Deductive Composition of Astronomical Software from Subroutine Libraries *

Mark Stickel and Richard Waldinger†
Artificial Intelligence Center
SRI International
Menlo Park, CA 94025

Michael Lowry, Thomas Pressburger, and Ian Underwood‡
Artificial Intelligence Research Branch
Recom Technologies
NASA Ames Research Center
Moffett Field, CA 94035

July 16, 1994

Abstract

Automated deduction techniques are being used in a system called Amphion to derive, from graphical specifications, programs composed from a subroutine library. The system has been applied to construct software for the planning and analysis of interplanetary missions.

The library for that application is a collection of subroutines written in FORTRAN-77 at JPL to perform computations in solar-system kinematics. An application domain theory has been developed that describes

*A preliminary version of this appears in the proceedings of the Twelfth International Conference on Automated Deduction, Nancy, France, June 1994, pages 341-355.

†{stickel,waldinger}@ai.sri.com

‡{lowry, pressburger,underwood}@ptolomy.arc.nasa.gov

the procedures in a portion of the library, as well as some basic properties of solar-system astronomy, in the form of first-order axioms.

Specifications are elicited from the user through a menu-driven graphical user interface; space scientists have found the graphical notation congenial. The specification is translated into a theorem, which is proved constructively in the astronomical domain theory by an automated theorem prover, SNARK. An applicative program is extracted from the proof and converted to FORTRAN-77. By the method of its construction, the program is guaranteed to meet the given specification and requires no further verification, provided, of course, that the specification, domain theory, and system itself are correct.

Amphion has successfully constructed more than a hundred programs to solve problems, formulated at NASA Ames, JPL, and Stanford, which involve typical computations involving the sun, planets, moons, and spacecraft. The system is currently being alpha tested at JPL.

1 Introduction

Automatic deductive program synthesis has been studied for many years but has never been used in practice. By restricting our attention to the construction of programs composed from subroutine libraries, rather than the primitive instructions of a programming language, and by adapting domain-specific control strategies, we have applied deductive methods to construct useful software.

Subroutine Libraries

Subroutine libraries are one of the most prevalent forms of software reuse, particularly within the scientific programming community. However, end users often do not make effective use of libraries. Sometimes this happens because the subroutines are not adequately documented. But even when excellent documentation is provided, users often have neither the time nor the inclination to familiarize themselves with it. In either case, the result is that most users lack the expertise to properly identify and compose the routines appropriate to their application. In domains with mature subroutine libraries, one can greatly improve the productivity and quality of software engineering by automating the effective use of those libraries.

Subroutines are commonly accessed by indexing key words in their documentation, a very approximate method. In the work of Rollins and Wing [RW 91],

logic programming techniques are invoked to retrieve appropriate subroutines, according to their specifications, but composing them is left up to the user. In this work, deductive methods—that is, methods of automated reasoning or theorem proving—are applied to the composition of subroutines into software. In that sense it most closely resembles the work of Tyugu and his associates [Tyu 88], in which software is also composed from subroutine libraries, to meet specifications expressed in intuitionistic propositional logic.

Although deductive methods are independent of the application domain, we discuss their application to the construction of software for performing computations in solar-system astronomy. Such computations are necessary in the planning and data analysis for interplanetary scientific missions. For example, observing the location of a moon of a nearby planet is often the best way of determining the position of the observing spacecraft.

Amphion

The Intelligent Software Project of the Artificial Intelligence Research Branch at NASA Ames, led by Michael Lowry, has been developing a system called Amphion[1] to automate the composition of software from subroutine libraries. Software requirements are specified in a graphical notation. An interactive interface, which is domain-independent but employs the vocabulary of the domain, presents the user with a menu of alternatives and elicits the specification gradually. The user need not know the contents of the library, the syntax of the specification language, or the target programming language.

The graphical specification is translated automatically into a first-order-logic theorem, and a program is developed from the logical form of the specification using SRI's automated deduction system SNARK, which has been implemented by Mark Stickel. The resulting program is subjected to common-subexpression elimination and translated into FORTRAN-77. The translation package invokes Refine™ transformations from Kestrel's KIDS system [Smi 90]. It would be a relatively small change to produce a final program in a language other than FORTRAN.

SPICE

Amphion's first application domain is software for planning and interpreting space-science observations. The software is based on SPICELIB, a library of

[1] Amphion built a wall around Thebes by charming the stones into place with a magic lyre.

procedures for solar-system geometry. These routines, written in FORTRAN-77 at the Navigation Ancillary Information Facility (NAIF) at JPL, perform basic computations involving the sun, planets, moons, and spacecraft. Various systems of time measurement (e.g., ephemeris time, which is used in astronomical tables, and spacecraft clock time) and multiple frames of reference come into play. Light is not assumed to travel instantaneously across astronomical distances.

The NAIF library procedures are used by astronomers and researchers as primitives to build more complex software. The subroutines embody considerable expertise and cannot easily be recreated. Although the routines are well documented, users seem reluctant to invest the time and effort to learn about them. They frequently attempt to reimplement routines that already exist in SPICELIB because they did not find them in the documentation, or if they are sufficiently influential, they prevail on the authors of the library to retrieve the appropriate routines and compose them into the required software.

2 Deductive Component

The emphasis of this paper is on the role of SNARK, the deductive subsystem, in Amphion. Other papers will focus on the astronomical aspects of the system and on the graphical interface.

Deductive Approach

A program is developed from the logical form of the specification by a deductive approach, which is based on work of Zohar Manna, of Stanford University, and Richard Waldinger [MW 92]. We prove a mathematical theorem that expresses the existence of an output that meets the specified conditions. The graphical specification language corresponds to only a subset of predicate logic, but in principle knowledgeable users can introduce logical specifications directly.

The proof is conducted in a classical logic but is restricted to be constructive—in other words, in proving the existence of the required output, we are forced to indicate a method for finding it. That method becomes the basis for a program to compute the output, which may be extracted from the proof. This program is guaranteed, by the way it was constructed, to meet the specification—it requires no further verification.

The structure of the program reflects the proof from which it was extracted.

If the proof relies on reasoning by cases (e.g., by application of the resolution rule [Rob 65]), the resulting program may contain a conditional expression. If the proof depends on the mathematical induction principle, the program may invoke recursion or other repetitive constructs.

The theorem is proved valid in an *application domain theory* that provides the knowledge on which the software depends. The specifications of the available subroutines, the constructs of the specification language, and properties of the application domain are expressed by axioms in the domain theory. The application domain theory also determines the options offered to the user by the graphical interface.

Program synthesis differs in its technical emphasis when its output is expressed in terms of subroutine calls rather than the primitives of a programming language. When most of the recursive and iterative constructs are embedded in subroutines, the major technical challenge is to effectively decompose the problem and glue together subroutines. While general program synthesis imposes severe demands on a deductive system, the theorems that arise when software is composed from a subroutine library appear to be within the range of existing deductive technology.

SNARK

To automate a deductive approach requires an automated deduction system, or theorem prover. SNARK is especially suitable for program synthesis and other applications in artificial intelligence and software engineering. SNARK is invoked as a subsystem of Amphion, but it can also be used independently or as a component of other systems.

The current implementation of SNARK, in COMMON LISP, includes the resolution [Rob 65] and paramodulation [WR 69] rules for handling the constructs of first-order logic with equality, like McCune's OTTER [McC 90]. It also will employ the principle of mathematical induction, like Boyer and Moore's NQTHM [BM 88]. Proofs are developed within Manna and Waldinger's *deductive tableau* framework [MW 93] and can be restricted to be constructive so that programs can be extracted. Clause form is optional—if the user prefers, formulas may employ a full set of logical connectives in arbitrary form.

It is intended that the SNARK user will be able to introduce new inference rules, but in the current implementation the user chooses among a fixed set of rules. An indexing mechanism allows the system to retrieve from its memory only those formulas that are syntactically relevant.

SNARK (like OTTER) is agenda-driven—it draws conclusions from a formula when that formula reaches the top of its agenda. The user does have the ability to influence the strategy adopted by the system, for example, by providing the function used to order the agenda. Although interactive handles are being attached to SNARK, the system is fundamentally automatic.

3 The Astronomical Domain

For the astronomical application, the specifications of a portion of the subroutines of the SPICELIB library are represented by axioms in the application domain theory. Other axioms describe properties of the specification constructs and the geometry and space kinematics on which the construction of the software depends. At this moment the domain theory consists of more than 200 axioms, all of which are available when we attempt to prove the specification theorem. It is beside the point of this paper to describe the domain theory (largely the work of Lowry and Pressburger) in any detail. But let us present enough of the theory to suggest its contents.

The Astronomical Domain Theory

A fundamental entity in the domain theory is a space-time location (sometimes called an event), a position in space at a certain time; for two events to be identical, they must correspond to the same position and time.

The relation *lightlike?*(e_1, e_2) holds if a photon could leave the position corresponding to event e_1, at the time corresponding to that event, and arrive at the position and time corresponding to event e_2; the symbol *lightlike?* is a specification construct, not a subroutine in the library.

The function *ephemeris-object-and-time-to-event* yields an event corresponding to the position of a given astronomical object (e.g., a planet or spacecraft) at a given time; this is also a specification concept. Objects and times are abstract entities, independent of any representation system for designating astronomical objects or units for measuring time.

The specification function *a-sent*(o,d,ta) computes the time a photon must leave the origin object o in order to arrive at the destination object d at time ta. This function is defined in part by the axiom *lightlike?-of-a-sent*:

```
(all (o d ta)
     (lightlike? (ephemeris-object-and-time-to-event o
```

```
                    (a-sent o d ta))
               (ephemeris-object-and-time-to-event d ta)))
```

(The axioms and theorems are written in LISP notation, e.g., `(a-sent o d ta)` instead of *a-sent*(*o,d,ta*).) In other words, a photon could leave object `o` at time `(a-sent o d ta)` and arrive at object `d` at time `ta`.

The specification constructs deal mainly with abstract entities. But each abstract entity corresponds to one or more concrete entities, which depend on a particular representation scheme or system of units. In particular, an abstract astronomical body such as Jupiter is assigned a NAIF library symbol, called its NAIF id; the NAIF id of Jupiter is 599. Each abstract time corresponds to a concrete ephemeris time and to a concrete spacecraft clock time. The function *abs*(*fn,c*) is used to denote the abstract entity corresponding to the concrete entity *c*; here, *fn* is the abstraction function that maps concrete entities into abstract ones. (For technical reasons, abstraction functions are reified; that is, they are denoted by constants and terms rather than by function symbols.) For example, `(abs ephemeris-time-to-time et)` stands for the abstract time corresponding to the ephemeris time `et`, and `(abs naif-id-to-body 599)` stands for Jupiter.

The subroutines in the library apply to concrete entities, not abstractions. For example, the subroutine `(sent onid dnid eta)` is analogous to the abstract function `(a-sent o d t)` but applies to concrete NAIF ids for origin and destination bodies, `onid` and `dnid`, respectively, and a concrete arrival time `eta` in ephemeris-time units, rather than their abstract counterparts. The precise relationship between the specification function `a-sent` and the subroutine `sent` is expressed by the following *a-sent-to-sent* axiom:

```
(all (onid dnid eta)
     (= (a-sent (abs naif-id-to-body onid)
                (abs naif-id-to-body dnid)
                (abs ephemeris-time-to-time eta))
        (abs ephemeris-time-to-time (sent onid dnid eta)))).
```

In other words, the result of first translating the concrete entities into abstractions and then computing `a-sent` is the same as the result of computing `sent` on the concrete entities and then translating to abstract time.

Figure 1: Where is the shadow of Io on Jupiter?

The Sample Problems

With the assistance of astronomers at JPL and Stanford, and based on his own experience, Underwood assembled a collection of fifteen sample problems representative of what might be requested of a NAIF consultant. The problems require solar-system computations typical of those required for scientific missions; some were from software that had been developed for the Hubble Space Telescope Science Institute. Although a NAIF expert would be able to construct programs to solve these problems in less than half an hour, NAIF experts are in short supply; a programmer unfamiliar with the NAIF library might require several days to learn its contents before composing the software.

Amphion was able to construct programs for all fifteen sample problems completely automatically, without user interaction. Once the specifications were elicited from the user, the system required less than three minutes to construct each program.

Let us look at one of the sample problems.

Shadow of Io

The first problem we considered involved determining the location of the shadow cast on Jupiter by its moon Io, as observed at a given time on Voyager 2 (Figure 1). The point *pi* indicates the shadow.

The corresponding graphical specification (Figure 2) may appear confusing, but would be much clearer if we could show the step-by-step interaction between user and system. After a one-hour tutorial, a novice user may require a half hour to construct such a specification; an experienced user can do it in a few minutes.

In Figure 2, PHOTON-SUN-IO designates a photon that passes from the sun at a certain time and reaches Io at another; the purpose of speaking about photons is to specify times. Similarly PHOTON-IO-JUPITER is perhaps the

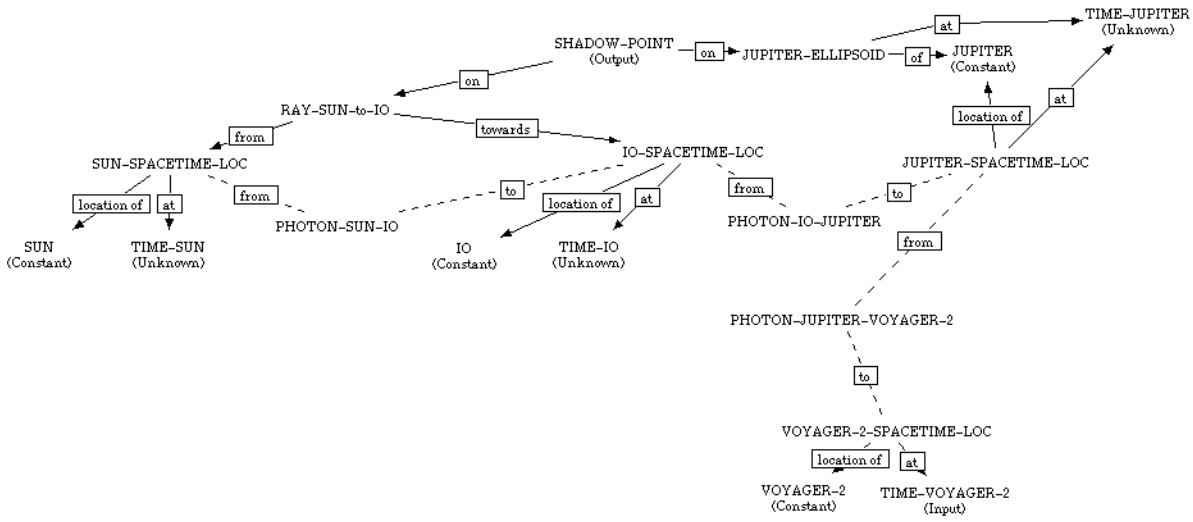

Figure 2: Shadow of Io Graphical Specification

same photon as it leaves Io and reaches Jupiter, and PHOTON-JUPITER-VOYAGER-2 is the photon as it leaves Jupiter and arrives at Voyager 2. The input to the program (indicated at the lower right) is the time that photon reaches Voyager 2.

RAY-SUN-to-IO is a ray (that is, a half-infinite line) that originates at the sun and passes through Io. JUPITER-ELLIPSOID is the surface of Jupiter at the time the photon reaches Jupiter; because Jupiter rotates and moves, its surface changes with time. SHADOW-POINT, the output of the program, is the first intersection of the ray with the surface of Jupiter.

The reader may observe that the user has chosen certain simplifications and approximations in specifying this problem. For instance, the sun and Io are regarded as points, and though Jupiter is sometimes regarded as a spheroid, PHOTON-IO-JUPITER arrives at the center of Jupiter, not its surface. The decision as to which simplifications may be made is left up to the user.

The theorem obtained from this specification is given in Figure 3.

```
(all (time-voyager-2-c)
     (find (shadow-point-c)
   (exists
```

```
    (time-sun sun-spacetime-loc time-io io-spacetime-loc
      time-jupiter jupiter-spacetime-loc time-voyager-2
      voyager-2-spacetime-loc shadow-point jupiter-ellipsoid
      ray-sun-to-io)
    (and
     (= ray-sun-to-io
(two-points-to-ray
 (event-to-position sun-spacetime-loc)
 (event-to-position io-spacetime-loc)))
     (= jupiter-ellipsoid
(body-and-time-to-ellipsoid jupiter
    time-jupiter))
     (= shadow-point
(intersect-ray-ellipsoid ray-sun-to-io jupiter-ellipsoid))
     (lightlike? jupiter-spacetime-loc voyager-2-spacetime-loc)
     (lightlike? io-spacetime-loc jupiter-spacetime-loc)
     (lightlike? sun-spacetime-loc io-spacetime-loc)
     (= voyager-2-spacetime-loc
(ephemeris-object-and-time-to-event voyager-2 time-voyager-2))
     (= jupiter-spacetime-loc
(ephemeris-object-and-time-to-event jupiter time-jupiter))
     (= io-spacetime-loc
(ephemeris-object-and-time-to-event io time-io))
     (= sun-spacetime-loc
(ephemeris-object-and-time-to-event sun time-sun))
     (= shadow-point (abs (coords-to-point j2000) shadow-point-c))
     (= time-voyager-2
(abs ephemeris-time-to-time time-voyager-2-c))))))
```

Figure 3: Shadow of Io Theorem

The quantifier `find` is a constructive version of the existential quantifier `exists`; in proving the existence of `shadow-point-c` (which corresponds to SHADOW-POINT in the graphical specification), the system is forced to indicate a method for finding it.

Although this and the other theorems required for the astronomical application are not mathematically deep, some of the authors of this paper will confess

to being unable to prove them from the axioms by hand. SNARK required about 40 seconds (on a Sun 670MP) to prove this one. The program extracted from the proof, as translated by Amphion into FORTRAN-77, is given in Figure 4.

```
      SUBROUTINE SHADOW ( TIMEVO, SHADOW )

C     Input Parameters
      DOUBLE PRECISION TIMEVO

C     Output Parameters
      DOUBLE PRECISION SHADOW ( 3 )

C     Function Declarations
      DOUBLE PRECISION SENT

C     Parameter Declarations
      INTEGER JUPITE
      PARAMETER (JUPITE = 599)
      INTEGER VOYGR2
      PARAMETER (VOYGR2 = -32)
      INTEGER SUN
      PARAMETER (SUN = 10)
      INTEGER IO
      PARAMETER (IO = 501)

C     Variable Declarations
      DOUBLE PRECISION RADJUP ( 3 )
      DOUBLE PRECISION TJUPIT
      DOUBLE PRECISION PJUPIT ( 3 )
      DOUBLE PRECISION TIO
      DOUBLE PRECISION MJUPIT ( 3, 3 )
      DOUBLE PRECISION PIO ( 3 )
      DOUBLE PRECISION TSUN
      DOUBLE PRECISION PSUN ( 3 )
      DOUBLE PRECISION DPSPI ( 3 )
      DOUBLE PRECISION DPJPS ( 3 )
      DOUBLE PRECISION XDPSPI ( 3 )
```

```
      DOUBLE PRECISION XDPJPS ( 3 )
      DOUBLE PRECISION P ( 3 )
      DOUBLE PRECISION DPJUPP ( 3 )

C     Dummy Variable Declarations
      INTEGER DMY0
      DOUBLE PRECISION DMY20 ( 3 )
      DOUBLE PRECISION DMY30 ( 3 )
      DOUBLE PRECISION DMY40 ( 3 )
      LOGICAL DMY90

      CALL BODVAR ( JUPITE, 'RADII', DMY0, RADJUP )
      TJUPIT = SENT ( JUPITE, VOYGR2, TIMEV0 )
      CALL FINDPV ( JUPITE, TJUPIT, PJUPIT, DMY20 )
      CALL BODMAT ( JUPITE, TJUPIT, MJUPIT )
      TIO = SENT ( IO, JUPITE, TJUPIT )
      CALL FINDPV ( IO, TIO, PIO, DMY30 )
      TSUN = SENT ( SUN, IO, TIO )
      CALL FINDPV ( SUN, TSUN, PSUN, DMY40 )
      CALL VSUB ( PIO, PSUN, DPSPI )
      CALL VSUB ( PSUN, PJUPIT, DPJPS )
      CALL MXV ( MJUPIT, DPSPI, XDPSPI )
      CALL MXV ( MJUPIT, DPJPS, XDPJPS )
      CALL SURFPT ( XDPJPS, XDPSPI, RADJUP ( 1 ), RADJUP ( 2 ),
     .RADJUP ( 3 ), P, DMY90 )
      CALL VSUB ( P, PJUPIT, DPJUPP )
      CALL MTXV ( MJUPIT, DPJUPP, SHADOW )

      END
```

Figure 4: Shadow of Io Program

Again there is little point to reading the entire program; we printed it to emphasize the difference between the program and its specification. After the long sequence of declarations and initializations, the program invokes the SPICELIB procedure `bodvar`, which computes the radii of Jupiter; because the surface of Jupiter is an ellipsoid, it has three radii, which are stored in an array. Then the library function `sent` computes the time `tjupit` a photon must have left

Jupiter to reach Voyager 2 at input time `timev0`. The procedures `findpv` and `bodmat` then compute the position and the orientation of Jupiter at time `tjupit`. The orientation of Jupiter is represented by a three-by-three matrix of double-precision numbers. And so on.

In short, the specification deals with abstract entities, such as planets, times, and ellipsoids; the program deals with integers, double-precision numbers, and matrices.

4 Strategic Considerations

We do not provide a systematic description of SNARK here, but we do describe some of the heuristic features that SNARK employed to solve the astronomical problems.

Recursive Path Ordering

SNARK employs term rewriting and the paramodulation rule [WR 69] for reasoning about equality. It has been found possible to avoid replacing one term with another if the second term is greater than the first with respect to a certain kind of ordering, a *recursive-path ordering* [Der 82]. The recursive-path orderings are syntactic relations defined on the terms of our language. SNARK allows the user to declare a recursive-path ordering before beginning a proof. The user provides an ordering on the constants and function symbols of the language, and that determines a corresponding ordering on the terms, which is used to control the paramodulation rule. It has been established [HR 91] that the recursive-path-ordering strategy is complete for first-order logic with equality. If a sentence has a proof, it can be proved with the strategy, regardless of the choice of ordering.

SNARK's success in this domain depends on its use of the recursive-path-ordering strategy and on the choice of a particular ordering. Indeed, there are examples in which SNARK found a proof in less than a minute with a plausible ordering, but failed to find a proof in a reasonable time if that ordering was reversed or if ordering information was omitted altogether.

We found that a good heuristic for ordering the terms was, roughly speaking, to direct SNARK to replace abstract, noncomputable symbols with concrete, computable ones, which could appear in the final program extracted from the proof. With little effort, it was possible to declare an ordering that would enable

SNARK to construct a program. A single ordering sufficed for all the problems in the astronomical domain. In general, we do not expect Amphion users to have to supply a recursive path ordering—that is done when the application domain theory is formulated.

The SPICE Agenda-Ordering Function

We have remarked that SNARK is an agenda-driven theorem prover. When it infers a new formula, it places it on an agenda, a list of formulas, to wait its turn to be processed. A formula is not processed until it reaches the head of the agenda; then it is removed from the agenda and all its immediate consequences are added.

The place at which a new formula is added to the agenda is determined by the *agenda-ordering function*. Although a default agenda-ordering function is provided with the system, the SNARK user may choose another or provide a new one, written in COMMON LISP. One of the ways SNARK has been specialized to the astronomical domain is with a new SPICE agenda-ordering function, written by Pressburger. This strategy gives special attention to goals with literals containing the predicate symbol `lightlike?` for which one of the arguments is ground (variable-free) and the other contains a variable; there are axioms in the domain theory, such as the axiom *lightlike?-of-a-sent* given previously, that are capable of solving any such literals. To a lesser extent, the strategy favors goals with fewer abstract function symbols. The effect of this strategy is to first determine the space-time locations of all the bodies in the problem, and then to replace all the abstract function symbols with concrete ones, which correspond to SPICELIB routines.

The choice of agenda-ordering function can be critical. One problem we have encountered requires less than three minutes with the SPICE agenda ordering but more than an hour with the SNARK default agenda ordering. All the astronomical problems were solved with this same SPICE agenda ordering; we do not expect Amphion users to have to change this ordering.

The Set-of-Support Strategy

The mathematical applications on which theorem provers are commonly tested require relatively deep proofs in theories with few axioms. In contrast, the astronomical domain, like most software-engineering applications, requires us to find mathematically less sophisticated proofs in theories with a large number

of axioms, which represent the subject knowledge of the domain. For such a problem, it is appropriate to invoke the *set-of-support* strategy [WRC 65] to focus attention on the goal—the theorem to be proved—at the expense of the axioms. This strategy requires that every formula we infer be descended from the goal. Otherwise, with so many axioms, it is hard to decide in advance which of them are relevant to the proof. In fact, the set-of-support strategy turned out to be crucial in the astronomical domain—theorems that are proved in under a minute with set of support cannot be proved within the available space without it.

When we employ the set-of-support and the recursive-path-ordering strategies at the same time, however, we lose completeness—there may be some valid theorems we will be unable to prove without violating the restrictions of one of the strategies. (In fact, once we combine the recursive-path-ordering strategy with the constructiveness restriction, which guarantees that we can extract programs from proofs, we may already have lost completeness.) The domain theory contains some logically redundant axioms to circumvent this incompleteness, but this is something of a stopgap measure. In the future, a hybrid strategy that allows some reasoning forward from axioms and some reasoning backward from the goal may be employed in combination with the recursive-path-ordering strategy.

5 Performance

Since the test cases were run, demonstrations of Amphion have been given by Lowry at NASA Ames, JPL, and other sites. Members of the audience unfamiliar with the system were invited to specify their own programs. In almost all cases, the graphical notation was adequate to specify the new program and SNARK was capable of proving the corresponding theorem and constructing the specified program.

In all our test cases, including those proposed by participants in NASA and JPL demonstrations, the specification has been formulated in less than half an hour; an experienced Amphion user needs just a few minutes. It is often more convenient to revise the stored specification of a similar problem than to construct a new specification from scratch. The theorems have required less than ten minutes—usually less than three minutes—for SNARK to prove, and the translation into FORTRAN is completed in seconds.

For one problem, the desired program relied on properties of subroutines in

SPICELIB that had not yet been axiomatized. It required less than half an hour to introduce the new axioms; the system was then able to construct the new program.

Once SNARK has found one proof and extracted the corresponding program, we can restart it to find other proofs and perhaps other programs. This ability is not used by Amphion, because we have not found that the various programs differed in any significant way; they were doing more or less the same things in different orders.

The system has recently been installed at NAIF so that JPL astronomers can use the system regularly, on an experimental basis.

6 What Next?

The problems solved so far have been relatively simple, none requiring more than two of three pages of FORTRAN code and none including if-statements or loops, except implicitly at the subroutine level. While SNARK does regularly introduce conditionals, for example by application of the resolution rule, its ability to introduce iterative or recursive constructs is rudimentary. It currently contains no induction rule, so we must provide the appropriate well-founded relation, on which the induction is based, and enter the induction hypothesis as an axiom. Although none were encountered in the sample problem set or in demonstrations, there are problems in the domain that do require iterations that cannot be relegated to subroutines. When we do employ induction, it may be advisable to use a nonclausal representation of formulas; so far, all formula have been kept in clausal form.

For more complex problems, it will be necessary to decompose the specification into subspecifications of more manageable modules. Simple decompositions might be achieved automatically, with the help of tactics that could be built into SNARK itself. Other decompositions will be performed interactively through the graphical interface. In this way, the user would specify the original problem and its decomposition into modules with the same mechanism.

Once SNARK has successfully constructed a module, its specification can be added to the theory as a new axiom. If that axiom is used in a proof, the module will be invoked by the corresponding program. Thus, if the decomposition is done appropriately, SNARK will be able to compose the modules to solve the main problem. Whether the decomposition is accomplished automatically or with user assistance, the correctness of the resulting program and its modules

is guaranteed by the method of their construction, provided that the domain theory is correct.

Nothing in the techniques we are using restricts us to SPICELIB or to the astronomical domain. We are currently considering other application domains in which the same technology would be valuable. Characteristic of a potentially fruitful domain are the existence of a mature subroutine library, many of whose users are imperfectly acquainted with its contents. Deductive methods are particularly attractive when the correctness of the derived software is critical. In such a domain, it is plausible that existing deductive technology will suffice to give computationally naive users access to a large library of subroutines and enable them to compose software of practical power and high reliability.

Acknowledgements

We would like to thank the National Science Foundation for support of some of this research, under Grant CCR-8922330.

References

[BM 88] R. S. Boyer and J S. Moore, *A Computational Logic Handbook*, Academic Press, Boston, MA (1988).

[Der 82] N. Dershowitz, Orderings for Term-Rewriting Systems, *Journal of Theoretical Computer Science*, 17,3 (1982), 279-301.

[HR 91] J. Hsiang and M. Rusinowitch, Proving Refutation Completeness of Theorem-Proving Strategies: The Transfinite Semantic Tree Method, *Journal of the ACM*, 38,3 (1991), 559-587.

[McC 90] W. McCune, *Otter 2.0 User's Guide*, Technical Report ANL-90/9, Argonne National Laboratory, Argonne, IL (1990).

[MW 92] Z. Manna and R. Waldinger, Fundamentals of Deductive Program Synthesis, *IEEE Transactions on Software Engineering*, 18,8 (1992), 674–704.

[MW 93] Z. Manna and R. Waldinger, *Deductive Foundations of Computer Programming*, Addison-Wesley, Reading, MA (1993).

[Rob 65] J. A. Robinson, A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM* 12 (1965) 23–41.

[RW 91] E. J. Rollins and J. M. Wing, Specifications as Search Keys for Software Libraries, *Eighth International Conference on Logic Programming*, Paris, June 1991.

[Smi 90] D. R. Smith, KIDS: A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering* 16,9 (1990) 1024–1043.

[Tyu 88] E. H. Tyugu, *Knowledge-Based Programming*, Turing Institute Press, Glasgow, Scotland, 1988.

[WR 69] L. Wos and G. Robinson, Paramodulation and Theorem Proving in First-Order Theories with Equality. In B. Meltzer and D. Michie (editors), *Machine Intelligence* 4, American Elsevier, New York, NY (1969) 135–150.

[WRC 65] L. Wos, G. A. Robinson, and D. F. Carson, Efficiency and Completeness of the Set-of-Support Strategy in Theorem Proving. *Journal of the ACM*, 12,4 (1965), 536–541.