A Formal Approach to Domain-Oriented Software Design Environments

Michael Lowry, Andrew Philpot, Thomas Pressburger, and Ian Underwood

Recom Technologies
AI Research Branch, M.S. 269-2
NASA Ames Research Center
Moffett Field, CA 94035

Abstract

This paper describes a formal approach to domain-oriented software design environments, based on declarative domain theories, formal specifications, and deductive program synthesis. 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 to users through an intuitive graphical interface that guides them in creating diagrams denoting formal specifications. Deductive program synthesis ensures that specifications are correctly implemented.

This approach has been implemented in AMPHION, *a generic KBSE system that targets scientific subroutine libraries.* AMPHION *has been applied to the domain of solar system kinematics.* 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.*

1: Introduction

This paper describes AMPHION[1]: an implemented Domain-Oriented Design Environment (DODE). In contrast to previous approaches to DODEs [4], AMPHION is based on formal specifications and automated deductive synthesis for program development. Nonetheless, from a user's viewpoint, AMPHION is similar to previously published accounts of DODEs. The thesis of this paper is that the development, modification, and reuse of problem specifications — not programs— are the central activities around which a domain-oriented KBSE life cycle should revolve. This is in consonance with the original vision for the knowledge-based software assistant [5]. For the high-assurance software that characterizes NASA's needs, basing a DODE on formal specifications offers many advantages, and, as reported in this paper, is eminently feasible.

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 [8]. 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 to DODEs separates the activity of domain formalization from the activity of individual problem formalization.

Users are also unaccustomed to the syntax and notation of mathematical logic. AMPHION incorporates techniques from visual programming and structured editing to guide users in creating domain-oriented diagrams that are 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.

Another impediment to domain-oriented formal frameworks is that 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 theorem proving tactics suitable for the specialized task of composing subroutines.

AMPHION is a generic architecture that is specialized to a particular domain and subroutine 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

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.

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. The objective of SPICELIB is to enable end-users in the planetary science community to construct their own application programs.

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 efficient, and appears to be the first use in practice of totally automatic deductive program synthesis. AMPHION synthesizes, from specifications, one- to two-page 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 was installed at JPL NAIF in December 1993. The technical leader of NAIF, who has no background in formal methods, has demonstrated AMPHION to other groups at JPL, generating considerable interest in applying AMPHION to other domains. Alpha testing of AMPHION is scheduled at other sites starting in late summer of 1994. AMPHION will be used in a scheduling system for the CASSINI mission to Saturn, in order to generate the programs for computing geometric constraints for science observations and Earth communications. Other NASA domains are under investigation, including numerical aerodynamic simulation and space shuttle trajectory planning.

Section 2 describes AMPHION's architecture. Section 3 presents an example of using AMPHION in the domain of solar system kinematics. Section 4 illustrates the structure of a domain theory needed for an AMPHION application. Section 5 describes the mechanisms for guiding an end-user in specification acquisition. Section 6 discusses the techniques for achieving efficient performance from deductive program synthesis, and analyzes timing results from thirty-eight specifications.

2: AMPHION System 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 both to check a specification and also to generate an applicative program. SNARK is a new first order logic (FOL) theorem prover developed at SRI International [12]. 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. To a first approximation, a diagram is an alternate surface syntax for a formal problem specification in FOL augmented with the lambda calculus. *Lambda* is used for binding input variables, while the constructive existential quantifier *find* is used for binding output variables. Diagrams are equivalent to specifications of the following form (more general specifications must currently be entered textually):

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

where each conjunct is either a constraint, *P(v1,..,vm)* , or an equality defining a variable through a function application, *vk = f(v1, .., vm)*.

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 under-constrained.

The program synthesis subsystem consists of a generator of an applicative program and a translator into the target programming language (e.g., Fortran-77 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. An applicative program is synthesized through constructive theorem proving[10]. During a proof, substitutions are generated for the existential variables through unification and equality re-

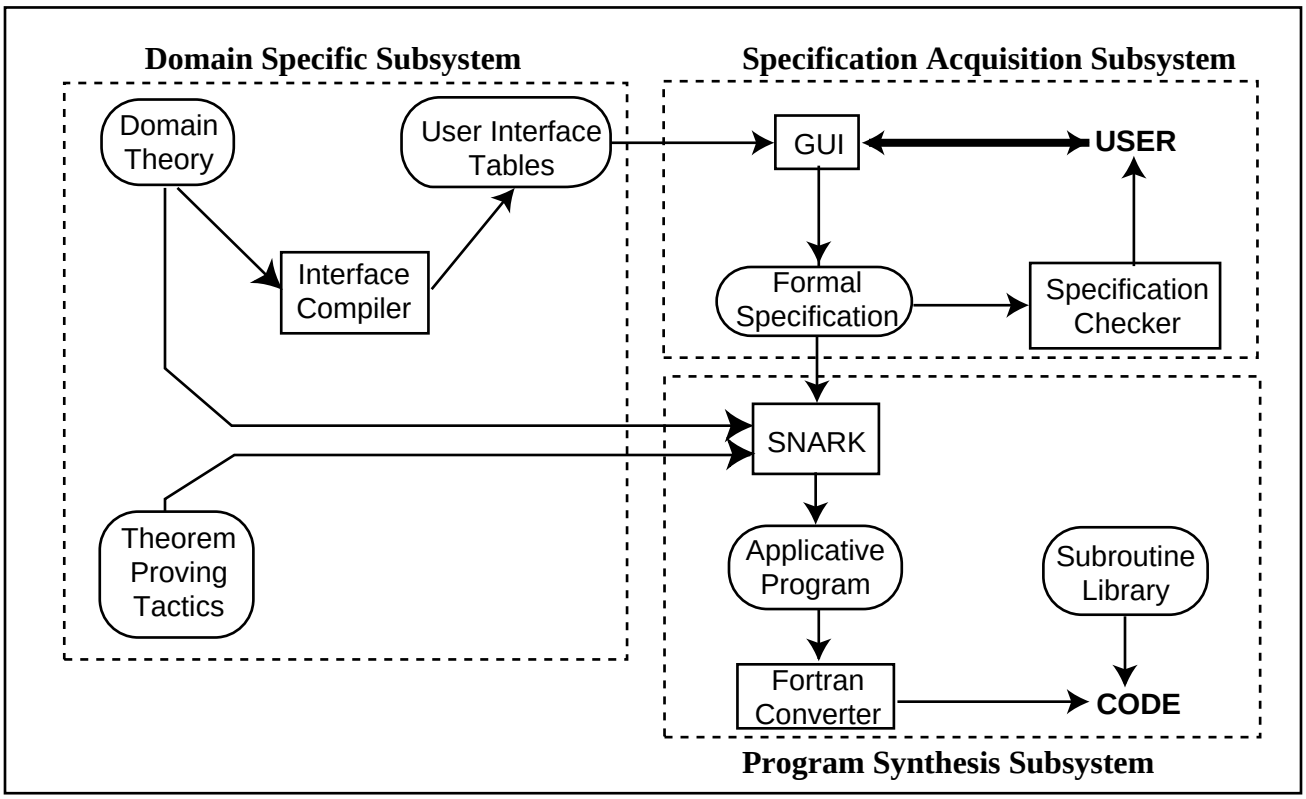

Figure 1: Flow diagram of AMPHION.

placement. 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.

The terms for the output variables are then translated into the output programming language through program transformations written in REFINE™ [11]. One set of transformations turns common subexpressions into lambda-bound variables in nested lambda applications. 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. Approximately two man-weeks of work would be required to output programs in a different target language such as C or UNIX shell scripts.

3: Specification-Based Software Engineering

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, rather than at the detailed level of subroutine calls. Within the scientific programming community, subroutine libraries are a ubiquitous form of software reuse. However, scientists often do not make effective use of libraries. Sometimes this happens because a subroutine library is developed without following good conventional software engineering practices, resulting in inadequate documentation, untrustworthy code, and a lack of coherence in the different functions performed by the individual subroutines.

However, even when a subroutine library is developed following the best conventional software engineering practices, users often have neither the time nor the inclination to fully familiarize themselves with it. The result is that most users lack the expertise to easily identify and assemble the routines appropriate to their problems. This represents an inherent knowledge barrier that lowers the utility of even the best-engineered software libraries: the effort to acquire the knowledge to effectively use a subroutine library is often perceived as being more than the effort to develop the code from scratch. AMPHION is an effective solution to this knowledge barrier.

Despite NAIF's outstanding software engineering practices and the excellent documentation for SPICELIB, few users take the time to study the documentation and become familiar with the full extent of its capabilities. Many simply prevail upon the NAIF group to write programs for them, thereby slowing down the introduction of new features in

SPICELIB. Those users who write their own programs seldom use more than a few of the core routines, such as those for planetary ephemerides (the position and velocities of planets as a function of time). In sampled user programs, there were numerous instances where users developed their own code for functions that already existed within SPICELIB, such as routines for analytic geometry.

3.1: Example Problem

Consider a planetary scientist working on the Galileo mission to Jupiter who wants a program that determines the solar incidence angle at the sub-spacecraft point of Galileo on the surface of Jupiter. The sub-spacecraft point is the point on a planet's surface closest to a spacecraft. The solar incidence angle is the angle between the surface normal and the apparent position of the sun. The solar incidence angle at the sub-spacecraft point would be used to help interpret images and particle/magnetic field data. Without AMPHION this scientist would need to manually code a program, e.g., the SOLAR program in Figure 3.

AMPHION's specification language for the NAIF domain is at the level of abstract geometry augmented with astronomical terms. There is no mention of coordinate frames, units, and so on, except in defining representations for inputs and outputs. Within the domain theory for NAIF, described in section 4, the solar incidence angle problem can be formalized as follows (variable names are in italics):

Let *Solar-Incidence-Angle* be the angle between two rays, *SurfaceNormal* and *Ray-Subspacecraft-Sun*.
Let *Subspacecraft-Point* be the point on *Jupiter-Body* nearest Galileo-Orbiter at time *TGalileo*.
Let *Jupiter-Body* be Jupiter at time *TJupiter*.
Let *Sun-Body* be the Sun at time *TSun*.
Let *Photon-Sun-Jupiter* be a photon from *Sun-Body* to *Jupiter-Body*.
Let *Photon-Jupiter-Galileo* be a photon from *Jupiter-Body* to Galileo-Orbiter arriving at time *TGalileo*.
Let *Ray-Subspacecraft-Sun* be the ray from the point *Subspacecraft-Point* towards *Sun-Body*.
Let *SurfaceNormal* be the ray normal to *Jupiter-Body* at the point *Subspacecraft-Point*.
Let the representation of *Solar-Incidence-Angle*, the output, be in radians.
Let the representation of *TGalileo*, the input, be a string in the format for Galileo's internal clock.

Except for syntax (AMPHION specifications are in a Lisp-like notation; each of the sentences above corresponds directly to an equality defining a variable) this is the specification given to AMPHION's program synthesis subsystem that generates the Fortran-77 program in Figure 3 in 52 seconds of CPU time on a Sparc 2. However, instead of writing textual specifications, users enter specifications graphically through a menu-guided interface, overviewed in Section 5, resulting in diagrams such as Figure 2. (The appearance of icons and arrows can be customized to a user's preference. In general, arrows are directed toward the object defined in terms of objects at the origin of the arrows. The labels on the arrows describe the relationship. Photons and rays are an exception to this convention.) From a user's viewpoint, this interface is similar to the construction kit of Fischer's DODEs. This interface automatically translates completed specification diagrams to the textual form for the program synthesis subsystem described in Section 6.

4: NAIF Domain Theory

There are presently over two hundred axioms in the theory for the NAIF domain. This section presents an overview of its structure, highlighting those aspects likely to be important in creating other AMPHION applications.

An AMPHION domain theory consists of an abstract theory that provides the background knowledge for formulating problems, a concrete theory for formalizing the subroutines, and an implementation relation between the abstract and concrete theory. The domain theory must include implementation details needed to correctly compose subroutines, such as subroutine preconditions and representational assumptions for subroutine input and output parameters. Although the domain theory does not need to include a first-principles axiomatization of the functions and relations in the abstract theory, it does need to include enough semantics to derive an abstract solution consisting of abstract operations from an abstract specification. Given an abstract solution in terms of abstract operations, the implementation relation is used to generate a concrete solution, taking into account subroutine preconditions.

4.1 Abstract Theory

At the abstract level, the NAIF domain theory includes types for objects in Euclidean geometry augmented with astronomical constructs such as photons, planets, and spacecraft. The abstract types exist independent of any particular representation. Abstract functions include constructors for derived types; the abstract relations include geometric predicates, such as whether one geometric object intersects another.

The semantics of functions and relations that correspond to concrete subroutines are defined by the implementation axioms. The semantics of the remaining functions and relations fall into two categories. First are those that are definitions based on other abstract functions and relations. For example, the angle between two planes is defined as the angle between their normals. The second category are non-definitional axioms that mutually constrain abstract func-

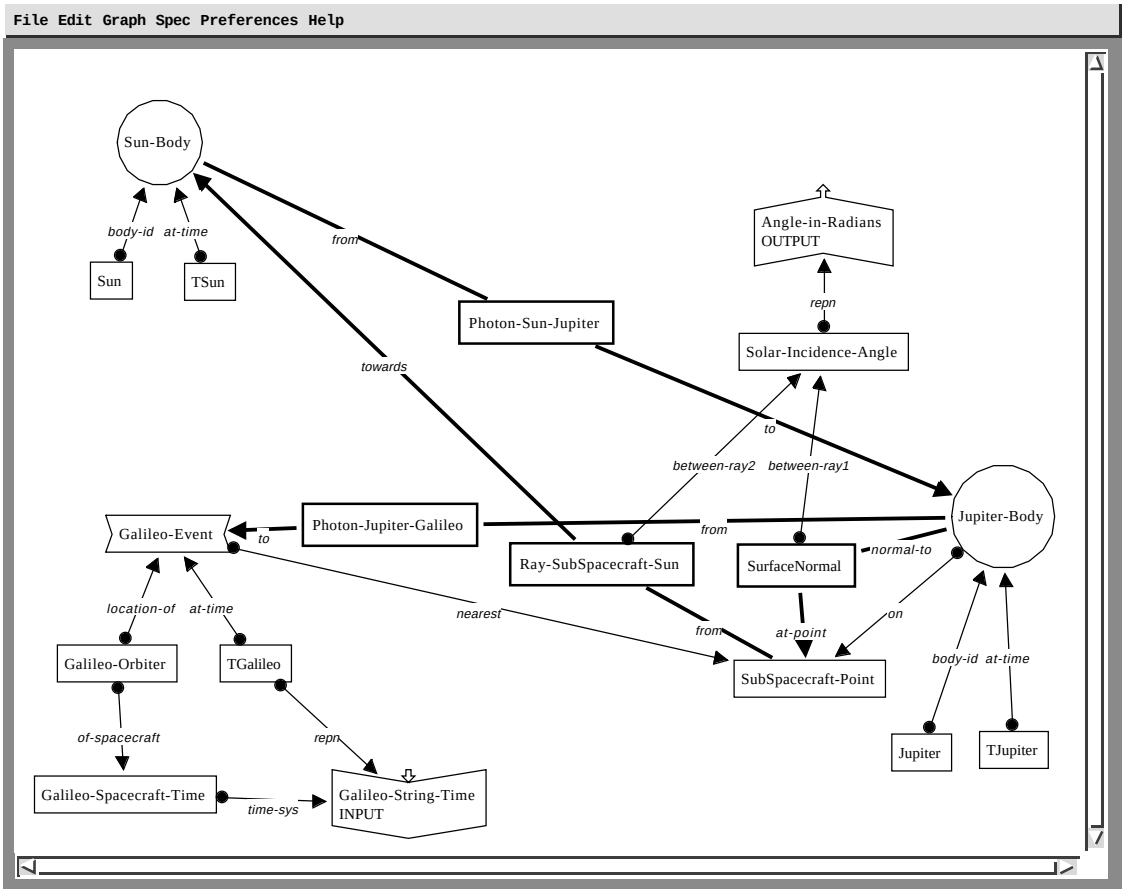

Figure 2: Diagram for solar incidence angle developed interactively with AMPHION.

```
      SUBROUTINE SOLAR ( GALILE, ANGLEI )
C     Input Parameters
      CHARACTER*(*) GALILE
C     Output Parameters
      DOUBLE PRECISION ANGLEI
C     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 RADJUP ( 3 )
      DOUBLE PRECISION E
      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
      INTEGER DMY10
      DOUBLE PRECISION DMY20 ( 6 )
      DOUBLE PRECISION DMY60 ( 6 )
      DOUBLE PRECISION DMY130
      CALL BODVAR ( JUPITE, 'RADII', DMY10, RADJUP )
      CALL SCS2E ( GALIL1, GALILE, E )
      CALL SPKSSB ( GALIL1, E, 'J2000', PVGALI )
      CALL SPKEZ ( JUPITE, E, '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', JUPITE,
     .             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 )
      RETURN
      END
```

Figure 3: SOLAR program generated by AMPHION from Figure 2.

tions and relations. For example, one set of axioms signifies that the relation *lightlike?* between two bodies at two different times holds if a photon leaving the center of the first body at the *sent* time would arrive at the center of the second body at the *receive* time. *Sent* and *receive* in turn are inverse functions that take two bodies and a time as input and return a time as output.

4.2: Concrete Theory and Implementation

The concrete theory defines types used in implementing a program or that are parameters used in defining a representation. The Galileo example uses the type 3Vector, which is a vector of 3 reals that variously represent a spatial position, direction, or the lengths of the 3 axes of an ellipsoid. In general, there is a many to many relation between abstract types and concrete types. However, any particular instance of a concrete type represents only one abstract type, defined through an abstraction map. The implementation relation is axiomatized in the style of Hoare[6] through these abstraction maps from concrete types to abstract types. These abstraction maps are often parameterized. To facilitate posting constraints on abstraction maps, the abstraction maps are reified. *Abs* is used to apply an abstraction map to a concrete object, e.g. *abs(coordinates-to-point(F), c)* denotes applying the abstraction map *coordinates-to-point*, parameterized on the coordinate frame *F*, to the point *c*. (Although there are a multitude of representation dimensions axiomatized in the NAIF domain theory, to simplify the presentation this paper only considers coordinate frames.)

The implementation relation is axiomatized as a set of equalities. For example, the following axiom describes how the abstract function of intersecting a ray and an ellipsoid is implemented by the subroutine *surfpt-intercept.* The *surfpt-intercept* subroutine takes three 3Vectors as arguments: the coordinates of the origin of the ray (*oc*), the coordinates of the direction of the ray (*dc*), and the lengths of the three axes of the ellipsoid (*radii*). It returns the coordinates of the intersection point of the ray and the ellipsoid. A precondition for correctly using the *surfpt-intercept* routine is that the ellipsoid frame, the observing frame, the direction frame, and the frame of the intersection point are all the same. This constraint is expressed through the common frame variable *F* in all the parameterized abstraction functions:

```
forall (F, radii, oc, dc)
  intersect-ray-ellipsoid (
          origin-and-direction-to-ray(
                    abs(coord-to-point(F), oc),
                    abs(coord-to-direction(F), dc)),
          abs(radii-to-ellipsoid(F), radii))
  = abs(coords-to-point(F), surfpt-intercept(oc,dc,radii))
```

This equation, like other implementation axioms, has the structure of a commutative diagram: applying the abstract function to the abstraction of the concrete inputs yields the same result as abstracting the result of applying the concrete function to the concrete inputs. This equation is illustrated in Figure 4.

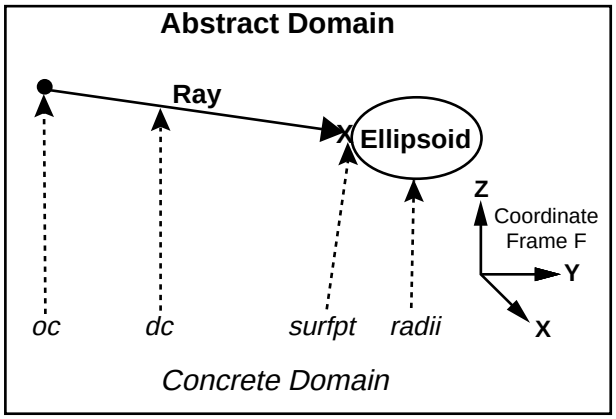

Figure 4: Intersection of a ray and ellipsoid implemented by Surfpt subroutine.

Only some of the subroutines directly correspond to functions and relations in the abstract specification domain. Other subroutines, such as subroutines that do representation conversion, are used to glue together these directly corresponding subroutines. For example, a solution to a problem usually involves several frames, so there must be a way to convert from one frame to another. The concrete function *coord-convert* is given two frames and a 3Vector as input: it returns the result of applying the transformation between the two frames to the 3Vector. A set of axioms define the properties of *coord-convert* as that of a group of transformations.

5. Specification Acquisition

AMPHION's GUI enables a specification to be developed as a diagram of objects and constraints. A user develops a problem specification by first defining a configuration of abstract objects and constraints. Then, a subset of the objects in a configuration are declared to be inputs or outputs of the desired target program.

As a result of iterative testing and refinement with users, several domain independent mechanisms have been developed that guide users in developing specifications. Without these mechanisms, users have considerable difficulty in developing valid specifications. With these mechanisms, new users of AMPHION are able to develop valid specifications by themselves after a one hour tutorial. This section presents an overview, a more detailed exposition is provided in [9].

These mechanisms are all instantiated by compiling user

interface tables from a domain theory. A user is guided when adding and refining objects or constraints by cascading menus that provide the functionality of structured editors. The direct manipulation mechanisms are cognizant of the underlying domain theory and ensure that a specification is well-typed and well-defined. Different styles of specification development are provided through bottom-up, top-down, and selected-object modalities.

The abstract functions and relations used for specification development are allowed to be overloaded. This reduces clutter in a diagram, and more importantly enables a user to think about the semantics of a problem rather than syntactic issues of typing. A table of coercions is given to the user interface compiler that define how one type can be coerced into another type. The interface compiler generates an expanded domain theory for the GUI including overloaded functions and relations. The interface compiler also generates the appropriate theory morphism from the expanded GUI theory to the more restricted domain theory used by the program synthesis subsystem.

After a well-defined specification is developed, AMPHION semantically checks the specification before generating code by attempting to solve an abstracted version of the specification. This serves two purposes: it is a necessary condition on whether a specification has a solution, and, if not, enables AMPHION to give the user feedback on correcting the specification. The abstracted specification is generated by removing conjuncts defining the input and output representation(s). The result of solving the abstract specification is a set of substitutions (terms) for the abstract existential variables. If all these substitutions are defined and ground with respect to the program input variables, then the abstract specification has a solution. Undefined substitutions indicate overconstrained variables, while non-ground substitutions indicate underconstrained variables.

AMPHION'S specification acquisition subsystem is especially well suited for specification modification and reuse. It achieves the benefits anticipated in the KBSA white paper [5] for specification-based software evolution. Instead of developing a specification from scratch, users typically modify an existing specification from a library. The abstract graphical notation makes it much easier to identify the required modifications than tracing through dependencies in code. AMPHION's editing operations facilitate making the required modifications. Most important, in contrast to code modification, there is no possibility of introducing bugs in the code, since AMPHION synthesizes the code from scratch for the modified specification.

A number of improvements are currently underway to improve AMPHION's specification acquisition component. First, although users can now access and modify previous specifications, AMPHION currently provides no help in selecting appropriate previous specifications. The DEDAL system's [1] conceptual indexing subsystem is being adapted to the task of indexing and retrieving formal specifications. Second, to enable users to develop their own graphical notation, a facility is being developed for users to record their layout preferences by example, through editing icon and link attributes. Users view other specifications through their own layout preferences. Third, at present AMPHION does not provide a simulation capability, and only generates the final program text. However, the applicative program generated directly by the theorem prover is in a format closely compatible to that used by a testing harness developed by the NAIF facility, and this testing harness will be integrated into AMPHION in the near future. Fourth, there is presently no on-line tutorial or help functionality; members of the NAIF group have agreed to populate a generic facility being developed.

As compared to Fischer's architecture for a DODE, the current specification acquisition component of AMPHION already includes functionality comparable to that of a construction kit, a catalog, and a construction analyzer. The current improvements underway will provide most of the remaining functionality.

6: Program Synthesis

Program synthesis whose target output consists of subroutine calls has a different technical emphasis than program synthesis whose target output consists of primitives in a programming language. Since most of the recursive and iterative constructs are embedded in the subroutines, the major technical challenge is effective problem decomposition, implementation of specification constructs in terms of the underlying concrete domain, and gluing together subroutines. In effect, the technical issues are similar to programming in the large, but with a predefined set of module specifications.

In AMPHION, a program is synthesized by turning the lambda form of a specification into a theorem in FOL, and then having SNARK construct a resolution refutation proof with the domain theory. SNARK incorporates very efficient equality reasoning — both paramodulation (general purpose, conditional, unoriented equality replacement) and demodulation (unconditional, oriented equality replacement) that compares favorably to more specialized inference systems that are restricted to equational logic. Many axioms of the NAIF domain theory are unconditional equalities defining the implementation relation. These are usually handled through demodulation; the confluent set of rewrite rules used by demodulation are generated by a Knuth-Bendix completion procedure [7].

By separating logic from control, the trade-off between expressiveness and efficiency for a logic becomes manage-

able for the purpose of composing subroutines. Where needed, the full expressive power of FOL is used; to gain efficiency, theorem proving tactics define a direction from abstract specification constructs to concrete subroutines. The tactics are implemented through various mechanisms provided by SNARK: recursive path orderings for orienting equalities into rewrite rules, predefined resolution refutation strategies such as set of support, and hooks for agenda ordering functions. The theorem proving tactics developed for the NAIF domain could likely be adapted and expanded for other domains, in which a domain theory had been structured as described in section 4. Once developed, they did not need to be tuned to individual problems. These tactics are also quite effective: constructive proofs that required hours or even days without these tactics are solved in under three minutes with these tactics. AMPHION's theorem proving tactics enable many of the efficiency advantages of program transformation approaches [2] and term rewriting systems to be embedded in the more expressive declarative framework of first-order logic. (The combination of these theorem proving strategies and tactics leads to loss of completeness; this is largely irrelevant in the context in which AMPHION is used):

Orienting equalities: The SNARK theorem prover can be given a set of ordering relations on function symbols that are expanded into recursive path orderings (RPO) [3] which orient its application of equational simplification (demodulation) and restrict its use of paramodulation. The equalities that axiomatize the implementation relation are oriented in the direction from abstract specification constructs to concrete subroutines by specifying that the abstract specification construct is greater in the RPO than the *abs* function and the functions and relations in the concrete theory used in the equality.

Agenda Ordering: Not all the equalities can be effectively oriented under an RPO, which also does not cover conditional equalities and non-equalities. A global inference direction from abstract specification constructs to concrete subroutines was implemented as an agenda ordering function. This function extends the directionality defined by the RPO for orientable equalities to unorientable equations and other axioms. Conceptually, this agenda ordering function consists of two parts in a lexicographic ordering. First, there is an ordering function that penalizes abstract specification constructs that are not themselves abstract operations. For example, the *lightlike?* relation is not an abstract operation, and is thus penalized. Second, abstract operations are penalized as compared to concrete subroutines. Both parts of this agenda ordering function are implemented by counting the number of penalized function and relation symbols in the terms of a clause placed on the agenda.

6.1: Deductive Synthesis Performance

The timing results in Figure 5 are taken from November 1993, when AMPHION was metered during tests with potential end-users. (Preliminary testing occurred from August through November 1993). First, as a practical matter, note that programs were synthesized well within the time limits suitable for an interactive system. Most programs were synthesized within a minute, the maximum amount of time being three minutes. The significance of these November timings are as empirical validation of the practicality of the deductive synthesis approach for composing subroutines.

The table in Figure 5 presents program synthesis performance data from thirty-eight different specifications. There is a fixed overhead of 236 steps (20 seconds of CPU time) to load the domain theory into SNARK for each derivation. Subtracting this number from the total number of steps in a program derivation gives the total number of resolution and paramodulation steps in searching for a proof. Each proof step incorporates full demodulation, so that inferences with orientable equalities (and only pattern matching, but not unification) are not counted as separate steps. Thus the number of steps in a proof search represents the premium paid for full first-order logic over just using a set of confluent rewrite rules. The length of the proof is the number of resolution and paramodulation steps in the derivation tree from the goal clause to the empty clause.

Neither the number of steps in a proof search nor the length of a proof is strongly correlated with the size of the specification or the size of the resulting program. The ratio of the proof length to the number of steps in the proof search indicates the search efficiency. Surprisingly, as shown in Figure 6, there is no correlation between the length of the proof and this ratio. If the number of search steps was exponential in the length of the final proof, this ratio should be decreasing exponentially. However, as shown in Figure 7, when the total CPU time is plotted against the length of a proof, there is a weak exponential growth.

One interpretation of Figures 6 and 7 is that as the length of a proof increases, the overhead (e.g. unification and demodulation) for each step grows multiplicatively, due to the increasing size of the term structures associated with longer proofs. Qualitative observations of the program derivations indicate that much of the search is due to unorientable equality reasoning (paramodulation), such as reasoning about invertible representation conversions. Many of these paramodulations will be subsumed under a unification algorithm that incorporated limited equality reasoning, i.e. RUE resolution, that is scheduled for a later version of SNARK. Another alternative under investigation is special purpose decision procedures for semantic fragments such as coordinate frame conversions that are called through the unification procedure.

7. Conclusion

This paper has described a formal approach to developing domain-oriented specification-based programming environments for domains with mature subroutine libraries. Raising development, modification, and reuse to the specification level eliminates an inherent knowledge barrier to using even the best engineered subroutine libraries. This approach has been implemented in the AMPHION system, which includes generic specification acquisition and automatic deductive program synthesis subsystems driven by a declarative domain theory. AMPHION is applied to a domain by developing a domain theory partitioned into an abstract specification theory, a concrete implementation theory, and an axiomatized implementation relation between the two. Specification development, modification, and reuse is well supported by the paradigm implemented in AMPHION. The tactics for the program synthesis subsystem of AMPHION enable efficient and totally automatic program generation.

The achievements to date are the first step toward the longer term goal of a generic shell that empowers domain experts to develop their own AMPHION applications.

Acknowledgments

Mark Stickel developed the SNARK FOL theorem prover and with Richard Waldinger adapted SNARK to deductive program synthesis. Waldinger aided reformulating a version of the NAIF domain theory adapted to a previous theorem proving platform to SNARK's notation, and collaborated in running initial test cases through SNARK. The following colleagues provided valuable advice for revising this paper: the anonymous reviewers, Jeffrey Van Baalen, Richard Waldinger, and Linda Wills.

References

[1] C. Baudin, J. Gevins, Z.V. Baya, A. Mabogunje: "Dedal: Using Domain Concepts to Index Engineering Design Information", Proceedings of the Meeting of the Cognitive Science Society, 1992.

[2] J. Darlington: "An Experimental Program Transformation and Synthesis System", Artificial Intelligence 16, 1981, pp. 1-46.

[3] N. Dershowitz "Termination of Rewriting", J. Symbolic Computation (1987) 3, pp. 69—116.

[4] G. Fischer: "Domain-Oriented Design Environments", Seventh KBSE, 1992, McLean, VA. pp. 204–213.

[5] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich: "Report on a Knowledge-Based Software Assistant", in C. Rich, R.C. Waters (eds.): Artificial Intelligence and Software Engineering. Los Altos: Morgan Kaufmann 1986.

[6] C.A.R. Hoare: "Proof of Correctness of Data Representations", Acta Informatica 1 (1972), pp. 271-281.

[7] D.E. Knuth and P.B. Bendix: "Simple Word Problems in Universal Algebras", in J. Leach (ed.): Computational Problems in Abstract Algebra. Pergammon Press, 1970, pp. 263-298.

[8] M. Lowry: "Methodologies for Knowledge-Based Software Engineering", in J. Komorowski and Z.W. Ras (eds.): Methodologies for Intelligent Systems, Lecture Notes in Artificial Intelligence 689, (1993) pp. 219-234.

[9] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood: "AMPHION: Automatic Programming for Scientific Subroutine Libraries" in ISMIS'94.

[10] Z. Manna and R. Waldinger: "Fundamentals of Deductive Program Synthesis," IEEE Transactions on Software Engineering (18) 8, August 1992, pp. 674-704.

[11] D.R. Smith: "KIDS: A Semiautomatic Program Development System", IEEE Transactions on Software Engineering 16,9 (1990) pp. 1024-1043.

[12] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood: "Deductive Composition of Astronomical Software from Subroutine Libraries", 1994, in CADE-12.