Amphion/NAV: Deductive Synthesis of State Estimation Software

Jon Whittle, Jeffrey Van Baalen, Johann Schumann
Peter Robinson, Tom Pressburger, John Penix
Phil Oh, Michael Lowry, Guillaume Brat

ASE Group: NASA, QSS, RIACS, Kestrel Tech., U. Wyoming
NASA Ames Research Center, Moffett Field, CA 94035

1 Introduction

Previous work on domain-specific deductive program synthesis [11, 12] described the Amphion/NAIF system for generating Fortran code from high-level graphical specifications describing problems in space system geometry. Amphion/NAIF specifications describe input-output functions that compute geometric quantities (e.g., the distance between two planets at a point in time, or the time when a radio communication path between a spacecraft and earth is occluded) by composing together Fortran subroutines from the NAIF subroutine library developed at the Jet Propulsion Laboratory. In essence, Amphion/NAIF synthesizes code for glueing together the NAIF components in a way such that the generated code implements the specification, with a concurrently generated proof that this implementation is correct. Amphion/NAIF demonstrated the success of domain-specific deductive program synthesis and is still in use today within the space science community. However, a number of questions remained open that we will attempt to answer in this short paper, namely:

- Can the deductive synthesis strategy be extended from the generation of input-output functions to iterative, imperative programs without incurring the computational complexity penalties entailed by most theoretical treatments of the formal derivation of imperative systems?
- Can the methodology for developing an Amphion-like program synthesis system be used in other domains, particularly where a well-defined component library does not already exist?
- Can the development of Amphion-like domain-specific program synthesis systems be modularized, with substantial reuse at the intersection of domains?
- How can the mechanized proof of implementation correctness be used in the software process by end-users unfamiliar with formal methods?

In order to investigate these questions, we developed Amphion for a new, much richer domain, namely Guidance, Navigation & Control (GN&C) algorithms, in particular single-mode geometric state estimation software for aerospace vehicles. AMPHION/NAV generates code for integrating a model of the vehicle dynamics with a temporal stream of data from multiple sensor sources in a statistically optimal way using one or more Kalman filters [1, 4]. GN&C algorithms are often complex, involving iterative loop algorithms and real-time considerations such as extrapolating sensor data so that data is integrated at the same point in time. Although there are standard components available for this domain (e.g., matrix manipulations, Kalman filter algorithms), there is no easily defined set of components that cover the domain fully.

AMPHION/NAV incorporates the Amphion/NAIF domain theory as one component in a much larger domain theory, demonstrating a form of theory reuse. In particular, the Amphion/NAIF domain theory provides the formulation of the geometric concepts. AMPHION/NAV is also an upgraded version of the same architecture as Amphion/NAIF, and thus demonstrates reusability across a number of components including the specification interface and the deduction engine. AMPHION/NAV incorporates an enhanced back-end code generator suitable for iterative programs, and significantly extends the explanation capabilities of the previous Amphion system [12]. The code generated by AMPHION/NAV is annotated with detailed explanations describing where each expression in the code came from. These explanations are constructed by tracing automatically through the proof that produced the code and composing explanations for each of the axioms used in the proof. As a result, each program expression can be explained in terms of the concepts in the specification from which they were derived. These explanations are given in the form of hyperlinked text in a standard notation of the GN&C domain.

Because GN&C algorithms are often used in safety-critical systems, detailed explanations are crucial to provide a means for a certification body such as the FAA to examine the code in detail and to know precisely where each code expression came from. Explanations are also crucial when the generated code needs to be modified or integrated it into a larger system.

The domain of state estimation turns out to be a good challenge domain for deductive synthesis. Developing state estimation software tends to be a black art. In principle, the engineer should develop a mathematical model that closely resembles the real-world characteristics of the problem. The output of simulation runs on this model should then be used to refine the model until a threshold level of accuracy is reached. In practice, however, engineers start off with a mathematical model but the time and cost constraints associated with the project mean that they merely "tweak" parameters in their code rather than reassessing the fidelity of the model. Program synthesis encourages analysis to take place at the modeling level and enables rapid design space exploration.

2 Background on State Estimation

The domain of interest for AMPHION/NAV is that of geometric state estimation, i.e., estimating the actual values of certain *state variables* (such as position, velocity, attitude) based on noisy data from multiple sensor sources. The standard technique for integrating multiple sensor data is to use a Kalman filter. A Kalman filter estimates the state of a linear dynamic system perturbed by Gaussian white noise using measurements linearly related to the state but also corrupted by Gaussian white noise. The Kalman filter algorithm is essentially a recursive version of linear least squares with incremental updates.

The state estimation problem can be represented by the following equations, given in vector form:

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t), t) + \mathbf{w}(t) \qquad (1)$$

$$\mathbf{z}(t) = h(\mathbf{x}(t), t) + \mathbf{v}(t) \qquad (2)$$

The first equation is the process model, a vector differential equation modeling how the state vector, $\mathbf{x}(t)$, changes over time. The second equation is the measurement model, relating the measured variables to the state variables. Specifically, $\mathbf{x}(t)$ is the state vector (with $\dot{\mathbf{x}}(t)$ the time derivative) of quantities to be estimated (e.g., position, attitude, etc.), $\mathbf{z}(t)$ is a vector of measurements (the state variables are not necessarily measured directly), $\mathbf{v}(t)$, $\mathbf{w}(t)$ are Gaussian white noise perturbances on the measurement and process model respectively and f and h are possibly nonlinear continuous functions that must be discretized for implementation purposes.

A Kalman filter is an iterative algorithm that returns a time sequence of estimates of the state vector, $\hat{\mathbf{x}}(t)$, by fusing the measurements with estimates of the state variables based on the process model in an optimal fashion, i.e., the estimates minimize the mean-square estimation error. In the case where either f or h is nonlinear, a Kalman filter can still be used by first linearizing around a "nominal" estimate. After linearization and discretization, f and h can be represented by matrices Φ (the state transition matrix) and H (the measurement matrix) respectively.

The standard implementation of a Kalman filter requires seven inputs: the Φ and H matrices, the covariance structure of the process and measurement noise ($\mathbf{w}(t)$ and $\mathbf{v}(t)$), an initial state estimate $\hat{\mathbf{x}}(t_0)$, an error covariance matrix of the initial estimate and, of course, the sequence of measurements. During each iteration of the filter, the state estimate is updated based on new measurements and the estimate error covariance is updated for the next iteration.

The AMPHION/NAV system takes as input a specification of the process model (a typical model is a description of the drift of an INS system over time), a specification of sensor characteristics, and a specification of the geometric constraints between an aerospace vehicle and any physical locations associated with the sensors - such as the position of radio navigation aids. The input specification also provides architectural constraints, such as whether there is one integrated Kalman filter or a federation of separate Kalman filters. AMPHION/NAV produces as output code that instantiates one or more Kalman filters. The user can run or simulate the code, determine that it is lacking (e.g., that the simulated estimate for altitude is not sufficiently accurate), reiterate the design (e.g., by adding a radio altimeter sensor to the specification), and then rerun the experiment.

3 Amphion/NAV System Architecture

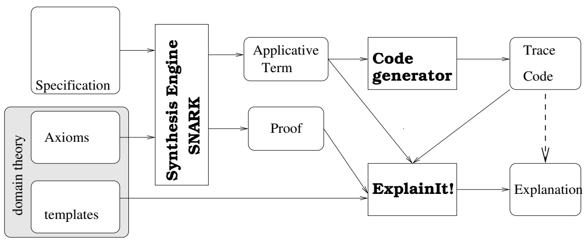

Figure 1. Architecture of Amphion/NAV with ExplainIt!

Figure 1 presents the architecture of the AMPHION/NAV system. The domain theory (Section 4) specifies the types and operation signatures in the domain, and contains axioms describing the implementation of the *abstract* operations (which are used in the problem specification) in terms of *concrete* operations (which are used in the implementation). The domain theory also contains explanation templates associated with each axiom (Section 5) providing documentation about their meaning.

The process of *deductive synthesis* [6, 9] submits the specification and the axioms of the domain theory to the synthesis engine, which is the SNARK[1] refutation-based theorem prover. The theorem prover proves that the specification is a consequence of the domain theory, and returns a proof and witness terms for the output variables, in our case an applicative term comprising the synthesized program. The code generator is given the applicative term and produces code in the target programming language by applying several program transformation phases. The target language in AMPHION/NAV is C++ and OCTAVE.[2] Amphion/NAIF generated Fortran code, but only the last phase of the code generator needed to be changed for AMPHION/NAV.

The code generator records a trace of the application of the transformations. The ExplainIt! component (Section 5) accepts the axiom explanation templates, the proof, the applicative term, and the code generator trace, and produces an explanation structure for the final code. This structure links portions of the target code to explanations. The explanation of a portion of target code is generated from the explanation templates associated with the axioms that were used in the creation of that portion of the code.

4 Engineering a Domain Theory for State Estimation

The state estimation domain theory represents both the operations and algorithms in the domain and how those domain elements are properly applied. For the initial version of AMPHION/NAV described in this paper, the scope is that of advanced graduate-level textbooks (e.g., [1]) state estimation examples. In all, six textbooks were used in developing the domain theory, ranging from general applied Kalman filters to specialized texts on INS and GPS systems. The examples used ranged from simple state estimation systems with radio beacons to complex INS/GPS systems with additional aiding sensors. The methodology followed was to work from concrete examples given in the textbooks, and, from these examples, to identify the concepts of the domain and the relationships between those concepts. Input from domain experts was solicited to validate these efforts. The domain theory is a collection of modular subtheories each containing a set of axioms describing the primitives in the subtheory and the relations between them.

Figure 2 shows the structure of the subtheories in the current domain theory. The arrows show which subtheories import other subtheories (e.g., the axioms for frame conversions import the NAIF axioms). In the synthesis proofs, the NAIF axioms and frame/coordinate axioms are applied using resolution, paramodulation and demodulation. All other axioms are applied using demodulation only. This was a restriction made to control the proof process. The arrows in Figure 2 also manifest themselves in the axioms: the rules refine primitives from one subtheory into primitives from a subtheory connected by an arrow. Note that the leaf nodes of Figure 2 are subtheories whose primitives appear in the final applicative term. In essence, high-level abstract primitives specifying Kalman filter architectures and sensor configurations are refined into primitives of Euclidean geometry and matrix/vector operations. Refinements also take place *within* the theories that refine primitives of those theories into primitives that are wrappers around code components.

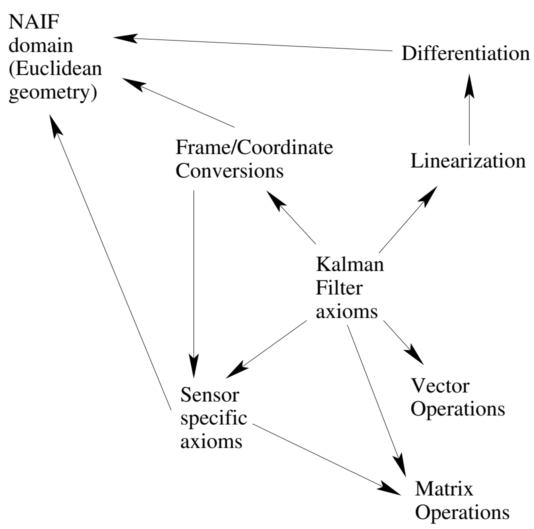

Figure 2. Amphion/NAV Domain Theory Organization

In general, the synthesis engine applies proof search to apply the axioms in a way that suits the current context. This may involve making pre-defined assumptions as to the nature of the current problem (e.g., that the nominal estimate is close enough to the true value to enable a Taylor series expansion to be accurate) but these assumptions appear explicitly in the final explanations presented to the user.

A fully declarative domain theory is ideal for expressing the concepts in a new domain and for communicating and validating their relationships. On the other hand, code generation (regardless of which synthesis engine is used) needs more guidance to be successful. As part of our methodology, we began with a highly declarative domain theory and

[1] URL: `http://www.ai.sri.com/ stickel/snark.html`
[2] A Matlab clone: `http://www.octave.org`

then extended this theory with operational elements to enable successful refinement.

Another way to limit the search space is to make use of *decision procedures* in the theorem proving process. The idea is to solve appropriate subtasks (over ground terms) that come up in the proof by calls to external routines rather than relying on the proof engine. Amphion/NAIF contained decision procedures for instantiating variables with the appropriate coordinate frame. AMPHION/NAV uses SNARK's procedural attachment mechanism to incorporate decision procedures from the KIF library[3] (list manipulations, numeric manipulations, etc.) and procedures for low-level matrix manipulations.

5 The ExplainIt! Documentation Generator

The code generated by AMPHION/NAV is annotated with detailed explanations describing where each statement and expression in the code came from. Intuitively, an explanation of a statement in the generated program is a collection of explained connections between the variables, functions and subroutines in that statement and objects, relations, and functions in the problem specification or domain theory respectively.

Our explanation technique works on the proof derivation of the generated program which is a tableau, a tree whose nodes are sets of formulas together with substitutions of the existentially quantified variables, and whose arcs are steps in the proof (i.e., they encode the "derived from" relation). Thus, an abstract syntax tree (AST) of the synthesized program and the empty clause is the root of this derivation tree. Its leaves are domain theory axioms and the problem specification. Since the AST and all formulas are represented as tree-structures terms, the derivation tree is essentially a tree of trees.

The explanation generation procedure traces back a position in the abstract syntax tree through the derivation tree extracting *explanation equalities* along the way. These equalities record the links between positions of different terms in the derivation. By reasoning with these equalities, *goal explanation equalities* are derived which relate elements of the generated program with terms in the specification and formulas in the domain theory.

With these explanation equalitites calculated, the appropriate explanation templates of the domain theory axioms are instantiated and composed. Finally, an XML document is assembled, containing an explanation for every executable statement in a synthesized program in a vocabulary that the domain expert understands. The explanation indicates why that statement is in the program and how the statement relates to the problem specification and the domain theory. These steps which are an extension of work reported in [12] will be described in the following.

Explanation Equalities. All pieces of a formula are identified using a position notation, described by a path from the root of the formula to that position. A path description is a sequence of argument position selectors, e.g., the path $\langle 2,1\rangle$ specifies the position of b in the term $f(a, g(b, c))$, i.e., $b@\langle 2,1\rangle$. *Explanation equalities* capturing the links between pieces of a formula are assertions of the form $\Phi_1@p_1 = \Phi_2@p_2$ between terms Φ_1, Φ_2 at positions p_1 and p_2 in the term, respectively. Explanation equalities are also extracted for variable substitutions generated during each derivation step.

Templates. The axioms of the domain theory are annotated with explanation templates which consist of text fragments and variables. All variables occurring in a template must also occur in the axiom to which the template is attached. Each axiom can have multiple templates each of which is associated with a different position in that axiom.

Template Instantiation and Composition. The explanation for a position in the generated program is composed from the templates associated with the explanation equalities. This is accomplished by constructing an equivalence class C_E w.r.t. the explanation equalities of the derivation. Then the desired goal explanation equalities linking a subterm to the specification and domain theory are contained in the corresponding equivalence class; the templates can be found in the set of templates attached to the formula positions in C_E. To construct the entire explanation, the templates in this set are instantiated and concatenated together according to the order in which they occur in the derivation.

Document Assembly. The final output of ExplainIt! is a document which explains each part of the synthesized code in a format suitable for the domain engineer. The structure of the explanation is reflected in the computational structure of the applicative term. Thus, explanations are constructed for each position in the applicative term. As a flexible intermediate format, XML is used, because it facilitates the generation of various document formats. Furthermore, hyper-links allow the user to transparently trace between the final code and the explanation document. This is necessary because the structure of the imperative C++ code does not necessarily coincide with the structure of the applicative program.

XSLT [8] is used to produce the final structured HTML version of the explanation. In order to enhance readability, all terms containing matrices are shown as HTML tables (in AMPHION/NAV they are represented as hard-to-read lists of lists). Fig. 6 shows the upper left corner of the 2x9 measurement matrix H. The XSLT parser can easily be modified to handle various syntactic transformations thus facilitating adaptation to other domains. ExplainIt! is thus configurable in a similar way like Hallgren's Proof Editor [7],

[3] URL: http://logic.stanford.edu/kif/kif.html

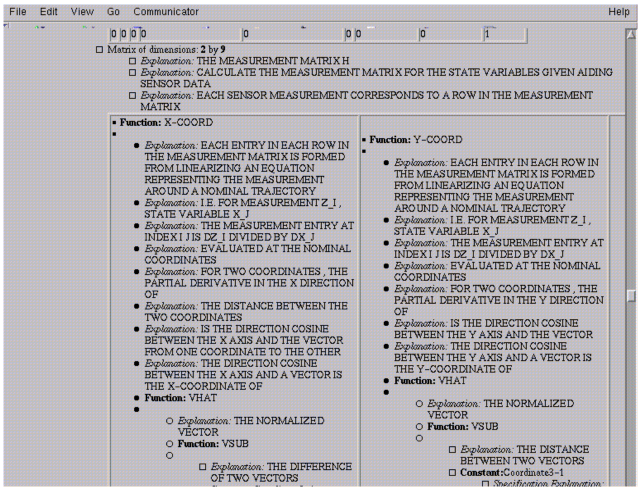

Figure 3. Screen dump of a part of the explanation document

or the ILF system [3].

6 Experiments and Results

We have used AMPHION/NAV to synthesize 5 groups of examples of single-mode geometric state estimation software. The examples use either an inertial navigation system (INS), or a GPS system as its basis. As aiding sensors, we have used models for distance measuring equipment (DME), VOR (measuring the angle between the aircraft and a fixed station), and a barometric altimeter. The following table gives an indication of the performance of the system for all examples. The size of the specification is given as the number of conjuncts in the textual (logic) representation of the specification. The domain theory consists of 622 axioms, 344 of which have been reused from the NAIF domain. T_{SNARK} depicts the run-time to find a proof (including loading of the prover's Lisp code) in seconds on a Sun Ultra 60. Finally, the number of lines of synthesized C++ is given. This number includes comments and rather lengthy interface code (300-350 lines per example).

measure	min	mean	max
size of spec	67	83	114
$T_{\text{SNARK}}[s]$	44	390	1311
C++ lines	712	950	1208

7 Conclusions

We have presented AMPHION/NAV, a deductive synthesis system for the automatic generation of highly documented state estimation software with Kalman filters. Although there have been many improvements over the old synthesis system with respect to domain complexity, usability, and generation of explanations, there is still a number of important issues to be addressed. During development of AMPHION/NAV it turned out that the graphical specification language, originally developed for space trajectory specifications, needs extensions for the state estimation domain. The original specifications were relational in nature, whereas a functional specification may be more appropriate for the new domain.

As described in the paper, the development of the domain theory turned out to be a central issue for our synthesis system. Although the old NAIF domain theory could be reused in an as-is manner, the structure and development process for the domain theory needs to be improved substantially. We are investigating in how far techniques from object-oriented software design can be of help to develop a domain theory in a much more structured and fundamental way.

In developing AMPHION/NAV, much effort was spent on the explanation system. Automatic generation of documentation is only a first step. Future work will investigate how far deductive synthesis can support computer-supported certification of safety-critical code by automatic generation of verification proof obligations, invariants, and other annotations for the synthesized code which then can be checked by a small and trusted proof checker.

References

[1] R. G. Brown and P. Hwang. *Introduction to Random Signals and Applied Kalman Filtering*. Wiley, 3rd ed., 1997.

[2] B. I. Dahn and A. Wolf. *Natural Language Presentation and Combination of Automatically Generated Proofs*, volume 3 of *Applied Logic Series*, pp. 175–192. Kluwer, 1996.

[3] A. Gelb (ed). *Applied Optimal Estimation*. MIT Press, 1974.

[4] C. C. Green. Application of theorem proving to problem solving. In *Proc. IJCAI*, pages 219–240, 1969.

[5] T. Hallgren and A. Ranta. An extensible proof text editor. In *Proc. LPAR'2000*, volume 1955 of *LNAI*, pages 70–84. Springer, 2000.

[6] M. Kay. *XSLT Programmer's Reference*. Wrox Press, 2000.

[7] Z. Manna and R. Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, 1994.

[8] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In *Proc. CADE 12*, pages 341–355, Springer, 1994.

[9] J. van Baalen, P. Robinson, M. Lowry, and T. Pressburger. Explaining synthesized software. In *Proc. ASE'98*, pages 240–248. IEEE, 1998.