Explaining Synthesized Software

Jeffrey Van Baalen, Peter Robinson*,
Michael Lowry, Thomas Pressburger*
NASA Ames Research Center
M.S. 269-2, Code IC
Moffett Field, CA 94035
{jvb,lowry,robinson,ttp}@ptolemy.arc.nasa.gov
http://ic-www.arc.nasa.gov/ic/projects/amphion/docs/amphion.html
* Recom Technologies, Inc.

Abstract

Motivated by NASA's need for high-assurance software, NASA Ames' Amphion project has developed a generic program generation system based on deductive synthesis. Amphion has a number of advantages, such as the ability to develop a new synthesis system simply by writing a declarative domain theory. However, as a practical matter, the validation of the domain theory for such a system is problematic because the link between generated programs and the domain theory is complex. As a result, when generated programs do not behave as expected, it is difficult to isolate the cause, whether it be an incorrect problem specification or an error in the domain theory.

This paper describes a tool we are developing that provides formal traceability between specifications and generated code for deductive synthesis systems. It is based on extensive instrumentation of the refutation-based theorem prover used to synthesize programs. It takes augmented proof structures and abstracts them to provide explanations of the relation between a specification, a domain theory, and synthesized code. In generating these explanations, the tool exploits the structure of Amphion domain theories, so the end user is not confronted with the intricacies of raw proof traces.

This tool is crucial for the validation of domain theories as well as being important in every-day use of the code synthesis system. It plays an important role in validation because when generated programs exhibit incorrect behavior, it provides the links that can be traced to identify errors in specifications or domain theory. It plays an important role in the every-day use of the synthesis system by explaining to users what parts of a specification or of the domain theory contribute to what pieces of a generated program. Comments are inserted into the synthesized code that document these explanations.

Introduction

The Amphion project in the Automated Software Engineering group at NASA Ames Research Center is investigating technology to support the development of high-assurance software. Amphion/NAIF [7, 5] is a domain-specific, high-assurance software synthesis system based on a specialization of the generic Amphion architecture. Amphion/NAIF takes an abstract specification of a problem in solar system observation geometry, such as "when will a signal sent from the Cassini spacecraft to Earth be blocked by the planet Saturn?", and automatically synthesizes a Fortran program to solve it.

Amphion greatly facilitates reuse of domain-oriented software libraries by enabling a user to state a problem in an abstract, domain-oriented vocabulary. The programs generated by Amphion/NAIF consist of assignment statements and calls to components from the SPICELIB software library in the NAIF toolkit. It takes significantly less time for an experienced user to develop a problem specification with Amphion than to manually generate and debug a program. More importantly, a novice user does not need to learn the details of the components in the library before using Amphion to create useful programs. This removes a significant barrier to the use of software libraries.

Amphion uses deductive synthesis in which programs are synthesized as a byproduct of theorem proving from an application domain theory, such as the domain of solar system observation geometry. In this paradigm, problem specifications are of the form $\forall \vec{x} \exists \vec{y} [P(\vec{x}, \vec{y})]$, where $\vec{x}$ and $\vec{y}$ are vectors of variables. The theorem prover generates constructive proofs in which witnesses have been produced for each of the variables in $\vec{y}$. Amphion/NAIF demonstrates that, using deductive synthesis, it is possible to create do-

main-specific systems that enable users to generate high-assurance software cost-effectively.

Deductive synthesis has several potential advantages over competing synthesis technologies. The first is the well-known but unrealized promise that developing a declarative domain theory costs less than developing a special-purpose synthesis engine through ad-hoc techniques. The second advantage is that synthesized programs are correct relative to a domain theory and the component library. The third advantage is that this relative correctness is rigorously documented in a verification proof, thereby potentially providing more understandable and readable code than even the best documented manually developed code. This latter potential advantage is considerable, as state-of-the-practice code generators produce programs that are unfit for human consumption or human maintenance. This potential of deductive synthesis has not been realized because raw, mechanically-generated proofs are also unfit for human consumption, and can only be understood through a laborious process by experts in theorem-proving technology.

This paper describes a tool we have developed (and have nearly finished implementation as of May 1998) to provide explanations of programs generated by Amphion. It will first be used to automatically insert comments into generated code that document the relation between program variables and parts of the specification and domain theory. It will then be used to provide a dynamic web-browsable explanation of all aspects of generated code, enabling a user to probe the rationale for a generated program in terms of the domain theory. Based on our past experience in validating and debugging Amphion domain theories, this facility is expected to enable domain experts to home in on parts of a domain theory that lead to faulty programs. This is part of the larger goal of the Meta-Amphion project: enabling domain experts to construct, validate, and maintain their own high-assurance software synthesis systems.

The next section of this paper introduces the explanation tool through a simple example from the NAIF domain. Section 3 then provides an overview of the Amphion deductive synthesis system, sufficient to understand the technical development in the rest of the paper, illustrated with this same example. Section 4 then illustrates the mechanics of generating an explanation. Section 5 develops the mathematical framework for generating explanations. Section 6 describes our tracing algorithms. Section 7 then relates this work to previous work and discusses future work.

Introduction to the Explanation Tool

We illustrate the need for an explanation tool with the following example. Figure 1 shows a simple specification given to the Amphion/NAIF system.

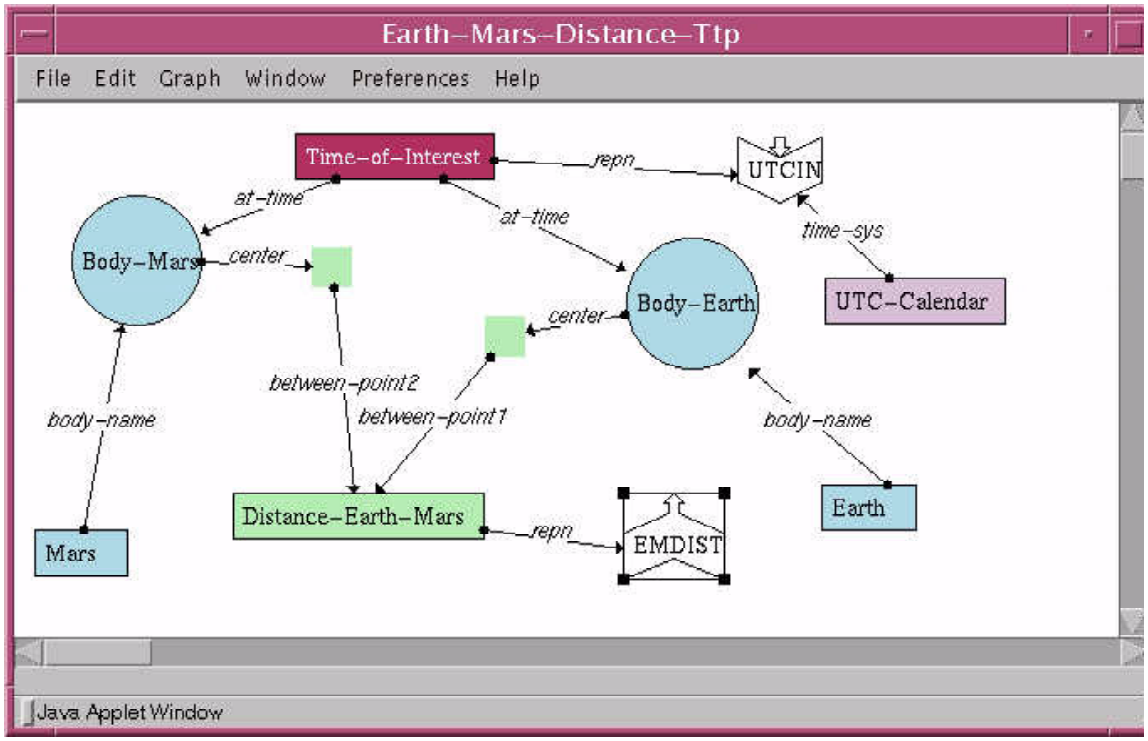

Figure 1: A simple specification for Amphion/NAIF

This specification depicts the constraints on a program that takes a time as input and produces as output the distance between the center of Earth and the center of Mars at that time. In general, specifications are given at an *abstract*

level and programs are generated at a *concrete* level (input/output parameters are exceptions to this). Abstract objects are free from implementation details; thus, a point is an abstract concept, while a Fortran array of three real numbers is a concrete, implementation-level construct. The concrete array may represent the point, in which case this representation must be further specified by a coordinate system and the origin and orientation of the coordinate axes.

The input in Figure 1(*UTCIN* in the upper left of the diagram) is depicted by a chevron with an arrow pointing into it. This chevron is connected to two other objects, *Time-of-Interest* and *UTC-Calendar*. *Time-of-Interest* is an abstract time, that is, an object representing the concept of a particular time. *UTC-Calendar* is a concrete time system. The arrows indicate that the abstract *Time-of-Interest* is represented in a program by a data object, *UTCIN*, which is interpreted as a time coordinate in *UTC-Calendar* format. *Body-Earth* and *Body-Mars* are abstract objects; each represents the *state* (space-time location and orientation) of its respective planet at the given time. Hence, *Body-Earth* is constrained to be the state of the planet with the name *Earth* at *Time-of-Interest*. The variable *Distance-Earth-Mars* is the abstract distance between the centers of the two bodies. Finally, the output *EMDIST* (the chevron in the lower right of the diagram) is the concrete representation of the abstract distance in kilometers.[1]Amphion/NAIF will generate the program shown in Figure 2 from this specification:[2]

```
      SUBROUTINE EARTH0 ( UTCIN, EMDIST )
          IMPLICIT NONE
          DOUBLE PRECISION SLOC(3)
      C   Code for EARTH-MARS-DISTANCE
      C   Request-id: REQ-1998-04-06-14-58-47-423
      C   Parameters
      C   MARSNA is Mars-NAIF-ID
          INTEGER MARSNA
          PARAMETER ( MARSNA = 499 )
      C   EARTHN is Earth-NAIF-ID
          INTEGER EARTHN
          PARAMETER ( EARTHN = 399 )
      C   Input variables
          CHARACTER*(*) UTCIN
      C   Output variables
          DOUBLE PRECISION EMDIST
      C   Functions
          DOUBLE PRECISION VDIST
          LOGICAL RETURN
      C   Local variables
          DOUBLE PRECISION E
          DOUBLE PRECISION SMARS ( 6 )
          DOUBLE PRECISION SEARTH ( 6 )
          DOUBLE PRECISION PSMARS ( 3 )
          DOUBLE PRECISION PSEART ( 3 )
      C   Error handling
          IF ( RETURN() ) THEN
            RETURN
          ELSE
            CALL CHKIN ( 'EARTH0' )
          END IF

          CALL UTC2ET(UTCIN,E)
          CALL
           SPKSSB(EARTHN,E,'J2000',SEARTH)
          CALL
           SPKSSB(MARSNA,E,'J2000',SMARS)
          CALL ST2POS(SEARTH,PSEART)
          CALL ST2POS(SMARS,PSMARS)
          EMDIST = VDIST(PSEART,PSMARS)

          CALL CHKOUT ( 'EARTH0' )
          RETURN
          END
```

Figure 2: Resulting Fortran program

It is difficult for the developers of Amphion/NAIF, let alone space scientist users that are not familiar with the component library, to understand the relationship between this program and the specification given above - and this is an extremely simple example. As the complexity of specifications increases, it becomes increasingly difficult to understand this relationship. The goal of the tracing tool is to make this relationship explicit.

The actual computation performed in this example consists of the sequence of subroutine and function calls that are shown in bold font. A good way to explain the links from this program to the specification is to explain the relationship between the Fortran variables and the variables in the specification. Our tool automatically inserts comments that provide this explanation. In doing so, the tool exploits the structure of Amphion domain theories, so the end user is not confronted with the intricacies of raw proof traces. For the program in Figure 2, the following comments are inserted before the sequence of calls:

```
      C   E represents Time-of-Interest in the ephe-
      C   meris time system
      C   SEARTH is an intermediate value used in
      C   computing the-center-of-Body-Earth
      C   SMARS is an intermediate value used in
      C   computing the-center-of-Body-Mars
      C   PSEART represents the-center-of-Body-
      C   Earth in rectangular coordinates J2000 at
      C   Time-of-Interest
      C   PSMARS represents the-center-of-Body-
```

[1] This version of the domain theory only has one representation for distances, namely kilometers. Thus this concrete object is not parameterized.

[2] Amphion/NAIF also generates a driver main program for this subroutine, but in the interest of simplifying the presentation, this is not shown here.

```
C     Mars in rectangular coordinates J2000 at
C     Time-of-Interest
C    EMDIST is Distance-Earth-Mars in kilome-
C     ters
```

The Amphion Deductive Synthesis System

This section describes the components of our deductive synthesis system in sufficient detail to motivate the implementation of the tracing tool. A more detailed treatment of the system can be found in [5, 7].

Amphion consists of three subsystems: a specification acquisition subsystem; a program synthesis subsystem; and a domain-specific subsystem. Figure 3 presents a flow diagram of the system, where the dotted lines enclose subsystems, the rectangles enclose major components, and the ovals enclose data.

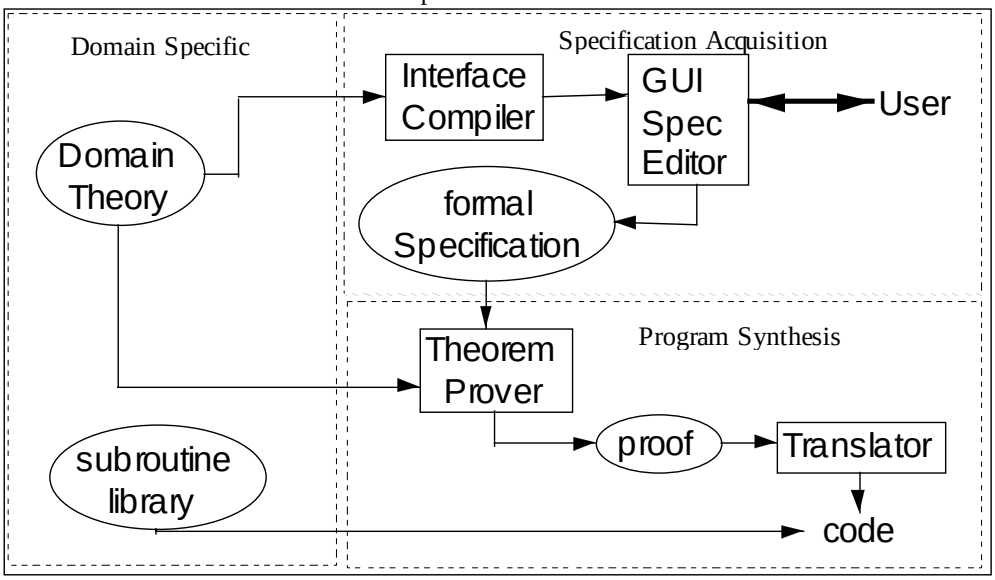

Figure 3: Amphion Block Diagram

1.1 Specification Acquisition

The specification acquisition system includes a graphical user interface that enables a user to interactively build a diagram representing a formal problem specification in first-order logic. A graphical specification is automatically converted into a formula to be proved. Thus, the graphical specification diagrams are equivalent to specifications (in first-order logic) of the following form:

lambda (inputs)
find (outputs)
exists (intermediates)
$conjunct_1$ &...& $conjunct_N$

The input variables of the specification (bound by the *lambda*) are universally quantified, while the output variables (bound by the *find*) are existentially quantified within the scope of the input variables. The intermediates (bound by *exists*) are also existentially quantified within the scope of the input variables. The conjuncts are all expressed in the abstract specification language, except for conjuncts expressing the relationships between concrete input or output variables and the abstract variables they represent. The first-order form of the specification depicted in Figure 1 is given in Figure 4.

```
(LAMBDA (UTCIN)
  (FIND (EMDIST)
   (EXISTS
     (Time-of-Interest Body-Earth Body-Mars
      the-center-of-Body-Earth
      the-center-of-Body-Mars
      Distance-Earth-Mars)
   (AND
    (= Body-Earth
       (BODY-ID-AND-TIME-TO-BODY
          Earth Time-of-Interest))
    (= Body-Mars
       (BODY-ID-AND-TIME-TO-BODY
          Mars Time-of-Interest))
    (= the-center-of-Body-Earth
      (BODY-TO-CENTER Body-Earth))
    (= the-center-of-Body-Mars
       (BODY-TO-CENTER Body-Mars))
    (= Distance-Earth-Mars
       (TWO-POINTS-TO-DISTANCE
            the-center-of-Body-Earth
            the-center-of-Body-Mars))
    (= Time-of-Interest
```

```
      (abs (coordinates-to-time UTC-Calendar)
           UTCIN))
   (= EMDIST
      (UIREPN-KILOMETERS-TO-DISTANCE
         Distance-Earth-Mars))))))
```

Figure 4: Formal Specification of the Earth-Mars-Distance problem

1.2 Program Synthesis

The program synthesis subsystem consists of a program generator (theorem prover) and a translator that generates code in the syntax of the target programming language. A functional (applicative) program is generated through deductive synthesis [4]. Amphion/NAIF uses the SNARK refutation-based theorem prover [7] to generate a proof that the specification is a theorem of the domain theory. SNARK has inference rules for binary resolution, paramodulation, and demodulation. The proof cycle consists of one application of either binary resolution or paramodulation followed by an arbitrary number of demodulations of the resulting clause. 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 the target language whose function symbols correspond to the components of the software library.

We refer to the answer term generated in the final step of the proof process (Figure 5) as the *applicative term* of the proof. We refer to the entire derivation process, including the proof and the transformation of the applicative term, as a *derivation* . The applicative term generated in proving the specification shown in Figure 4 is shown in Figure 5.

```
vdist
 (findp
   (earth-naif-id,
     convert-time(utc-calendar,ephemeris-time-ts,utcin)),
  findp
   (mars-naif-id,
    convert-time(utc-calendar,ephemeris-time-ts,utcin)))
```

Figure 5: Applicative term for the Earth-Mars distance problem

The functional program is translated into a target programming language, such as Fortran for the NAIF domain, through program transformations. One set of transformations introduces a bound variable for each complex subterm (e.g. E for convert-time(utc-calendar,ephemeris-time-ts,utcin)). Another set of transformations handles subroutines with multiple output values. In the final stage, variable declarations and the sequence of component calls are generated in the syntax of the target language. Recall that the Fortran program resulting in our example is shown in Figure 2.

1.3 Domain Specific Components

The domain specific components of an Amphion application consist of a domain theory and a component library. An Amphion domain theory has three parts: an abstract theory whose language is suitable for problem specifications, a concrete theory that includes the target component specifications, and an implementation relation between the abstract and concrete theories. The implementation relation is axiomatized through abstraction maps using a method described by Hoare [3]. The domain theory is created by acquiring knowledge from a domain expert, developing appropriate domain abstractions and pre/post conditions for the software components, and then constructing axioms.

Generating an Explanation through Tracing a Derivation

This section informally illustrates the process of generating an explanation through the example of the Earth-Mars-Distance-Ttp program. The succeeding sections develop the mathematics and algorithms of the formal explanation generation process. The process starts with the generated Fortran program, considered to be the root of a derivation tree, and traces backwards to the specification and domain theory, considered to be the leaves of the derivation tree. Both the proof, resulting in an applicative answer term, and the trace of the program transformations, that result in an abstract syntax term for the Fortran program, are part of this derivation tree. An explanation is an equality connecting parts of the Fortran program to constructs in the specification and domain theory, from which it was derived. This equality is extracted from the derivation tree. Note, however, that even though explanations are represented as equalities, our technique is not limited to equational theories. For example, Section 0 describes explanation generation for proofs containing paramodulations. Niether of the formulas used in a paramodulation step is required to be an equality.

In this paper we will focus on explaining the variables in a Fortran program, but in general all the constructs in a generated program are connected to specification and domain theory axioms.

Consider the problem of connecting the Fortran variable PSMARS (see Figure 2) to the specification term representing the center of body Mars in order to generate the comment “PSMARS is the position of Mars in rectangular coordinates J2000 at the Time-of-Interest.” In the generated Fortran program (Figure 2) PSMARS appears as an output variable of a call of the Fortran subroutine ST2POS whose input is the variable SMARS. SMARS in turn is the output of a call of the subroutine SPKSSB whose inputs are

MARSNA, E, and J2000. By inspecting the program transformation trace, it can be determined that this call to SPKSSB was generated by the program transformations from the subterm *findp(mars-naif-id,convert-time(utc-calendar,ephemeris-time-ts,utcin))* of the applicative term.

The remainder of the trace process proceeds backwards through the proof. This *findp(...)* term is generated by the demodulation step shown in Figure 6.

Demodulation using the axiom named
Ephemeris-object-and-time-to-position-to-findp:

```
(= (ephemeris-object-and-time-to-position
      (abs (naif-id-to-body-id) Bnid)
      (abs (coordinates-to-time ephemeris-time-ts) Et))
   (abs (coordinates-to-point rectangular j2000)
      (findp Bnid Et)))
```

on the subterm

```
(ephemeris-object-and-time-to-position
  (abs(naif-id-to-body-id) mars-naif-id)
  (abs (coordinates-to-time ephemeris-time-ts)
       (convert-time utc-calendar ephemeris-time-ts utcin)))
```

yields

```
(abs (coordinates-to-point rectangular j2000)
     (findp mars-naif-id
        (convert-time utc-calendar ephemeris-time-ts utcin)))
```

Figure 6: An example demodulation step

This step illustrates one of the complexities of the trace process, namely, the term being traced (*findp(...)*) is a subterm of the term introduced by this demodulation. The trace process needs to track the evolution of this subterm backwards through the derivation tree, where at each node it is only a subpart of the modified part formula.

Another complexity is that while *findp* is a concrete function symbol, *ephemeris-object-and-time-to-position* is abstract. Hence, this is a step where the term being traced has moved from the concrete level to the abstract level. In general, such steps are where additional information is inserted into a derivation, such as a quantity's units and representation. In this example, the demodulation step provides the units and representation for PSMARS, i.e., rectangular coordinates in the J2000 frame. This type of information is inserted into the explanation.

Continuing to trace backwards through the derivation tree, the next step is the demodulation shown in Figure 7.

Demodulation using the axiom
Body-to-center-to-ephemeris-object-and-time-to-position:

```
(= (body-to-center
      (body-id-and-time-to-body B T))
   (ephemeris-object-and–time-to-position B T))
```

on the subterm

```
(body-to-center
```

```
 (body-id-and-time-to-body (abs (naif-id-to-body-id)
                                mars-naif-id)
 (abs (coordinates-to-time ephemeris-time-ts)
      (convert-time utc-calendar ephemeris-time-ts,utcin))
```

yields

```
(ephemeris-object-and-time-to-position
  (abs(naif-id-to-body-id) mars-naif-id)
  (abs (coordinates-to-time ephemeris-time-ts)
       (convert-time utc-calendar ephemeris-time-ts utcin)))
```

Figure 7: Another example demodulation step

Tracing back from this term through several more demodulations, we arrive at the leaves of the derivation tree-namely, the subterm*(BODY-TO-CENTER Body-Mars)* in the specification . This subterm is equated to the specification variable *the-center-of-Body-Mars*. (see Figure 4). Hence, the trace back to the specification is complete.

Mathematical Framework

Derivations consist of annotated proof trees and annotated transformation trees. Neither the SNARK theorem prover nor the transformation system record sufficient information to enable explanations to be extracted from derivations. The first step in implementing the tracing algorithms described in the next section was to augment both these Amphion subsystems to provide the needed annotations. These annotations explicitly record those parts of a formula changed in each step in a proof or program transformation. Furthermore, the generated Fortran programs are represented as abstract syntax terms. This provides a uniform representation for the nodes of a derivation from the specification to the generated program. Derivations consist of nodes which are formulas and arcs between nodes which correspond to proof steps or program transformations. Because transformations are treated as just another type of derivation step, we make no distinction in what follows between the proof process and the transformation process.

A subformula is specified by a path description from the root of the formula to the root of the subformula. A path description is a sequence of argument position selectors, e.g., using notation similar to [8], the path [2, 1] specifies subterm *b* of the term *f(a,g(b,c))*. The expression *paths(t)* is the set of valid paths in the term *t*. The subterm of a term *t* selected by a valid path *p* of *t* is written *t@p*. The concatenation of two paths *p* and *q* is denoted by *p+q*. For paths *p* and *q*, we write $p<q$ if *p* is a proper prefix of *q*. In this case, *t@q* is a subterm of *t@p*.

In generating explanations, it is important to avoid getting "mixed up" when the same term occurring in different places has different derivations. To avoid this, we extend the notion of a path to that of a *location*. A location is a pair $<n,p>$, where *n* is a formula number and *p* is a valid path in that formula. Every formula in a derivation is assigned a

unique number. If the same formula is used more than once in a derivation, the different uses have different numbers. If l is $\langle n,p \rangle$, then $@l$ is defined to be $formula_n@p$. We label each nonlogical symbol in a formula by its location and we label each variable with its formula number. For example,

$formula_n=(P_{n,[0]}\ (f_{n,[1,0]}\ x_n)\ (g_{n,[2,0]}\ (f_{n,[2,1,0]}\ y_n\ x_n)))$.

Formally, a derivation is a directed acyclic graph whose nodes are derivation steps and whose arcs encode the "derived from" relation. The root of a derivation contains the final formula and answer term resulting from a derivation, and the leaves contain the specification formula and axioms of the domain theory. Each derivation step is a triple $\langle F,T,A \rangle$, where A is an inference rule application, and F and T are the resulting formula and answer term, respectively. Each application of an inference rule specifies: the inference rule that was applied (one of demodulation, paramodulation, or resolution); the input formulae $F_1, ..,F_n$; and locations of the subterms to which the rule applied. In steps of the derivation, subterms of formulas will be substituted into the answer term T.

In general, an explanation of a Fortran program in terms of the abstract specification and domain theory is a collection of explained connections between the variables, functions and subroutines in the Fortran program and the objects, relations and functions in the problem specification or domain theory. For example, the purpose of the Fortran variable PSMARS in Figure 2 is explained by identifying its relationship to the specification object representing the center of body MARS (see Figure 4). We represent the collection of explained connections as a set of equalities between locations (paths in formulas) in a derivation. Then an explanation of the relationship between a Fortran variable and a specification object is an equality consequence of the specification formula union the explanation equalities of the derivation.

Formally, we can associate with each step a set of *explanation equalities* which are equality assertions of the form $t_1=t_2$, where t_1 is $\Phi@p_1$ and t_2 is $\Psi@p_2$, or of the form $x=t$ which comes from a substitution. Each equality is a consequence of the semantics of the inference rule and of the input formulae. In our implementation, the explanation equalities of a step are represented implicitly by the functions **map-back** and **pass-through** described in Section 0. The explanations are a theory of only the connections that were created by the derivation, not of all possible equality relations that might be inferred from the input specification and the domain theory.

A *goal explanation equality* for a program variable is an equality of the form:

(= <a possibly empty composition of selectors applied to an abstract variable>
<the abstraction of the concrete variable to be explained>)

e.g.,

```
(= the-center-of-Body-Mars
   (abs(coordinates-to-point rectangular j2000)
       PSMARS)),
```

As an example of computing a goal explanation equality, consider the following explanation equalities of the PSMARS derivation (in which we have omitted the location labels, but keep in mind that these are really equalities between subtrees):

```
(= PSMARS
  (findp mars-naif-id
    (convert-time utc-calendar ephemeris-time-ts utcin)))

(= (abs (coordinates-to-point rectangular j2000)
        (findp mars-naif-id
          (convert-time utc-calendar ephemeris-time-ts
                        utcin)))
   (ephemeris-object-and-time-to-position
     (abs (naif-id-to-body-id) mars-naif-id)
     (abs (coordinates-to-time ephemeris-time-ts)
          (convert-time utc-calendar ephemeris-time-ts
                        utcin))))
```

{This explanation equality comes from the demodulation step in Figure 6.}

```
(= (ephemeris-object-and-time-to-position
     (abs (naif-id-to-body-id) mars-naif-id)
     (abs (coordinates-to-time ephemeris-time-ts)
          (convert-time utc-calendar ephemeris-time-ts
                        utcin)))
   (body-to-center
     (body-id-and-time-to-body
       (abs (naif-id-to-body-id) mars-naif-id)
       (abs (coordinates-to-time ephemeris-time-ts)
            (convert-time utc-calendar ephemeris-time-ts
                          utcin)))))
```

{This comes from the demodulation step in Figure 7.}

Additionally, the following equalities are a consequence of the specification (Figure 4):

```
(= (body-id-and-time-to-body
        (abs (naif-id-to-body-id) mars-naif-id)
        (abs (coordinates-to-time ephemeris-time-ts)
             (convert-time utc-calendar ephemeris-time-ts
                           utcin)))
   Body-Mars)

(= the-center-of-Body-Mars  (body-to-center Body-Mars))
```

The derived goal explanation equality :

```
(= the-center-of-Body-Mars
  (abs (coordinates-to-point  rectangular j2000) PSMARS))
```

is an equality consequence of this set of equalities. Standard methods for reasoning about sets of ground equalities can be used to generate this consequence. For instance, congruence closure can finitely represent the possibly infinite set of consequences of a set of ground equalities. Given this goal explanation equality, the comment explaining PSMARS is generated directly through template instantiation.

We now formalize the definition of a demodulation step that uses the equality $s=t$ to rewrite the subterm s' of a formula Φ specified by the path p (i.e. $s'=\Phi@p$). Let σ be a substitution such that $s'=s\sigma$. The result of the demodulation step $demod(\Phi,p,s=t)$, is a formula Ψ. The explanation equalities of this application of demodulation are as follows. Because $s=t$, substitution of $t\sigma$ for s' does not change the denotation of any subterm of Φ that is not a subterm of s'. More formally, for any $q \in paths(\Phi)$, if $\neg(p<q)$ then $\Phi@q=\Psi@q$. Explanation equalities between subterms of $\Phi@p$ and $\Psi@p$ are specified by the substitutions $mgu(s,\Phi@p)$ and $mgu(t,\Psi@p)$. Here mgu is the most general unifier of two terms ignoring the symbol labels. Those labels are, however, kept in the resulting unifier.

This approach generalizes to all of the inference rules in our system. As another example, the explanation equalities of a paramodulation step are as follows: Let Γ be an axiom of the form $(s=t \vee Q)$, where Q is a disjunction of literals. Let $paramodulate(\Phi,p,\Gamma)$ be Ψ, the result of paramodulating Γ into $\Phi[s']$ at p, where $s'=\Phi@p$ and $\sigma=mgu(s,s')$. The resulting formula Ψ is a renaming of $\Phi[t]\sigma \vee Q\sigma$. The explanation equalities are σ, σ_Φ, and σ_Q, where $\sigma_\Phi = mgu(\Psi@[1]+p,\Phi@p)$ and $\sigma_Q=mgu(\Psi@[2],Q)$.

The following extract from the derivation of the Fortran program in Figure 2 illustrates the explanation equalities of a paramodulation step:

```
Paramodulating
Ephemeris-object-and-time-to-position-to-findp
into
Φ= [(≠ (two-points-to-distance
          p(ephemeris-object-and-time-to-position
            (abs(naif-id-to-body-id) p',earth-naif-id)
            (abs(coordinates-to-time X)
              p'',(convert-time utc-calendar X utcin)))
          (ephemeris-object-and-time-to-position
            (abs (naif-id-to-body-id) mars-naif-id)
            (abs (coordinates-to-time X)
                 (convert-time utc-calendar X utcin))))
        (abs (kilometers-to-distance) Y))]
resulting in
Ψ=[(≠ (two-points-to-distance
          p(abs(coordinates-to-point rectangular j2000)
            (findp q',earth-naif-id
              q'',(convert-time utc-calendar ephemeris-time-ts
                                utcin))
          (ephemeris-object-and-time-to-position
            (abs (naif-id-to-body-id) mars-naif-id)
            (abs (coordinates-to-time q''',ephemeris-time-ts)
              (convert-time utc-calendar ephemeris-time-ts
                            utcin))))
        (abs (kilometers-to-distance) Y))]

Γ=[(= (ephemeris-object-and-time-to-position
        (abs (naif-id-to-body-id) Bnid)
        (abs (coordinates-to-time rephemeris-time-ts) Et))
      (abs (coordinates-to-point rectangular j2000)
        (findp Bnid Et)))]

Ψ@p=Φ@p
Ψ@q'=Φ@p'
Ψ@q''=Φ@p''
Ψ@q'''=Γ@r
```

Figure 8: An example paramodulation step

The explanation equalities of the other inference rules in our derivation system, namely resolution and transformation, are defined similarly.

The Explanation Tool

Rather than computing explanations by explicitly computing the consequences of all the explanation equalities of a derivation, our implemented algorithm uses a more focused approach that traces back through a derivation from a term of interest in the abstract syntax tree of the Fortran program and assembles a (usually small) subset of the explanation equalities in the derivation. It uses congruence closure to compute the goal explanation equality from this focused subset. Pseudo-code for the algorithm is now presented.

```
Procedure generate-traces(program, derivation)
  For each variable in program do
    loc ← last location of variable in
            answer-term(root(derivation))
    trace ← compute-trace(derivation, loc, {})
    traces ←traces ∪ trace
  End
  return(traces)
End generate-traces

Procedure compute-trace
          (derivation, loc, explanation)
  If contains-goal-equality(explanation)
  then return(goal-equality(explanation))
  step←root(derivation)
  If location-affected-by-step?(loc, step)
  then
    starting-loc, derivation, exp-eqs←
        map-back(loc, step, derivation)
    explanation←
      congruence-closure(explanation ∪ exp-eqs)
  Else
    starting-loc, derivation←
          pass-through(loc, step, derivation)
  Endif
  Return(Compute-trace
          (derivation, starting-loc, explanation))
  End
End compute-trace
```

Generate-traces finds the location of the last occurrence of each Fortran variable in the abstract syntax tree generated in the last step of the **derivation**. It calls **compute-trace** with the location of each program variable. **Compute-trace** takes a derivation, a location and the set of explanation equalities assembled so far (starts out empty) and maps the location back through the derivation structure, picking up explanation equalities. As it assembles explanation equalities, it incrementally computes the congruence closure of those equalities and, hence, represents the equality consequences. It continues this mapping process until a goal explanation equality is derived for the Fortran variable. The function **generate-comment** (not shown here) massages the goal explanation equality into a human readable comment through template instantiation.

Loc is the location of interest described as a path into the formula in a derivation step. **Starting-loc** is the location of interest, similarly described, in the previous formula. This is the location from which **loc** is derived. The function **location-affected-by-step?** determines whether or not a given location is affected by a step by checking whether the location of interest is contained in an affected subformula of the **step**. If it is not, the map back to the location of interest in the previous formula is the identity. For example, in a demodulation step, if the location of interest (**loc**) is not in the subformula that is the result of the demodulation, then the subformula of interest is the same location in the previous formula and is unchanged by the step. This identity connection is implemented by **pass-through** which returns **loc** and a partial derivation whose last step is the previous formula. Identity connections are not included in explanation traces, so **pass-through** does not return any explanation.

If the location of interest is affected by a step, the function **map-back** is called to identify the location of interest in the previous formula and to assemble the relevant explanation equalities from the derivation step (node). This function returns the new location of interest and the assembled explanation equalities.

Related and Future work

The technical approach described in this paper is most closely related to the literature on origin tracking in the rewrite community [8]. Origin tracking relates variables and subexpressions in a final program to the initial program by tracking the evolution of the abstract syntax tree under rewriting. Our framework based on equalities handles the full range of deductive synthesis rules, including paramodulation and resolution, not just rewriting. Because of the more general framework of deductive synthesis, we also need to allow explanations to terminate in portions of the domain theory, and not just the goal clause.

The mechanism described in this paper for providing explanations of variables in synthesized programs can also be used for several other purposes including: providing explanations of function and subroutine invocations in the generated program, providing explanations of fragments of intermediate steps in deductive synthesis derivations, and inserting assertions of expected program behavior into generated code. This latter capability will be used in future work to explore the use of formal explanations to provide feedback loops from generated program execution to specification revision. The explanations will be extended to provide descriptions of expected program behavior at intermediate points as related back to the specification. Discrepancies between expected and actual program behavior will be used to modify specifications, either in an advisory or automated capacity.

References

[1] Y. Bertot “Occurrences in Debugger Specifications”, Proceedings of the ACM Conference on Programming Language Design and Implementation, pp. 327 - 337, 1991.

[2] Felty, A., and Miller, D. “Proof Explanation and Revision”, University of Pennsylvania Technical Report MS-CIS-88-17, 1987.

[3] Hoare, C.A.R., “Proof of Correctness of Data Representations,” Acta Informatica, pp. 271-281, 1973.

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

[5] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood, “A Formal Approach to Domain-Oriented Software Design Environments,” Proceedings of the Ninth Knowledge-Based Software Engineering Conference, 1994.

[6] M. Lowry and J. Van Baalen, “META-Amphion: Synthesis of Efficient Domain-Specific Program Synthesis Systems”, *Automated Software Engineering*, vol 4, pp. 199-241, 1997.

[7] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood, “Deductive Composition of Astronomical Software from Subroutine Libraries,” *CADE-12*, 1994.

[8] van Deursen, A., Klint, P. and Tip, F., “Origin Tracking,” Journal of Symbolic Computation 15:523-545, 1993.