Towards Certifying Domain-Specific Properties of Synthesized Code – Extended Abstract –

Grigore Roşu
NASA Ames Research Center - USRA/RIACS
grosu@ptolemy.arc.nasa.gov

Jon Whittle
NASA Ames Research Center - QSS Group Inc
jonathw@ptolemy.arc.nasa.gov

Abstract

We present a technique for certifying domain-specific properties of code generated using program synthesis technology. Program synthesis is a maturing technology that generates code from high-level specifications in particular domains. For acceptance in safety-critical applications, the generated code must be thoroughly tested which is a costly process. We show how the program synthesis system AUTOFILTER *can be extended to generate not only code but also proofs that properties hold in the code. This technique has the potential to reduce the costs of testing generated code.*

1. Introduction

Program synthesis systems generating fully executable code from high level specifications are rapidly maturing (see, for example, [9, 8]), in some cases to the point of commercialization (e.g., SciNapse [1]). However, the use of such systems is limited by concerns about the correctness of the generated code. If program synthesis systems can be augmented to generate correctness guarantees as well as the code, then some of the testing costs (typically 80% of development for mission critical systems) can be avoided. There are two ways to provide these correctness guarantees. The first is to verify the program synthesis system itself. However, these systems tend to be highly complex and dynamic, so such an approach is not viable. In this paper, we follow an alternative approach of generating correctness proofs along with the code. This requires extending the synthesis system to generate proofs for the programs generated but these programs are much simpler than the synthesis system that generates them and hence the proofs are also simpler.

We focus on certifying crucial domain-specific properties in safety or mission critical domains; a safety policy certifier was presented in [6], for the domain of coordinate frames which is crucial to astronomical navigation. Many software products are developed for complex domains that involve a significant body of mathematical knowledge. 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 complex domains such as the one presented next. Therefore, user intervention is often needed to insert domain-specific knowledge into the programs to be certified, usually under the form of code annotations. In our approach, the domain-specific knowledge can be inserted automatically by the program synthesis system. These annotations are in the form of model specifications, assertions and proof scripts.

2. Domain-Specific Program Synthesis

AUTOFILTER is a program synthesizer for state estimation problems. By state estimation, we mean estimating the state of an object (e.g., its position, attitude or noise characteristics) based on noisy sensor measurements. The most common way of solving a state estimation problem is to use a recursive update algorithm known as the Kalman Filter [2] which provides a statistically optimal estimate of a state based on noisy sensor measurements. A Kalman Filter requires additional information to make this estimate, namely a model of the dynamics of the problem and a model of how the sensor measurements relate to the state, such as:

$$x_{k+1} = \Phi_k x_k + w_k \quad (1)$$

$$z_k = H_k x_k + v_k \quad (2)$$

$$E[w_k] = E[v_k] = 0 \quad (3)$$

$$E[w_k w_i^\top] = \delta(k-i) Q_k \quad (4)$$

$$E[v_k v_i^\top] = \delta(k-i) R_k \quad (5)$$

x_k is a vector of state variables at time k. In a typical attitude estimation problem, for example, the state vector, x_k, might contain three variables representing rotation angles of a spacecraft. This is what the Kalman Filter will estimate. Equation (1) is the process model which describes the dynamics of the 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 often cannot be measured directly. The measurement vector, z_k, is related to the state by matrix H_k. v_k is the noise in this relationship. Simplifying Kalman Filter assumptions state that all noises must be gaussian processes with zero mean and there must be no correlation between the noise over time (see (4) and (5) where $a^\top$ is the transpose of vector a and $\delta(j)$ evaluates to 1 when $j = 0$ and 0 otherwise. Q_k and R_k are matrices which represent the noise characteristics of the process model noise and measurement model noise, respectively).

Given models of this form, a Kalman Filter can be implemented that optimally estimates x_k. A schematic algorithm for this Kalman Filter is given in Figure 1. The estimate, $\hat{x}_k$, can be proved to be optimal, in the sense that the mean squared error (also known as the error covariance matrix), $E[(x_k - \hat{x}_k)(x_k - \hat{x}_k)^\top]$, is minimized (see the next section). Any Kalman Filter should satisfy this minimization property. In Figure 1, P_k is the error covariance matrix and is updated on each iteration of the filter. P_k gives an indication of the error in the filter estimates and so is used as a check whether or not the filter is converging.

As an example of how Kalman Filters work 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 startracker. In this formulation, the process equation (1) would model how the gyroscopes degrade and the equation (2) would model the relationship between the startracker measurements and the three rotation angles that form the state. From these models, a Kalman Filter implementation would produce an optimal estimate of the current attitude, where the uncertainties in the problem (gyro degradation, startracker noise, etc.) have been minimized.

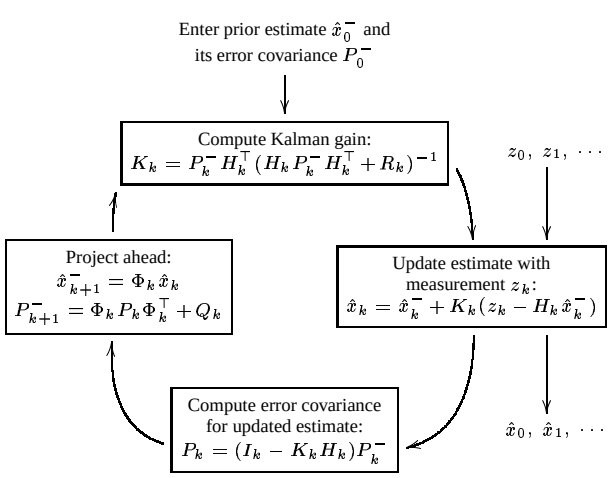

Figure 1. Kalman Filter Loop

AUTOFILTER takes as input a mathematical specification including equations (1) - (5) but also descriptions of the noise characteristics and filter parameters. From this specification, it generates code that implements (some variant of) the algorithm in Figure 1. In fact, AUTOFILTER generates code in our own intermediate language which is then translated into C++ or Matlab. In this paper, we only consider code in the intermediate language. It should be noted that Figure 1 represents just one of many possible variations and configurations of the filter. In fact, the abundance of variations is what makes this domain ideal for synthesis.

3. An Informal Optimality Proof

In this paper, we describe techniques for certifying domain-specific properties of code generated by AUTOFILTER. In particular, we consider the optimality proof introduced in the previous section — of the minimization of the mean squared error. Despite the apparent simplicity of the code in Figure 2 that AUTOFILTER generates, the proof of optimality is quite complex. The main task is to show that `xhat(k)` (corresponding to $\hat{x}_k$ in the previous section) is the best estimate, under simplifying assumptions, of the state vector x_k at time k. This is a standard proof in state estimation and is usually presented in books as an informal mathematical proof several pages long. We next sketch the proof, emphasizing those aspects which are particularly relevant for automating the proof, especially the *assumptions*.

The very first assumption made in all books is that the initial estimates, $\hat{x}_0^-$ and P_0^-, are the best prior estimate and its error covariance, i.e., $E[(x_0 - \hat{x}_0^-)(x_0 - \hat{x}_0^-)^\top]$, respectively. At a given time k, if $\hat{x}_k^-$ is the best prior estimate then one can use (2) to conclude that the most probable measurement error is $z_k - H_k\hat{x}_k^-$. Another assumption is that "the best estimate is a linear combination of the best prior estimate and the measurement error". Formally, this says that $\hat{x}_k$ is somewhere in the image of the function $\hat{x}_k(y) := \lambda y.(\hat{x}_k^- + y * (z_k - H_k\hat{x}_k^-))$, where the coefficient y is a matrix having as many rows as $\hat{x}_k^-$ and as many columns as rows z_k. We are looking for the y corresponding to the minimum error covariance matrix, $P_k(y) := E[(x_k - \hat{x}_k(y))(x_k - \hat{x}_k(y))^\top]$, that is, the solution of the derivative of $P_k(y)$ with respect to y. In fact, 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 giving the minimum of $P_k(y)$ is the solution of the equation $d(trace(P_k(y)))/dy = 0$, where the trace of a matrix is the sum of the elements on its first diagonal and for a (standard) function $f(y_{11}, y_{12}, ...)$ on the elements of a matrix y, its derivative df/dy is the matrix $(df/dy_{ij} \mid y_{ij} \in y)$ having the same dimensions as y. Assuming that P_k^- is the error covariance of the best *prior* estimate of $\hat{x}_k$, that is, $E[(x_k - \hat{x}_k^-)(x_k - \hat{x}_k^-)^\top]$, then after thousands of basic proof steps one gets the solution $K_k := P_k^- H_k^\top (H_k P_k^- H_k^\top + R_k)^{-1}$, which is what

```
1.  input xhatmin(0), pminus(0);
2.  for(k,0,n) {
3.    gain(k) := pminus(k) * mtrans(h(k)) * minv(h(k) * pminus(k) * mtrans(h(k)) + r(k));
4.    xhat(k) := xhatmin(k) + (gain(k) * (z(k) - (h(k) * xhatmin(k)))));
5.    p(k) := (id(n) - gain(k) * h(k)) * pminus(k);
6.    xhatmin(k + 1) := phi(k) * xhat(k);
7.    pminus(k + 1)  := phi(k) * (p(k) * mtrans(phi(k))) + q(k); }
```

Figure 2. Kalman Filter code calculating the best estimate incrementally.

line 3 in Figure 2 calculates. One can also calculate the best estimate now, namely $\hat{x}_k(K)$ (line 4) and also the error covariance matrix of the best estimate, $P_k(K)$ (line 5).

In order to complete the proof, one needs to show that $\hat{x}^-_{k+1}$ and P^-_{k+1} are the best prior estimate and its error covariance matrix at time $k+1$, respectively. The former follows from another accepted assumption 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 noise is *ignored*; the intuition for this assumption is that the current best estimate is random anyway, so the noise with mean 0 can be ignored. We do this in line 6. The latter can be also obtained by calculations, also taking several thousand basic proof steps, transforming the expression $P^-_{k+1} = E[(x_{k+1} - \hat{x}^-_{k+1})(x_{k+1} - \hat{x}^-_{k+1})^\top]$ by replacing $\hat{x}_{k+1}$ as in equation (1) and $\hat{x}^-_{k+1}$ as in line 6.

4. A Framework for Formalizing the Proof

To generate and automatically certify proofs such as the above, we need to formalize the domain knowledge, which includes matrices, functions on matrices, and differentiation. The formalization was done using the executable specification language Maude [4]. Maude implements both rewriting logic and membership equational logic (MEL), 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.

4.1. Matrices

Most of the operations and axioms/lemmas in matrix theory are *partial*. For example, multiplication is defined iff the number of columns of the first matrix equals the number of rows of the second, and the commutativity and associativity of addition hold true iff the matrices involved have the same dimensions. Maude provides support for partiality; the partial infix operation of multiplication and the total transpose operation are defined as follows:

```
op _*_ : Matrix Matrix -> [Matrix]
op mtrans : Matrix -> Matrix
```

In order to define their semantics and properties, we need two (total) operations that give the numbers of columns and rows of a matrix, `c` and `r`, of arity `Matrix -> Nat`. Now we can express definedness of multiplication together with appropriate equations for the new columns and rows:

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

Axioms relating various operators on matrices such as

```
ceq mtrans(P*Q)=mtrans(Q)*mtrans(P) if P*Q : Matrix
```

are also needed, together with more than 50 others, most of them conditional and involving memberships.

4.2. Functions on Matrices

One step in optimality proofs is stating that the best estimate of the actual state is a linear combination of the best prior estimate and the measurement error. The coefficient of this linear dependency is calculated such that the error covariance matrix 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 $\lambda y.(\langle \textit{prior estimate}\rangle + y * \langle \textit{measurement error}\rangle)$. For this function to be well defined, y must be a matrix having appropriate dimensions as in Section 3. Hence, we need to formally define functions on matrices together with 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 Matrix -> MatrixFun
op _ _ : MatrixFun Matrix -> [Matrix]
```

Appropriate (conditional) axioms for functions are specified, also taking into account partiality, such as:

```
ceq (/\y.(P+Q))(X)=(/\y.P)(X)+(/\y.Q)(X) if P+Q : Matrix
ceq (/\y.y)(X) = X if c(X) == c(y) and r(X) == r(y)
```

among many others.

4.3. Differentiation

If P is a square matrix then *trace*(P) is the sum of all P's elements on the main diagonal. 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. For example, the only property used so far linking optimality 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, to avoid going into deep axiomatizability of mathematics, we have just defined a "derived" operation

```
op d(trace_)/d_ : MatrixFun MatrixVar -> MatrixFun
```

giving directly the derivative of the trace of a function, and have declared some properties of it, such as the equation

```
ceq d(trace(/\Y.(Y*P)))/d(Y) = /\Y.mtrans(P)
   if Y*P : Matrix and r(Y) == c(P)
```

stating that the derivative of $\lambda y.(y * P)$ is $\lambda y.P^{\top}$ whenever $y * P$ is a well-defined square matrix. One could, of course, prove this property from more basic properties of traces, functions and differentiations, but one would need to add a significant body of mathematical knowledge to the system.

5. A Formal Optimality Proof

We next explain how we formalized the informal proof in Section 3, using the axiomatization of the abstract domain in Section 4. This formalization was done manually, using an interactive theorem prover, ITP [3], implemented in Maude. ITP provides support for automatic proofs or proof hints can be given, such as for example (`apply -distr to 1.2 at 2.3`) which says that the distributivity axiom should be applied backwards to proof task number `1.2` at position `2.3`. Simplifications are automatically done after each hint; we used hundreds of hints in the next proofs.

5.1. Specifying the Statistical Model

To reason about the code in Figure 2, one must know where the matrices `z`, `h`, etc., come from and what is their meaning, or in other words, one needs the *specification* of this particular Kalman Filter together with all its *assumptions*. These are needed *in addition* to the abstract domain knowledge in Section 4. Hence, the very first step is to expand the abstract domain with this Kalman Filter's specification, that we denote SPEC_{KF}, declaring all the matrices/vectors involved and their dimensions, such as

```
ops x phi w : MachineInt -> Matrix
ops z h v : MachineInt -> Matrix
ops r q : MachineInt -> Matrix
ops n m k : -> MachineInt
var K : MachineInt
eq r(x(K)) = n .    eq c(x(K)) = 1
eq r(phi(K)) = n .  eq c(phi(K)) = n
eq r(z(K)) = m .    eq c(z(K)) = 1
...
```

as well as model equations/assumptions, such as

```
eq x(K + 1) = (phi(K) * x(K)) + w(K)
eq z(K) = (h(K) * x(K)) + v(K)
eq r(K) = E[v(K) * mtrans(v(K))]
eq q(K) = E[w(K) * mtrans(w(K))]
...
```

Other axioms/assumptions not formalized here 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`.

This specification has about 35 axioms/assumptions and the interested reader can get it from the authors. A major advantage of our approach to combine synthesis and certification is that specifications can be generated *automatically* from the problem description input to the synthesis engine.

5.2. Modularizing the Proof

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 above 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. The resulting annotated code is shown in Figure 3, where we replaced the more formal (and longer) assertions by English. The proof assertions in Figure 3 should be read as follows: proof assertion n is a proof of assertion n in its current environment.

The best we can assert between instructions 1 and 2 is that `xhatmin(0)` and `pminus(0)` are initially the best prior estimate and error covariance matrix, respectively. This assertion is an assumption in SPEC_{KF}.

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 3 is that `gain(k)` minimizes the covariance matrix of the error between the real (unknown) state of the system and a linear combination of the best prior estimate and our current measurement error. This formal assertion is rather technical and takes a few lines of Maude code, so we do not show it here. It was, however, the most difficult part of the proof. Its proof script contains 7 lemmas and it has 142 steps, that is, there are 142 uses of labeled sentences. This means that there are at least 142 places where an automatic equational theorem prover would try both directions of an equation — 2^{142} combinations. The assertion between lines 4 and 5 says that `xhat(k)` is the best estimate of the actual state and follows now immediately from the previous assertion. After line 5, however, we have the assertion that `p(k)` is the error covariance matrix of the best estimate and its proof needs 4 lemmas and has about 110 proof script steps. After line 6, due to an assumption in SPEC_{KF}, we can assert and easily show that `xhatmin(k+1)` is the best prior estimate at time `k+1`, which together with the instruction on line 7 implies the assertion between lines 2 and 3, so we have completed our proof of optimality of the code in Figure 2 by induction. It took ITP a bit more than 30 seconds to check all this proof, which makes us predict at least 100,000 axiom applications.

6. Synthesizing Annotated Kalman Filters

AUTOFILTER synthesizes code by exhaustive, layered application of *schemas*. A schema is a program template with open slots and a set of applicability conditions. The

```
  /* Specification of the state estimation problem ... about 35 axioms/assumptions in Maude */
1.  input xhatmin(0), pminus(0);
  /* 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 its error covariance matrix */
3.    gain(k) := pminus(k) * mtrans(h(k)) * minv(h(k) * pminus(k) * mtrans(h(k)) + r(k));
  /* Proof assertion 3: (142 ITP hints including those below)
         (lem (l(pminus(k))) = (n) to (1) .)
         (apply assertion-1-2 to (1 . 0 . 1) at (1) .)
         ... the 138 other hints are omitted ...
         (apply  pminuskmtrans to (1) at (1 . 2 . 1 . 1 . 1) .)
         (apply  comm+ to (1) at (1 . 2) .) */
  /* Assertion 3: (in English)
         gain(k) minimizes the error covariance matrix */
4.    xhat(k) := xhatmin(k) + (gain(k) * (z(k) - (h(k) * xhatmin(k))));
  /* Proof assertion 4: (... omitted) */
  /* Assertion 4: (the main goal)
         xhat(k) is the best estimate */
5.    p(k) := (id(n) - gain(k) * h(k)) * pminus(k);
  /* Proof assertion 5: (... omitted; 110 ITP hints) */
  /* Assertion 5:
         p(k) error covariance matrix of xhat(k) */
6.    xhatmin(k + 1) := phi(k) * xhat(k);
  /* Proof assertion 6: (... omitted) */
  /* Assertion 6:
         xhatmin(k + 1) best prior estimate at time k + 1 */
7.    pminus(k + 1)  := phi(k) * (p(k) * mtrans(phi(k))) + q(k);
  /* Proof assertion 2: (... omitted; 31 ITP hints) */ }
```

Figure 3. Annotated Kalman Filter code calculating the best estimate.

slots are filled in with code fragments by the synthesis system calling the schemas recursively. The conditions constrain how the slots can be filled — they must be proven to hold in the given specification before the schema can be applied. Some of the schemas contain calls to symbolic equation solvers, others contain entire skeletons of statistical or numerical algorithms. By recursively invoking schemas and composing the resulting code fragments, AUTOFILTER is able to automatically synthesize programs of considerable size and internal complexity.

Figure 4 gives an abstraction of a top-level schema for generating Kalman Filter code. $name(\%)$ are schema slots. They are filled in by an assignment of the form $\%name :=$ .... In some cases, the top-level schema will fill slots directly. In other cases, the slots are filled by recursively invoking other schemas. The numbers in square parentheses refer to line numbers in Figure 2.

For certification purposes, the schema must also generate a *proof* that the schema is correct — in this case, the optimality proof in Figure 3. In order to generate this proof, the schemas are extended to generate also the assertions and proof assertions from Figure 3. In this way, each line of code generated comes optionally with assertions or proof assertions. Currently, the proof assertions are ITP proof scripts, i.e., a sequence of applications of axioms/lemmas (along with variable substitutions). These proofs typically are very complex and involve the exploration of a large search space. The hints in the high-level proof scripts always occur at a choice point in the proof. Hence, given the proof script, it is possible to reconstruct the entire proof without the need for any search. The reconstruction of the

```
/* applicability conditions */
 ... process noise is Gaussian
 ... measurement noise is Gaussian
 ... process/measurement noise are independent
                ...
/* set up template */
result := kalman(local(%), initialize(%), loop(%),
                 postloop(%))),
                ...
%loop := for(pvar,0,n,                      // [2]
             update(zupdate(%), phiupdate(%), hupdate(%),
                    gain(%),                // [3]
                    estimateUpdate(%),      // [4]
                    covarUpdate(%),         // [5]
                    storeOutput(%),
                    propagateEstimate(%), // [6]
                    propagateCovar(%)))     // [7]
/* fill in some slots */
                ...
/* recursively invoke schemas to fill
   remaining slots */
                ...
```

Figure 4. (Part of) a Kalman Filter schema

entire proof is currently done in the certifier, but it could just as easily have been given explicitly in the schema.

7. 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 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 longer. 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.

The state estimation certifier works as follows. It first extends the abstract domain with the specification of the program (extracted from the beginning of the 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 filter are correctly specified, then for any annotated program as above our certifier returns "yes" iff 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.

8. Conclusions and Future Work

In this paper, we have shown how to extend program synthesis systems to generate not only code but also proofs of properties of that code. This work was carried out in the context of AUTOFILTER, a synthesizer of state estimation programs, which was augmented to output a proof that the code implements an optimal estimator. This proof is a highly complex proof that cannot be proved automatically but by encoding the key steps of the proof in AUTOFILTER's knowledge base, it is able to generate a proof that can easily be checked by an independent certifier. Such results will encourage the acceptance of code generators in safety-critical domains since the generators will produce not only the code but also certificates that the code is correct.

Our certification approach requires more sophisticated reasoning than in *proof-carrying code (PCC)* [7], since our abstract domains admit complex axiomatizations and verifying the safety of each line of code may need tens of thousands of inference steps. *Extended Static Checker (ESC)* [5] finds errors at compile time, such as array index bounds errors and nil dereferences. The use of ESC is limited to language definable types and properties that can be proved automatically. By allowing proof scripts, we extend the usability of our certifiers to whatever provable properties. However, domain-specific proofs can be very complex, so, even if possible in theory, we do not anticipate that our certifiers will be used independently from synthesis engines.

The certifier used in the work described 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.

References

[1] R. Akers, E. Kant, C. Randall, S. Steinberg, and R. Young. Scinapse: A problem-solving environment for partial differential equations. *IEEE Comp. Sci. and Eng.*, 4:32–42, 1997.

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

[3] M. Clavel. The ITP tool. In A. N. et al., editor, *Logic, Language and Information. Proceedings of the First Workshop on Logic and Language*, pages 55–62. Kronos, 2001.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. `http:/maude.csl.sri.com`.

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

[6] M. Lowry, T. Pressburger, and G. Roşu. Certifying domain-specific policies. In *Automated Software Engineering (ASE'01)*, pages 81–90. IEEE, 2001.

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

[8] Y. V. Srinivas and R. Jüllig. Specware: Formal support for composing software. In *Mathematics of Program Construction (MPC'95)*, volume 947 of *LNCS*. Springer, 1995.

[9] 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 *Automated Software Engineering (ASE'01)*, IEEE, 2001.