# AUTOMATIC DOMAIN-ORIENTED SOFTWARE DESIGN USING FORMAL METHODS

**Thomas T. Pressburger**
**Michael R. Lowry**
Recom Technologies
NASA Ames Research Center
Moffett Field, California

## ABSTRACT

This paper describes a formal approach to automating domain-oriented software design. The formal approach ensures that a user's problem specification is correctly implemented, given a validated domain theory. A declarative domain theory defines the semantics of a domain-oriented specification language and its relationship to implementation-level subroutines. Formal specification development and reuse is made accessible through an intuitive graphical interface that incorporates the advantages of structured editors and visual programming environments at the specification level.

This approach has been implemented in AMPHION, a generic KBSE system that automates component-based programming from specifications. AMPHION has been applied to the domain of solar system kinematics; other NASA applications are under development. AMPHION enables space scientists to develop, modify, and reuse specifications an order of magnitude more rapidly than manual program development. Program synthesis is efficient and completely automatic. Both sequential and iterative programs are synthesized.

## INTRODUCTION

This paper describes AMPHION[1]: an automated software design tool. Like corresponding mechanical or VLSI design tools, AMPHION is based on the underlying mathematics of its design domain, i.e. the mathematics behind software. "Formal methods" are the application of mathematical techniques to software engineering. A classical use of formal methods is to prove that a program correctly implements its specification. This requires a mathematical theory for the programming language, a mathematical theory for the specification language, and a mathematical theory of correct implementations. When these ingredients are available, then the statistical assurance derived through empirical software testing can be replaced by the mathematical certainty derived through logical proof.

Formal methods are attracting substantial interest from customers requiring high-assurance software. However, there is often a high price to pay: the mathematical techniques in formal methods require advanced training and take considerably more resources than the largely informal techniques usually employed in software engineering. Thus a long-term research goal has been the automation of formal methods.

In fact, not only could automated formal methods be used to verify existing software, but in principle they could be used to synthesize new software, just as automatic logic synthesis is used in digital VLSI design. Instead of proving that an existing program correctly implements its specification, one method of program synthesis involves proving that there exists an output that satisfies the specification for every valid input. One type of proof – a constructive proof – entails constructing an expression, parameterized on the input, that describes the output. When this constructed expression is restricted to the terms of a programming language, then this expression is itself a correct program. The constructive proof approach to program synthesis is called *deductive synthesis;* a tutorial article can be found in Manna and Waldinger [4].

Automatic deductive synthesis would seem to solve the programming problem. Frameworks for automatic deductive synthesis were developed over twenty-five years ago (see Green [9] and Manna and Waldinger [4]). However, there are still numerous difficulties, both from a technical viewpoint and from a human interaction viewpoint. Up to now, automatic deductive synthesis has had little impact on real software engineering practice.

This paper describes an application of automatic deductive synthesis that overcomes these difficulties for component-based

1. AMPHION was Zeus's son who used his magic lyre to charm the stones around Thebes into position to form the city's walls.

software engineering. In component-based software engineering, the implementation language is not restricted to the primitives of a programming language; the implementation language also includes pre-existing components from a library. These components typically embed much of the algorithmic complexity in a software application, thus making software development easier – for both humans and machines. Component-based programming also typically has a strong application-domain orientation. This means that the specification language can be tailored to a particular application domain, thus overcoming some of the human interaction difficulties previously associated with general-purpose specification languages. Applying automatic deductive synthesis to component-based programming also complements other applications of formal methods to high-assurance software. In particular, manual or interactive formal methods could be applied to verifying or generating the component library itself. These proofs, which can be more complex than is feasible to generate completely automatically, are then reused as lemmas when the automatic system deductively synthesizes a program that calls these components.

Previous papers have described the domain theory structure required in an AMPHION application [3], the user interface [3], the mathematical basis for AMPHION's program synthesis [8], and the mechanisms for making this synthesis efficient, along with empirical timing results of the program synthesis subsystem in AMPHION [2]. The next section of this paper presents an overview of AMPHION, so refer to these other papers for more details.

For component-based programming, many problems only require simple data-flow composition of components, as described in previous papers. However, some problems require more complicated composition of components. The Iterative Program Synthesis section of this paper describes an extension of AMPHION's previous capabilities that generates iterative programs. It is based on the concept of higher-order components, typically implemented as iterative drivers, that take other components as arguments. The example used throughout this third section is related to planning observations of the impact sites on Jupiter of the Shoemaker-Levy comet fragments.

## AMPHION OVERVIEW

AMPHION is a generic architecture that is specialized to a particular domain and component library through a domain theory and domain-specific theorem-proving tactics. As the first application domain for AMPHION, solar system kinematics was chosen, as implemented in the SPICELIB subroutine library developed by the Navigation Ancillary Information Facility (NAIF) at NASA's Jet Propulsion Laboratory (JPL). NAIF is charged with developing software to support planning and data analysis for interplanetary scientific missions; in particular, software that computes the geometric information needed to plan and interpret observations. SPICELIB functionality includes access to ephemeris data (position and velocity of solar system objects as a function of time) and extensive routines for analytic geometry. AMPHION prototypes for other NASA application domains are under development, in particular the domain of numerical aerodynamic simulation and the domain of space shuttle flight planning.

The objective of AMPHION is to enable users familiar with the basic concepts of an application domain to program at the level of abstract domain-oriented problem specifications. This objective is similar to that for application generators. Like other knowledge-based approaches, AMPHION's automated reasoning techniques are more powerful than the limited compiler-based techniques used in application generators. This enables the specification language to be more abstract, and hence at a further conceptual distance from the details of the implementation language. In contrast to previous automated knowledge-based approaches to domain-specific software design tools, AMPHION is unique in being based on formal specifications and formal methods.

Formal specifications provide an abstract and unambiguous representation of a user's requirements. Formal methods ensure that a program is a correct implementation of a formal specification. AMPHION addresses several difficulties that have impeded formal frameworks being used in practice:

First, users without a background in formal mathematics find that developing a formal problem specification is usually more difficult than developing code manually. In part, this is due to the need to formalize the domain concepts necessary to state a problem. Our approach separates the activity of domain formalization from the activity of individual problem formalization.

Second, users are also unaccustomed to the syntax and notation of mathematical logic, which necessarily underlies any formal specification language. AMPHION incorporates techniques from visual programming and structured editing to guide users in creating domain-oriented diagrams that are then translated into formal specifications. AMPHION also includes a number of effective knowledge-based mechanisms, all driven by a declarative domain theory, that aid a user in formulating a problem while ensuring that the resulting specification is valid [3].

Third, program synthesis must be totally automatic for users without an extensive background in formal methods. The combinatorial explosion inherent in automated reasoning for general purpose program synthesis has prevented completely automatic deductive program synthesis. AMPHION avoids this combinatorial explosion through automatic theorem proving tactics suitable for the specialized task of composing subroutines [2].

### AMPHION User Experience

AMPHION is more than a research prototype: it has already undergone substantial testing with planetary scientists over a period of six months and is currently undergoing further enhancements in preparation for distribution to the large NAIF user community. The specification acquisition component is easy to learn: users are able to develop their own specifications after only an hour's tutorial.

Observations over six months indicate at least an order of magnitude improvement for specification development over manual program development. Programs which would take the better part of a day to develop for someone only casually familiar with the subroutine library can be specified in fifteen minutes after the tutorial introduction to AMPHION. Experienced AMPHION users can develop specifications in five minutes for programs that would take the subroutine library developers an hour to code manually.

AMPHION's program synthesis component is robust and effi-

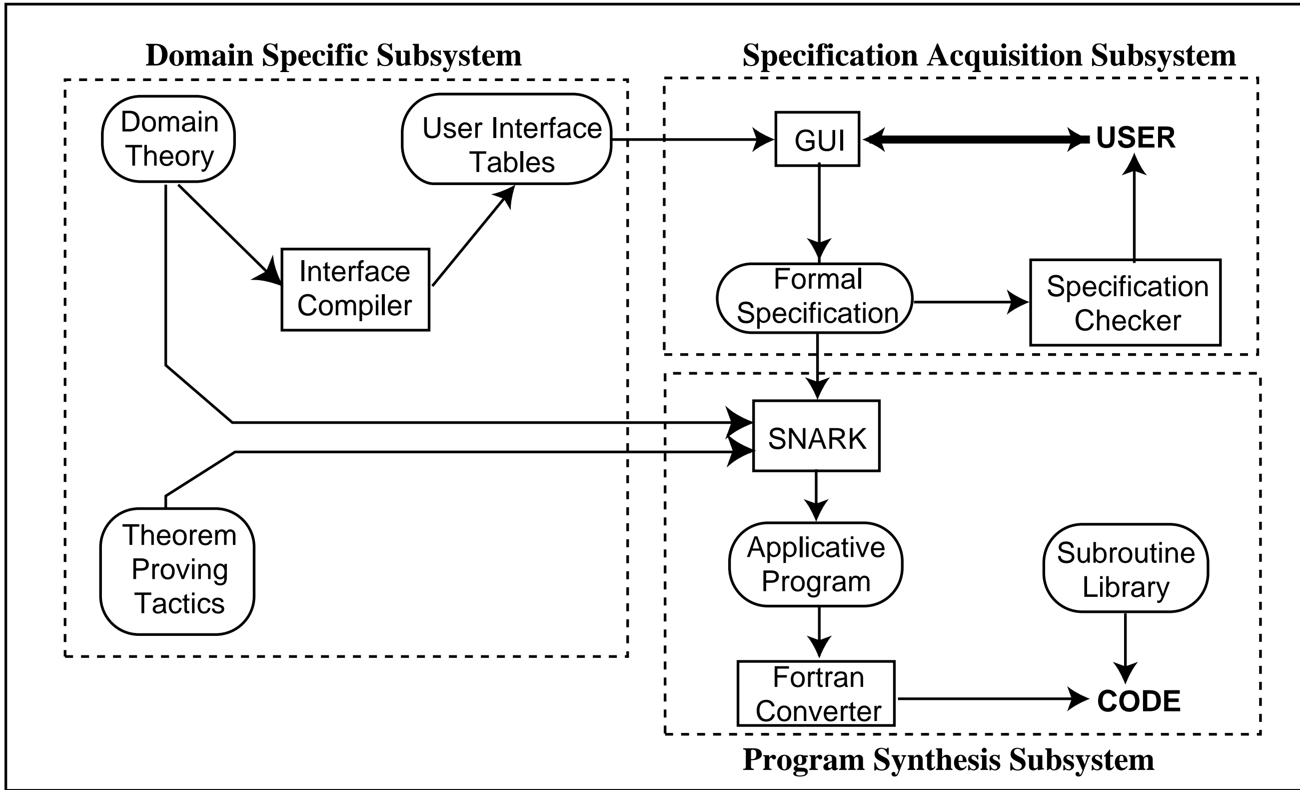


FIGURE 1: AMPHION FLOW DIAGRAM

cient, and appears to be the first use in practice of totally automatic deductive program synthesis. AMPHION synthesizes, from specifications, one- to two-page FORTRAN77 programs consisting of one- to three-dozen calls to SPICELIB subroutines in just a few minutes. In over a hundred programs generated by AMPHION to date for the NAIF domain, the CPU time to synthesize a program never exceeded five minutes of CPU time.

## AMPHION Architecture Overview

Figure 1 presents a flow diagram of AMPHION, where the dotted lines enclose subsystems, the rectangles enclose major components, and the rounded boxes enclose data. AMPHION is applied to a new domain by defining a domain theory and theorem-proving tactics. The domain theory is automatically translated into tables that drive the graphical user interface. The domain theory together with the theorem proving tactics are used by the SNARK theorem prover (see Stickel, et. al. [8]) both to check a specification and also to generate an applicative version of the program. These three sources of information — the domain theory, derived user interface tables, and theorem proving tactics — constitute the domain specific subsystem of an AMPHION application.

The graphical user interface and the specification checker constitute the specification acquisition subsystem. AMPHION enables a user to interactively build a diagram representing a formal problem specification, such as Figure 2. A diagram is an alternate surface syntax for a formal problem specification stated in mathematical logic with the predicates and functions from the domain theory. AMPHION checks a specification by attempting to solve an abstracted version of the problem. If AMPHION cannot solve the abstracted problem, it employs heuristics to localize the problem in the specification and give the user appropriate feedback. For example, if an output or intermediate variable cannot be solved in terms of the input variables, then that variable is underconstrained.

The program synthesis subsystem consists of: (1) a theorem-prover based generator of an applicative program; and (2) a converter of the applicative program into the target programming language (e.g., FORTRAN77 for the JPL SPICELIB subroutine library). After a valid specification is developed, it is converted into a theorem to be proved. The input variables of the specification are universally quantified and the output variables are existentially quantified within the scope of the input variables. The English schematic for these theorems is: “For all *valid* inputs, there is an output satisfying the *input-output relation*.” In any particular theorem, *valid* and *input-output relation* would be definitions for the particular problems in the specification language.

An applicative program is synthesized through constructive theorem proving [4]. Conceptually, an applicative program is equivalent to a data flow graph. During a proof, substitutions are generated for the output variables through unification and equality replacement. The substitutions for the output variables are constrained to be terms in the applicative target language whose function symbols correspond to the subroutines in a library. Unification is a type of two-way pattern matching used in theorem proving. For example, if part of a problem specification matches

the specification of a component, then unification would match them together and replace the output variable with an expression, part of which would be a subexpression containing the function symbol corresponding to the component. Equality replacement is particularly useful in rewriting abstract specification-level constructs into implementation-level constructs closer to the component level.

The expressions for the output variables are then translated into the output programming language through program transformations written in REFINE™ [6]. One set of transformations introduces a variable to hold the value of each subexpression, thus "flattening" the structure of an expression into a sequence of function applications to variables. This also has the effect of combining common subexpressions. Another set of transformations handles subroutines with multiple outputs. Only the very last stage of the translation is programming-language specific: variable declarations and the sequence of subroutine calls are generated in the syntax of the target language. So it is not difficult to retarget AMPHION to a new programming language. For example, in applying AMPHION to the shuttle flight design domain, we were able to generate C++ programs in a few days.

## ITERATIVE PROGRAM SYNTHESIS

The straight-line composition of SPICE subroutines provides a basic capability for solving NAIF domain problems that don't require search. These problems often have the form "Does situation *S* occur at time *t*" or "What is the value of *f* at time *t*". See [2,8] for a description of AMPHION's capabilities for generating straight-line code. However, space scientists frequently want solutions to 'when' questions, such as "When does situation *S* occur in time interval $[t_1, t_2]$ " or "When does *f* take on a minimum value in the time interval $[t_1, t_2]$ ". These problems require iterative search techniques. This section describes an extension to our approach for component-based automatic programming that synthesizes iterative search code. The basic idea is to extend the composition obtained through data flow connections with the composition obtained through nesting one operation within another operation. To be precise, we extend the first-order operations which only take data values as inputs with second-order operations called *functionals* which can take other operations as inputs. These functionals are implemented as iterative drivers that repeatedly call the first-order operations.

The next subsection describes the second-order operations for the NAIF domain. The following subsection describes the diagram, logic, and FORTRAN77 representations for applying a second-order operation to a first-order operation. The following subsection describes the transformations and automated inference techniques which synthesize component-based Fortran implementations given the graphical specification developed by an end-user. The last subsection describes the program synthesis and domain knowledge needed to extend these techniques to produce more robust and efficient code.

### PERCY: Functionals for Iterative Search

Our approach of composition-based program synthesis starts with components developed and validated by experts. The JPL NAIF group developed a set of iterative FORTRAN77 drivers called PERCY that find the set of times when a particular situation occurs. A situation is either:

1. A predicate holding true - for example, an angle being within a specified range.
2. A function achieving (or approaching) an absolute (or relative) extrema - for example, the distance between a planet and one of its moons reaching a local minima.

PERCY represents a set of times by the *window* datatype, which is a set of non-overlapping closed time intervals:

$$\{ [t_1, t_2], \ldots, [t_{2n-1}, t_{2n}] \} \text{ where } t_1 \le t_2 < \ldots < t_{2n-1} \le t_{2n} .$$

(A point in time is represented by an interval with identical endpoints, e.g., $t_1 = t_2$.) The search operations over windows are the PERCY functionals; different functionals correspond to different situation types. PERCY also provides basic set-theoretic operations for manipulating windows such as intersecting two windows and deciding whether one window is a sub-window of another window.

PERCY was developed under contract to the Space Telescope Science Institute as a tool used in scheduling observations by the Hubble Space Telescope (HST). The NAIF group also built a command-line interface to PERCY called MOSS (Moving Object Support System, the name referring to the fact that only solar system objects change position in astronomical observations against the fixed background of the stars). A MOSS user specifies an input window, a situation type, and one of a set of predefined predicates or functions (about two dozen); such as eclipse, occult, distance, or angle separation. The result of the command execution is another window. This window could be used directly for scheduling HST observations, or could be given as an input window to another MOSS command. Typically, a MOSS user starts a session with an initial window consisting of a single time interval, and then refines this window with successive searches to find good observation times. During this refinement process the MOSS user can also apply an operation such as window intersection, or the discarding of time intervals shorter than the minimum time needed to set up an HST observation.

To carry out this refinement process, a MOSS user may need substantial mathematical and astronomical knowledge to reformulate a problem into an efficient sequence of search commands; in particular, because each command is restricted to the predefined functions and predicates. Extending MOSS with a new predefined function or predicate would be a laborious process requiring many changes to the code in the MOSS system. AMPHION also provides access to the PERCY routines, but, in contrast to MOSS, additionally provides the capability to automatically generate code for arbitrary functions and predicates called by the functionals. This by itself makes the refinement process more flexible for generating good windows for HST scheduling and other NAIF applications. Future work, described in the section on Knowledge-Based Program Optimization, will include program synthesis techniques that incorporate domain knowledge in order

to completely automate the window refinement process.

## Diagram, Logic, and Fortran Representations

This subsection describes the representations, particularly for functionals and functional applications, at the diagram specification level, at the logic specification level, and at the Fortran program level generated by AMPHION. The example problem used throughout is for planning observations by the Galileo spacecraft of impact sites on Jupiter, such as those created by the Shoemaker-Levy comet in July of 1994. The problem is to find the times within a given time-interval when the angle of sunlight at the point on Jupiter below the spacecraft is within a specified range. The idea is that a low angle of sunlight makes for long, high-relief shadows of the vertical structures of an impact site and its associated atmospheric disturbances.

**Diagram Representation.** The diagram representation has been extended with functionals corresponding to the situation types provided in PERCY. For example, the functional *local-max-times*, given a window and a real-valued function on time, returns a window denoting a set of points where the function attains a local maximum. Another example is the functional *filter* which, given a window and a predicate on time, returns the subwindow representing those times for which the predicate holds. This paper will concentrate on the *filter* functional.

Figure 2 shows the diagram that specifies the problem of determining good observation times of impact sites on Jupiter. The icons in the diagram represent variables, and the edges in the diagram represent relationships between variables. A user can modify the visual appearance of icons and edges through graphical editors, and generalize these preferences to types of objects and relationships. The convention followed for many of the edges in Figure 2 is that they are directed from a variable to another variable, the latter being defined in terms of the former, and possibly other, variables via the application of a single domain function. The label on the edge denotes the parameter of the domain function being instantiated. For example, in the upper right corner the Boolean variable *Angle-Within-Range?* is defined as the Boolean conjunction of the variables *Angle-Above-Low?* and *Angle-Below-High?* The labels on the directed edges are the parameter names for Boolean conjunction: *conjunct1* and *conjunct2.* In Figure 2, photons, rays, and surface-normals do not follow this convention for edge direction. Note that input and output variables are denoted by chevrons with arrows going in or out. These variables are always attached to a corresponding abstract variable which they represent through a representation (*repn*) function.

The trapezoidal icon *Time-Window-from-Start-to-End* denotes the operation that constructs the initial window from two input variables to the Fortran program, *Time-Coord-Start* and *Time-Coord-End*. The trapezoidal icon *Time-Window-When-Angle-is-in-Range* denotes the *filter* operation that searches through the initial window for the intervals when the predicate holds. The predicate is specified in the subdiagram consisting of the lower half and right half of Figure 2 by defining the relationship of the *independent* variable, *TGalileo*, to the *dependent* variable *Angle-within-Range?*. If the scientist was interested in a program that only computed this predicate for a particular point of time (as opposed to a time window), then the independent variable would be an input to the program and the dependent variable would be the output. Note that the subdiagram has two additional inputs, *Radians-Low* and *Radians-High*; these describe the desirable range of angles, and are inputs to the entire program. Since the functional *filter* only takes a predicate with a single dependent variable, these additional inputs are encapsulated through environment variables in the logic representation of the predicate. These variables also appear in the main FORTRAN77 program that calls the predicate as an entry subroutine. The output specified in the diagram is *Time-Window-Repn-When-Angle-is-in-Range*: it denotes the subwindow of the initial window where the predicate holds true.

**Logic Representation.** The logic representation created by AMPHION for the specification diagram in Figure 2 is shown in Figure 3. The logic representation makes use of first-order logic and the *lambda calculus,* which is a mathematical notation devised for complex function expressions such as functionals, functional applications, and locally-defined functions without a global name. In this notation *lambda* is used for binding input variables, while *find* is used for binding output variables. *Exists* is used for binding variables which are neither input nor output—it is the standard existential quantifier of first-order logic. Specification diagrams are equivalent to logic representations of the following form.

*lambda (inputs)*
  *find (outputs)*
    *exists (intermediates)*
      *conjunct1 & .. & conjunctN*

Each conjunct is either a constraint, $P(v_1, \ldots, v_m)$, or an equality defining a variable through a function application, $v_k = f(v_1, \ldots, v_k)$.

An application of the filter functional to a window and a predicate is represented by an equality with the following form in the lambda calculus:

$$\text{window}_{\text{out}} = \mathit{filter}(\text{window}_{\text{in}},\ \mathit{lambda}\ (t)\ \mathit{find}\ (x_{\text{dep}})\ \mathit{exists}\ (\overrightarrow{x_\exists})\quad P\left(t, x_{\text{dep}}, \overrightarrow{x_\exists}, \overrightarrow{x_{\text{env}}}\right)$$

The embedded *lambda* expression denotes a locally-defined function whose input is *t*, and whose output is the boolean $x_{\text{dep}}$. Its form is the same as the logic representation for an entire specification diagram. The predicate *P* is itself a set of conjuncts. Note that the predicate *P* may depend not only on the variables *t*, $x_{\text{dep}}$, and existentially quantified variables $\overrightarrow{x_\exists}$; but also on the *environment* in which the embedded *lambda* expression is situated. In logic, the environment of a subexpression is the set of variables which are bound in expressions that contain the subexpression. The environment $\overrightarrow{x_{\text{env}}}$ of the embedded *lambda* expression above includes the input variables to the problem, *inputs;* the output variables of the problem, *outputs*; and also the existential variables, *intermediates.* In the Galileo specification, these embedded

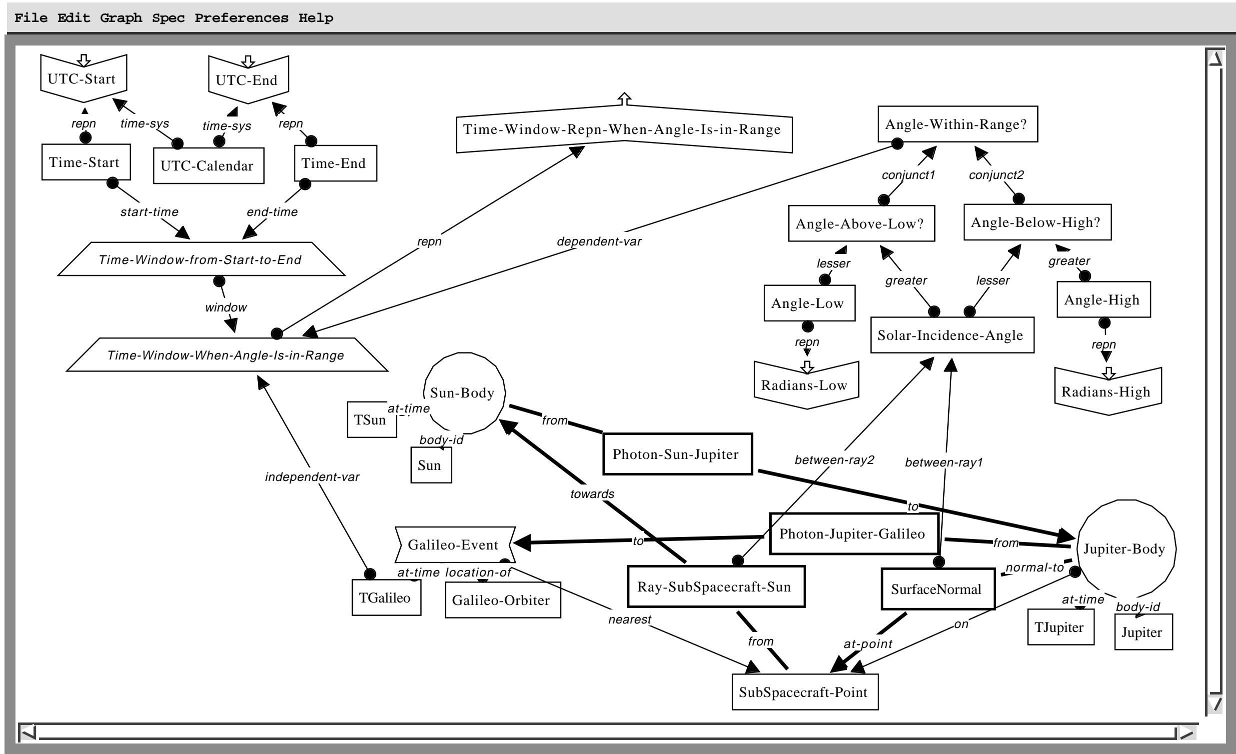


FIGURE 2: DIAGRAM OF GALILEO PROBLEM CREATED INTERACTIVELY WITH AMPHION.

variables include the inputs *Radians-Low* and *Radians-High.*

Some programming languages such as Lisp (which was modeled after the lambda calculus), provide *lexical closures* [1] for constructing locally-defined, unnamed functions whose value depends on variables in their environment. The next subsection describes how to implement a closure in FORTRAN77 for the PERCY functionals; analogous techniques would be used for C and similar programming languages.

**Fortran77 Representation.** The final FORTRAN77 program, `FNDTMS` is shown in Figure 4. The input and output variable names are transformed from those in the diagram and logic representation to be no more than six characters. The body of `FNDTMS` first declares all the variables required for executing the program, including the `ENTRY` subroutine `SOLARI` corresponding to the embedded lambda. In essence, all the environment variables for the embedded lambda are promoted into the main routine. `FNDTMS` then calls PERCY window routines in order to set up the initial window. The PERCY subroutine that implements the *filter* functional is called `PRSOLV`. Its input arguments are a window and a subroutine. This subroutine in turn must take a time as input and return a logical as output. `PRSOLV` itself returns as output the subwindow consisting of the time intervals for which the subroutine returns logical true. The behavior of `PRSOLV` is also affected by two other parameters: the size of the time step used to sample the window looking for changes between logical true and logical false, and the desired accuracy tolerance for the location of the endpoints of the time intervals. These will be discussed further in the section on Knowledge-Based Program Optimization.

The *lambda* expression embedded in the *filter* operation is represented in FORTRAN77 as an `ENTRY` subroutine with a single input, a time, and a single output, a Fortran logical. The `ENTRY` subroutine `SOLARI` implements the predicate that determines whether the incidence angle is in the range between `RLOW` and `RHIGH` at a given time. `SOLARI` references variables `RLOW` and `RHIGH` which are set by the main routine.

## Program Synthesis

The first step in the synthesis of a program from the specification diagram is translating the diagram into its logic representation. The translation of an icon that denotes a definition of a

```
(LAMBDA (UTC-End UTC-Start Radians-Low Radians-High)
   (FIND (Time-Window-Repn-When-Angle-Is-In-Range)
      (EXISTS (Time-Start Time-End Time-Window-from-Start-to-End Time-Window-When-Angle-Is-in-Range)
         (AND (= Time-Window-Repn-When-Angle-Is-In-Range
                 (PERCY-TIME-WINDOW-REPN (TIME-WINDOW Time-Window-When-Angle-Is-in-Range)))
              (= UTC-Start
                 (TIME-TO-COORDINATES (TIME Time-Start) (TIME-SYSTEM UTC-Calendar)))
              (= UTC-End
                 (TIME-TO-COORDINATES (TIME Time-End) (TIME-SYSTEM UTC-Calendar)))
              (= Time-Window-from-Start-to-End
                 (WINDOW-CREATE (TIME Time-Start) (TIME Time-End)))
              (= Time-Window-When-Angle-Is-in-Range
                 (FILTER
                    (TIME-WINDOW Time-Window-from-Start-to-End)
                    (LAMBDA (TGalileo)
                       (FIND (Angle-Within-Range?)
                          (EXISTS (Angle-Low Angle-Above-Low? Angle-High Angle-Below-High?
                                   Ray-SubSpacecraft-Sun TJupiter TSun
                                   Photon-Sun-Jupiter Photon-Jupiter-Galileo Jupiter-Body Sun-Body
                                   Galileo-Event SubSpacecraft-Point SurfaceNormal Solar-Incidence-Angle)
                             (AND (= Radians-Low
                                     (ANGLE-TO-RADIANS (ANGLE Angle-Low)))
                                  (= Angle-Above-Low?
                                     (ANGLE-COMPARISON (ANGLE Angle-Low) (ANGLE Solar-Incidence-Angle)))
                                  (= Ray-SubSpacecraft-Sun
                                     (TWO-POINTS-TO-RAY (POINT SubSpacecraft-Point) (BODY Sun-Body)))
                                  (= Sun-Body
                                     (BODY-ID-AND-TIME-TO-BODY (BODY-ID Sun) (TIME TSun)))
                                  (= Photon-Sun-Jupiter
                                     (TWO-EVENTS-TO-PHOTON (BODY Sun-Body) (BODY Jupiter-Body)))
                                  (= Jupiter-Body
                                     (BODY-ID-AND-TIME-TO-BODY (BODY-ID Jupiter) (TIME TJupiter)))
                                  (= Photon-Jupiter-Galileo
                                     (TWO-EVENTS-TO-PHOTON (BODY Jupiter-Body) (EVENT Galileo-Event)))
                                  (= Galileo-Event
                                     (EPHEMERIS-OBJECT-AND-TIME-TO-EVENT
                                        (SPACECRAFT-ID Galileo-Orbiter) (TIME TGalileo)))
                                  (= SubSpacecraft-Point
                                     (ELLIPSOID-POINT-NEAREST-POINT
                                        (BODY Jupiter-Body) (EVENT Galileo-Event)))
                                  (= SurfaceNormal
                                     (SURFACE-NORMAL-RAY (BODY Jupiter-Body) (POINT SubSpacecraft-Point)))
                                  (= Solar-Incidence-Angle
                                     (TWO-RAYS-TO-ANGLE (RAY SurfaceNormal) (RAY Ray-SubSpacecraft-Sun)))
                                  (= Radians-High
                                     (ANGLE-TO-RADIANS (ANGLE Angle-High)))
                                  (= Angle-Below-High?
                                     (ANGLE-COMPARISON (ANGLE Solar-Incidence-Angle) (ANGLE Angle-High)))
                                  (= Angle-Within-Range?
                                     (BOOLEAN-AND
                                        (BOOLEAN Angle-Above-Low?) (BOOLEAN Angle-Below-High?)))))))))))))
```

FIGURE 3: AMPHION TRANSLATION INTO LOGIC OF FIGURE 2.

```
      SUBROUTINE FNDTMS (UTSTR, UTEND, RLOW, RHIGH, WNAIR)
C     Input Parameters
      DOUBLE PRECISION UTCSTR
      DOUBLE PRECISION UTCEND
      DOUBLE PRECISION RLOW
      DOUBLE PRECISION RHIGH
C     Output Parameters
      INTEGER LBCELL
      PARAMETER (LBCELL = -5)
      INTEGER WNSIZE
      PARAMETER (WNSIZE = 100)
      DOUBLE PRECISION WNAIR (LBCELL:WNSIZE)
C     Variable Declarations
      DOUBLE PRECISION ETSTR
      DOUBLE PRECISION ETEND
      DOUBLE PRECISION WNINIT (LBCELL:WNSIZE)

C     Declarations for ENTRY subroutine SOLARI
C     Input Parameters
      DOUBLE PRECISION ET
C     Output parameters
      LOGICAL ANWIRA
      Function Declarations
      DOUBLE PRECISION VSEP
C     Parameter Declarations
      INTEGER JUPITE
      PARAMETER (JUPITE = 599)
      INTEGER GALIL1
      PARAMETER (GALIL1 = -77)
      INTEGER SUN
      PARAMETER (SUN = 10)
C     Variable Declarations
      DOUBLE PRECISION ANGLEI
      DOUBLE PRECISION RADJUP ( 3 )
      DOUBLE PRECISION PVGALI ( 6 )
      DOUBLE PRECISION LTJUGA
      DOUBLE PRECISION V1 ( 3 )
      DOUBLE PRECISION X
      DOUBLE PRECISION PVJUPI ( 6 )
      DOUBLE PRECISION LTSUJU
      DOUBLE PRECISION MJUPIT ( 3, 3 )
      DOUBLE PRECISION V2 ( 3 )
      DOUBLE PRECISION X1
      DOUBLE PRECISION DV2V1 ( 3 )
      DOUBLE PRECISION PVSUN ( 6 )
      DOUBLE PRECISION XDV2V1 ( 3 )
      DOUBLE PRECISION V ( 3 )
      DOUBLE PRECISION N ( 3 )
      DOUBLE PRECISION PN ( 3 )
      DOUBLE PRECISION DV2N ( 3 )
      DOUBLE PRECISION XDV2N ( 3 )
      DOUBLE PRECISION DXDV2V ( 3 )
      DOUBLE PRECISION XDXDV2 ( 3 )
C     Dummy Variable Declarations
                  (omitted)

      CALL SSIZED ( WNSIZE, WNAIR )
      CALL SCARDD ( 0, WNDAIR )
      CALL SSIZED ( WNSIZE, WNINIT )
      CALL SCARDD ( 0, WNINIT )
      CALL UTC2ET ( UTCSTR, ETSTR )
      CALL UTC2ET ( UTCEND, ETEND )
      CALL WNINS ( ETSTR, ETEND, WNINIT )
      CALL PRSOLV ( WNINIT, SOLARI, WNAIR )
      RETURN

      ENTRY SOLARI (ET, ANWIRA)
      CALL BODVAR ( JUPITE, 'RADII', DMY10, RADJUP )
      CALL SPKSSB ( GALIL1, ET, 'J2000', PVGALI )
      CALL SPKEZ ( JUPITER, ET, 'J2000', 'NONE', GALIL1,
     .            DMY20, LTJUGA )
      CALL VEQU ( PVGALI ( 1 ), V1 )
      X = E - LTJUGA
      CALL SPKSSB ( JUPITE, X, 'J2000', PVJUPI )
      CALL SPKEZ ( SUN, X, 'J2000', 'NONE', JUPITER,
     .            DMY60, LTSUJU )
      CALL BODMAT ( JUPITE, X, MJUPIT )
      CALL VEQU ( PVJUPI ( 1 ), V2 )
      X1 = X - LTSUJU
      CALL VSUB ( V1, V2, DV2V1 )
      CALL SPKSSB ( SUN, X1, 'J2000', PVSUN )
      CALL MXV ( MJUPIT, DV2V1, XDV2V1 )
      CALL VEQU ( PVSUN ( 1 ), V )
      CALL NEARPT ( XDV2V1, RADJUP ( 1 ), RADJUP ( 2 ),
     .             RADJUP ( 3 ), N, DMY130)
      CALL SURFNM ( RADJUP ( 1 ), RADJUP ( 2 ),
     .             RADJUP ( 3 ), N, PN )
      CALL VSUB ( N, V2, DV2N )
      CALL MTXV ( MJUPIT, DV2N, XDV2N )
      CALL VSUB ( V, XDV2N, DXDV2V )
      CALL MXV ( MJUPIT, DXDV2V, XDXDV2 )
      ANGLEI = VSEP ( XDXDV2, PN )
      ANWIRA = ANGLEI .GT. RLOW .AND. ANGLEI .LT. RHIGH
      RETURN
      END
```

FIGURE 4: FORTRAN77 CODE GENERATED BY AMPHION FROM FIGURE 3.

variable is an equality defining the variable. The translation of an icon defined by a functional requires determining the logic specification of the function or predicate passed to the functional. For example, in Figure 2, it must be determined which equalities and constraints should be included in the predicate passed to the *filter* functional. This corresponds to determining the extent of the subdiagram that defines the dependent variable *Angle-within-Range?*. In Figure 2 this subdiagram is everything between the edges labeled *independent-var* and *dependent-var*.

The definition of the function or predicate is determined by finding the constraints and variables that define the dependent variable, and then the constraints and variables that define those variables, and so on recursively. A well-defined diagram is required to have no cycles in its definitions or constraints; this requirement is enforced by the user interface. This requirement ensures the termination of the computation for finding the subdiagram for a function or predicate.

The second synthesis step is to deductively synthesize an implementation given the logic specification. The deductive framework for synthesizing implementations from logical specifications [2,7] works, in principle, the same for simple specifications as for specifications that include functionals and *lambda* expressions. However, functionals and *lambda* expressions are second-order constructs, hence requiring second-order theorem proving for the deductive synthesis. While second-order theorem provers can be used in interactive theorem proving where they are guided by a human expert, they are unsuitable for automatic theorem proving because the search spaces are too large. Hence AMPHION uses a first-order theorem prover. Thus in AMPHION a logic specification is first transformed into an equivalent first-order form. Embedded *lambda* expressions are transformed into named functions that are deductively synthesized separately. The functionals are treated as first-order functions that take named functions as arguments; in the deductive synthesis of the main program these named functions are treated as first-order objects. Transforming embedded *lambda* expressions into named functions is called *closure conversion* in Appel [1]. In AMPHION, where all variables have distinct names, and such a function is only passed downward into an environment containing the environment of the lambda expression, the function can be implemented by an `ENTRY` subroutine in the scope of the variables in the main subroutine. An example is the subroutine `SOLARI` in Figure 4.

During synthesis, representations must be chosen for the input variable of the *lambda* expression. Note that the diagram in Figure 2 does not explicitly state the representation of TGalileo. The current implementation of AMPHION chooses a standard representation for time called “ephemeris time”. This choice is appropriate because most of the SPICE routines with an input time parameter require the “ephemeris” representation. For those few SPICE routines that require other time representations, SPICE provides representation conversion functions from ephemeris time that AMPHION will automatically insert. The ephemeris representation is also amenable to the arithmetic operations used in the `PRSOLV` algorithm, e.g. increment by a time step and find the midpoint between two times. These operations would be difficult in some of the other representations, e.g. character string representations of time.

## Knowledge-Based Program Optimization

This section describes future work in an area of program synthesis that requires combining domain, mathematical, and algorithmic knowledge. The goal is to apply various sources of knowledge to the problem of producing more efficient and robust programs. This section describes some of that knowledge for NAIF domain problems similar to scheduling observation times for the Hubble Space Telescope. The research issue is the extent to which the application of this knowledge can be automated. Some user guidance may be required, which raises the research issue of human-computer interaction during program synthesis.

MOSS provides window operations that, if called inappropriately, can be very inefficient or produce incorrect results. A MOSS user employs knowledge about astronomy, mathematics, and the algorithms underlying PERCY to develop a sequence of commands that invoke these operations appropriately. This section describes some of the knowledge necessary to invoke `PRSOLV` effectively. This requires some understanding of the algorithm employed by `PRSOLV`.

The algorithm employed by `PRSOLV` is as follows. `PRSOLV` searches each interval in the input window. The algorithm starts at the left endpoint of an interval in the window and repeatedly increments by a time step $\delta$, testing for a transition in the value of the predicate. When a transition is found, binary search is used to locate the time of the transition to the specified accuracy. `PRSOLV` returns a subwindow of the input window.

The behavior of `PRSOLV` is dependent on the time step, which is a parameter that can be set. A large time step will allow `PRSOLV` to skip through the window rapidly. However, too large a time step will allow `PRSOLV` to skip over a transition, yielding incorrect results. For this reason, `PRSOLV` is guaranteed to find all of the transitions in a window only if the time step is smaller than the smallest time difference between two adjacent transitions occurring within the window. Finding situations of short duration would seem to require a time step shorter than the shortest duration. However, there are several knowledge-based optimization tactics that enable large time steps to be used and still produce accurate results.

A PERCY search can often be made more efficient and robust by taking advantage of set-theoretic reformulations and domain-specific knowledge. As an example, consider an extension of the Galileo problem where it is also required that the Galileo spacecraft be near one of the impact sites. A naive implementation, which might require a small timestep, would search for both a low sunlight angle and a near location. This implementation could be refined into the intersection of the windows found by two searches: one finding the time window when sunlight is low, the other finding the time window when the spacecraft is near a site. A search for a conjunctive predicate can be implemented as a window intersection of two searches as follows:

$$\mathrm{PRSOLV}(w,\ P_1 \mathrm{and} P_2,\ \delta_{\mathrm{step12}}) =$$
$$\mathrm{PRSOLV}(w, P_1, \delta_{\mathrm{step1}}) \cap \mathrm{PRSOLV}(w, P_2, \delta_{\mathrm{step2}})$$

This rule is useful when the transitions of the individual predicates are spaced farther apart than the transitions of their con-

junction. (Consider the case where $P_1$ holds in a long interval that just barely overlaps the long interval in which $P_2$ holds.) If this is the case, larger step sizes may be used in each of the two searches than in the original search.

Another way of optimizing the search employed by PERCY is to translate a search for a short-duration situation to a search for a different situation that occurs at the same time. For example, though the interval when a function is very near some particular value $x$ may be short, the interval when the function is greater than or equal to $x$ may be large, as is the interval when the function is less than or equal to $x$. An astronomical example is in finding the times when a moon is behind a planet, as seen from an observer at a great distance. This may be a very short-duration situation; however, the predicate of the moon being on one side of the planet describes a long-duration situation. The time when the latter predicate makes a transition is also the time the moon is behind the planet (along one dimension).

Another optimization technique that combines general program synthesis knowledge with domain-specific knowledge is to use necessary conditions on a situation to generate a restricted window over which to search for the full situation, as described in Mostow [5] and Smith [7]. Also, for certain constraints, the set of elements satisfying the constraints can be reformulated as a generator of the elements [5]. For example, solar eclipses are of short duration, which seems to necessitate a small time step. However, because a necessary condition for a solar eclipse is that it occurs during a new moon, and because new moons occur every 29 days, the search for a solar eclipse can be restricted to set of short intervals generated 29 days apart, reducing the search by a factor of 29.

## CONCLUSION

This paper has described an approach to automated software design founded on the application of formal methods to domain-oriented software composition. Difficulties previously associated with automating formal methods and using them with naive users have been addressed. In particular, it was found that a diagram notation that represents logic statements provides an intuitive specification language. It was also found that a theorem prover could effectively synthesize useful programs given explicit, declarative facts about components in a library. Future work will introduce knowledge from a variety of domains into the optimization process. Prototype AMPHION applications are being developed for other domains, namely space shuttle flight planning and numerical aerodynamic simulation.

## ACKNOWLEDGMENTS

Andrew Philpot implemented the graphical interface in Garnet. Arthur Reyes helped improve the exposition of this paper. Thanks are due to the NAIF group at JPL, and especially to Bill Taber, who created the PERCY and MOSS code with Ian Underwood, and patiently described it to us, and to Chuck Acton, head of the NAIF group, for his advice and for allowing his group to work with us.

## REFERENCES

[1] Appel, A., 1992, *Compiling with Continuations,* Cambridge University Press, Cambridge, Chapters 2.4 and 10.8.

[2] Lowry, M., Philpot, A., Pressburger, T., and Underwood, I., 1994, "A Formal Approach to Domain-Oriented Software Design Environments", in *Proc. 9th Knowledge-Based Software Engineering Conference*, Sept. 20-23, Monterey, California, pp. 2, 48-57.

[3] Lowry, M., Philpot, A., Pressburger, T., and Underwood, I., 1994, "AMPHION: Automatic Programming for Scientific Subroutine Libraries", in *Proc. 8th Intl. Symp. on Methodologies for Intelligent Systems*, Charlotte, North Carolina, October 16-19.

[4] Manna, Z. and Waldinger, R., 1992, "Fundamentals of Deductive Program Synthesis," *IEEE Transactions on Software Engineering*, Vol. 18, No. 8, August 1992, pp. 674-704.

[5] Mostow, J., 1983, "Machine Transformation of Advice into a Heuristic Search Procedure" In *Machine Learning: An Artificial Intelligence Appraoch,* eds. R. S. Michalski, J. Carboneel, and T. Mitchell, Morgan Kaufman, Los Altos, Calif., pp. 367-404.

[6] Reasoning Systems, 1992, *REFINE User's Guide.* Reasoning Systems, Palo Alto, Calif.

[7] Smith, D., 1991,"KIDS—A Knowledge-Based Software Development System", in *Automating Software Design,* Lowry, M. and McCartney, R., eds, AAAI Press, Menlo Park, Calif., 1991, pp. 483-514, see pp. 494.

[8] Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., and Underwood, I., 1994, "Deductive Composition of Astronomical Software from Subroutine Libraries", in *12th Conference on Automated Deduction*, June 28-July 1, Nancy, France.

[9] Green, C., 1969, "Application of Theorem Proving to Problem Solving", in *Proc. Intl. Joint Conf. on Artifical Intelligence,* May 7-9, Washington D.C., pp. 219-240.