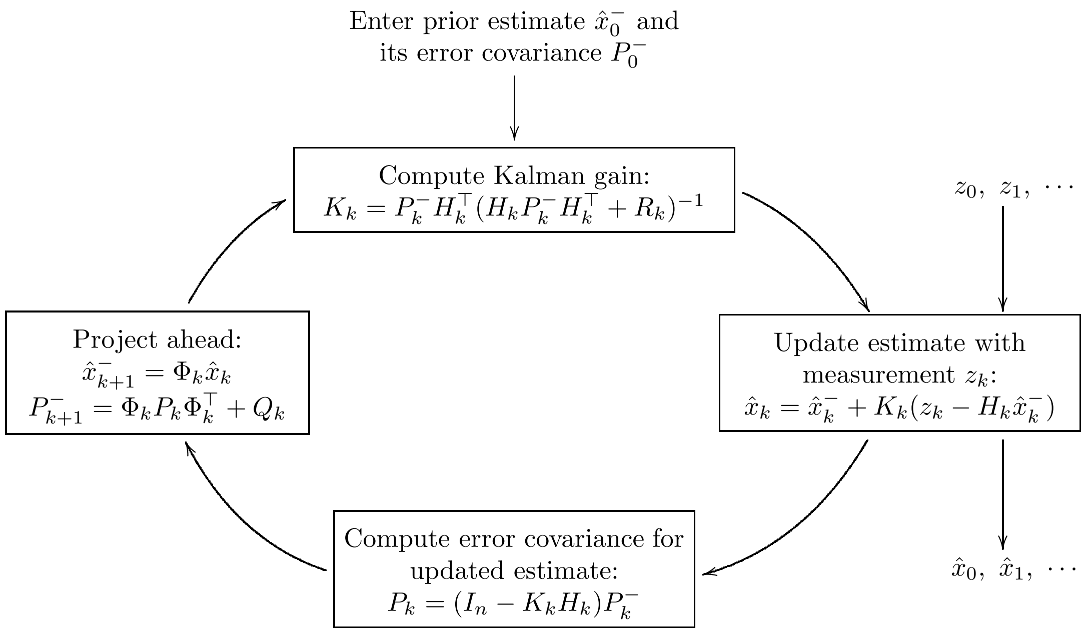
The $n \times m$ matrix K_k in equation (13) is chosen to be the *gain*, or *blending factor*, that minimizes the a posteriori estimate error covariance P_k. One form of the Kalman gain is

$$K_k \;=\; P_k^- H_k^\top (H_k P_k^- H_k^\top + R_k)^{-1}. \tag{14}$$

The covariance matrix associated with the optimal estimate may now be computed. Using equations (11), (13), and (14), we get

$$P_k \;=\; (I_n - K_k H_k) P_k^-. \tag{15}$$

Figure 2 gives a complete picture of the operations of the simple Kalman filter.



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

Figure 2: Simple Kalman filter loop.

AutoFilter takes as input a mathematical specification including equations (1) - (7) and also descriptions of the noise characteristics and filter parameters, and generates code that implements the algorithm in Figure 2. More precisely, AutoFilter generates code in an intermediate language which is then translated into C++ or Matlab. In this paper, we only consider code in the intermediate language. Figure 3 shows a simple Kalman filter code calculating the best estimate incrementally. In this code, `xhatmin(k)` and `xhat(k)` correspond to $\hat{x}_k^-$ and $\hat{x}_k$ (respectively), `Pminus(k)` stands for P_k^-, `zminus(k)` for z_k^-, and `gain(k)` is the Kalman gain K_k.

Despite the apparent simplicity of the code in Figure 3, that AutoFilter generates, 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

```
input xhatmin(0), Pminus(0);
for(k, 0, n) {
  zminus(k) := H(k) * xhatmin(k);
  gain(k) := (Pminus(k) * mtrans(H(k))) * minv(((H(k) * Pminus(k)) * mtrans(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)) * mtrans(Phi(k))) + Q(k);}
```

Figure 3: Simple Kalman filter intermediate code.

assumptions. The optimality proof is a standard proof in state estimation and it is usually presented in books as an informal mathematical proof several pages long (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 the initial estimates, $\hat{x}_0^-$ and P_0^-, are the best prior estimate and its error covariance matrix. Another assumption is that the most probable measurement z_k^- at a given time k is given by equation (12). With the assumption of a prior estimate $\hat{x}_k^-$, we seek to use the measurement z_k to improve the prior estimate. The most important assumption is that the best estimate $\hat{x}_k$ is a linear blending of the residual and the prior estimate (13). 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, 18] 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. We are looking for the particular y that yields an estimate that is optimal using the minimum mean-square error as the performance criterion. If

$$P_k(y) \quad := \quad E[(x_k - \hat{x}_k(y))(x_k - \hat{x}_k(y))^\top]$$

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

$$\frac{d(\mathit{trace}(P_k(y)))}{dy} = 0,$$

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.

Let us detail the optimization process. Firstly, using the Kalman filter equations and noting that $x_k - \hat{x}_k^-$ is uncorrelated with the measurement noise v_k, we get

$$P_k(y) \quad = \quad E[((x_k - \hat{x}_k^-) - y(z_k - z_k^-))((x_k - \hat{x}_k^-) - y(z_k - z_k^-))^\top]$$

$$
\begin{aligned}
&= E[((I_n - yH_k)(x_k - \hat{x}_k^-) - yv_k)((I_n - yH_k)(x_k - \hat{x}_k^-) - yv_k)^\top] \\
&= E[(I_n - yH_k)((x_k - \hat{x}_k^-)(x_k - \hat{x}_k^-)^\top)(I_n - yH_k)^\top - (I_n - yH_k)((x_k - \hat{x}_k^-)v_k^\top)y^\top \\
&\quad - y(v_k(x_k - \hat{x}_k^-)^\top)(I_n - yH_k)^\top + y(v_k v_k^\top)y^\top] \\
&= (I_n - yH_k)P_k^-(I_n - yH_k)^\top - (I_n - yH_k)E[(x_k - \hat{x}_k^-)v_k^\top]y^\top \\
&\quad - yE[v_k(x_k - \hat{x}_k^-)^\top](I_n - yH_k)^\top + yR_k y^\top \\
&= (I_n - yH_k)P_k^-(I_n - yH_k)^\top + yR_k y^\top \\
&= P_k^- - yH_kP_k^- - P_k^-H_k^\top y^\top + y(H_kP_k^-H_k^\top + R_k)y^\top.
\end{aligned}
$$

We proceed now to differentiate the trace of $P_k(y)$ with respect to y. In order to do so, we use two matrix differentiation formulas:

$$
\begin{aligned}
\frac{d(\mathit{trace}(yA))}{dy} &= A^\top \ \text{ if } yA \text{ is a square matrix} \\
\frac{d(\mathit{trace}(yAy^\top))}{dy} &= 2yA \ \text{ if } A \text{ is a symmetric matrix.}
\end{aligned}
$$

Noting that the matrices P_k^- and $H_kP_k^-H_k^\top + R_k$ are symmetric, and that the trace of $yH_kP_k^-$ is equal to the trace of its transpose $P_k^-H_k^\top y^\top$, the result is

$$
\frac{d(\mathit{trace}(P_k(y)))}{dy} = -2(H_kP_k^-)^\top + 2y(H_kP_k^-H_k^\top + R_k).
$$

We now set the derivative equal to zero and solve for the optimal gain. One gets the solution given by equation (14)

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

which is what line 4 in Figure 3 calculates. The above calculations formally involve several thousands of uses of basic properties of matrices and differentiation, as argued in Section 4.

One can now calculate the a posteriori best estimate, $\hat{x}_k(K_k)$, and the covariance matrix, $P_k(K_k)$, associated with the optimal estimate. By routine substitution of the Kalman gain expression, we get

$$
\begin{aligned}
P_k(K_k) &= P_k^- - K_kH_kP_k^- - P_k^-H_k^\top K_k^\top + K_k(H_kP_k^-H_k^\top + R_k)K_k^\top \\
&= P_k^- - K_kH_kP_k^- - P_k^-H_k^\top K_k^\top + P_k^-H_k^\top K_k^\top \\
&= P_k^- - K_kH_kP_k^- \\
&= (I_n - K_kH_k)P_k^-,
\end{aligned}
$$

which is what line 6 in Figure 3 does.

The updated estimate $\hat{x}_k$ is easily projected ahead via the transition matrix

$$
\hat{x}_{k+1}^- = \Phi_k\hat{x}_k. \tag{16}
$$

The fact that $\hat{x}_{k+1}^-$ is the best prior estimate at time $k+1$ follows by another assumption, namely 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.

Finally, we must prove that

$$P_{k+1}^- = \Phi_k P_k \Phi_k^\top + Q_k \tag{17}$$

is the prior estimate error covariance matrix $E[(x_{k+1} - \hat{x}_{k+1}^-)(x_{k+1} - \hat{x}_{k+1}^-)^\top]$ at time $k+1$. This is obtained by replacing x_{k+1} as in equation (1) and $\hat{x}_{k+1}^-$ as in equation (16), and noting that w_k and $x_k - \hat{x}_k$ have zerro-crosscorrelation, because w_k is the process noise for the steap ahead of time k.

$$\begin{aligned}
P_{k+1}^- &= E[(x_{k+1} - \hat{x}_{k+1}^-)(x_{k+1} - \hat{x}_{k+1}^-)^\top] \\
&= E[(\Phi_k(x_k - \hat{x}_k) + w_k)(\Phi_k(x_k - \hat{x}_k) + w_k)^\top] \\
&= E[\Phi_k((x_k - \hat{x}_k)(x_k - \hat{x}_k)^\top)\Phi_k^\top + \Phi_k((x_k - \hat{x}_k)w_k^\top) + (w_k(x_k - \hat{x}_k)^\top)\Phi_k^\top + w_k w_k^\top] \\
&= \Phi_k P_k \Phi_k^\top + \Phi_k E[(x_k - \hat{x}_k)w_k^\top] + E[w_k(x_k - \hat{x}_k)^\top]\Phi_k^\top + Q_k \\
&= \Phi_k P_k \Phi_k^\top + Q_k.
\end{aligned}$$

2.2 Information Filter

The simple Kalman filter equations can be algebraically manipulated into a variety of forms. An alternative form that is especially useful is known as the *information filter* [2], which additionally assumes that the matrices P_k^-, P_k, and R_k admit inverses. One should think of $(P_k^-)^{-1}$ as a measure of the information content of the a priori estimate, that is, before the new measurement information is assimilated into the estimate. Then $P_k = \infty$, that is $(P_k^-)^{-1} = 0$, corresponds to infinite uncertainty, or zero information. This leads to the indeterminate form $\frac{\infty}{\infty}$ in the Kalman gain expression (14), so one can not apply the simple Kalman filter due to rounding errors. The information filter accomodates this situation. It is based on the following equations:

$$P_k^{-1} = (P_k^-)^{-1} + H_k^\top R_k^{-1} H_k \tag{18}$$

$$K_k = P_k H_k^\top R_k^{-1}. \tag{19}$$

Justification of equation (18) is straightforward: one can simply form the product of the right sides of equations (15) and (18) and show that this reduces to the identity matrix.

$$\begin{aligned}
&[(I_n - K_k H_k)P_k^-][(P_k^-)^{-1} + H_k^\top R_k^{-1} H_k] \\
&= [P_k^- - P_k^- H_k^\top (H_k P_k^- H_k^\top + R_k)^{-1} H_k P_k^-][(P_k^-)^{-1} + H_k^\top R_k^{-1} H_k] \\
&= I_n - P_k^- H_k^\top[(H_k P_k^- H_k^\top + R_k)^{-1} + (H_k P_k^- H_k^\top + R_k)^{-1} H_k P_k^- H_k^\top R_k^{-1} - R_k^{-1}]H_k \\
&= I_n - P_k^- H_k^\top[(H_k P_k^- H_k^\top + R_k)^{-1}(I_m + H_k P_k^- H_k^\top R_k^{-1}) - R_k^{-1}]H_k \\
&= I_n - P_k^- H_k^\top[(H_k P_k^- H_k^\top + R_k)^{-1}(R_k + H_k P_k^- H_k^\top)R_k^{-1} - R_k^{-1}]H_k \\
&= I_n - P_k^- H_k^\top[R_k^{-1} - R_k^{-1}]H_k \\
&= I_n,
\end{aligned}$$

$$\begin{aligned}
&[(P_k^-)^{-1} + H_k^\top R_k^{-1} H_k][(I_n - K_k H_k)P_k^-] \\
&= [(P_k^-)^{-1} + H_k^\top R_k^{-1} H_k][P_k^- - P_k^- H_k^\top (H_k P_k^- H_k^\top + R_k)^{-1} H_k P_k^-] \\
&= I_n - H_k^\top[(H_k P_k^- H_k^\top + R_k)^{-1} - R_k^{-1} + R_k^{-1} H_k P_k^- H_k^\top (H_k P_k^- H_k^\top + R_k)^{-1})]H_k P_k^- \\
&= I_n - H_k^\top[(I_m + R_k^{-1} H_k P_k^- H_k^\top)(H_k P_k^- H_k^\top + R_k)^{-1} - R_k^{-1}]H_k P_k^- \\
&= I_n - H_k^\top[R_k^{-1}(R_k + H_k P_k^- H_k^\top)(H_k P_k^- H_k^\top + R_k)^{-1} - R_k^{-1}]H_k P_k^- \\
&= I_n - H_k^\top[R_k^{-1} - R_k^{-1}]H_k P_k^- \\
&= I_n.
\end{aligned}$$

Equation (19) gives an alternative expression for the Kalman gain and it can be derived from equation (14) using (18).

$$\begin{aligned}
K_k &= P_k^- H_k^\top (H_k P_k^- H_k^\top + R_k)^{-1} \\
&= P_k P_k^{-1} P_k^- H_k^\top [(H_k P_k^- H_k^\top R_k^{-1} + I_m) R_k]^{-1} \\
&= P_k P_k^{-1} P_k^- H_k^\top R_k^{-1} (H_k P_k^- H_k^\top R_k^{-1} + I_m)^{-1} \\
&= P_k [(P_k^-)^{-1} + H_k^\top R_k^{-1} H_k] P_k^- H_k^\top R_k^{-1} (H_k P_k^- H_k^\top R_k^{-1} + I_m)^{-1} \\
&= P_k (H_k^\top R_k^{-1} + H_k^\top R_k^{-1} H_k P_k^- H_k^\top R_k^{-1})(H_k P_k^- H_k^\top R_k^{-1} + I_m)^{-1} \\
&= P_k H_k^\top R_k^{-1} (I_m + H_k P_k^- H_k^\top R_k^{-1})(H_k P_k^- H_k^\top R_k^{-1} + I_m)^{-1} \\
&= P_k H_k^\top R_k^{-1}.
\end{aligned}$$

Notice that the updated error covariance, P_k, can be computed without first finding the gain, and that the expression for gain is now involving P_k. Consequently, K_k must be computed *after* P_k, so the order in which the P_k and K_k computations appear in this alternative algorithm is reversed. The matrix P_k^{-1} is often referred to as the *information matrix* and this gives the name of the algorithm. Figure 4 shows the information filter loop and intermediate code.

2.3 Extended Kalman Filter

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

$$\begin{aligned}
x_{k+1} &= f(x_k, u_k) + w_k && (20) \\
z_k &= h(x_k) + v_k, && (21)
\end{aligned}$$

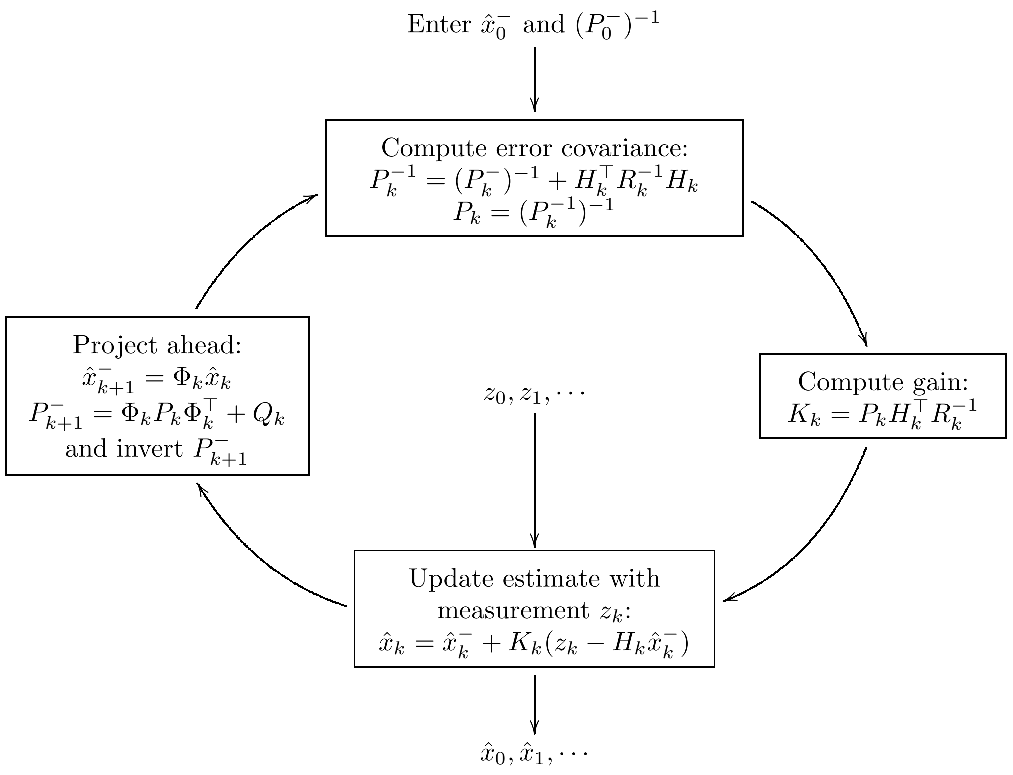
where f and h are non-linear functions, u_k is a deterministic forcing function (regard it as an input), and the random vectors w_k and v_k again represent the process and the measurement noise and satisfy the same conditions as for the simple Kalman filter.

To simplify computations and to make the problem implementable, one typical technique is to 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*). We will use the same notations for a priori and a posteriori state estimates, and generally the same conceptual assumptions as for simple Kalman filter; in particular, 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 follows:

$$\begin{aligned}
\hat{x}_{k+1}^- &= f(\hat{x}_k, u_k), && (22) \\
z_k^- &= h(\hat{x}_k^-). && (23)
\end{aligned}$$

Let Φ_k and H_k be the Jacobian matrices of partial derivatives of f and h with respect to x:

$$\begin{aligned}
\Phi_k &= (\frac{\delta f_i}{\delta x_j}(\hat{x}_k, u_k))_{i,j} && (24) \\
H_k &= (\frac{\delta h_i}{\delta x_j}(\hat{x}_k^-))_{i,j} && (25)
\end{aligned}$$


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

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

Figure 4: Information filter loop and intermediate code.

We approximate the non-linear functions f and h with Taylor series expansions retaining only first order terms:

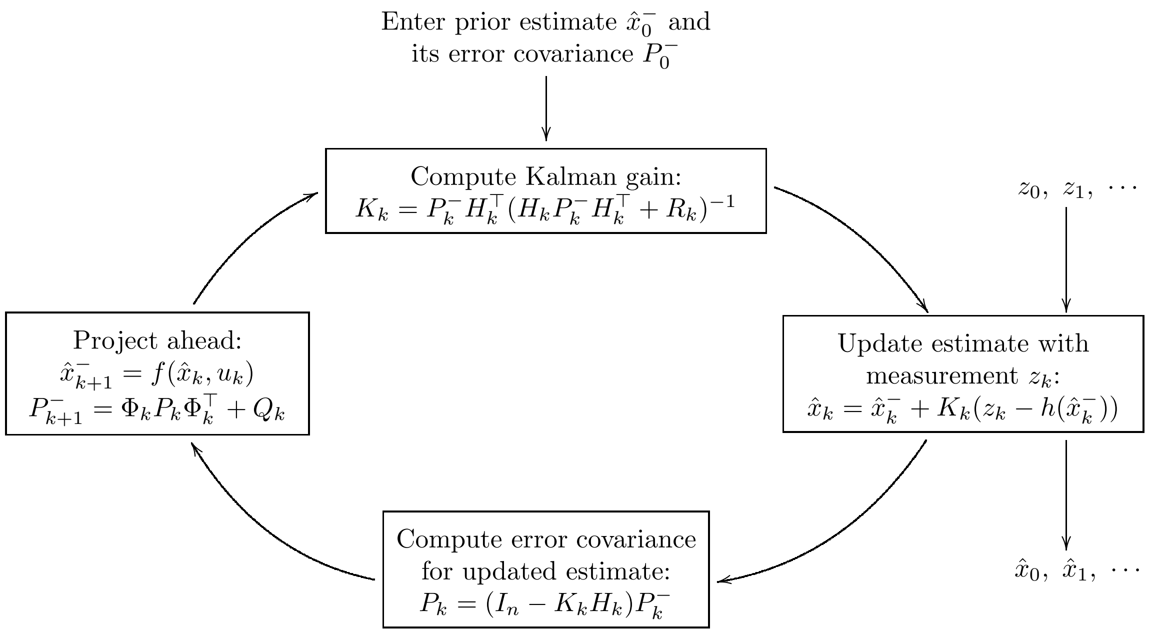
$$f(x_k, u_k) = f(\hat{x}_k, u_k) + \Phi_k(x_k - \hat{x}_k), \tag{26}$$

$$h(x_k) = h(\hat{x}_k^-) + H_k(x_k - \hat{x}_k^-). \tag{27}$$

Then we get the new governing equations that linearize an estimate about equations (22) and (23)

$$x_{k+1} = \hat{x}_{k+1}^- + \Phi_k(x_k - \hat{x}_k) + w_k, \tag{28}$$
$$z_k = z_k^- + H_k(x_k - \hat{x}_k^-) + v_k. \tag{29}$$

The basic operation of the EKF is very similar to that of the simple Kalman filter, as shown in Figure 5.


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: $\hat{x}_k = \hat{x}_k^- + K_k(z_k - h(\hat{x}_k^-))$
$\hat{x}_0,\ \hat{x}_1,\ \cdots$
Compute error covariance for updated estimate: $P_k = (I_n - K_k H_k) P_k^-$
Project ahead: $\hat{x}_{k+1}^- = f(\hat{x}_k, u_k)$ $P_{k+1}^- = \Phi_k P_k \Phi_k^\top + Q_k$

```
 1.  input xhatmin(0), Pminus(0);
 2.  for(k,0,n) {
 3.    zminus(k) := h(xhatmin(K));
 4.    H(K) := Jacob_h(xhatmin(K));
 5.    gain(k) := (Pminus(k) * mtrans(H(k))) * minv(((H(k) * Pminus(k)) * mtrans(H(k))) + R(k));
 6.    xhat(k) := xhatmin(k) + (gain(k) * (z(k) - zminus(k)));
 7.    P(k) := (id(n) - (gain(k) * H(k))) * Pminus(k);
 8.    xhatmin(k + 1) := f(xhat(k), u(k));
 9.    Phi(K) := Jacob_f(xhat(K), u(K));
10.    Pminus(k + 1)  := ((Phi(k) * P(k)) * mtrans(Phi(k))) + Q(k);}
```

Figure 5: Extended Kalman filter loop and intermediate code.

The optimality proof is also similar to the one for the simple Kalman filter, but there are some differences. Firstly, in order to calculate $P_k(y)$, we have to use equation (27):

$$\begin{aligned}
P_k(y) &= E[((x_k - \hat{x}_k^-) - y(z_k - z_k^-))((x_k - \hat{x}_k^-) - y(z_k - z_k^-))^\top] \\
&= E[((x_k - \hat{x}_k^-) - y(h(x_k) + v_k - h(\hat{x}_k^-)))((x_k - \hat{x}_k^-) - y(h(x_k) + w_k - h(\hat{x}_k^-)))^\top \\
&= E[((x_k - \hat{x}_k^-) - y(H_k(x_k - \hat{x}_k^-) + v_k))((x_k - \hat{x}_k^-) - y(H_k(x_k - \hat{x}_k^-) + v_k))^\top] \\
&\ \vdots \quad \text{as in the case of the simple Kalman filter} \\
&= (I_n - yH_k)P_k^-(I_n - yH_k)^\top + yR_k y^\top \\
&= P_k^- - yH_kP_k^- - P_k^- H_k^\top y^\top + y(H_k P_k^- H_k^\top + R_k)y^\top .
\end{aligned}$$

The optimization process proceeds now exactly as for the simple Kalman filter; we get the same Kalman gain, given by equation (14), and we can calculate $\hat{x}_k(K_k)$ and $P_k(K_k)$, as lines 6 and 7 of the code from Figure 5 do.

The updated estimate is projected ahead using equation (23). Thus, as for the simple Kalman filter, the best prior estimate at the next step follows the state equation (20) using the best estimate at the current state, but where the contribution of the process noise w_k is ignored. The proof of the fact that the a priori estimate error covariance matrix at time $k+1$ is

$$P^-_{k+1} \quad = \quad \Phi_k P_k \Phi_k^\top + Q_k$$

follows the one for the simple Kalman filter, using equations (22) and (26) as extra-assumptions:

$$\begin{aligned}
P^-_{k+1} &= E[(x_{k+1} - \hat{x}^-_{k+1})(x_{k+1} - \hat{x}^-_{k+1})^\top] \\
&= E[(f(x_k, u_k) + w_k - f(\hat{x}_k, u_k))(f(x_k, u_k) + w_k - f(\hat{x}_k, u_k))^\top] \\
&= E[(\Phi_k(x_k - \hat{x}_k) + w_k)(\Phi_k(x_k - \hat{x}_k) + w_k)^\top] \\
&\vdots \quad \text{as in the case of the simple Kalman filter} \\
&= \Phi_k P_k \Phi_k^\top + Q_k.
\end{aligned}$$

3 A Framework for Formalizing the Proof

To generate and automatically certify the optimality proofs for the three Kalman filters, we need to formalize the domain knowledge, which includes matrices, random matrices, functions on matrices, and matrix differentiation. We first discuss the formal language that we chose for this purpose together with its underlying logic. The reader can see in Appendix A our abstract domain formalization.

3.1 Maude and Membership Equational Logic

Maude [4] is a freely distributed high-performance executable specification system in the OBJ [6] family, supporting both rewriting logic [9] and membership equational logic [10]. 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.

Membership equational logic (MEL) [10] is a variant of equational logic which, in addition to atomic equalities $t = t'$, allows atomic *memberships* $t : s$ stating that the term t has the sort s. In Maude, conditional equations and memberships are declared with the keywords `ceq` and `cmb`, respectively, while the unconditional ones with `eq` and `mb`. For example, the conditional membership

```
cmb X / Y : Real if Y =/= 0 .
```

states that for any reals `X` and `Y`, `X/Y` is a real, or has the sort `Real`, if `Y` is non-zero. Sorts are grouped in *kinds* and operations are defined only on kinds, but Maude provides convenient syntactic sugar conventions. For example, a subsort declaration

```
subsort Nat < Real .
```

is syntactic sugar for the membership

```
cmb X : Real if X : Nat .
```

The operation declaration

```
op _+_ : Real Real -> Real .
```

is syntactic sugar for[2]

```
cmb X + Y : Real if X : Real / Y : Real .
```

`[Real]` is a shorthand for the kind containing the sort `Real`; the operation declaration

```
op _/_ : Real Real -> [Real] .
```

says that a quotient term might not have a sort, i.e., it might be *undefined* in a partial algebraic terminology. There is an automatic translation from partial equational logic, or more precisely from its more general variant called *partial membership equational logic (PMEL)*, into MEL explored in detail in [10, 11]. Maude's *implicit* support for partiality was a major factor in choosing Maude as a logic and implementation engine for our certification tools, because all the specifications of abstract domains that we encountered so far involve partial operators. However, we warn the reader that Maude does *not* explicitly support PMEL, i.e., it is a *total* MEL engine[3]. That means that, for example if one declares the equation

```
eq X / (Y / Z) = (X * Z) / Y .
```

then one should not expect Maude to implicitly prove that `Y / Z` is defined before applying the equation; it is the user's responsibility to test this, i.e., to use an equation of the form

```
ceq X / (Y / Z) = (X * Z) / Y if Y / Z : Real .
```

Since we use partial specifications often and since it is so easy to omit such membership checks, we have developed an automatic criterion that checks whether a MEL theory is *duplex*, i.e., if it can be safely regarded as a specification in PMEL [11].

Our axiom abstract domain presented next, having around 100 axioms, passed the duplex criterion, so we know that Maude's total reasoning is sound for our domain.

3.2 Matrices and Random Matrices

Equational reasoning with (random) matrices is extensively used in all state estimation optimality proofs that we are aware of. In fact, most of the proof steps in our scripts and all our lemmas are equational. We used two sorts `Matrix, MatrixExp` for representing matrices and, similarly, two sorts `RandomMatrix, RandomMatrixExp` for random matrices. The reason why we had to make this distinction between `(Random)Matrix` and `(Random)MatrixExp` will become clear in the next subsection, on functions on matrices. The subsort relations between these four sorts are:

```
subsort Matrix < MatrixExp .
subsort Matrix < RandomMatrix .
subsort MatrixExp < RandomMatrixExp .
subsort RandomMatrix < RandomMatrixExp .
```

[2]Together with an appropriate operation declaration on kinds.

[3]We are not aware of any tool providing explicit executional support for partial equational logics.

Most of the operations and axioms/lemmas in matrix theory are *partial*. For example, addition is defined iff the two matrices have the same dimensions, multiplication is defined iff the number of columns of the first matrix equals the number of rows of the second, the inverse of a matrix is defined only if the matrix is square and its determinant is non-zero, and the commutativity and associativity of addition hold true iff the matrices involved have the same dimensions, etc.... It was a big benefit, if not the biggest, that Maude provided support for partiality, thus allowing us to compactly specify matrix theory and do partial proofs. The partial operations of addition and multiplication are defined as follows:

```
op _+_ : MatrixExp MatrixExp -> [MatrixExp] .
op _*_ : MatrixExp MatrixExp -> [MatrixExp] .
```

One example of a total operation is the transpose of a matrix:

```
op mtrans : MatrixExp -> MatrixExp .
```

In order to define their semantics and properties, we need two total operations that give the numbers of columns and rows of a matrix

```
ops c r : MatrixExp -> MachineInt .
```

Now we can express definedness of addition and multiplication together with appropriate conditional equations computing the new columns and rows:

```
vars P Q R : MatrixExp .

cmb (P + Q) : MatrixExp if (r(P) == r(Q) and c(P) == c(Q)) .
ceq c(P + Q) = c(P) if (P + Q : MatrixExp) .
ceq r(P + Q) = r(P) if (P + Q : MatrixExp) .

cmb (P * Q) : MatrixExp if c(P) == r(Q) .
ceq c(P * Q) = c(Q) if (P * Q : MatrixExp) .
ceq r(P * Q) = r(P) if (P * Q : MatrixExp) .
```

Axioms relating various operators on matrices are also needed, such as:

```
ceq mtrans(P * Q) = mtrans(Q) * mtrans(P) if (P * Q : MatrixExp) .
ceq (P * (Q + R)) = ((P * Q) + (P * R)) if (Q + R : MatrixExp and
                                            P * Q : MatrixExp) .
ceq (mtrans(minv(P))) = (minv(mtrans(P))) if (minv(P) : MatrixExp) .
```

together with more than 60 others, most of them conditional and involving memberships. We have also defined the *expectation* of a random matrix as an operation

```
op E|_| : RandomMatrixExp -> MatrixExp .

vars RP RQ : RandomMatrixExp .
eq (r(E| RP |)) = (r(RP)) .
eq (c(E| RP |)) = (c(RP)) .
```

There are 7 axioms specifying properties of this operation, some of the most used being

```
ceq (E| P * RQ |) = (P  * E| RQ |) if (P * RQ  : RandomMatrixExp) .
ceq (E| RP * Q |) = ( E| RP |  * Q) if (RP * Q  : RandomMatrixExp) .
```

3.3 Functions on Matrices

One step in the optimality proofs is stating that the best estimate of the actual state is a linear combination of the best prior estimate and the measurement innovation (13). 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

$$\lambda y.(\langle \textit{prior estimate} \rangle + y * \langle \textit{measurement innovation} \rangle).$$

In order for this function to be well defined, y must be a matrix having appropriate dimensions. Hence, we need to formally define functions on matrices together with their properties. We do it by declaring new sorts, `MatrixVar` and `MatrixFun`, the first being a subsort of `Matrix`, together with operations for defining functions and for applying them, respectively:

```
op /\_._ : MatrixVar MatrixExp -> MatrixFun .
op __ : MatrixFun MatrixExp -> [MatrixExp] .

var X : MatrixVar .
vars P R : MatrixExp .
cmb ((/\ X . P)(R)) : MatrixExp if (X + R : MatrixExp) .
```

We also need an operation

```
op _in_ : MatrixVar MatrixExp -> Bool .
```

such that `X in P` is `true` iff X appears in the expression of P, and `false` otherwise. The infix operation `in` is defined inductively:

```
var Y : Matrix .
eq (X in X) = (true) .
ceq (X in Y) = (false) if (X =/= Y).
ceq (X in (P * Q)) = ((X in P) or (X in Q)) if (P * Q : MatrixExp) .
ceq (X in (P + Q)) = ((X in P) or (X in Q)) if (P + Q : MatrixExp) .
...
```

Hence, we can see that, in order to define it, we need to make the distinction between `Matrix` and `MatrixExp`. That's why we had to use two sorts for representing matrices. Using the operation `in` appropriate (conditional) axioms for functions are specified, also taking into account partiality, such as:

```
ceq ((/\ X . X)(R)) = (R) if (X + R : MatrixExp) .
ceq ((/\ X . P)(R)) = (P) if (X + R : MatrixExp and not(X in P)) .
ceq ((/\ X .(P + Q))(R)) = (((/\ X . P)(R)) + ((/\ X . Q)(R)))
    if (X + R : MatrixExp and P + Q : MatrixExp) .
```

among many others.

3.4 Matrix Differentiation

As we saw in Subsection 2.1, in order to prove that $\hat{x}_k$ is the best estimate, under simplifying assumptions, of the state vector x_k at time k, we use a differential calculus approach. Axiomatization of functions on matrices with their derivatives 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 when possible. For example, the only property that we used so far linking optimality of estimates to differentiation is that a matrix K minimizes a function $\lambda y.P$ iff $(d(trace(\lambda y.P))/dy)(K) = 0$. For that reason, in order to avoid going into deep axiomatizability of mathematics, we have just defined a “derived” operation

```
op d|trace_|/d_ : MatrixFun MatrixVar -> MatrixFun .
```

which gives directly the derivative of the trace of a function on matrices, and declared some basic properties of it, such as

```
ceq ((d|trace(/\ X . X)|/d(X))(R)) = (id(r(X))) if (X + R : MatrixExp) .
ceq ((d|trace(/\ X . P)|/d(X))(R)) = (zero(r(X),c(X)))
    if (X + R : MatrixExp and not(X in P)) .
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) .
```

We also used two matrix differentiation formulas given by the following conditional equations

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

One could, of course, prove these properties from more basic properties of traces, functions and differentiations, but one would need to add a significant body of mathematical knowledge to the system. We will eventually do it when our certification system becomes more stable, but for now we prefer to just give these provable properties as axioms of the abstract domain.

4 Formal Optimality Proofs

In this section we explain how we formalized the informal optimality proofs for the three Kalman filters, using the axiomatization of the abstract domain in Section 3. This formalization was done using version 2.3.7 of ITP, an interactive theorem prover implemented in Maude and presented below.

4.1 The ITP Tool

The ITP tool [3] is an experimental interactive inductive theorem prover for proving properties of Maude equational specifications, i.e., specifications in membership equational logic with an initial algebra semantics. The ITP tool has been written entirely in Maude and is in fact an *executable* *specification* in rewriting logic of the formal inference system that it implements. This inference system includes a "structural induction" principle, that allows proving that a property holds in the initial model of an equational specification by proving that it holds for each constructor whenever it holds for the arguments of that constructor.

The input of ITP is a pair `specification |- sentence`, called a *goal*. Thus, the sentence to be proved is submitted to ITP as the initial goal, and then this goal is succesively transformed by controlled rewriting - using the ITP inference rules as rewrite rules - into different sets of subgoals, until no subgoals are left if the proof succeeds. If the proof succeeds, then the user gets the message `q.e.d.`. All (sub-)goals in a proof are labelled with a list of natural numbers separated by dots, the initial goal being automatically labelled with 1.

The interface of ITP accepts Maude functional modules with labelled equations. There are more label categories, providing information about the status of an equation, but the most used are `ax>` and `ax*>`.

```
[ax*> | distr*+]
  ceq (P * (Q + R)) = ((P * Q) + (P * R))
     if (Q + R : MatrixExp and P * Q : MatrixExp) .
[ax> | zero-*-left]
  ceq (zero(N,M) * P) = (zero(N,c(P))) if (r(P) == M) .
```

The equation should be applied automatically if it is without * (this is the case with the second equation, labelled `zero-*-left`) and only on demand if it is with * (like the first equation, labelled `distr*+`). The ITP tool has many commands, but the most important are `set, set*, rwr` and `apply`. To set the *-status of an equation or of a set of equations we use `set` and `set*`:

```
(set distr*+ in (1) .)
(set* zero-*-left in (1) .)
```

To prove the sentence by automatically applying the equations that are set (that is, the equations having the label category `ax>`), we use the `rwr` command.

```
(rwr (1) .)
```

If the sentence can not be automatically proved using `set` and `rwr` - which, in general, does not imply its falsehood - the tool will return some simplified form of the sentence to be proved and print the message "the strategy can not be applied." In these situations we must provide *hints* by using the `apply` command. For example,

```
(apply distr*+ to (1.2) at (2.2.3) .)
```

says that the distributivity axiom should be applied to subgoal number `1.2` at position `2.2.3`. Simplifications by rewriting are automatically done after each hint. There are rigorous conventions in ITP for labelling the equations and the (sub-)goals, and also in accessing positions in terms, but these are not important for this presentation and may change in the near future as ITP improves, so we omit them.

4.2 Specifying the Statistical Models

In order to reason about the codes for the three Kalman filters, one needs the *specifications* of the Kalman filters together with all their *assumptions*. These are needed in *addition* to the abstract domain knowledge in Section 3. Therefore, the very first step is to expand the abstract domain with three specifications, which we denote

$$\textsc{Spec}_{\text{KF}},\ \textsc{Spec}_{\text{IF}},\ \textsc{Spec}_{\text{EKF}}.$$

The specifications contain operations declaring all the matrices and vectors involved together with their dimensions, such as, for all three specifications,

```
ops x z v w : MachineInt -> RandomMatrix .
ops R Q Phi H : MachineInt -> Matrix .
ops n m p : -> MachineInt .

var K : MachineInt .
eq (r(x(K))) = (n) .      eq (c(x(K))) = (1) .
eq (r(w(K))) = (n) .      eq (c(w(K))) = (1) .
eq (r(Phi(K))) = (n) .    eq (c(Phi(K))) = (n) .
eq (r(z(K))) = (m) .      eq (c(z(K))) = (1) .
eq (r(v(K))) = (m) .      eq (c(v(K))) = (1) .
eq (r(H(K))) = (m) .      eq (c(H(K))) = (n) .
...
```

and, only in the case of the extended Kalman filter,

```
op u : MachineInt -> Matrix .
op f : RandomMatrix  RandomMatrix ->  [RandomMatrix] .
op h : RandomMatrix ->  [RandomMatrix] .

eq (r(u(K))) = (p) .       eq (c(u(K))) = (1) .

vars V1 V2 :  RandomMatrix .
cmb (f(V1, V2)) :  RandomMatrix
   if (r(V1) == n and r(V2) == p and c(V1) == 1 and c(V2) == 1)   .
cmb (h(V1)) :  RandomMatrix if (r(V1) == n and  c(V1) ==  1) .
ceq (r(f(V1, V2))) = (n) if (f(V1, V2) :  RandomMatrix) .
ceq (c(f(V1, V2))) = (1) if (f(V1, V2) : RandomMatrix) .
ceq (r(h(V1))) = (m) if (h(V1) :  RandomMatrix) .
ceq (c(h(V1))) = (1) if (h(V1) :  RandomMatrix) .
```

Other axioms specify model equations/assumptions, 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) * mtrans(v(K)) |) .
[ax*> | QK]
  eq (Q(K)) = (E| w(K) * mtrans(w(K)) |) .
```

in $\textsc{Spec}_{\mathrm{KF}}$ and $\textsc{Spec}_{\mathrm{IF}}$, and

```
[ax*> | EKF-next]
  eq (x(K + 1)) = (f(x(K), u(K)) + w(K)) .
[ax*> | EKF-measurement]
  eq (z(K)) = (h(x(K))  + v(K)) .
```

in $\textsc{Spec}_{\mathrm{EKF}}$. 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 was that these and other assumptions not mentioned here are so well (and easily) accepted by experts that they don't even make their use explicit in their informal proofs; this was of course unavoidable in the context of formal proving, so we had to declare them explicitly as axioms.

The specifications $\textsc{Spec}_{\mathrm{KF}}$ and $\textsc{Spec}_{\mathrm{IF}}$ for the simple Kalman and information filters have about 45-50 axioms, while $\textsc{Spec}_{\mathrm{EKF}}$ has about 60 axioms.

4.3 Domain-specific Lemmas

We want the formal optimality proofs for the three Kalman filters to be as automatic as possible; that is, we want to minimize the number of hints (`apply` commands). We obtained this by using more domain-specific (matrix theory, in our case) lemmas like, for example

```
[lem*> | lemma-2]
 ceq (((RP * M1) + M2) - (RP * M3)) = ((RP * (M1 - M3))+ M2)
     if (r(RP) == r(M2) and c(RP) == r(M1) and c(RP) == r(M3) and c(M1) == c(M2)
         and c(M1) == c(M3)) .
```

This lemma is used 6 times in the three optimality proofs. We used 34 lemmas, which were added to the abstract domain. In Appendix B we present the lemmas and a table specifying the number of times each lemma was used for each of the three optimality proofs. As we shall see in Subsection 4.4, the use of these lemmas greatly improved the proof scripts, reducing very much the number of hints. Since these lemmas are formulated and proved independent of the Kalman filters, they can be stored in a database together with their proofs; that's what we intend to do in the future.

4.4 The Annotated Program

In order to machine check the proof of optimality, the proof must be decomposed and linked to the actual code. This is done by adding the specification of the statistical model at the beginning of the code and 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. Proofs are also added as annotations where needed. Notice that by "proof" we here mean a series of hints that ITP uses to guide the proof. In this way we obtain three annotated programs, one for each Kalman filter. The reader can see them in Appendices C, D, and E.

A sketch of the annotated program for the simple Kalman filter is shown in Figure 6, where we replaced the more formal (and longer) assertions by English (see Appendix C for the full annotated program). The proof assertions in Figure 6 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

```
  /* Specification of the state estimation problem
        ... about 45 axioms/assumptions in Maude */
1.  input xhatmin(0), Pminus(0);
  /* Proof assertion 1: ... */
  /* Assertion 1: (in English) xhatmin(0) and Pminus(0) are the
        best prior estimate and its error covariance matrix */
2.  for(k,0,n) {
  /* Assertion 2: (in English) xhatmin(k) and Pminus(k) are the
        best prior estimate and ts error covariance matrix */
3.    zminus(k) := H(k) * xhatmin(k);
4.    gain(k) := (Pminus(k) * mtrans(H(k))) *
                 minv(((H(k) * Pminus(k)) * mtrans(H(k))) + R(k));
  /* Proof assertion 3: ... */
  /* Assertion 3: (in English) 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) error covariance matrix of xhat(k) */
7.    xhatmin(k + 1) := Phi(k) * xhat(k);
8.    P(k + 1)  := ((Phi(k) * P(k)) * mtrans(Phi(k))) + Q(k);
  /* Proof assertion 2: ... */
    }
```

Figure 6: Annotated program for the simple Kalman filter.

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 $\text{S}\textsc{pec}_{\text{KF}}$ and so can be immediately checked. Between 2 and 3 we assert that `xhatmin(k)` and `Pminus(k)` are the best prior estimate and error covariance matrix, respectively. This is obvious for the first iteration of the loop, but needs to be proved for the other iterations. Therefore, we do an implicit proof by induction. The assertion after line 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 in Figure 7 its proof script.
Hence, we first set 44 axioms and lemmas to be applied and after that we simplify automatically using `rwr` command. Note that the proof of this assertion is done completely automatically. That is the benefit of using domain-specific lemmas; we can see that in this proof we use 11 domain-specific lemmas. The assertion between lines 5 and 6 says that `xhat(k)` is the best estimate of the actual state and its proof is the only one that uses a hint. After line 6, we have the assertion that `P(k)` is the error covariance matrix of the best estimate; its proof uses 12 domain-specific lemmas, 8 of them being used also in the proof of assertion after line 4. After line 8, we complete the proof by induction of assertion between lines 2 and 3. Due to an assumption in $\text{S}\textsc{pec}_{\text{KF}}$, we can easily show that `xhatmin(k+1)` is the best prior estimate at time `k+1`. The fact that `Pminus(k+1)` is its error covariance matrix is also proved automatically, with the help of 2 domain-specific lemmas.

Using domain-specific lemmas, we minimized the number of hints. In a previous proof [15, 14] for the simple Kalman filter there were used more than 500 `apply` commands, now we use only one.

```
[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*mtrans minus-def1 misc-is E- E+
      E*left E*right mtrans-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)) * mtrans(x(K) - Estimate(K, y)) |)))
     = (true) .
------------------------------------------------------------------ */
```

Figure 7: Proof script of assertion 3: `gain(K)` minimizes error covariance matrix.

It took ITP 35 seconds to check all this proof. Taking into account the speed of Maude (3 million rewrites per second) and pessimistically assuming that its ITP slows it down 1000 times, then we predict that there are at least 105.000 uses of the axioms and lemmas in the abstract domain and SPEC$_{KF}$ in this proof.

As we can see in Appendix D, the proof scripts from the annotated program for the extended Kalman filter have many similarities with the ones for the simple Kalman filter. We need only one hint, but we use more domain-specific lemmas. In the case of the information filter (see Appendix E), because of the increased number of algebraic properties used by this algorithm, we need 3 `apply` commands, and we make use of 32 lemmas.

5 Certifying Annotated Kalman Filters

There are various types and levels of certification, including testing and human code review. In this paper we address certifying conformance of programs to domain-specific properties. The general problem is known to be intractable, but by using program synthesis to annotate code with assertions and proof scripts, complex properties can be certified automatically.

Our current certifier uses a combination of theorem proving and proof checking. It takes as input a PMEL specification of the abstract domain and an annotated program and returns "yes" or "no". It extracts proof tasks together with their proof scripts from the annotated program, and then calls Maude's ITP tool to validate them. The proof tasks are generated from both annotations and code, while the proof scripts are extracted from annotations. It answers "yes" if and only if all the proof scripts are valid for the proof tasks that it generates. Therefore, like in proof carrying code, the code, the assertions and the proofs are interdependent, so one cannot maliciously modify either of them.

The state estimation certifier is very restrictive at this stage, but this is acceptable because we only use it on programs synthesized with AutoFilter. It only accepts programs written in a generic matrix-assignment based programming language like the ones in Figures 3, 4, 5 ; additionally, those programs are not allowed to redefine variables (except the loop counter) and must consist of exactly one loop which iteratively calculates the best estimate. These programs must be annotated like in Figure 6, i.e., they must start with an annotation containing the specification of the Kalman filter for which the subsequent code is claimed, and then contain lines of code, proof scripts and assertions. All the annotations use directly Maude and/or ITP notation. Each assertion that cannot be proved by straightforward rewriting should come with its proof script annotation. This simple approach works because our synthesized state estimation programs are not concurrent.

The certifier works as follows. It first extends the abstract domain specification with the specification of the statistical model (extracted from the beginning of the annotated program). Then it follows the steps of a proof by mathematical induction on `k`, the loop index. More precisely, it first proof checks the first assertion in the code in which it replaces `k` by 0. Then it incrementally visits each line of code in the loop, adding the assignments to the specification as ordinary MEL equations and proof checking the assertions. In order to proof check an assertion, it calls the ITP tool with the current specification, the assertion, and the proof script provided in the code as annotation. At the end of the loop, it also proof checks the first assertion in the loop in which `k` is replaced by `k+1`.

Therefore, our current certifier simulates the execution of the code modifying its environment (specification) and checking a provided proof whenever an assertion is found. Assuming that the abstract domain and the Kalman filters are correctly specified, then for any annotated program as above our certifier returns "yes" if and only if the program calculates the best estimate at each iteration. Notice that the certifier was specifically designed to be totally independent from the synthesis engine. The proofs and the annotations are orders of magnitude larger than the real code, but fortunately, they can be automatically generated by the synthesis engine once generic proofs are provided with the program schemas.

6 Conclusions and Future Work

In this paper, we built a certifier for three Kalman filters: simple Kalman filter, information filter and extended Kalman filter, extending previous work done by Roşu and Whittle [15, 14] for the simple Kalman filter. The language we chose for this purpose was Maude, a high-performance executable specification system in the OBJ family. For the optimality proofs for the three Kalman filters we used ITP, an inductive theorem prover implemented in Maude.

We used many domain-specific (matrix theory) lemmas, which were formulated and proved independent of the Kalman filters, so they can be stored in a database together with their proofs. With the help of these lemmas, we automatized the proofs, minimizing the number of hints. Thus, we had only 5 hints in the optimality proofs for all three Kalman filters, while in [15, 14] there were used more than 400 hints only for the simple Kalman filter.

The certifier used in our paper is itself a substantial program, including the Maude system and the ITP tool. Maude and ITP are used both to generate the proofs and to check them. For practical purposes, a certifier must be as simple as possible. Certification authorities will only trust a certification tool if it has been formally verified, and hence it must be small. This means that whilst Maude and ITP are good generic engines for developing domain-specific proofs scripts of individual schemas, the final product will most likely incorporate a kernel certifier with a minimal knowledge base and minimal proving technology.

Our long term goal is to develop an automated state estimation certifier which:

- is simple, so that it can be easily validated by ordinary code reviewers;
- is general, so it works on a large variety of programs;
- reduces the amount of domain-specific knowledge to be trusted to a few easily readable properties, so that it can be validated by domain experts;
- is independent from the domain-specific synthesis system, so that the likelihood that the two systems have common abstract domain errors is minimized and therefore can be safely used together.

There are sensible trade-offs between these desired features. For example, if the certifier uses a specialized theorem prover then the synthesis engine can generate fewer and simpler annotations, but the certifier is itself complex and the certification process can take a long time. On the other hand, if the certifier is a simple proof checker then certification can be done relatively quickly and can be more easily accepted even by skeptical users, but one needs to generate very detailed proofs of correctness together with the code.

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] Robert G. Brown and Patrick Hwang. *Introduction to Random Signals and Applied Kalman Filtering.* John Wiley & Son, 3rd edition, 1997.

[3] Manuel Clavel. ITP tool. Department of Philosophy, University of Navarre, `http://sophia.unav.es/ clavel/itp/`.

[4] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude: specification and programming in rewriting logic. SRI International, January 1999, `http://maude.csl.sri.com`.

[5] Compaq. Extended Static Checking for Java, 2000. URL: `www.research.compaq.com/SRC/esc`.

[6] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action*, pages 3–167. Kluwer, 2000.

[7] Rudolf E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME-Journal of Basic Engineering*, (82):35–45, 1960.

[8] Michael Lowry, Thomas Pressburger, and Grigore Roşu. Certifying domain-specific policies. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 81–90. IEEE, 2001. Coronado Island, California.

[9] José Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, pages 73–155, 1992.

[10] José Meseguer. Membership algebra as a logical framework for equational specification. In *Proceedings, WADT'97*, volume 1376 of *LNCS*, pages 18–61. Springer, 1998.

[11] José Meseguer and Grigore Roşu. A total approach to partial algebraic specification. In *International Conference on Automata, Languages and Programming (ICALP'02)*, Lecture Notes in Computer Science, 2002. To appear.

[12] George C. Necula. Proof-carrying code. In *Proceedings of the 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119. ACM Press, 1997.

[13] PolySpace. URL: `http://www.polyspace.com`.

[14] Grigore Roşu and Jonathan Whittle. Towards certifying domain-specific properties of synthesized code. In *Proceedings, VCL*, 2002.

[15] Grigore Roşu and Jonathan Whittle. Towards certifying domain-specific properties of synthesized code (extended abstract). In *Proceedings, ASE*, 2002.

[16] 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 *Lecture Notes in Computer Science*, pages 302–305. Springer, April 1998.

[17] 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 *Lecture notes in computer science*, Kloster Irsee, Germany, 1995. Springer.

[18] Greg Welch and Gary Bishop. An Introduction to the Kalman Filter. Course, SIGGRAPH 2001.

[19] 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 Conference on Automated Software Engineering (ASE01)*, San Diego, CA, USA, 2001.

Appendix A

```
--- ------------------------------------------------
--- ABSTRACT DOMAIN FORMALIZATION---
--- ------------------------------------------------

fmod DOMAIN is
  including MACHINE-INT .

  sorts Real .
  sorts Matrix MatrixExp  MatrixExp? .
  sorts RandomMatrix RandomMatrixExp RandomMatrixExp? .
  subsort MachineInt < Real .
  subsort Matrix < MatrixExp .
  subsort MatrixExp < MatrixExp? .
  subsort Matrix < MatrixExp? .
  subsort RandomMatrix < RandomMatrixExp .
  subsort RandomMatrixExp < RandomMatrixExp? .
  subsort RandomMatrix < RandomMatrixExp? .
  subsort Matrix < RandomMatrix .
  subsort Matrix < RandomMatrixExp .
  subsort Matrix < RandomMatrixExp? .
  subsort MatrixExp < RandomMatrixExp .
  subsort MatrixExp < RandomMatrixExp? .
  subsort MatrixExp? < RandomMatrixExp? .

  ops c r : RandomMatrixExp -> MachineInt .

  vars RP RQ RR : RandomMatrixExp .
  vars M1 M2 M3 M4 M5 : RandomMatrixExp .
  vars P Q R  : MatrixExp .
  vars M N N1 : MachineInt .

  op zero : MachineInt MachineInt -> Matrix .
  eq (r(zero(N,M))) = (N) .
  eq (c(zero(N,M))) = (M) .

  op _+_ : RandomMatrixExp RandomMatrixExp -> RandomMatrixExp? .
  op _+_ : MatrixExp MatrixExp -> MatrixExp? .
 cmb (RP + RQ) : RandomMatrixExp if (r(RP) == r(RQ) and c(RP) ==  c(RQ)) .
 cmb ( P +  Q) : MatrixExp       if (r( P) == r( Q) and c( P) ==  c( Q)) .
 ceq (r(RP + RQ)) = (r(RP))      if (RP + RQ : RandomMatrixExp) .
 ceq (c(RP + RQ)) = (c(RP))      if (RP + RQ : RandomMatrixExp) .
[ax*> | assoc+]
 ceq ((RP + RQ) + RR) = (RP + (RQ + RR))
    if (RP + RQ : RandomMatrixExp and RQ + RR : RandomMatrixExp) .
[ax*> | comm+]
 ceq (RP + RQ) = (RQ + RP) if (RP + RQ : RandomMatrixExp) .
[ax> |  zero-+-left]
 ceq (zero(N,M) + RP) = (RP) if (r(RP) == N and c(RP) == M) .
[ax> |  zero-+-right]
 ceq (RP + zero(N,M)) = (RP) if (r(RP) == N and c(RP) == M) .
```

```
  op -_  : RandomMatrixExp -> RandomMatrixExp .
  op -_  : MatrixExp -> MatrixExp .
  eq (r(- RP)) = (r(RP)) .
  eq (c(- RP)) = (c(RP)) .
[ax*> | minus-def1]
  eq ((- RP) +  RP) = (zero(r(RP),c(RP))) .
[ax*> | minus-idemp]
  eq (- (- RP)) = (RP) .
[ax*> | minus-suma]
  eq (- (RP + RQ)) = ((- RP) + (- RQ)) .

  op _-_ : RandomMatrixExp RandomMatrixExp -> RandomMatrixExp? .
  op _-_ : MatrixExp MatrixExp -> MatrixExp? .
[ax*> | minus]
 ceq (RP - RQ) = (RP + (- RQ)) if (r(RP) == r(RQ) and c(RP) == c(RQ)) .
 cmb (RP - RQ) : RandomMatrixExp if (r(RP) == r(RQ) and   c(RP) ==  c(RQ)) .
 cmb ( P -  Q) : MatrixExp       if (r( P) == r( Q) and   c( P) ==  c( Q)) .
 ceq (RP - zero(N, M)) = (RP)   if (r(RP) == N and c(RP) == M) .
 ceq (zero(N, M) - RP) = (- RP) if (r(RP) == N and c(RP) == M) .
 ceq (r(RP - RQ)) = (r(RP))      if (RP + RQ : RandomMatrixExp) .
 ceq (c(RP - RQ)) = (c(RP))      if (RP + RQ : RandomMatrixExp) .
[ax*> | minus-def2]
  eq (RP - RP) = (zero(r(RP),c(RP))) .


  op id  : MachineInt -> Matrix .
  eq (c(id(N))) = (N) .
  eq (r(id(N))) = (N) .

  op _*_ : RandomMatrixExp RandomMatrixExp -> RandomMatrixExp? .
  op _*_ : MatrixExp MatrixExp -> MatrixExp? .
 cmb (RP * RQ) : RandomMatrixExp if (c(RP) == r(RQ)) .
 cmb ( P *  Q) : MatrixExp       if (c( P) == r( Q)) .
 ceq (r(RP * RQ)) = (r(RP)) if (RP * RQ : RandomMatrixExp) .
 ceq (c(RP * RQ)) = (c(RQ)) if (RP * RQ : RandomMatrixExp) .
[ax>  | zero-*-left]
 ceq (zero(N,M) * RP) = (zero(N, c(RP))) if (r(RP) == M) .
[ax>  | zero-*-right]
 ceq (RP * zero(N,M)) = (zero(r(RP), M)) if (c(RP) == N) .
[ax*> | id*left]
 ceq (id(N) * RP) = (RP) if (r(RP) == N) .
[ax*> | id*right]
 ceq (RP * id(N)) = (RP) if (c(RP) == N) .
[ax*> | assoc*]
 ceq ((RP * RQ) * RR) = (RP * (RQ * RR))
    if (RP * RQ : RandomMatrixExp and RQ * RR : RandomMatrixExp) .
[ax*> | distr*+]
 ceq (RP * (RQ + RR)) = ((RP * RQ) + (RP * RR))
    if (RQ + RR : RandomMatrixExp and  RP * RQ : RandomMatrixExp) .
[ax*> | distr+*]
 ceq ((RQ + RR) * RP) = ((RQ * RP) + (RR * RP))
    if (RQ + RR : RandomMatrixExp and  RQ * RP : RandomMatrixExp) .
[ax*> | distr-*left]
  eq ((- RP) * RQ) = (- (RP * RQ)) .
[ax*> | distr-*right]
  eq (RP * (- RQ)) = (- (RP * RQ)) .
  op _*_ : MachineInt RandomMatrixExp -> RandomMatrixExp .
  op _*_ : MachineInt MatrixExp -> MatrixExp .
  eq (r(N * RP)) = (r(RP)) .
  eq (c(N * RP)) = (c(RP)) .
  eq (N1 * zero(N,M)) = (zero(N,M)) .
[ax*> | scalar*-1]
 ceq ((N * RP) * RQ) = (N * (RP * RQ)) if (RP * RQ : RandomMatrixExp) .
[ax*> | scalar*-2]
 ceq (RP * (N * RQ)) = (N * (RP * RQ)) if (RP * RQ : RandomMatrixExp) .
[ax*> | scalar+]
 ceq (N * (RP + RQ)) = ((N * RP) + (N * RQ)) if (RP + RQ : RandomMatrixExp) .
[ax*> | scalar-]
 ceq (N * (RP - RQ)) = ((N * RP) - (N * RQ)) if (RP + RQ : RandomMatrixExp) .
[ax*> | scalar-minus]
  eq (N * (- RP)) = (- (N * RP)) .
[ax*> | scalar-trans]
  eq (mtrans(N * RP)) = (N * mtrans(RP)) .

  op mtrans : RandomMatrixExp -> RandomMatrixExp .
  op mtrans : MatrixExp -> MatrixExp .
  eq (r(mtrans(RP))) = (c(RP)) .
  eq (c(mtrans(RP))) = (r(RP)) .


[ax*> | id-trans]
  eq (mtrans(id(N))) = (id(N)) .
[ax*> | mtrans-minus]
  eq (mtrans(- RP)) = (- mtrans(RP)) .
[ax*> | mtrans-idemp]
  eq (mtrans(mtrans(RP))) = (RP) .
[ax*> | distr+mtrans]
 ceq (mtrans(RP + RQ)) = (mtrans(RP) + mtrans(RQ))
    if (RP + RQ : RandomMatrixExp) .
[ax*> | distr*mtrans]
 ceq (mtrans(RP * RQ)) = (mtrans(RQ) * mtrans(RP))
    if (RP * RQ : RandomMatrixExp) .

  op minv : RandomMatrixExp -> RandomMatrixExp? .
  op minv : MatrixExp -> MatrixExp? .
 ceq (r(minv(RP))) = (r(RP)) if (minv(RP) : RandomMatrixExp) .
 ceq (c(minv(RP))) = (c(RP)) if (minv(RP) : RandomMatrixExp) .
 cmb (minv(minv(RP))) : RandomMatrixExp if (minv(RP)) : MatrixExp .
 cmb (minv(minv(P))) : MatrixExp if (minv(P)) : MatrixExp .
[ax*> | minv-def1]
 ceq ((minv(RP)) * RP) = (id(r(RP))) if (minv(RP) : RandomMatrixExp) .
[ax*> | minv-def2]
 ceq (RP * (minv(RP))) = (id(r(RP))) if (minv(RP) : RandomMatrixExp) .
[ax*> | mtrans-inv]
 ceq (mtrans(minv(RP))) = (minv(mtrans(RP))) if (minv(RP) : RandomMatrixExp) .
[ax*> | idemp-inv]
  ceq (minv(minv(RP))) = (RP)  if (minv(RP) : RandomMatrixExp) .
[ax*> | minv-of-matrix]
 ceq (minv(RP)) = (RQ) if (RP * RQ is id(r(RP)) and RQ * RP is id(r(RP))) .
```

```
  op E|_| : RandomMatrixExp -> MatrixExp .
  eq (r(E| RP |)) = (r(RP)) .
  eq (c(E| RP |)) = (c(RP)) .
[ax*> | E+]
 ceq (E| RP + RQ |) = (E| RP | + E| RQ |) if (RP + RQ : RandomMatrixExp) .
[ax*> | Eopus]
  eq (E| - RP |) = (- E| RP |) .
[ax*> | E-]
 ceq (E| RP - RQ |) = (E| RP | - E| RQ |) if (RP + RQ : RandomMatrixExp) .
[ax*> | Emtrans]
  eq (E| mtrans(RP) |) = (mtrans(E| RP |)) .
[ax*> | mtrans-E]
 eq (mtrans(E| RP * mtrans(RP) |)) = (E| RP * mtrans(RP) |) .
[ax*> | E*left]
 ceq (E| P * RQ |) = (P * E| RQ |) if (P * RQ  : RandomMatrixExp) .
[ax*> | E*right]
 ceq (E| RP * Q |) = ( E| RP |  * Q) if (RP * Q  : RandomMatrixExp) .

  sorts MatrixVar MatrixFun  .
  subsort MatrixVar < Matrix .
  subsort MatrixVar < MatrixExp .
  subsort MatrixVar < MatrixExp? .
  subsort MatrixVar < RandomMatrix .
  subsort MatrixVar < RandomMatrixExp .
  subsort MatrixVar < RandomMatrixExp? .

--- X in P----
--- here is the only place where we need the distinction
--- between Matrix and MatrixExp: in-2.

op _in_ : MatrixVar RandomMatrixExp -> Bool .
var RY : RandomMatrix .
[ax> | in-1]
  eq (X in X) = (true) .
[ax> | in-2]
  ceq (X in RY) = (false) if (X =/= RY) .
[ax> | in-3]
  ceq (X in (RP * RQ)) = ((X in RP) or (X in RQ)) if (RP * RQ : RandomMatrixExp) .
[ax> | in-4]
  ceq (X in (RP + RQ)) = ((X in RP) or (X in RQ))  if (RP + RQ : RandomMatrixExp) .
[ax> | in-5]
  ceq (X in (RP - RQ)) = ((X in RP) or (X in RQ)) if (RP + RQ : RandomMatrixExp) .
[ax> | in-6]
  eq (X in (N * RP)) = (X in RP) .
[ax> | in-7]
  eq (X in (mtrans(RP))) = (X in RP) .
[ax> | in-8]
  eq (X in (E| RP |)) = (X in RP) .

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

[ax*> | fun-def1]
 ceq ((/\ X . X)(R)) = (R) if (X + R : MatrixExp) .
```

```
[ax*> | fun-def2]
 ceq ((/\ X . P)(R)) = (P) if (X + R : MatrixExp and not(X in P)) .
[ax*> | fun-sum]
 ceq ((/\ X . (P + Q))(R)) = (((/\ X . P)(R)) + ((/\ X . Q)(R)))
    if (P + Q : MatrixExp and X + R : MatrixExp) .
[ax*> | fun-scalar-mult]
 ceq ((/\ X . (N * P))(R)) = (N * ((/\ X . P)(R))) if (X + R : MatrixExp).
[ax*> | fun-mult]
 ceq ((/\ X . (P * Q))(R)) = (((/\ X . P)(R)) * ((/\ X . Q)(R)))
    if (P * Q : MatrixExp and X + R : MatrixExp) .
[ax*> | fun-minus]
 ceq ((/\ X . (- P))(R)) = (- ((/\ X . P)(R)))
    if (X + R : MatrixExp) .
[ax*> | fun-trans]
 ceq ((/\ X . (mtrans(P)))(R)) = (mtrans((/\ X . P)(R)))
    if (X + R : MatrixExp) .

  op d|trace_|/d_ : MatrixFun MatrixVar -> MatrixFun .
[ax*> | tracem-def1]
 ceq ((d|trace(/\ X . X)|/d(X))(R)) = (id(r(X))) if (X + R : MatrixExp) .
[ax*> | tracem-def2]
 ceq ((d|trace(/\ X . P)|/d(X))(R)) = (zero(r(X),c(X)))
    if (X + R : MatrixExp and not(X in P)) .
[ax*> | trace+]
 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) .
[ax*> | trace-]
 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) .
[ax*> | trace-lemma1]
 ceq (d|trace(/\ X . (X * P))|/d(X)) = (/\ X . (mtrans(P)))
    if (X * P : MatrixExp and not(X in P)) .
[ax*> | trace-lemma2]
 ceq (d|trace(/\ X . ((X * P) * mtrans(X)))|/d(X)) = (/\ X . (2 * (X * P)))
    if (X * P : MatrixExp and P * mtrans(X) : MatrixExp and
        mtrans(P) == P and not(X in P)) .
[ax*> | trace-lemma1*]
 ceq (d|trace(/\ X . (P * mtrans(X)))|/d(X)) = (/\ X . P)
    if (P * mtrans(X) : MatrixExp and not(X in P)) .

  op _minimizes_ : MatrixExp MatrixFun -> Bool .
[ax*> | minimizes]
  eq (Q minimizes (/\ X . P)) =
     ((d|trace(/\ X . P)|/d(X))(Q) is zero(r(X),c(X))) .

  op _is_ : MatrixExp MatrixExp -> Bool .
[ax*> | misc-is]
  eq (P is P) = (true) .

 cmb (minv((M1 * (M2 * minv(M3))) + id(N))) : MatrixExp
    if (N == r(M3) and
        minv(M3) : MatrixExp and minv((M1 * M2) + M3) : MatrixExp) .

endfm
```

Appendix B

```
--- -------------------------------------
--- LEMMAS
--- -------------------------------------


[lem*> | lemma-1]
 ceq (RP - (RQ + RR)) = ((RP - RQ) - RR)
    if (r(RP) == r(RQ) and r(RQ) == r(RR) and c(RP) == c(RQ)
        and c(RQ) == c(RR)) .
[lem*> | lemma-2]
 ceq (((RP * M1) + M2) - (RP * M3)) = ((RP * (M1 - M3))+ M2)
    if (r(M2) == r(RP) and r(M1) == c(RP) and r(M3) == c(RP)
        and c(M2) == c(M1) and c(M3) == c(M1)) .
[lem*> | lemma-3]
 ceq (RP * ((RQ * (M1 - M2)) + RR )) = (((RP * RQ) * (M1 - M2)) + ( RP * RR))
    if (c(RP) == r(RQ) and c(RQ) == r(M1) and r(M1) == r(M2) and
        c(M1) == c(M2) and c(M1) == c(RR) and r(RQ) == r(RR)) .
[lem*> | lemma-4]
 ceq ((M1 - M2) - ((RP * RQ) * (M1 - M2))) = ((id(r(M1)) - (RP * RQ)) * (M1 - M2))
    if (r(M1) == r(M2) and  c(M1) == c(M2) and c(RP) == r(RQ)  and r(RP) == r(M1)
        and c(RQ) == r(M1)) .
[lem*> | lemma-5]
 ceq (((RP * (M1 - M2)) - (RQ * RR)) * mtrans((RP * (M1 - M2)) - (RQ * RR))) =
     (((((RP * ((M1 - M2) * mtrans(M1 - M2))) * mtrans(RP)) +
        ((RQ * (RR * mtrans(RR))) * mtrans(RQ))) -
       ((RP * ((M1 - M2) * mtrans(RR))) * mtrans(RQ))) -
      ((RQ * (RR * mtrans(M1 - M2))) * mtrans(RP)))
    if (r(M1) == r(M2) and c(M1) == c(M2) and c(RP) == r(M1) and c(RQ) == r(RR)
        and r(RP) == r(RQ) and c(M1) == c(RR)) .
[lem*> | lemma-6]
 ceq (((M1 - (M2 * M3) ) * RP) * mtrans(M1 - (M2 * M3) )) =
     (((((M1 * RP ) * mtrans(M1)) - ((( M1 * RP) * mtrans(M3)) * mtrans(M2))) -
       (M2 * ((M3 * RP) * mtrans(M1)))) +
      ((M2 * ( (M3 * RP) * mtrans(M3))) * mtrans(M2)))
    if (c(M2) == r(M3) and r(M1) == r(M2) and c(M1) == c(M3) and
        c(M1) == r(RP) and c(RP) == c(M1)) .
[lem*> | lemma-7]
 ceq ((mtrans(mtrans(RP))) * (RQ * RR))  = ((RP * RQ) * RR)
    if (c(RQ) == r(RR) and c(RP) == r(RQ)) .
[lem*> | lemma-8]
 ceq ((N * (RP * M1)) + (N * (RP * M2))) = (N * (RP * (M1 + M2)))
    if (c(RP) == r(M1) and r(M1) == r(M2) and c(M1) == c(M2)) .
[lem*> | lemma-9]
 ceq ((((- RP - RQ) + RR) + M1)) = (- (RP + RQ) + (RR + M1))
    if (c(RP) == c(RQ) and  r(RP) == r(RQ) and c(RR) == c(RQ) and r(RR) == r(RQ)
        and c(M1) == c(RQ) and r(M1) == r(RQ)) .
[lem*> | lemma-10]
 ceq  (((RP * RQ) * minv(M1)) * M1) = (RP * RQ)
    if (c(RP) == r(RQ) and c(M1) == r(M1) and minv(M1) : RandomMatrixExp and c(RQ) == r(M1)) .
```

```
[ax*> | lemma-11]
  eq (RP + RP) = (2 * RP) .
[lem*> | lemma-12]
  ceq ((RR + (((RP * M1) * RQ) * M3)) + (((RP * M2) * RQ) * M3)) =
    (RR + (((RP * (M1 + M2)) * RQ) * M3))
    if (c(RR) == c(M3) and r(RR) == r(RP) and c(RP) == r(M1) and c(M1) == r(RQ)
     and c(RQ) == r(M3) and c(RP) == r(M2) and c(M2) == r(RQ)) .
[lem*> | lemma-13]
  ceq (RP * (RQ * (mtrans(mtrans(RR)) * M1))) = ((RP * RQ) * (RR * M1))
      if (c(RP) == r(RQ) and c(RQ) == r(RR) and c(RR) == r(M1)) .
[lem*> | lemma-14]
  ceq ((RP - RQ) + RQ) = (RP)
      if (r(RP) == r(RQ) and c(RP) == c(RQ)) .
[lem*> | lemma-15]
  ceq (RP - (RQ * (RR * RP))) = ((id(r(RP)) - (RQ * RR)) * RP)
      if (r(RP) == r(RQ) and c(RQ) == r(RR) and c(RR) == r(RP)) .
[lem*> | lemma-16]
  ceq (((RP * (M1 - M2)) + RQ) * mtrans((RP * (M1 - M2)) + RQ )) =
     ((((RP * ((M1 - M2) * mtrans(M1 - M2))) * mtrans(RP)) + (RQ * mtrans(RQ))) +
      ((RP * ((M1 - M2) * mtrans(RQ))) + ((RQ * mtrans(M1 - M2)) * mtrans(RP))))
      if (c(RP) == r(M1) and r(M1) == r(M2) and c(M1) == c(M2) and r(RP) == r(RQ)
          and c(M1) == c(RQ)) .
[lem*> | lemma-17]
 ceq (((RP + RQ) + RR ) - RP) = (RQ + RR)
    if (c(RP) == c(RQ) and r(RP) == r(RQ) and c(RR) == c(RQ) and r(RR) == r(RQ)) .
[lem*> | lemma-18]
  ceq ((M1 * M2) * minv(M3 + M4)) = ((RP * ((minv(RP) * M1) * (M2 * minv(RQ)))) *
   (RQ * minv(M3 + M4)))
      if (c(M1) == r(M2) and c(M2) == r(M3) and c(M3) == c(M4) and r(M3) == r(M4)
       and c(M3) == r(M3) and minv(M3 + M4) : RandomMatrixExp and c(RP) == r(RP)
       and minv(RP) : RandomMatrixExp and c(RQ) == r(RQ)
       and minv(RQ) : RandomMatrixExp and c(RP) == r(M1) and c(RQ) == r(M3)) .
[lem*> | lemma-19]
 ceq ((id(N) + ((M1  * M2 ) * M3)) * M1) = (M1  * (id(c(M1)) + ((M2 * M3) * M1)))
    if (r(M1) == (N) and c(M3) == r(M1) and c(M1) == r(M2) and c(M2) == r(M3)) .
[lem*> | lemma-20]
 ceq (M1 * minv((M2 * M3) + M1)) = (minv((M2 * (M3 * minv(M1))) + id(r(M1))))
    if (c(M1) == r(M1) and minv(M1) : RandomMatrixExp and c(M2) == r(M3) and r(M2) == r(M1)
        and c(M3) == c(M1) and minv((M2 * M3) + M1) : RandomMatrixExp) .
[lem*> | lemma-21]
  ceq ((M3 * (M4 * (M1 + M2))) * minv(M2 + M1)) = (M3 * M4)
      if (c(M3) == r(M4) and c(M4) == r(M1) and r(M1) == r(M2) and c(M1) == c(M2)
       and minv(M2 + M1) : RandomMatrixExp) .
[lem*> | lemma-22]
  ceq (((M1 - M2) * RP) * (M3 + M4)) = ((M1 * (RP * M3)) + ((M1 * (RP * M4)) -
                                    ((M2 * (RP * M3)) + (M2 * (RP * M4)))))
      if (r(M1) == r(M2) and c(M1) == c(M2) and c(M1) == r(RP) and c(RP) == r(M3)
          and r(M3) == r(M4) and c(M3) == c(M4)) .
[lem*> | lemma-23]
  ceq (((M1 * RP) * M2) + (((M1 * RP) *  M3) * RQ)) = ((M1 * RP) *  (M2 + (M3 * RQ)))
      if (c(M1) == r(RP) and c(RP) == r(M2) and c(RP) == r(M3) and c(M3) == r(RQ)
          and c(RQ) == c(M2)) .
[lem*> | lemma-24]
  ceq (M1 + (M2 * (M3 * (M4 * M1)))) = ((id(r(M1)) + ((M2 * M3) * M4)) * M1)
      if (r(M1) == r(M2) and c(M2) == r(M3) and c(M3) == r(M4) and c(M4) == r(M1)) .
[lem*> | lemma-25]
```

```
 ceq (id(N) + (M1 * (M2 * minv(RP)))) = ((RP + (M1 * M2)) * minv(RP))
    if (r(RP) == (N) and r(M1) == (N) and c(M1) == r(M2) and c(M2) == r(RP) and c(RP) == (N)
        and minv(RP): RandomMatrixExp ) .
[lem*> | lemma-26]
  ceq (((M1 * M2) * (minv(RP + RQ))) * (((RQ + RP) * M3) * M4)) = (M1 * ((M2 * M3) * M4))
      if (c(M1) == r(M2) and c(M2) == r(RP) and r(RP) == r(RQ) and c(RP) == c(RQ)
          and r(RP) == c(RP) and c(RP) == r(M3) and c(M3) == r(M4)
          and minv(RP + RQ) : RandomMatrixExp) .
[lem*> | lemma-27]
  ceq ((M1 + M2) * ((M3 - M4) * RP)) = (((M1 * M3) * RP) + (((M2 * M3) * RP) -
                                     (((M1 * M4) * RP) + ((M2 * M4) * RP))))
      if (r(M1) == r(M2) and c(M1) == c(M2) and c(M1) == r(M3) and r(M3) == r(M4)
          and c(M3) == c(M4) and c(M4) == r(RP)) .
[lem*> | lemma-28]
  ceq (minv(RP) * ((( RP * M1) * M2) * M3)) = ((M1 * M2) * M3)
      if (r(RP) == c(RP) and minv(RP) : RandomMatrixExp and c(RP) == r(M1) and c(M2) == r(M3)) .
[lem*> | lemma-29]
  ceq ((((RP * M1) * M2) * RQ) + ((((RP * M3) * M4) * M5) * RQ)) = ((RP * ((M1 * M2) +
                                                        ((M3 * M4) * M5))) * RQ)
      if (c(RP) == r(M1) and c(M1) == r(M2) and c(M2) == r(RQ) and c(RP) == r(M3)
          and c(M3) == r(M4) and c(M4) == r(M5) and c(M5) == r(RQ)) .
[lem*> | lemma-30]
  ceq ((RP * M1)  + (M3 * ((M4 * RP) * M1))) = (((id(r(RP)) + (M3 * M4)) * RP) * M1)
       if (c(RP) == r(M1) and r(RP) == r(M3) and c(M3) == r(M4) and c(M4) == r(RP)) .
[lem*> | lemma-31]
  ceq (id(N) + ((minv(RP)* M1) * (M2 * M3))) = (minv(RP)* (RP + (((M1 * M2) * M3))))
      if (c(RP) == r(RP) and r(RP) == (N) and minv(RP) : RandomMatrixExp and c(RP) == r(M1)
          and c(M1) == r(M2) and c(M2) == r(M3)) .
[lem*> | lemma-32]
  ceq ((M1 * (RP + RQ)) * minv(RQ + RP)) = (M1)
      if (c(M1) == r(RP) and r(RP) == r(RQ) and c(RP) == c(RQ)
          and minv(RQ + RP) : RandomMatrixExp) .
[lem*> | lemma-33]
  ceq ((((M1 * M2) * M3) * M4) - ((M1 * (M2 * M3)) * M4)) = (zero(r(M1), c(M4)))
      if (c(M1) == r(M2) and c(M2) == r(M3) and c(M3) == r(M4)) .
[lem*> | lemma-34]
  ceq ((minv(M1) + M2) * M1) = (id(r(M1)) + (M2 * M1))
      if (r(M1) == c(M1) and minv(M1) : RandomMatrixExp and r(M1) == r(M2) and c(M1) == c(M2)) .
```

Lemma	KF	IF	EKF
lemma-1	2	2	2
lemma-2	3	3	3
lemma-3	2	2	2
lemma-4	2	2	2
lemma-5	2	2	2
lemma-6	2	2	2
lemma-7	2	2	2
lemma-8	1	1	1
lemma-9	1	1	1
lemma-10	2	2	2
lemma-11	1	1	1
lemma-12	1	1	1
lemma-13	1	1	1
lemma-14	1	1	1
lemma-15	1	1	1
lemma-16	1	1	1
lemma-17	0	0	3
lemma-18	0	1	0
lemma-19	0	1	0
lemma-20	0	1	0
lemma-21	0	1	0
lemma-22	0	1	0
lemma-23	0	1	0
lemma-24	0	1	0
lemma-25	0	1	0
lemma-26	0	1	0
lemma-27	0	1	0
lemma-28	0	1	0
lemma-29	0	1	0
lemma-30	0	1	0
lemma-31	0	1	0
lemma-32	0	1	0
lemma-33	0	1	0
lemma-34	0	1	0

Appendix C

```
/* -----------------------------------------------
---  Kalman Filter Specification
--- ---------------------------------------------

--- 44 axioms---

---initial equations---

  ops x z v w : MachineInt -> RandomMatrix .
  ops n m k : -> MachineInt .
  ops Phi H :  MachineInt -> Matrix .

  var K : MachineInt .

  eq (r(x(K))) = (n) .        eq (c(x(K))) = (1) .
  eq (r(z(K))) = (m) .        eq (c(z(K))) = (1) .
  eq (r(v(K))) = (m) .        eq (c(v(K))) = (1) .
  eq (r(w(K))) = (n) .        eq (c(w(K))) = (1) .
  eq (r(Phi(K))) = (n) .      eq (c(Phi(K))) = (n) .
  eq (r(H(K))) = (m) .        eq (c(H(K))) = (n) .

[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)) .

---initial suppositions---

ops Q R : MachineInt -> Matrix .

[ax*> | RK]
  eq (R(K)) =  (E| v(K) * mtrans(v(K)) |) .
[ax*> | QK]
  eq (Q(K)) = (E| w(K) * mtrans(w(K)) |) .

var J : MachineInt .

[ax*> | white_noise_w_1]
  eq (E| w(K) |) = (zero(n,1)) .
[ax*> | white_noise_v_1]
  eq (E| v(K) |) = (zero(m,1)) .
[ax*> | white_noise_w_2]
  ceq (E| w(K) * mtrans(w(J)) |) = (zero(n,n)) if (K =/= J) .
[ax*> | white_noise_v_2]
  ceq (E| v(K) * mtrans(v(J)) |) = (zero(m,m)) if (K =/= J) .
[ax*> | zerro_cross_v_w]
  eq (E| w(K) * mtrans(v(J)) |) = (zero(n,m)) .

---a priori and a posteriori estimates---

  ops  xhat xhatmin : MachineInt -> RandomMatrix .
  ops  Pminus P : MachineInt -> Matrix .
  eq (r(xhat(K))) = (n) .     eq (c(xhat(K))) = (1) .
  eq (r(xhatmin(K))) = (n) .  eq (c(xhatmin(K))) = (1) .

[ax*> | Pminus-0 ]
  eq (Pminus(0)) = (E| (x(0) - xhatmin(0)) * mtrans(x(0) - xhatmin(0)) |) .
[ax*> | zerro-cross_v_x-xhatmin_1]
  eq (E| (x(K) - xhatmin(K)) * mtrans(v(K)) |) = (zero(n, m)) .
[ax*> | zerro-cross_v_x-xhatmin_2]
  eq (E| v(K) *  mtrans(x(K) - xhatmin(K)) |) = (zero(m, n)) .
[ax*> | zerro-cross_w_x-xhatmin_1]
  eq (E| ( x(K) - xhat(K) ) * mtrans(w(K)) |) = (zero(n,n)) .
[ax*> | zerro-cross_w_x-xhatmin_2]
  eq (E| w(K) * mtrans( x(K) - xhat(K) ) |) = (zero(n,n)) .

---measurement innovation---

  op zminus : MachineInt -> RandomMatrix .
  eq (r(zminus(K))) = (m) .  eq (c(zminus(K))) = (1) .

---gain---

  op gain  : MachineInt -> Matrix .
  eq (r(gain(K))) = (n) .    eq (c(gain(K))) = (m) .

--- best estimate---

  op y : -> MatrixVar .
  eq (r(y)) = (n) .          eq (c(y)) = (m) .

  ops BestPriorEstimate BestEstimate : MachineInt -> RandomMatrixExp .
  op Estimate : MachineInt MatrixExp -> RandomMatrixExp .

  var G : MatrixExp .

  eq (r(Estimate(K, G))) = (n) .  eq (c(Estimate(K, G))) = (1) .

[ax*> | Estimate]
  eq (Estimate(K, G)) = (BestPriorEstimate(K) + (G * (z(K) - zminus(K)))) .
[ax*> | BestPriorEstimate-0]
  eq (BestPriorEstimate(0)) = (xhatmin(0)) .
[ax*> | BestEstimate]
 ceq (Estimate(K, G)) = (BestEstimate(K))
    if  (G minimizes /\ y . (E| (x(K) - Estimate(K, y)) * mtrans(x(K) - Estimate(K, y)) |)) .
[ax*> | BestPriorEstimate]
  eq (BestPriorEstimate(K + 1)) = (Phi(K) * BestEstimate(K)) .

---additional hypothesis---

mb (minv (((H(K) * E| (x(K) - xhatmin(K)) * mtrans(x(K) - xhatmin(K)) |) *
    mtrans(H(K))) + E| v(K) * mtrans(v(K)) |)) :  MatrixExp .

---------------------------------------------------------------------- */

input xhatmin(0), Pminus(0)

/* ----------------------------------------------------------------------
```

```
[proof assertion-1-1, first iteration]

(set BestPriorEstimate-0 in (1) .)
(rwr (1) .)
(idt (1) .)
---------------------------------------------------------------------- */

/* ----------------------------------------------------------------------
[proof assertion-1-2, first iteration]

(set Pminus-0 in (1) .)
(rwr (1) .)
(idt (1) .)
--------------------------------------------------------------------- */

for(k,0,n)
{
/* ----------------------------------------------------------------------
[lem*> | assertion-1-1]
  eq (BestPriorEstimate(k)) = (xhatmin(k)) .

[lem*> | assertion-1-2]
  eq (Pminus(k)) = (E| (x(k)  - xhatmin(k)) * mtrans(x(k)  - xhatmin(k)) |) .
---------------------------------------------------------------------- */

// [lem*> | instruction-1-1]
  zminus(k) := H(k) * xhatmin(k) ;

// [lem*> | instruction-2]
  gain(k) := (Pminus(k) * mtrans(H(k))) * minv(((H(k) * Pminus(k)) * mtrans(H(k))) + R(k)) ;

/* ----------------------------------------------------------------------
[proof assertion-2]

(set (instruction-1-1 instruction-2 assertion-1-1 assertion-1-2) in (1) .)
(set (Estimate KF-measurement zerro-cross_v_x-xhatmin_1 zerro-cross_v_x-xhatmin_2
      RK) in (1) .)
(set (id*left id*right id-trans distr*mtrans minus-def1 misc-is) in (1) .)
(set (E- E+ E*left E*right mtrans-E) in (1) .)
(set (fun-scalar-mult fun-mult fun-trans fun-def1 fun-def2) in (1) .)
(set (trace+ trace- tracem-def1  tracem-def2 trace-lemma1 trace-lemma1* trace-lemma2
      minimizes) in (1) .)
(set (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) .)
---------------------------------------------------------------------- */

/* ----------------------------------------------------------------------
[lem*> | assertion-2]
  eq (gain(k)
     minimizes
     (/\ y . (E| (x(k) - Estimate(k, y)) *
         mtrans(x(k) - Estimate(k, y)) |)))
     = (true) .
---------------------------------------------------------------------- */
```

```
// [lem*> | instruction-3]
  xhat(k) := Estimate(k, gain(k)) ;

/* ---------------------------------------------------------------------
[proof assertion-3]

(set (instruction-3 assertion-2 ) in (1) .)
(apply BestEstimate to (1) at (none) with (G <- (gain(k))) .)
(idt (1) .)
------------------------------------------------------------------- */

/* ---------------------------------------------------------------------
[lem*> | assertion-3]
  eq (xhat(k)) = (BestEstimate(k)) .
------------------------------------------------------------------- */

// [lem*> | instruction-3]
  P(k) := (id(n) - (gain(k) * H(k))) * Pminus(k);

/* ---------------------------------------------------------------------
[proof assertion-4]

(set (instruction-1-1 instruction-2 instruction-3 instruction-4 assertion-1-1
      assertion-1-2) in (1) .)
(set (Estimate KF-measurement RK zerro-cross_v_x-xhatmin_1 zerro-cross_v_x-xhatmin_2)
      in (1) .)
(set (distr*mtrans mtrans-inv distr+mtrans id*left id*right id-trans) in (1) .)
(set (E- E+ E*left E*right mtrans-E) in (1) .)
(set (lemma-1 lemma-2 lemma-3 lemma-4 lemma-5 lemma-6 lemma-7 lemma-10 lemma-12
      lemma-13 lemma-14 lemma-15) in (1) .)
(rwr (1) .)
(idt (1) .)
------------------------------------------------------------------- */

/* ---------------------------------------------------------------------
[lem*> | assertion-4]
  eq (P(k)) = (E| (x(k) - xhat(k)) * mtrans(x(k) - xhat(k)) |) .
------------------------------------------------------------------- */

// [lem*> | instruction-5]
  xhatmin(k + 1) := Phi(k) * xhat(k);

// [lem*> | instruction-6]
  Pminus(k + 1) := ((Phi(k) * P(k)) * mtrans(Phi(k))) + Q(k);

/* ---------------------------------------------------------------------
[proof assertion-1-1]

(set (instruction-5 assertion-3 BestPriorEstimate) in (1) .)
(rwr (1) .)
(idt (1) .)
------------------------------------------------------------------- */

/* ---------------------------------------------------------------------
[proof assertion-1-2]
```

```
(set (instruction-5 instruction-6 assertion-4) in (1) .)
(set (KF-next QK zerro-cross_w_x-xhatmin_1 zerro-cross_w_x-xhatmin_2) in (1) .)
(set (E+ E*left E*right) in (1) .)
(set (lemma-2 lemma-16) in (1) .)
(rwr (1) .)
(idt (1) .)
--------------------------------------------------------------------- */
}
```

Appendix D

```
/* ------------------------------------------------------------------
---  Information Filter Specification-----
--- ------------------------------------------------

---51 axioms---

--- initial equations---

  ops x z v w : MachineInt -> RandomMatrix .
  ops n m k : -> MachineInt .
  ops Phi H :  MachineInt -> Matrix .

  var K : MachineInt .

  eq (r(x(K))) = (n) .         eq (c(x(K))) = (1) .
  eq (r(z(K))) = (m) .         eq (c(z(K))) = (1) .
  eq (r(v(K))) = (m) .         eq (c(v(K))) = (1) .
  eq (r(w(K))) = (n) .         eq (c(w(K))) = (1) .
  eq (r(Phi(K))) = (n) .       eq (c(Phi(K))) = (n) .
  eq (r(H(K))) = (m) .         eq (c(H(K))) = (n) .

[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)) .

---initial suppositions---

ops Q R : MachineInt -> Matrix .

[ax*> | RK]
  eq (R(K)) =  (E| v(K) * mtrans(v(K)) |) .
[ax*> | QK]
  eq (Q(K)) = (E| w(K) * mtrans(w(K)) |) .

var J : MachineInt .

[ax*> | white_noise_w_1]
  eq (E| w(K) |) = (zero(n,1)) .
[ax*> | white_noise_v_1]
  eq (E| v(K) |) = (zero(m,1)) .
[ax*> | white_noise_w_2]
  ceq (E| w(K) * mtrans(w(J)) |) = (zero(n,n)) if (K =/= J) .
[ax*> | white_noise_v_2]
  ceq (E| v(K) * mtrans(v(J)) |) = (zero(m,m)) if (K =/= J) .
[ax*> | zerro_cross_v_w]
  eq (E| w(K) * mtrans(v(J)) |) = (zero(n,m)) .

---a priori and a posteriori estimates---

  ops  xhat xhatmin : MachineInt -> RandomMatrix .
  ops  Pminus P : MachineInt -> Matrix .
  eq (r(xhat(K))) = (n) .     eq (c(xhat(K))) = (1) .
  eq (r(xhatmin(K))) = (n) .  eq (c(xhatmin(K))) = (1) .
[ax*> | Pminus-0 ]
  eq (Pminus(0)) = (E| (x(0) - xhatmin(0)) * mtrans(x(0) - xhatmin(0)) |) .
[ax*> | zerro-cross_v_x-xhatmin_1]
  eq (E| (x(K) - xhatmin(K)) * mtrans(v(K)) |) = (zero(n, m)) .
[ax*> | zerro-cross_v_x-xhatmin_2]
  eq (E| v(K) *  mtrans(x(K) - xhatmin(K)) |) = (zero(m, n)) .
[ax*> | zerro-cross_w_x-xhatmin_1]
  eq (E| ( x(K) - xhat(K) ) * mtrans(w(K)) |) = (zero(n,n)) .
[ax*> | zerro-cross_w_x-xhatmin_2]
  eq (E| w(K) * mtrans( x(K) - xhat(K) ) |) = (zero(n,n)) .

---measurement innovation---

  op zminus : MachineInt -> RandomMatrix .
  eq (r(zminus(K))) = (m) .  eq (c(zminus(K))) = (1) .

---gain---

  op gain  : MachineInt -> Matrix .
  eq (r(gain(K))) = (n) .    eq (c(gain(K))) = (m) .

--- best estimate---

  op y : -> MatrixVar .
  eq (r(y)) = (n) .          eq (c(y)) = (m) .

  ops BestPriorEstimate BestEstimate : MachineInt -> RandomMatrixExp .
  op Estimate : MachineInt MatrixExp -> RandomMatrixExp .

  var G : MatrixExp .

  eq (r(Estimate(K, G))) = (n) .  eq (c(Estimate(K, G))) = (1) .

[ax*> | Estimate]
  eq (Estimate(K, G)) = (BestPriorEstimate(K) + (G * (z(K) - zminus(K)))) .
[ax*> | BestPriorEstimate-0]
  eq (BestPriorEstimate(0)) = (xhatmin(0)) .
[ax*> | BestEstimate]
 ceq (Estimate(K, G)) = (BestEstimate(K))
    if (G minimizes /\ y . (E| (x(K) - Estimate(K, y)) * mtrans(x(K) - Estimate(K, y)) |)) .
[ax*> | BestPriorEstimate]
  eq (BestPriorEstimate(K + 1)) = (Phi(K) * BestEstimate(K)) .

---additional hypothesis---

  mb (minv (((H(K) * E|(x(K) - xhatmin(K)) * mtrans(x(K) - xhatmin(K))|) * mtrans(H(K)))
      + E| v(K) * mtrans(v(K))|)) :  MatrixExp .

--- Specific to INFORMATION FILTER---

  op InvP : MachineInt -> Matrix .

---this is the inverse of P, so P will be the inverse of InvP---
```

```
  mb (minv(InvP(K))) : MatrixExp .

  mb (minv(Pminus(K))) : MatrixExp .
  mb (minv(R(K))) : MatrixExp .
  mb (minv(P(K))) : MatrixExp .

  eq (r(Pminus(K))) = (n) .  eq (c(Pminus(K))) = (n) .
  eq (r(R(K))) = (m) .       eq (c(R(K))) = (m) .
  eq (r(P(K))) = (n) .       eq (c(P(K))) = (n) .
  eq (r(InvP(K))) = (n) .    eq (c(InvP(K))) = (n) .

  mb (minv(minv(Pminus(K)) + ((mtrans(H(K)) * minv(R(K))) * H(K)))) : MatrixExp .
  mb (minv(((H(K) * Pminus(K)) * mtrans(H(K))) + R(K))) : MatrixExp  .



---------------------------------------------------------------------- */

input xhatmin(0), Pminus(0)

/* ----------------------------------------------------------------------
[proof assertion-1-1, first iteration]

(set BestPriorEstimate-0 in (1) .)
(rwr (1) .)
(idt (1) .)
---------------------------------------------------------------------- */

/* ----------------------------------------------------------------------
[proof assertion-1-2, first iteration]

(set Pminus-0 in (1) .)
(rwr (1) .)
(idt (1) .)
---------------------------------------------------------------------- */

for(k,0,n)
{
/* ----------------------------------------------------------------------
[lem*> | assertion-1-1]
  eq (BestPriorEstimate(k)) = (xhatmin(k)) .

[lem*> | assertion-1-2]
  eq (Pminus(k)) = (E| (x(k)  - xhatmin(k)) * mtrans(x(k)  - xhatmin(k)) |) .
---------------------------------------------------------------------- */

//[lem*> | instruction-0-1]
 InvP(k) := minv(Pminus(k)) + ((mtrans(H(k)) * minv(R(k))) * H(k));

//[lem*> | instruction-0-2]
 P(k) := minv(InvP(k));

// [lem*> | instruction-0-3]
  gain(k) := P(k) * (mtrans(H(k)) * minv(R(k))) ;
```

```
/* --------------------------------------------------------------------
[proof assertion-1-3]

 (apply-r  lemma-18 to (1) at (none) with ((RP <- (P(k)) ); (RQ <- (R(k)))) .)
 (set (instruction-0-1 instruction-0-2 instruction-0-3) in (1) .)
 (set idemp-inv in (1) .)
 (set (lemma-19 lemma-20 lemma-21 lemma-34) in (1) .)
 (rwr (1) .)
 (idt (1) .)
-------------------------------------------------------------------- */

/* --------------------------------------------------------------------
[lem*> | assertion-1-3]
 eq (gain(k)) = ((Pminus(k) * mtrans(H(k))) * minv(((H(k) * Pminus(k)) * mtrans(H(k))) + R(k))) .
-------------------------------------------------------------------- */

/* --------------------------------------------------------------------
[proof assertion-1-4]

(lem ((((id(n) - (gain(k) * H(k))) * Pminus(k)) * InvP(k)) is (id(n))) = (true)
         to (1) lbl inv-cond1 .)
(set (instruction-0-1 assertion-1-3) in (1 . 0 . 1 ) .)
(set (id*left id*right minv-def2 minus-def2 misc-is) in (1 . 0 . 1) .)
(set (lemma-22 lemma-23 lemma-24 lemma-25 lemma-26) in (1 . 0 . 1) .)
(rwr (1 . 0 . 1 ) .)
(idt (1 . 0 . 1) .)

(lem ((InvP(k) * ((id(n) - (gain(k) * H(k))) * Pminus(k))) is (id(n))) = (true) to (1)
      lbl inv-cond2 .)
(set (instruction-0-1 assertion-1-3) in (1 . 0 . 2 ) .)
(set (id*left id*right minv-def1 misc-is) in (1 . 0 . 2) .)
(set (lemma-27 lemma-28 lemma-29 lemma-30 lemma-31 lemma-32 lemma-33) in (1 . 0 . 2 ) .)
(rwr (1 . 0 . 2 ) .)
(idt (1 . 0 . 2)  .)

(set  instruction-0-2 in (1) .)
(rwr (1) .)
(set (inv-cond1 inv-cond2) in (1) .)
(apply minv-of-matrix to (1) at (none) with ((RP <- (InvP(k)));
      (RQ <- ((id(n) - (gain(k) * H(k))) * Pminus(k)))) .)
(idt (1) .)
-------------------------------------------------------------------- */

/* --------------------------------------------------------------------
[lem*> | assertion-1-4]
 eq (P(k)) = ((id(n) - (gain(k) * H(k))) * Pminus(k)) .
-------------------------------------------------------------------- */

// [lem*> | instruction-1-1]
zminus(k) := H(k) * xhatmin(k) ;

/* --------------------------------------------------------------------
[proof assertion-2]

(set (instruction-1-1 assertion-1-1 assertion-1-2 assertion-1-3) in (1) .)
(set (Estimate KF-measurement zerro-cross_v_x-xhatmin_1 zerro-cross_v_x-xhatmin_2
      RK) in (1) .)
(set (id*left id*right id-trans distr*mtrans minus-def1 misc-is) in (1) .)
(set (E- E+ E*left E*right mtrans-E) in (1) .)
(set (fun-scalar-mult fun-mult fun-trans fun-def1 fun-def2) in (1) .)
(set (trace+ trace- tracem-def1 tracem-def2 trace-lemma1 trace-lemma1* trace-lemma2
      minimizes) in (1) .)
(set (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) .)
---------------------------------------------------------------------- */

/* ----------------------------------------------------------------------
[lem*> | assertion-2]
  eq (gain(k)
     minimizes
     (/\ y . (E| (x(k) - Estimate(k, y)) *
         mtrans(x(k) - Estimate(k, y)) |)))
     = (true) .
---------------------------------------------------------------------- */

// [lem*> | instruction-3]
  xhat(k) := Estimate(k, gain(k)) ;

/* ----------------------------------------------------------------------
[proof assertion-3]

(set (instruction-3 assertion-2) in (1) .)
(apply BestEstimate to (1) at (none) with (G <- (gain(k))) .)
(idt (1) .)
---------------------------------------------------------------------- */

/* ----------------------------------------------------------------------
[lem*> | assertion-3]
  eq (xhat(k)) = (BestEstimate(k)) .
---------------------------------------------------------------------- */

/* ----------------------------------------------------------------------
[proof assertion-4]

(set (instruction-1-1 instruction-3 assertion-1-1 assertion-1-2 assertion-1-3
      assertion-1-4) in (1) .)
(set (Estimate KF-measurement RK zerro-cross_v_x-xhatmin_1 zerro-cross_v_x-xhatmin_2) in (1) .)
(set (distr*mtrans mtrans-inv distr+mtrans id*left id*right id-trans) in (1) .)
(set (E- E+ E*left E*right mtrans-E) in (1) .)
(set (lemma-1 lemma-2 lemma-3 lemma-4 lemma-5 lemma-6 lemma-7 lemma-10 lemma-12
      lemma-13 lemma-14 lemma-15) in (1) .)
(rwr (1) .)
(idt (1) .)
---------------------------------------------------------------------- */

/* ----------------------------------------------------------------------
[lem*> | assertion-4]
  eq (P(k)) = (E| (x(k) - xhat(k)) * mtrans(x(k) - xhat(k)) |) .
---------------------------------------------------------------------- */
```

```
// [lem*> | instruction-5]
  xhatmin(k + 1) := Phi(k) * xhat(k);

// [lem*> | instruction-6]
  Pminus(k + 1) := ((Phi(k) * P(k)) * mtrans(Phi(k))) + Q(k) ;

/* ----------------------------------------------------------------------
[proof assertion-1-1]

(set (instruction-5 assertion-3 BestPriorEstimate) in (1) .)
(rwr (1) .)
(idt (1) .)
---------------------------------------------------------------------- */

/* ----------------------------------------------------------------------
[proof assertion-1-2]

(set (instruction-5 instruction-6 assertion-4) in (1) .)
(set (KF-next QK zerro-cross_w_x-xhatmin_1 zerro-cross_w_x-xhatmin_2) in (1) .)
(set (E+ E*left E*right) in (1) .)
(set (lemma-2 lemma-16) in (1) .)
(rwr (1) .)
(idt (1) .)
---------------------------------------------------------------------- */
}
```

Appendix E

```
/* -----------------------------------------------------------------
--- Extended Kalman Filter Specification-----
--- ------------------------------------------

--- 57 axioms---

---initial equations---

  ops x z v w : MachineInt -> RandomMatrix .
  op u : MachineInt -> Matrix .
  op f :  RandomMatrix  RandomMatrix ->  RandomMatrixExp? .
  op h :   RandomMatrix ->  RandomMatrixExp? .
  ops n m p k : -> MachineInt .

  var K : MachineInt .
  vars V1 V2 :  RandomMatrix .


  eq (r(x(K))) = (n) .        eq (c(x(K))) = (1) .
  eq (r(z(K))) = (m) .        eq (c(z(K))) = (1) .
  eq (r(v(K))) = (m) .        eq (c(v(K))) = (1) .
  eq (r(w(K))) = (n) .        eq (c(w(K))) = (1) .
  eq (r(u(K))) = (p) .        eq (c(u(K))) = (1) .

  cmb (f(V1, V2)) :  RandomMatrix if (r(V1) == n and r(V2) == p and c(V1) == 1
                                          and c(V2) == 1)  .
  cmb (h(V1)) :  RandomMatrix if (r(V1) == n and  c(V1) ==  1) .

  ceq (r(f(V1, V2))) = (n) if (f(V1, V2) :  RandomMatrix) .
  ceq (c(f(V1, V2))) = (1) if (f(V1, V2) : RandomMatrix) .
  ceq (r(h(V1))) = (m) if (h(V1) :  RandomMatrix) .
  ceq (c(h(V1))) = (1) if (h(V1) :  RandomMatrix) .

[ax*> | EKF-next]
  eq (x(K + 1)) = (f(x(K), u(K)) + w(K)) .
[ax*> | EKF-measurement]
  eq (z(K)) = (h(x(K))  + v(K)) .

---initial suppositions---

  ops Q R : MachineInt -> Matrix .

[ax*> | RK]
  eq (R(K)) =  (E| v(K) * mtrans(v(K)) |) .
[ax*> | QK]
  eq (Q(K)) = (E| w(K) * mtrans(w(K)) |) .

  var J : MachineInt .

[ax*> | white_noise_w_1]
  eq (E| w(K) |) = (zero(n,1)) .
[ax*> | white_noise_v_1]
```

```
  eq (E| v(K) |) = (zero(m,1)) .
[ax*> | white_noise_w_2]
  ceq (E| w(K) * mtrans(w(J)) |) = (zero(n,n)) if (K =/= J) .
[ax*> | white_noise_v_2]
  ceq (E| v(K) * mtrans(v(J)) |) = (zero(m,m)) if (K =/= J) .
[ax*> | zerro_cross_v_w]
  eq (E| w(K) * mtrans(v(J)) |) = (zero(n,m)) .

---a priori and a posteriori estimates---

  ops  xhat xhatmin : MachineInt -> RandomMatrix .
  ops  Pminus P : MachineInt -> Matrix .
  eq (r(xhat(K))) = (n) .     eq (c(xhat(K))) = (1) .
  eq (r(xhatmin(K))) = (n) .  eq (c(xhatmin(K))) = (1) .

[ax*> | Pminus-0 ]
  eq (Pminus(0)) = (E| (x(0) - xhatmin(0)) * mtrans(x(0) - xhatmin(0)) |) .
[ax*> | zerro-cross_v_x-xhatmin_1]
  eq (E| (x(K) - xhatmin(K)) * mtrans(v(K)) |) = (zero(n, m)) .
[ax*> | zerro-cross_v_x-xhatmin_2]
  eq (E| v(K) *  mtrans(x(K) - xhatmin(K)) |) = (zero(m, n)) .
[ax*> | zerro-cross_w_x-xhatmin_1]
  eq (E| ( x(K) - xhat(K) ) * mtrans(w(K)) |) = (zero(n,n)) .
[ax*> | zerro-cross_w_x-xhatmin_2]
  eq (E| w(K) * mtrans( x(K) - xhat(K) ) |) = (zero(n,n)) .

---measurement innovation---

  op zminus : MachineInt ->  RandomMatrix .
  eq (r(zminus(K))) = (m) .  eq (c(zminus(K))) = (1) .

---gain---

  op gain  : MachineInt -> Matrix .
  eq (r(gain(K))) = (n) .     eq (c(gain(K))) = (m) .

---Phi+H+Jacobians---

  ops Phi H  :  MachineInt -> Matrix .
  op  Jacob-f :  RandomMatrix  RandomMatrix -> MatrixExp .
  op  Jacob-h :  RandomMatrix ->  MatrixExp .

  var V10 : RandomMatrix .

  cmb (Jacob-f(V1, V2)) :  Matrix if (f(V1, V2) :  RandomMatrix) .
  cmb (Jacob-h(V1)) :  Matrix if (h(V1) :  RandomMatrix ) .
  ceq (r(Jacob-f(V1, V2))) =(r(f(V1, V2))) if (Jacob-f(V1, V2) :  Matrix) .
  ceq (c(Jacob-f(V1, V2))) =(r(V1)) if (Jacob-f(V1, V2) :  Matrix) .
  ceq (r(Jacob-h(V1))) = (r(h(V1))) if (Jacob-h(V1) : Matrix) .
  ceq (c(Jacob-h(V1))) = (r(V1)) if (Jacob-h(V1) : RandomMatrix) .

[ ax*> | Jacob-f]
   eq  (f(V1, V2) ) = (f(V10, V2) + (Jacob-f(V10, V2) * (V1 - V10)) ).
[ ax*> | Jacob-h]
   eq  (h(V1)) = (h(V10) + (Jacob-h(V10) * (V1 - V10))) .
```

```
---best estimate---

  op y : -> MatrixVar .
  eq (r(y)) = (n) .           eq (c(y)) = (m) .

  ops BestPriorEstimate BestEstimate : MachineInt -> RandomMatrixExp .
  op Estimate : MachineInt MatrixExp -> RandomMatrixExp .

  var G : MatrixExp .

  eq (r(Estimate(K, G))) = (n) .  eq (c(Estimate(K, G))) = (1) .

[ax*> | Estimate]
  eq (Estimate(K, G)) = (BestPriorEstimate(K) + (G * (z(K) - zminus(K)))) .
[ax*> | BestPriorEstimate-0]
  eq (BestPriorEstimate(0)) = (xhatmin(0)) .
[ax*> | BestEstimate]
 ceq (Estimate(K, G)) = (BestEstimate(K))
    if  (G minimizes /\ y . (E| (x(K) - Estimate(K, y)) * mtrans(x(K) - Estimate(K, y)) |)) .
[ax*> | BestPriorEstimate]
  eq (BestPriorEstimate(K + 1)) = (f(BestEstimate(K), u(K))) .

---additional hypotheses---

  mb  (minv(((Jacob-h(xhatmin(K)) * E| (x(K) - xhatmin(K)) * mtrans (x(K) - xhatmin(K) ) |) *
       mtrans(Jacob-h(xhatmin(K)))) + E| v(K) * mtrans(v(K)) |)) :  MatrixExp .

[ax*> | Jacob-h-x]
  eq  (h(x(K))) = (h(xhatmin(K)) + (Jacob-h(xhatmin(K)) * (x(K) - xhatmin(K)))) .
[ax*> | Jacob-f-x]
  eq  (f(x(K), u(K)) ) = (f(xhat(K), u(K)) + (Jacob-f(xhat(K), u(K)) * (x(K) - xhat(K))) ).

--------------------------------------------------------------------- */
input xhatmin(0), Pminus(0)

/* ---------------------------------------------------------------------
[proof assertion-1-1, first iteration]

(set BestPriorEstimate-0 in (1) .)
(rwr (1) .)
(idt (1) .)
--------------------------------------------------------------------- */

/* ---------------------------------------------------------------------
[proof assertion-1-2, first iteration]

(set Pminus-0 in (1) .)
(rwr (1) .)
(idt (1) .)
--------------------------------------------------------------------- */

for(k,0,n)
{
/* ---------------------------------------------------------------------
[lem*> | assertion-1-1]
  eq (BestPriorEstimate(k)) = (xhatmin(k)) .
```

```
[lem*> | assertion-1-2]
  eq (Pminus(k)) = (E| (x(k)  - xhatmin(k)) * mtrans(x(k)  - xhatmin(k)) |) .
---------------------------------------------------------------------- */

// [lem*> | instruction-1-1]
zminus(k) := h(xhatmin(k)) ;

// [lem*> | instruction-1-2]
H(k) := Jacob-h(xhatmin(k));

/* ----------------------------------------------------------------------
[proof assertion-1-3]

(set (instruction-1-1 EKF-measurement Jacob-h-x) in (1) .)
(rwr (1) .)
(idt (1) .)
---------------------------------------------------------------------- */

/* ----------------------------------------------------------------------
[lem*> | assertion-1-3]
  eq (z(k)) = ((zminus(k) + (Jacob-h(xhatmin(k)) * (x(k) - xhatmin(k )))) + v(k)) .
---------------------------------------------------------------------- */

// [lem*> | instruction-2]
  gain(k) := (Pminus(k) * mtrans(H(k))) * minv(((H(k) * Pminus(k)) * mtrans(H(k))) + R(k)) ;

/* ----------------------------------------------------------------------

[proof assertion-2]

(set (instruction-1-1 instruction-1-2 instruction-2 assertion-1-1 assertion-1-2) in (1) .)
(set (Estimate EKF-measurement zerro-cross_v_x-xhatmin_1 zerro-cross_v_x-xhatmin_2 RK
      Jacob-h-x) in (1) .)
(set (id*left id*right id-trans distr*mtrans minus-def1 misc-is) in (1) .)
(set (E- E+ E*left E*right mtrans-E) in (1) .)
(set (fun-scalar-mult fun-mult fun-trans fun-def1 fun-def2) in (1) .)
(set (trace+ trace- tracem-def1  tracem-def2 trace-lemma1 trace-lemma1* trace-lemma2
      minimizes) in (1) .)
(set (lemma-1 lemma-2 lemma-3 lemma-4 lemma-5 lemma-6 lemma-7 lemma-8 lemma-9 lemma-10
      lemma-11 lemma-17) in (1) .)
(rwr (1) .)
(idt (1) .)
---------------------------------------------------------------------- */

/* ----------------------------------------------------------------------
[lem*> | assertion-2]
  eq (gain(k)
     minimizes
     (/\ y . (E| (x(k) - Estimate(k, y)) *
         mtrans(x(k) - Estimate(k, y)) |)))
     = (true) .
---------------------------------------------------------------------- */

// [lem*> | instruction-3]
  xhat(k) := Estimate(k, gain(k)) ;
```

```
/* ------------------------------------------------------------------
[proof assertion-3]

(set (instruction-3 assertion-2 ) in (1) .)
(apply BestEstimate to (1) at (none) with (G <- (gain(k))) .)
(idt (1) .)
--------------------------------------------------------------- */

/* ------------------------------------------------------------------
[lem*> | assertion-3]
  eq (xhat(k)) = (BestEstimate (k)) .
--------------------------------------------------------------- */

// [lem*> | instruction-4]
  P(k) := (id(n) - (gain(k) * H(k))) * Pminus(k);

/* ------------------------------------------------------------------
[proof assertion-4]

(set (instruction-1-1 instruction-1-2 instruction-2 instruction-3 instruction-4
      assertion-1-1 assertion-1-2) in (1) .)
(set (Estimate EKF-measurement RK zerro-cross_v_x-xhatmin_1 zerro-cross_v_x-xhatmin_2
      Jacob-h-x) in (1) .)
(set (distr*mtrans mtrans-inv distr+mtrans id*left id*right id-trans) in (1) .)
(set (E- E+ E*left E*right mtrans-E) in (1) .)
(set (lemma-1 lemma-2 lemma-3 lemma-4 lemma-5 lemma-6 lemma-7 lemma-10 lemma-12
      lemma-13 lemma-14 lemma-15 lemma-17) in (1) .)
(rwr (1) .)
(idt (1) .)
--------------------------------------------------------------- */

/* ------------------------------------------------------------------
[lem*> | assertion-4]
  eq (P(k)) = (E| (x(k) - xhat(k)) * mtrans(x(k) - xhat(k)) |) .
--------------------------------------------------------------- */

// [lem*> | instruction-5]
  xhatmin(k + 1) := f(xhat(k), u(k));

// [lem*> | instruction-6-0]
 Phi(k) := Jacob-f(xhat(k), u(k)) ;

/* ------------------------------------------------------------------
[proof assertion-5]

(set (instruction-5 instruction-6-0 EKF-next Jacob-f-x) in (1) .)
(rwr (1) .)
(idt (1) .)
--------------------------------------------------------------- */

/* ------------------------------------------------------------------
[lem*> | assertion-5]
  eq (x(k + 1)) = ((xhatmin(k + 1) + (Jacob-f(xhat(k), u(k)) * (x(k) - xhat(k)))) + w(k)) .
--------------------------------------------------------------- */
```

```
// [lem*> | instruction-6]
  Pminus(k + 1) := ((Phi(k) * P(k)) * mtrans(Phi(k))) + Q(k) ;

/* ----------------------------------------------------------------------
[proof assertion-1-1]

(set (instruction-5 assertion-3 BestPriorEstimate) in (1) .)
(rwr (1) .)
(idt (1) .)
---------------------------------------------------------------------- */

/* ----------------------------------------------------------------------
[proof assertion-1-2]

(set (instruction-5 instruction-6-0 instruction-6 assertion-4 assertion-5) in (1) .)
(set (EKF-next QK zerro-cross_w_x-xhatmin_1 zerro-cross_w_x-xhatmin_2 Jacob-f-x) in (1) .)
(set (E+ E*left E*right) in (1) .)
(set (lemma-2 lemma-16 lemma-17) in (1) .)
(rwr (1) .)
(idt (1) .)
---------------------------------------------------------------------- */
}
```