

# Flexible Generation of Kalman Filter Code

Julian Richardson (1), Edward Wilson (2)  
(1) RIACS, NASA Ames Research Center, julianr@riacs.edu  
(2) Intellization, ed.wilson@intellization.com

*Abstract*—Domain-specific program synthesis can automatically generate high quality code in complex domains from succinct specifications, but the range of programs which can be generated by a given synthesis system is typically narrow.

Obtaining code which falls outside this narrow scope necessitates either 1) extension of the code generator, which is usually very expensive, or 2) manual modification of the generated code, which is often difficult and which must be redone whenever changes are made to the program specification.

In this paper, we describe adaptations and extensions of the AUTOFILTER Kalman filter synthesis system which greatly extend the range of programs which can be generated. Users augment the input specification with a specification of code fragments and how those fragments should interleave with or replace parts of the synthesized filter. This allows users to generate a much wider range of programs without their needing to modify the synthesis system or edit generated code.

We demonstrate the usefulness of the approach by applying it to the synthesis of a complex state estimator which combines code from several Kalman filters with user-specified code. The work described in this paper allows the complex design decisions necessary for real-world applications to be reflected in the synthesized code. When executed on simulated input data, the generated state estimator was found to produce comparable estimates to those produced by a hand-coded estimator.

## TABLE OF CONTENTS

- 1 INTRODUCTION
- 2 AUTOFILTER SYNTHESIS OF KALMAN FILTERS
- 3 VARIABILITY IN SYNTHESIS
- 4 BARRIERS TO SYNTHESIS IN PRACTICE
- 5 GENERATING COMPONENTS
- 6 IMPERATIVE SPECIFICATION LANGUAGE
- 7 EXAMPLE
- 8 FURTHER WORK
- 9 CONCLUSIONS

## 1. INTRODUCTION

Domain-specific program synthesis can automatically generate high quality code in complex domains from succinct specifications. An experiment reported in [7] reported that performance of individuals using a domain-specific code generator was three times faster, with half as many errors, as implementing code for the same tasks by manually editing templates. Similarly, a survey of domain-specific languages (DSLs) [9] describes speed ups by factors of 3-6 for using DSLs compared to manual coding.

Obtaining code which falls outside this narrow scope necessitates either 1) extension of the code generator, which is typically very expensive, or 2) manual modification of the generated code, which is often difficult and which must be redone whenever changes are made to the program specification.

In this paper, we describe adaptations and extensions of the AUTOFILTER Kalman filter synthesis system which greatly extend the range of programs which can be generated. Given an input specification, the system synthesizes a set of procedures or components for implementing that specification, in addition to a program which implements the specification using those synthesized procedures. The user can then specify code fragments which are combined with the synthesized components to generate programs which could not be generated by the synthesis system alone.

The solution we describe allows users to generate a much wider range of programs without their needing to modify the synthesis system or edit generated code.

We demonstrate the usefulness of the approach by applying it to the synthesis of a complex state estimator based on a manually designed estimator developed and applied to a planetary rover prototype. This estimator combines code from several Kalman filters with user-specified code. When executed on simulated input data, the generated state estimator was found to produce comparable estimates to those produced by a hand-coded estimator.

The paper is structured as follows: in section 2 we describe how the AUTOFILTER system synthesizes state estimation code from succinct, mathematical specifications of the code's desired behavior. We outline how (section 3) such synthesis can enable programs to be developed in an iterative, rapid prototyping style. We then describe (section 4) how a number of problems need to be overcome before we can attain this rapid prototyping ideal: the synthesis system needs to allow new algorithms or variants of existing algorithms to

be synthesized without necessitating costly modification of the synthesis system, and this needs to be done without requiring the user to modify synthesized code. In sections 5 and 6, we describe how these needs can be met by modifying the system to synthesize components for implementing the specification, and allowing the user to specify how those components should be combined with special-purpose code to generate the required program. We describe (section 7) the application of the system to the synthesis of a complex state estimator and demonstrate some of the modifications enabled by our work. In section 8 we describe further work, including increased formality and support for certification, and in section 9 we conclude.

This paper’s main contributions are to:

- show that synthesizing *libraries of components* rather than single procedures increases the range of programs which can be synthesized,
- define a language for expressing non-standard aspects of algorithms, which can be integrated with synthesized components,
- show that the approach can scale to real-world examples.

## 2. AUTOFILTER SYNTHESIS OF KALMAN FILTERS

The Automated Software Engineering group at NASA Ames Research Center has previously developed a number of systems for generating program code in NASA-relevant domains, including AMPHION/NAV [10] for generating code for mission planning and AUTOBAYES [4] for generating data analysis code.

AUTOFILTER [3, 8, 11], which has been developed at NASA Ames Research Center, synthesizes Kalman filters — a standard class of algorithm for *state estimation*, i.e. the estimation of *state variables*, for example position, heading or velocity, from possibly noisy measurements. Kalman filters provide a computationally efficient class of state estimators. In a Kalman filter, the relationship between the measurements and the state variables which are to be estimated is encoded by a set of *measurement equations*. The behavior of the system is encoded by a set of *process equations*, which express how the state variables are expected to evolve, typically with time. The other main parts of the Kalman filter specification are matrices representing the covariance structure of the measurement and state variables, and the initial values of these and the state variables.

AUTOFILTER takes as input a textual specification of a Kalman filter, and automatically outputs a program implementing a Kalman filter meeting that specification. The input specification language contains constructs appropriate to the domain: specification of noise distributions, difference and differential equations, declarations for scalar, vector and matrix constants and variables. By default, AUTOFILTER synthesizes a C++ program using OCTAVE matrix libraries —

```

model driving as 'FIDO driving mode KF'.

/* Process model:  $\dot{\mathbf{x}} = \mathbf{F}_t \mathbf{x} + \mathbf{w}$  */
/* Process noise:  $\mathbf{w} \sim \mathcal{N}(0, \mathbf{q} \cdot \mathbf{I}) \Leftrightarrow w_i \sim \mathcal{N}(0, q_i)$  */

const nat n := 3 as '# state variables'.
const double geometryR as 'wheel radius'.
const double geometryB as 'half wheel base'.
const double ts := 1/200 as 'Update interval'.
const nat n_steps := 200000.

double x(1..n) as 'state variable vector'.
double w(1..n) as 'process noise vector'.
double q(I) as 'variance of process noise'.
double u(time) as 'driver function'.
w(I) ~ gauss(0, q(I)).

equations process_eqs are [
  dot x(0) := geometryR*cos(x(2)) + w(0),
  dot x(1) := geometryR*sin(x(2)) + w(1),
  dot x(2) := geometryR/geometryB*u(tvar) + w(2)].

/* Measurement model:  $\mathbf{z} = \mathbf{x} + \mathbf{v}$  */
/* Measurement noise:  $\mathbf{v} \sim \mathcal{N}(0, \mathbf{r} \cdot \mathbf{I}) \Leftrightarrow v_i \sim \mathcal{N}(0, r_i)$  */

const nat m := 1 as '# measurement variables'.
data double phi_imu(1..m,time) as 'Integrated gyro'.
double v(1..m) as 'measurement noise vector'.
double r(I) as 'variance of measurement noise'.
v(I) ~ gauss(0, r(I)).

equations measurement_eqs are [
  phi_imu(0,tvar) := x(2) + v(0)].

/* Filter architecture */
estimator driving.
driving::process_model      ::= process_eqs.
driving::measurement_model ::= measurement_eqs.
driving::steps              ::= n_steps.
driving::propagate_by_integration ::= true.
driving::update_interval   ::= ts.
driving::initials          ::= xinit().
output driving.

```

**Figure 1.** Part of FIDO driving mode filter specification.

a different target language or library can be requested using command line flags. In a 2002 case study [11], part of the attitude control system for Deep Space 1<sup>1</sup> was specified in AUTOFILTER’s specification language. The full DS1 specification consists of 50 non-blank, non-comment (NBNC) lines (193 with comments). From that specification, AUTOFILTER automatically synthesizes an extended Kalman filter with 281 NBNC (439 with comments) lines of C. The synthesized code was compared to the original DS1 attitude estimation code. Parts of the code from the deployed DS1 state estimator were manually replaced by code synthesized from the above specification, and both versions run in the DS1 simulator. The estimates produced by both versions were found to be essentially identical.

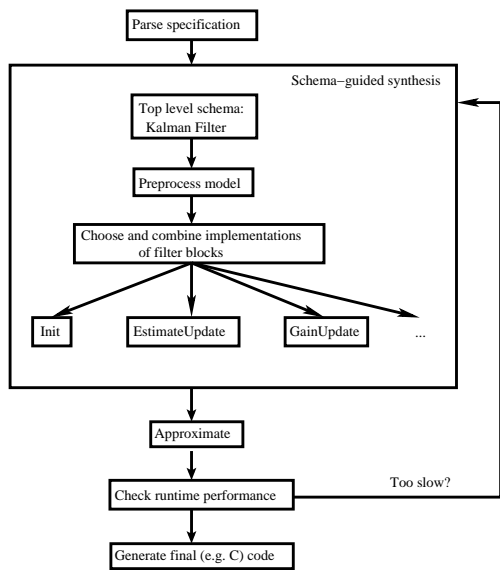
Figure 1 shows part of the specification for one of the Kalman filters which comprises the FIDO state estimator described in more detail in section 7. The specification is succinct, and readily comprehensible by engineers, using a natural syntax for expressing differential equations, and standard statistical language for describing noise distributions.

<sup>1</sup>The Deep Space I (DS1) probe was deployed as a testbed for a range of experimental NASA technologies.

AUTOFILTER employs *schema-guided synthesis* [5]. AUTOFILTER first parses the input specification, then applies a succession of *synthesis schemas* to it. Some schemas generate code fragments, for example updating the Kalman gain; some schemas combine previously generated code fragments into bigger blocks; some schemas make large scale determinations, for example what kind of filter is required.

Figure 2 depicts the structure of the AUTOFILTER system. Synthesis proceeds as follows:

- The specification is parsed and stored internally.
- Code is generated in a simplified imperative language called the AUTOBAYES Intermediate Representation (ABIR). The intermediate language contains constructs for loops, variable and function declarations and calls etc, but avoids complex features of real programming languages like tests with side effects. The economy of the intermediate language facilitates its analysis and manipulation, and simplifies the back-end translation to the desired target language.
- Approximations, if specified, are applied to simplify the intermediate code, and the intermediate code is checked to ensure it meets runtime requirements. These aspects are described in more detail in [8].
- The intermediate code is translated by the back end into the desired target language, e.g. C.



**Figure 2.** Structure of AUTOFILTER. The main part of the synthesis takes place in the middle large box.

A schema implementing a part of a Kalman filter effectively encodes the knowledge which we would expect to find in a text book description of that implementation technique: when engineering a schema-guided synthesis system, a good book is extremely useful, e.g. [2, 6]. In AUTOFILTER, there is currently a single top level schema, which carries out some pre-processing of the model — in particular linearization of the

process equations when they are non-linear, for example in the case of an extended Kalman filter — and determines how the synthesis of a filter should be broken down into the synthesis of a number of simpler components.

If code fragments for all the components can be synthesized, they are composed by the top level schema into code implementing a Kalman filter. By default, the synthesized code takes as input a matrix containing measurements for every time step, and returns as output a matrix of all state estimates.

An important capability of the AUTOFILTER and AUTOBAYES systems is *certification* [3]. The synthesis systems can include annotations into synthesized code which guide a theorem prover to prove that various safety policies are adhered to, for example that the program does not violate array bounds, does not use uninitialized variables, or does not violate the symmetry of certain matrices used in the Kalman filter.

### 3. VARIABILITY IN SYNTHESIS

Implementing a Kalman filter involves a number of trade offs: a simple model with a small number of state variables may not be able to accurately model the behavior of the system — it may be necessary to increase the number of state variables in order to achieve good estimates. Due to the heavy use of matrix operations (including multiplication and inverse), increasing the number of state variables in a Kalman filter very significantly increases the computational complexity of the filter. Code deployed in radiation hard environments must satisfy tough timing constraints: compare Mars Pathfinder’s 20 MHz RAD6000 processor running at 2 MIPS with an old 1 GHz desktop Pentium III running at 3 MIPS. In addition, problems may arise in which some state variables become unobservable and their values no longer adequately constrained by the input measurements.

Implementation of Kalman filters for new aerospace applications is an iterative process. Typically, the filter is prototyped in a simulation environment such as MATLAB until a satisfactory mathematical model has been found. At this stage, decisions are made as to whether and how to simplify or otherwise manipulate the process and measurement equations, what kind of Kalman filter to use (an extended Kalman filter, or a linearized one, continuous or discrete etc), what kind of updates to use (simultaneous updates to cope with possibly correlated measurement noise, Bierman, Carlson square root etc). Once the prototype filter has been tested, it is then recoded for deployment on a target platform in a low-level language such as C. The code is tested and adjusted until its computational characteristics (e.g. runtimes or numerical accuracy) meet the requirements. In the worst case, this can lead to a redesign of the mathematical model. AUTOFILTER can simplify this process in several ways: It can rapidly turn around model changes, which allows the exploration of the filter design space, it can generate multiple alternative implementations for the same specification, which allows the

exploration of the implementation design space, it can generate both MATLAB code (for prototyping) and target platform code (for deployment) from the same specification, it can generate documentation at the same time as code, and it can automatically demonstrate that the code it generates is free from various classes of defects.

#### 4. BARRIERS TO SYNTHESIS IN PRACTICE

As mentioned in section 3, synthesis has the potential to greatly accelerate iterative development of Kalman filters. In practice, a number of barriers need to be overcome in order to realize this potential:

*Often the required program lies outside the range of the synthesis system.* Consultation with state estimation domain experts at JPL revealed that in practice, new state estimators are not simple Kalman filters, but frequently require adaptations to Kalman filter code, for example to perform part of the estimation in a non-standard way, to preprocess sensor inputs, to insert monitoring statements for debugging purposes, or to implement some parts of the filter in a way which departs from the standard definition of Kalman filters.

Ideally, if we find that the AUTOFILTER cannot synthesize the algorithm needed for a particular Kalman filter deployment, we add a new schema to AUTOFILTER to synthesize the parts of the filter that differ from what can already be synthesized. Two problems stand in the way: first, adding new schemas typically requires very deep knowledge of the principles and implementations of the synthesis system and so is feasible for system developers but not for system users, and second, adding new synthesis schemas is extremely time-consuming. Some recent experiments carried out by the first author suggest that making small variations to synthesized code requires around 10 lines of changed schema code for each single line of changed synthesized code. It is only worth making changes to the synthesis schemas if those changes are going to be used again many times in the future.

*We can rapidly resynthesize whenever the specification changes.* If changes to the synthesized code are made by hand, then they must be redone every time the code is resynthesized. This is time consuming and error prone.

We tackle these two problems by restructuring the AUTOFILTER schemas (section 5) and implementing a compiler for specifying the non-standard aspects of the code (section 6).

#### 5. GENERATING COMPONENTS

A Kalman filter algorithm can typically be broken down into four or five blocks: initialization, gain calculation, measurement update, covariance update, state propagation. It is natural for the synthesis schemas to reflect this block structure in the Kalman filter domain. Where possible, each of those blocks should be synthesized independently of the others in order to maximize the number of combinations which can be

```
propagate(covariance, integration):matrix pminus, matrix xhatmin, matrix
q,double delta_t → matrix pminus,matrix pplus
propagate(covariance):matrix pplus,matrix phi,matrix q → matrix pminus
update(kalman_gain): matrix pminus, matrix h, matrix r → matrix gain
```

**Figure 3.** The types of some synthesized components for one mode of the FIDO rover filter (see Section 5). Users can employ the components in their own customized filters.

generated and therefore maximize the chances that the user will get a program which meets their needs. Synthesis in other domains is also typically modular.

Many useful variations of Kalman filters can be obtained by removing, replacing, or extending one of the synthesized blocks. Providing a means for a user (rather than a developer) to specify modifications to the synthesized Kalman filter which remove, replace or extend one of its synthesized blocks significantly extends the range of useful programs which can be synthesized. In order to provide this means, we adapted AUTOFILTER in the following ways:

- We modified the schemas to synthesize self-contained code blocks, corresponding to the block structure of the domain. Each synthesized block is realized as a procedure in order to clearly identify its dependencies on the rest of the synthesized code.
- We designed and implemented a simple imperative programming language which allows the user to specify code fragments in a way which is independent of the target programming language and can interface easily to the synthesized algorithm blocks.

Figure 3 shows the types of some of the components synthesized for the turn-in-place mode of the FIDO filter (see section 7 for more details). Note that in this case two alternative implementations for covariance update are generated — when incorporating them into their own code (see section 6), the user can choose which one to use, or use both. Note also that the synthesized procedure for propagation of covariance by integration uses `pminus`, the prior covariance matrix, as both an input and an output.

#### 6. IMPERATIVE SPECIFICATION LANGUAGE

##### Introduction

In this section, we describe our design and implementation of a simple imperative language, sufficient to allow the user to specify how a synthesized Kalman filter should be adapted to meet the demands of real state estimation.

Our aims in designing the language are that it should:

- Be sufficient to express general kinds of algorithms, in particular for data processing.
- Provide a way for the user to specify how their hand-written code interleaves with the synthesized Kalman code.

The basis for a language meeting the above requirements is already present in AUTOFILTER. AUTOFILTER synthesizes code in two steps: first, code is synthesized from the specification in an intermediate language called the ABIR, and second this ABIR code is compiled by the back end to code in the target language: C or C++ using various different function interfaces and libraries for performing matrix calculations (a back end for MODULA2 also exists but is rarely used). ABIR was designed to be sufficiently expressive to be able to express fairly general algorithms, especially for numerical data processing, and to be quite close to imperative languages such as C — the difficult part of synthesis should be the generation of the algorithm implementing the specification; translating that code to a particular C-like language should be straightforward.

We therefore based the imperative specification language (ISL) on the AUTOBAYES Intermediate Representation, ABIR. We added syntactic sugar and type checking to assist the user, but the most important extension was the addition of a mechanism for referring to variables and procedures synthesized from different specifications; this allows the user to combine code arising from different specifications, and to refer to variables and procedures without having to know the variable/procedure names chosen by the synthesis system.

### ISL Language Features

ISL code included by the user in the specification uses the following constructs:

- *Constants*: Float and integer literals. MATLAB-syntax matrix literals, e.g. `(0;0;0)`, which denotes a  $3 \times 1$  matrix whose double values are all 0.0.
- *Variable references* (see section 6 for more details). References can be to a scalar, vector, matrix, or an element of a vector, `vector(expression)`, or to an element of a matrix, `matrix(expression, expression)`.
- *Expressions*: variable references, arithmetic, vector and matrix operations using binary operators `+`, `-`, `*`, `/`, unary operators `++` (as in C/C++, this is used to increment a variable, but returns a value also), `abs`, boolean expressions `<`, `>`, `=(expression)` etc.
- *Statements*: while loops `while(condition, body)`, conditionals `if(condition, then, else)`, scalar, vector, and matrix assignment. The statement `tic` increments the clock, which is a variable associated with the `timevar` attribute of the main library. Section section 6 below discusses two other statement types: calling synthesized components, and a special form of the assignment operator which permits assignment of an entire row or column of a matrix.
- *Comments*, e.g. `c(['No comment'])`, become attached to the immediately following statement.
- *Macros* are easily implemented using Prolog variables.

### Variable References

Variables and synthesized procedures are associated with the specification from which they are derived. The user refers

to them using references of the form `Library::handle`, where `Library` is the name of the specification. If the prefix `Library::` is missing, the default library is used. When a reference is used, the compiler looks in the input attributes for an element `Handle` in the specified library, which yields the variable name to use in the generated code, and the variable's type. This is particularly useful when combining code from several synthesized components: the user does not need to know the name the synthesis system chose for an internal variable, only the handle of that variable, which denotes its *purpose* in the filter. For example `drivingKF::kalman_gain` denotes the variable used to hold the Kalman gain in the filter synthesized from specification `drivingKF`.

The ISL does not currently have constructs for declaring variables — variables and their types must currently be specified in the input synthesis environment; either they must be declared in one of the specifications whose synthesis is incorporated into the synthesis environment, or they must be manually added to it. This restriction is somewhat arbitrary and easily overcome (section 8).

### Procedure Calls and Matrix Row Assignments

A special form of the assignment operator which permits assignment of an entire column or column of a matrix: when the term `'_'` is an index in both the left and right hand sides of a matrix assignment, a `for` loop is generated which loops over every element of the appropriate row or column of the matrix (depending on whether the `_` is the first or second index of the LHS matrix). For example, if `m` is a  $4 \times 1000$  matrix, then the assignment `m(_, t+1) = m(_, t) + m(_, t-1)` generates ABIR code which corresponds to the C/C++ loop `for(i=0; i<4; i++) m(i, t+1) = m(i, t) + m(i, t-1)`.

Synthesized components take the form of procedures — this is a good way to ensure proper encapsulation and make explicit what data the component assumes and what it produces, i.e. its inputs and outputs. If `schema` is the name of a synthesis schema, then the syntax `Library::call(schema)` represents a call to the component of `Library` synthesized by `schema`.

In the current version of the language, procedure calls have no additional parameters. The specified component is called with the same variable names and types as specified in its (synthesized) declaration. An obvious extension is to allow calls to components to specify all or some of the arguments for that component.

### Compiling ISL

ISL code is compiled given a *synthesis environment*, which stores the results of prior code generation, including synthesized procedures and generated variable names and types. The ISL is compiled to an ABIR term which can be passed directly to the synthesis system back end (in fact, the back

end does most of the hard work).

An important post-processing step is applied to the resulting ABIR term: partial evaluation unfolds procedure definitions where they are called. This allows us to avoid the overhead of parameter passing. It also allows synthesized procedures to use multiple outputs, and input-output variables, even when these are not allowed by the target back end.

## 7. EXAMPLE

The extended AUTOFILTER system was used to specify and synthesize code for a complex state estimator described in [1]. The estimator has a number of features which make it impossible to synthesize using the old version of AUTOFILTER:

- The estimator effectively combines code from several different filters: a discrete linear Kalman filter to estimate gyro drift, an extended Kalman filter to combine heading with estimated rover position, and a filter to integrate rover position given inputs from gyro and wheel rotation sensors.
- The main technique used to estimate rover position is not a regular Kalman filter, but instead integrates sensor readings directly. Covariance matrices are updated using non-standard equations.

Figure 4, adapted from [1], shows the overall estimator architecture. The following pseudocode corresponds roughly to this architecture:

```

init();
while (!done) {
  if (start of traverse) get sun sensor reading;
  while (stopped) estimate angular velocity;
  let gyro_bias = final angular velocity estimate;
  while (moving) {
    heading+=delta_heading-gyro_bias;
    if (turning in place)
      propagate state and covariance matrices
      using turn-in-place mode equations;
    else
      propagate state and covariance matrices
      using driving mode equations;
  }
  fuse integrated state and angle using EKF; }

```

A few words about how the estimator works: motion is divided into traverses. At the end of each traverse, the rover stops and takes a sun sensor reading, which gives an accurate estimate of true heading. Whenever the rover is stopped, it uses a linear Kalman filter to estimate its angular velocity. Since, when stopped, it is known that the rover’s angular velocity should be zero, this estimate is in fact the gyro bias. When the rover is moving, readings from the gyro (corrected by the gyro bias previously estimated) and the wheel rotation sensors, are integrated to give heading and total wheel rotation. Differential equations of motions which depend on the rover mode — either driving forwards, or turning on the spot — are then solved to update the rover state and the error covariance matrices. Finally, at the end of a motion, an extended

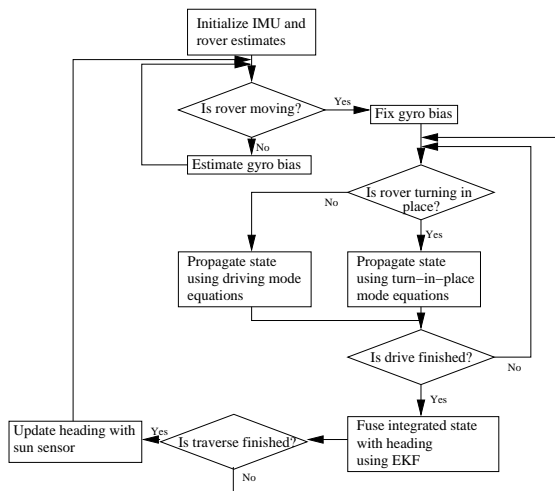


Figure 4. FIDO estimator architecture adapted from [1].

Kalman filter (EKF) is used to fuse integrated heading with estimated state.

Initially, many of the details of the required estimator were unclear. The second author independently wrote an implementation in MATLAB of the state estimator. This served two purposes:

- It provided a standard against which we could test the results produced by running our synthesized estimator.
- It refined many details of the estimator, which were not explicit in [1] or were difficult for someone who is not a control engineer to extract.

Figure 5 shows excerpts from the ISL for the FIDO example. Specifying it in AUTOFILTER without ISL and the other modifications described in this paper would have required the addition of a large number of very specialized schemas: for example, obtaining the current heading would require a schema for summing an input adjusting for some offset, taking a sun sensor reading would require a schema for overwriting an estimate with an input quantity, and the architecture of the whole filter would itself require a specialized schema; it would be very impractical.

By contrast, ISL provides exactly the constructs needed to specify the filter, leveraging the components synthesized by AUTOFILTER. Figure 5 illustrates many of the language features described in section 6:

For this example, AUTOFILTER’s synthesis engine is run three times:

- On a specification of the gyro bias estimation filter. This synthesizes a procedure which implements the gyro bias estimation filter using a library of six synthesized ABIR components (i.e. procedures) for initialization, Kalman filter gain calculation, estimate update, covariance update, state propa-

```

TV=timevar,
Moving=(locomotion(TV)>0),
Turninplace=(locomotion(TV)=1),
NewTraverse=(locomotion(TV)=-1),
...
TV = 0; theta_l = 0; theta_r = 0;
gyro::call(init) ;
driving::call(init) ;
posterior_state_estimate = (0;0;0);
while(TV < ((n_steps)-2), (
  if(NewTraverse,(
    c(['If this is a new traverse, get
      reading from sun sensor']);
    phi_imu(0,TV) = sunsensor(TV);
    prior_state_estimate(2,0) = sunsensor(TV);
    posterior_state_estimate(2,0) =
      sunsensor(TV);
    kalman_output(_,TV) =
      posterior_state_estimate(_,0)
  ),
  (
    c(['otherwise carry on filtering']);
    ...
    c(['Copy final gyro bias estimate
      to phi_dot_bias.']);
    phi_dot_bias =
      gyro::posterior_state_estimate(0,0) ;
    while((Moving and (TV < ((n_steps)-2))), (
      c(['integrate heading using corrected
        gyro reading']);
      phi_imu(0,TV) = phi_imu(0,TV-1)+
        deltat(TV)*(gyro::phi_dot_meas(0,TV)
          - phi_dot_bias) ;
      theta_l = theta_l+dtheta_l(TV);
      theta_r = theta_r+dtheta_r(TV);
      if(Turninplace,(
        delta_t = (dtheta_l(TV)-dtheta_r(TV))/2;
        turninplace::call(update(loop_dependents));
        turninplace::call(propagate(state,integration));

```

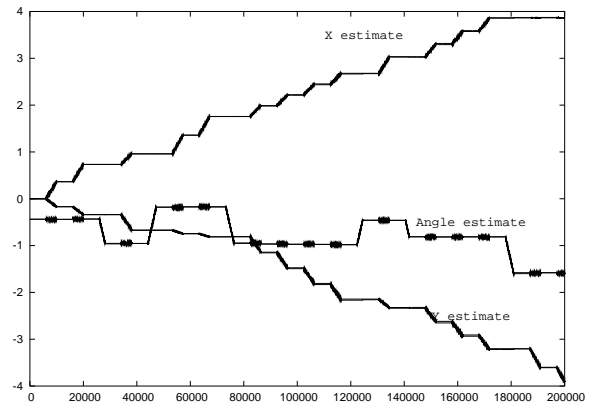
**Figure 5.** Code excerpts for FIDO example.

gation, and covariance propagation,

- On a specification of the driving mode. As for the gyro bias estimator, this yields a library of six ABIR components, plus two additional components for propagating the rover state and covariance by integrating the driving mode equations of motion.
- On a specification of the turn-in-place mode. As above, this yields a library of six ABIR components, plus two additional components for propagating the rover state and covariance by integrating the turn-in-place mode equations of motion. The components for propagating the state and covariance by integration are used during motion; the components for propagating the state and covariance using the regular extended Kalman filter propagation equations are used at the end of each motion.

Section 3 listed the components synthesized for the turn-in-place mode.

The three specifications together comprise 115 NBNC lines of specification. The results from the three synthesis runs are merged into a single synthesis environment, which is then input to the ISL compiler along with 74 lines of ISL code. After unfolding of the component calls, the final result is



**Figure 6.** Outputs from the FIDO estimators. At the right edge of the graph, estimated X coordinate, Y coordinate and angle are at the top, bottom, and middle respectively.

496 NBNC lines of C++ code. The C++ code was compiled into the Octave computer algebra system environment, yielding an Octave command which took as argument the readings from wheel rotation sensors and gyroscope and produced rover state estimates as output.

The MATLAB code was modified to save both the estimates produced by its filter, and the simulated inputs which it generated. These simulated inputs were passed as arguments to the synthesized Octave command, and when compared with the MATLAB outputs, the estimates were found to be comparable. Figure 6 shows three state estimates (X position, Y position, heading) produced by the synthesized filter.

## 8. FURTHER WORK

The most pressing area of further work is integration with AUTOFILTER's certification engine. In current usage, certification provides a rubber stamp for synthesis — except during rare occasions where an error in the synthesis engine is uncovered, certification of synthesized programs almost always succeeds. We can expect to gain much greater leverage of certification by extending it to the flexible synthesis described in this paper. When combining components with hand-written code, it is non-trivial to ensure that the correct elements of input or intermediate matrices are used at the correct times. The *initialization*, *input-usage* and *array bounds* policies will be very helpful here and provide significant advantages over purely hand-written code.

The ISL language should also be consolidated, adding obvious constructs such as variable declarations (which are currently made by including them in the Kalman filter specification). This would have a pleasant side-effect of making ISL a portable programming language, capable of being compiled down to any of AUTOFILTER's target programming languages.

## 9. CONCLUSIONS

Domain-specific program synthesis can automatically generate high quality code in complex domains from succinct specifications. This can enable a powerful iterative form of development in which a specification is initially developed, the synthesized code tested, the specification revised and so on.

In practice, synthesis is only part of the story. Synthesized code may form the basis of deployed code but must frequently be modified in order to incorporate aspects which do not fit within the domain of the specification language. Carrying out such modifications by hand spoils the arguments for rapid prototyping and correctness, since these modifications must be redone each time the specification is changed and the code resynthesized.

In this paper, we outlined modifications to the AUTOFILTER program synthesis system to synthesize *libraries of components* rather than single procedures. We defined a language (ISL) for specifying imperative code in such a way that it can be integrated automatically into a synthesized program.

We described the application of the modified system and ISL to the synthesis of a complex state estimator. The estimator design used in this application had been developed and implemented on a planetary rover prototype. The many human design decisions, approximations, and estimator modifications present in this example are commonly found in many complex estimator design applications; the work described in this paper enables those design decisions to be implemented using the synthesis system. The results of running our synthesized code were found to agree with the results of running a hand-coded MATLAB implementation of the same filter.

## REFERENCES

- [1] E. Baumgartner, H. Aghazarian, and A. Trebi-Ollennu. Rover localization results for the FIDO rover. In *Proc. SPIE Conf. Sensor Fusion and Decentralized Control in Autonomous Robotic Systems*, 2001.
- [2] R. Brown and P. Hwang. *Introduction to Random Signals and Applied Kalman Filtering*. John Wiley & Sons, 3rd edition, 1997.
- [3] E. Denney, B. Fischer, J. Schumann, and J. Richardson. Automatic certification of Kalman filters for reliable code generation. In *Proceedings of 2005 IEEE Aerospace Conference*, 2005.
- [4] B. Fischer and J. Schumann. AutoBayes: A system for generating data analysis programs from statistical models. 13(3):483–508, May 2003.
- [5] P. Flener, K.-K. Lau, M. Ornaghi, and J. Richardson. An abstract formalisation of correct schemas for program synthesis. *Journal of Symbolic Computation*, 30(1):93–127, 2000.
- [6] M. S. Grewal and A. P. Andrews. *Kalman Filtering:*

*Theory and Practice Using MATLAB*. Wiley Interscience, 2001. 2nd edition.

- [7] R. B. Kiebert, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 542–552, Washington, DC, USA, 1996. IEEE Computer Society.
- [8] J. Richardson, J. Schumann, B. Fischer, and E. Denney. Rapid exploration of the design space during automatic generation of Kalman filter code. In *Proceedings of 2005 IEEE Aerospace Conference*, 2005.
- [9] A. Van Deursen and P. Klint. Little languages: Little maintenance? *Journal Of Software Maintenance*, (10):75–92, 1998.
- [10] J. Whittle, J. V. Baalen, J. Schumann, P. Robinson, T. Pressburger, J. Penix, P. Oh, M. Lowry, and G. Brat. Amphion/NAV: Deductive synthesis of state estimation software. In *Proceedings of the 16th IEEE Conference on Automated Software Engineering*, 2001.
- [11] J. Whittle and J. Schumann. Automating the implementation of Kalman filter algorithms. *ACM Transactions on Mathematical Software*, 2005. To appear.



*Dr. Julian Richardson* (PhD University of Edinburgh, 1995) is a Research Scientist in the Automated Software Engineering Group, NASA Ames. He is engaged in research on automatic program generation, in particular of Kalman filters, and in research on software risk and the effectiveness of verification and validation techniques for aerospace software. He has published around 30 technical papers, in areas including automated theorem proving, program synthesis and transformation, and 3 magazine articles.



*Dr. Edward Wilson* is President of Intellization, a consulting business applying and extending intelligent systems technologies for optimization in the aerospace and metals industries since 1995. Attended M.I.T. from 1983-1987, receiving one S.M. and two S.B. degrees. Ph.D. in M.E. from Stanford University, 1995. Previously at: Hughes Aircraft; US Air Force; Director of Research at Neural Applications Corporation; Visiting Scholar and Lecturer in the Stanford Aero-Astro department. Areas of interest and expertise include: fault detection and isolation; process optimization; identification; estimation; signal processing; control; and other applications of advanced data analysis technologies.