AMPHION: Automatic Programming for Scientific Subroutine Libraries

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

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

Abstract. This paper describes AMPHION[1], a knowledge-based software engineering (KBSE) system that guides a user in developing a formal specification of a problem and then implements this specification as a program consisting of calls to subroutines from a library. AMPHION is domain independent and is specialized to an application domain through a declarative domain theory. A user is guided in creating a diagram that represents the formal specification through menus based upon the domain theory and the current state of the specification. The diagram also serves to document the specification. Program synthesis is based upon constructive theorem proving, and is efficient and totally automatic.

1 Introduction

Subroutine libraries are one of the most prevalent forms of software reuse, particularly within the scientific programming community. However, users seldom have the time or inclination to fully familiarize themselves with even well-documented libraries. The result is that most users lack the expertise to properly identify and assemble the routines appropriate to their applications. 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 library is often perceived as being more than the effort to develop the code from scratch. In domains with mature subroutine libraries, intelligent systems technology can greatly improve the productivity and quality of software engineering by automating the effective use of those libraries.

This paper describes a methodology for constructing an intelligent system that guides a user in creating a formal problem specification, and then automatically generates a program for this specification composed of subroutines from a library. The objective is to enable users who are 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. The domain-independent part of this methodology has been implemented in the AMPHION system, which includes generic specification acquisition and program synthesis subsystems. AMPHION is applied to an application domain by developing a declarative domain theory through the methodology described in this paper.

This methodology will be described by presenting AMPHION's application to the domain of solar system kinematics. (AMPHION applications in the domains of numerical aerodynamic simulation and space shuttle flight planning are currently under development.) A domain theory was developed that includes an abstract formalization of the domain suitable for expressing problems, and also includes the knowledge needed to

1. Amphion, son of Zeus, played his magic lyre to charm the stones around Thebes into position to form the city's walls.

implement solutions to these problems using JPL's SPICELIB subroutine library. SPICELIB provides a tool kit for planetary scientists to construct programs analyzing the geometry of science observations for interplanetary missions. Typical problems include eclipses, occultations, moon shadows, and illumination angles. SPICELIB subroutines provide access to planetary ephemerides (the positions and velocities of planets as a function of time), spacecraft trajectories, and operations in analytic geometry. Complicating factors include the necessity of light-time correction over astronomical distances, and a plethora of formats and representations for locations, directions, and time.

AMPHION has undergone substantial testing with planetary scientists and has been installed at JPL for alpha testing in preparation for distribution to the planetary science community. New users are able to use AMPHION after a one hour tutorial. It usually takes a user an order of magnitude less time to develop a specification with AMPHION than to write and debug the corresponding program. In particular, experienced AMPHION users develop specifications in five minutes that take the SPICELIB developers an hour to code manually. New AMPHION users can develop a specification within fifteen minutes that take new users of SPICELIB a couple of days to develop a correctly working program. Further productivity gains are achieved through specification reuse and modification: the abstract graphical notation makes it much easier to identify the required modifications in a diagrammatic specification than it is to trace through dependencies in code. AMPHION's editing operations facilitate making the required modifications. Furthermore, there is no possibility of introducing bugs in the code, since AMPHION generates a new program from scratch for the modified specification. The specification in Figure 2 was generated in a few minutes by modifying a previous specification. The minor modifications to the specification resulted in substantially different programs, illustrating the advantages of specification modification versus program modification. These productivity improvements are consonant with the anticipated benefits of a specification-based software-engineering life cycle envisioned in [1].

AMPHION uses deductive synthesis [7] to routinely generate programs in this domain consisting of dozens of subroutine calls in under three minutes of CPU time on a Sun Sparc 2. Over a hundred programs have been generated to date. Because of the deductive synthesis, the programs are guaranteed to be correct implementations of users' specifications with respect to the domain theory.

To date, AMPHION has demonstrated the following capabilities essential for real-world KBSE: Users without training in formal methods readily develop domain-oriented diagrams denoting formal specifications using AMPHION's specification acquisition tools. Users can reuse, modify, and maintain previously developed specifications; thereby elevating software evolution from the code level to the specification level. Automatic deductive program synthesis achieves acceptable performance, given an appropriately structured domain theory and moderate use of theorem-proving tactics.

Section 2 of this paper presents an overview of the AMPHION system. Section 3 presents an example problem, its specification, and the program synthesized by AMPHION. Section 4 describes the domain-engineering methodology for the domain theory and overviews key steps in the example program synthesis. Section 5 describes AMPHION's generic specification acquisition subsystem. Section 6 summarizes the main points of the paper and compares it to previous work. The theorem-proving component is described in [9]. The theorem-proving tactics that make deductive program synthesis tractable for the specialized task of subroutine composition as well as an empirical analysis of program synthesis performance are presented in [6].

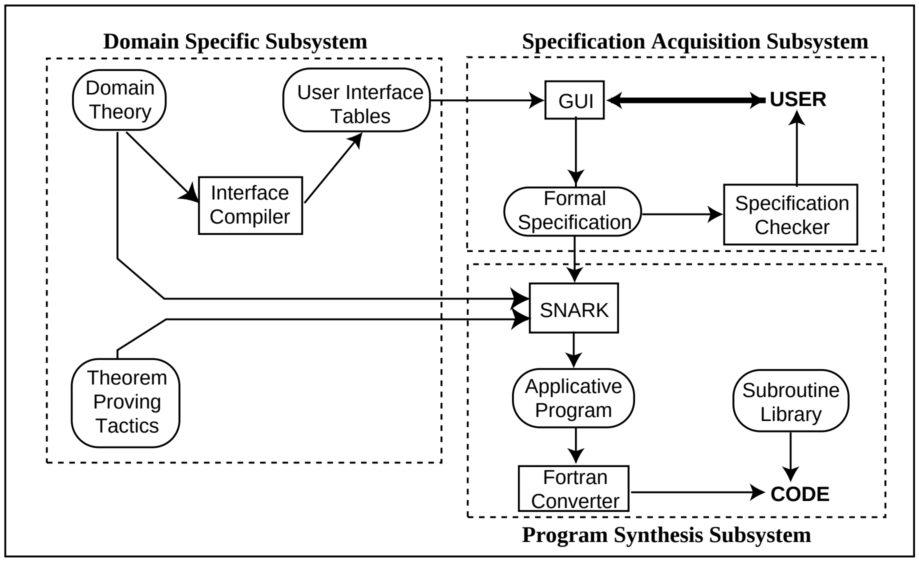

Figure 1: Flow diagram of AMPHION.

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 information. 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 first order logic (FOL) theorem prover [9] both to check a specification and also to generate an applicative 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 (GUI) and the specification checker constitute the specification acquisition subsystem. A diagram developed interactively with the GUI 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 equation 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 errors 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 an applicative program generator 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. During a proof, substitutions are generated for the existential variables through unification and equality replacement. The substitutions for the output variables are constrained to be terms in an applicative language whose function symbols correspond to the subroutines in a library.

The terms for the output variables are then translated into the target programming language through program transformations written in REFINE™ [8]. 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. Targeting other programming languages would only require minor modifications.

3 Example Problem

The following problem will be used in this paper to illustrate the structure of the domain theory and key steps in program synthesis: A planetary scientist working on the Galileo mission to Jupiter wants to compute the solar incidence angle at the point on Jupiter's surface at the center of the boresight of an instrument on Galileo. The solar incidence angle is the angle between the surface normal and the apparent position of the sun. This problem is formalized as follows within the domain theory (variable names in italics):

Let *Solar-Incidence-Angle* be the angle between two rays, *SurfaceNormal* and *Ray-Intersection-Sun*.
Let *SurfaceNormal* be the ray normal to *Jupiter-Body* at the point *Boresight-Intersection*.
Let *Ray-Intersection-Sun* be the ray from the point *Boresight-Intersection* to *Sun-Body*.
Let *Boresight-Intersection* be the intersection point of the ray *Boresight* and *Jupiter-Body* .
Let *Boresight* be the ray from the location of Galileo-Orbiter at time *TGalileo* in the direction of *Direction-2*.
Let *Direction-2* be the direction pointed by the instrument *Galileo-Camera* 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 the representation of *Solar-Incidence-Angle*, the output, be in radians.
Let the representation of *TGalileo*, one input, be a string in the format for Galileo's internal clock.
Let the representation of *Galileo-Camera*, the other input, be an integer which identifies a particular instrument on Galileo-Orbiter.

Each of these sentences corresponds to a conjunct in the lambda form of the specification. AMPHION's specification language for this domain is at the level of abstract Euclidean geometry (e.g., points, rays, ellipsoids, and intersections) augmented with astronomical terms (e.g., photons and planets). There is no mention of implementation level coordinate frames, units, and so on, except in defining representations for inputs and outputs. These are introduced during program synthesis.

Figure 2 is the diagram created interactively with AMPHION for this problem. AMPHION translated this diagram to the lambda form of the specification, and then the program synthesis subsystem generated the FORTRAN-77 program in Figure 3 in 96 seconds

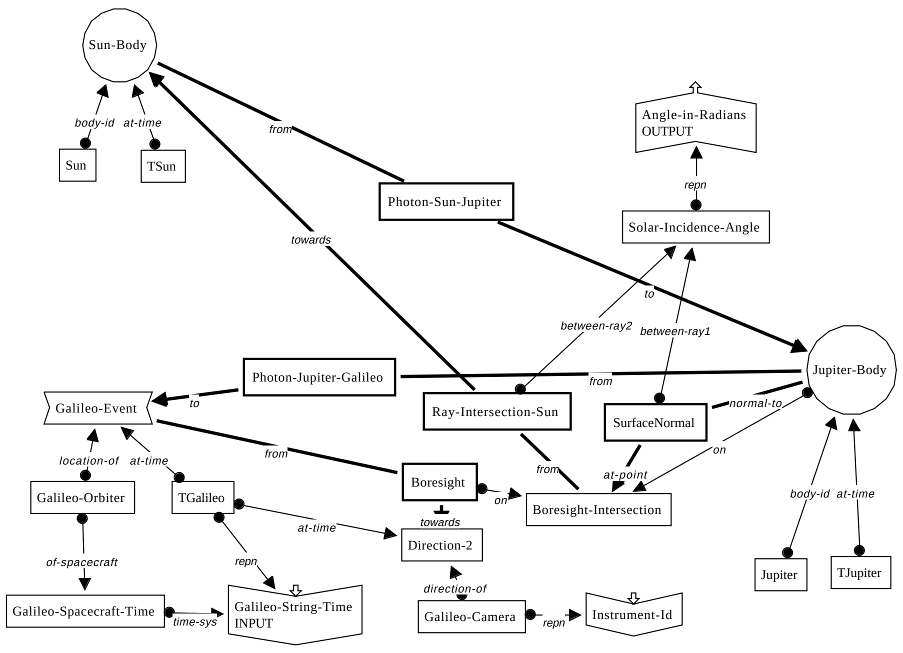

Figure 2: Diagram for solar incidence angle developed interactively with AMPHION.

```
SUBROUTINE SOLAR
        (INSTRU, GALILE, ANGLEI )
 C     Input Parameters
 INTEGER INSTRU
 CHARACTER*(*) GALILE
 C     Output Parameters
 DOUBLE PRECISION ANGLEI
 C     Function Declarations
 DOUBLE PRECISION VSEP
 C     Parameter Declarations
 INTEGER JUPIT
 PARAMETER (JUPIT = 599)
 INTEGER GALIL
 PARAMETER (GALIL = -77)
 INTEGER SUN1
 PARAMETER (SUN1 = 10)
 DOUBLE PRECISION TIKTOL
 PARAMETER (TIKTOL = 0.01)
 C     Variable Declarations
 Deleted for lack of space
 C   Dummy Variable Declarations
 INTEGER DMY10
 DOUBLE PRECISION DMY20 ( 6 )
 DOUBLE PRECISION DMY60
 DOUBLE PRECISION DMY61
 LOGICAL DMY62
 DOUBLE PRECISION DMY80 ( 6 )
 LOGICAL DMY170
CALL BODVAR ( JUPIT, 'RADII', DMY10, RADJUP )
CALL SCS2E ( GALIL, GALILE, E )
CALL SCE2T ( INSTRU, E, S )
CALL SPKSSB ( GALIL, E, 'J2000', PVGALI )
CALL SPKEZ ( JUPIT, E, 'J2000', 'NONE', GALIL,
                   DMY20, LTJUGA )
CALL CKGPAV ( INSTRU, S, TIKTOL, 'J2000', C,
                   DMY60, DMY61, DMY62 )
CALL VEQU ( PVGALI ( 1 ), V1 )
X = E - LTJUGA
CALL VEQU ( C ( 3, 1 ), V3 )
CALL SPKSSB ( JUPIT, X, 'J2000', PVJUPI )
CALL SPKEZ ( SUN1, X, 'J2000', 'NONE', JUPIT,
                   DMY80, LTSUJU )
CALL BODMAT ( JUPIT, X, MJUPIT )
CALL MXV ( MJUPIT, V3, XV3 )
CALL VEQU ( PVJUPI ( 1 ), V2 )
X1 = X - LTSUJU
CALL VSUB ( V1, V2, DV2V1 )
CALL SPKSSB ( SUN1, X1, 'J2000', PVSUN1 )
CALL MXV ( MJUPIT, DV2V1, XDV2V1 )
CALL VEQU ( PVSUN1 ( 1 ), V )
CALL SURFPT ( XDV2V1, XV3, RADJUP ( 1 ),
   RADJUP ( 2 ), RADJUP ( 3 ), P, DMY170 )
CALL SURFNM ( RADJUP ( 1 ), RADJUP ( 2 ),
   RADJUP ( 3 ), P, PP )
CALL VSUB ( P, V2, DV2P )
CALL MTXV ( MJUPIT, DV2P, XDV2P )
CALL VSUB ( V, XDV2P, DXDV2V )
CALL MXV ( MJUPIT, DXDV2V, XDXDV2 )
ANGLEI = VSEP ( XDXDV2, PP )
      RETURN
      END
```

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

of CPU time on a Sparc 2. The appearance of icons and edges in a diagram can be modified to a user's preference. The general convention for edges in Figure 2 is that they are directed from defining variables to defined variables; the label on an edge describes the role of a defining variable in defining a defined variable. The function for defining a variable in terms of other variables is not displayed, but can be readily inferred from the edges leading into a defined variable. Rays and photons are an exception to this convention for edge direction. Photons denote constraints for light-time correction.

4 Domain Engineering

An AMPHION domain theory encodes the knowledge needed to correctly use the subroutines in a library. AMPHION's purpose is to map a problem specification to the functionality embedded in a subroutine library; the resulting program then solves the problem. AMPHION does not synthesize or verify the subroutines themselves. An AMPHION domain theory consists of an abstract theory that provides the background knowledge and specification language for formulating problems, a concrete theory for formalizing the subroutines, and an implementation relation between the abstract and concrete theory. The style of axiomatization is similar to algebraic specifications for abstract data types: in essence abstract types and operations are created for the abstract theory, and then also for the concrete theory. The operations are then axiomatized, with particular emphasis on the implementation relation between abstract and concrete types and operations.

Developing the abstract theory is not a straightforward process of formalizing an existing ontology, rather it is a creative iterative process that, if done well, results in a conceptual framework that guides users in formulating their problems. An iterative refinement methodology was used for creating the abstract domain theory for solar system kinematics, similar to that espoused by Lakatos [4]. Starting with informal requirements for a set of nine typical problems in this domain, a domain expert and experts in KBSE concurrently developed the vocabulary and formal specifications for these problems. Several iterations were required over a period of a couple of weeks to develop a satisfactory vocabulary in which all the problems could be succinctly and naturally described.

The structure of an AMPHION domain theory will be illustrated in the following subsections using fragments of the domain theory needed to solve the Galileo example. The axioms have been simplified for purposes of presentation, mainly by including only one dimension of representational choice – coordinate frames – and omitting other dimensions of representational choice, such as time systems and coordinate systems.

4.1 Abstract Theory

The abstract domain theory includes types for objects in Euclidean geometry augmented with astronomical constructs such as photons, planets (modeled by ellipsoids), and spacecraft. Abstract types are independent of any particular representation. Abstract functions include constructors for derived types, such as the constructor for a ray from a point and a direction, and their corresponding selectors. Geometric operations are also included as abstract functions, such as intersecting one geometric object with another. 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 subsection on the implementation relation below describes how the function *intersect-ray-ellipsoid*, which determines the point where a given ray first intersects an ellipsoid, is axiomatized. 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 equatorial plane of a planet is defined as the plane perpendicular to the north pole at the center of a planet. The second category are non-definitional axioms among abstract functions and relations. For example, the relation *lightlike?* between two bodies at two different times holds if a photon leaving the center of the first body at the earlier time would arrive at the center of the second body at the later time. Photons in a diagram are rewritten into *lightlike?* relations. The *lightlike?* relation is axiomatized in terms of two abstract functions, which in turn are axiomatized in the implementation relation: the function *a-received* returns the time a signal would be received on the *r-body* if it was sent from *s-body* at time *s-time*; the function *a-sent* is the inverse. The following axioms mutually define the *lightlike?* relation and the *a-received* and *a-sent* functions (all variables are universally quantified):

lightlike?(s-body,s-time, r-body, a-received(s-body,r-body, s-time))
lightlike?(s-body,a-sent(s-body, r-body, r-time), r-body, r-time)

Given a set of *lightlike?* relations translated from photons in a diagram, the theorem prover generates substitution terms consisting of applications of the *a-received* and *a-sent* functions for all the sending- and received-time logical variables. This is done through unification during a unit resolution of the (negated) specification and one of the axioms above. For the Galileo example in Figure 2, two such terms are generated (where *TGalileo* is declared to be an input to the program):

TJupiter ← *a-sent(Jupiter, Galileo-Orbiter, TGalileo)*
TSun ← *a-sent(Sun, Jupiter, TJupiter)*

These abstract terms are later transformed into subroutine calls involving light-time correction (e.g., the *spkez* SPICELIB subroutine), taking into account necessary time-system conversions, in a manner similar to the derivation described below.

4.2 Concrete Theory and Implementation Relation

The concrete theory defines types used in implementing a program. The Galileo example uses the type 3Vector, which is a vector of three reals that variously represent a spatial position, direction, or the lengths of the three radii of an ellipsoid. In general, there is a many-to-many correspondence 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 [3] 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 also reified. *Abs* is used to apply a reified abstraction map to a concrete object, e.g., *abs(coord-to-point(F), c)* denotes applying the abstraction map *coord-to-point*, parameterized on the coordinate frame *F*, to the 3Vector *c*. A frame is an origin and three perpendicular axes. *Coord-to-dir* is a similar abstraction map for directions. *Radii-to-ellipsoid* maps the lengths of three radii to an ellipsoid, given a coordinate frame assumed to be aligned along the ellipsoid's axes.

The functions of the concrete theory denote subroutines in the target subroutine library. The SPICELIB subroutine *surfpt*, given the coordinates of an observation point, the coordinates of an observation direction, and three radii of an ellipsoid assumed to be aligned along the coordinate frame; returns the coordinates where the observing vector first intersects the ellipsoid. The observation point and direction, as well as the intercept coordinate, must all be represented in the coordinate frame defined by the ellipsoid.

The following equation is a typical implementation axiom: it defines how the abstract function *intersect-ray-ellipsoid* – used in the Galileo example – is implemented by the target language function *surfpt:*

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

This equation, like all the 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 output of applying the concrete function to the concrete inputs. This axiom also expresses the constraint that the concrete *surfpt* function implements the abstract *intersect-ray-ellipsoid* function only when the ellipsoid frame, the observing frame, the direction frame, and the frame for the intersection point are all identical (e.g., *F*).

To use this axiom in the Galileo example, the theorem prover must introduce a coordinate conversion between the coordinate frame for the ellipsoid – *Jupiter-Frame* – and the coordinate frame for the *Boresight* ray. The latter is the standard *J2000* frame, a helio-centered frame whose z-axis points to the north star in the year 2000. The ephemerides for planets and spacecraft are represented in *J2000*. The introduction of this coordinate conversion is done through one of the axioms defining coordinate frame conversions (the other axioms define these conversions as a group of transformations):

```
abs(coord-to-point(f1),v) =  abs(coord-to-point(f2), coord-convert(f1,f2,v))
```

The following conjunct defines *Boresight-intersection* at an intermediate step in the program derivation:

```
Boresight-Intersection =
 intersect-ray-ellipsoid (origin-and-direction-to-ray(
                                        abs(coord-to-point(J2000), Galileo-Event),
                                        abs(coord-to-dir(J2000), Direction-2)),
              abs(radii-to-ellipsoid(Jupiter-Frame), Jupiter-radii))
```

Equality replacement (paramodulation) with the coordinate conversion axiom above and a similar one for direction conversion yields a transformed conjunct with two new logical variables for frames, *F1* and *F2*:

```
Boresight-Intersection =
   intersect-ray-ellipsoid (origin-and-direction-to-ray(
                         abs(coord-to-point(F1), coord-convert(J2000,F1,Galileo-Event)),
                         abs(coord-to-dir(F2), dir-convert(J2000,F2,Direction-2))),
              abs(radii-to-ellipsoid(Jupiter-Frame), Jupiter-radii))
```

The right hand side of this equation unifies with the left hand side of the *intersect-ray-ellipsoid* implementation equation, with the following substitutions:

F ← *Jupiter-Frame* *F1* ← *Jupiter-Frame* *F2* ← *Jupiter-Frame*

Then, again through paramodulation, the following concrete-level definition is generated for the *Boresight-Intersection* point:

```
Boresight-Intersection =
           abs(coord-to-point(Jupiter-Frame),
                 surfpt-intercept(coord-convert(J2000,Jupiter-Frame,Galileo-Event),
                                        dir-convert(J2000,Jupiter-Frame,Direction-2),
                                        Jupiter-radii)
```

5 Specification Acquisition

AMPHION's GUI achieves the benefits associated with structured editors and visual programming paradigms - but at the specification level rather than the program level. Furthermore, the information needed to instantiate the GUI is derived from the declarative domain theory. This guarantees consistency between the specification acquisition subsystem and the program synthesis subsystem when a domain theory is updated. The GUI's cascading menus for adding and refining objects in a specification diagram incorporate the functionality of a structured editor by presenting a user with a template for defining/refining an object according to the syntax of the domain theory. The slots in these templates are themselves menus enumerating the possible choices, given the current state of the specification. A user can also directly manipulate the icons and edges in a specification diagram, as in a visual programming environment, with the GUI disallowing actions that would violate the domain theory. The net effect of these capabilities is that a user does not need to learn the syntax of a textual specification language or the terminology of formal logic to develop problem specifications with AMPHION.

The specification acquisition subsystem incorporates a number of additional mechanisms that greatly facilitate users' developing correct specifications. The GUI ensures that specifications are well-defined by enforcing a well-founded ordering based on one object being used in the definition of another object. The GUI has top-down, bottom-up, and selected-object(s) modalities that support different methodologies for developing well-defined specifications. A specification is also semantically checked before program synthesis by attempting to solve an abstracted version of the specification, obtained by deleting conjuncts defining concrete representations for program inputs and outputs. If a valid abstract program cannot be derived, then the user is given feedback identifying errors such as over-constrained and under-constrained variables.

A domain theory developer can also allow overloading of the abstract functions and relations used in developing a specification. 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. In general, whenever it is obvious how one type can be coerced into another type; then overloaded types, functions, and relations are defined for use by the GUI. From a table of coercions, AMPHION creates an expanded domain theory for the GUI. Axioms are generated defining the types, functions, and relations in the expanded GUI theory in terms of the types, functions, and relations in the domain theory. These axioms enable AMPHION to translate specifications developed in the expanded GUI theory to the more restricted theory used for program synthesis. The expanded GUI domain theory abstracts away irrelevant syntactic detail for the user while still ensuring that specifications are typed correctly for program synthesis**.**

For example, a body (a planet or moon) can be viewed as the point defined by its center. In the expanded GUI theory, a new supertype is generated called *Coercible-to-point*, of which body is a subtype. Functions like *distance* are overloaded with these supertypes, and the axioms defining these overloaded functions are generated:

overloaded-distance(x,y) = distance(coerce-to-point(x), coerce-to-point(y))

The function *coerce-to-point* coerces its argument, whatever type it may be, into a point, according to the entries in the table of coercions. The action of *coerce-to-point* on bodies could be defined as follows:

isbody?(b) => coerce-to-point(b) = center-of(b)

However, in AMPHION this coercion axiom is reformulated as an unconditional equality which is used as a more efficient unconditional rewrite rule:

coerce-to-point(body(b)) = center-of(b)

In the axiom above, the function *body* is a *retract* [10] which serves the same role as the type predicate in the conditional equality.

6 Related Work and Summary

This work is most closely related to that of Tyugu [11] in which software is also composed from subroutine libraries, but in contrast to Tyugu uses full predicate logic instead of intuitionistic propositional logic. The formal approach to domain-specific software design environments taken in this work contrasts with the ad-hoc approach championed by Fisher [2] and previous approaches to domain-specific automatic programming [5].

This paper has described a methodology for creating 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 methodology has been implemented in the AMPHION system, which includes generic specification acquisition and automatic program synthesis subsystems driven by a declarative domain theory. AMPHION is applied to a domain by developing a domain theory structured according to the methodology described in this paper.

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 domain theory adapted to a previous theorem proving platform to SNARK's notation, and collaborated in running initial test cases through SNARK. The anonymous reviewers, Jeffrey Van Baalen, and Linda Wills provided valuable advice for revising this paper.

References

1. C. Green, D. Luckham, R. Balzer, T. Cheathan, 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, 337-428.
2. G. Fischer: Domain-Oriented Design Environments, KBSE'92, 204–213.
3. C.A.R. Hoare: Proof of Correctness of Data Representations, Acta Informatica 1, 271-281.
4. E. Lakatos: Proofs and Refutations, J. Worrall (ed.), Cambridge University Press, 1976.
5. M. Lowry and R. McCartney (eds.): Automating Software Design, MIT Press 1991.
6. M. Lowry, A. Philpot, T. Pressburger, and I. Underwood: A Formal Approach to Domain-Oriented Softwre Design Environments, KBSE'94.
7. Z. Manna and R. Waldinger: Fundamentals of Deductive Program Synthesis, IEEE Transactions on Software Engineering (18) 8, August 1992, 674-704.
8. D.R. Smith: KIDS: A Semiautomatic Program Development System, IEEE Transactions on Software Engineering 16,9 (1990), 1024-1043.
9. M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood: Deductive Composition of Astronomical Software from Subroutine Libraries, 1994, in CADE-12.
10. J. Stoy: Denotation Semantics: the Scott-Strachey approach to programming language semantics. MIT Press, 1977. *Retract defined on page 133.*
11. E.H. Tyugu, *Knowledge-Based Programming*, Turing Institute Press, Glasgow, Scotland, 1988.