Automating Software Reuse with Amphion

Thomas Pressburger and Michael Lowry
Recom Technologies
NASA Ames Research Center MS 269-2
Moffett Field, CA 94035
{ttp,lowry}@ptolemy.arc.nasa.gov
http://ic-www.arc.nasa.gov/ic/projects/amphion

The construction of libraries of reusable software components is the standard software engineering solution for improving software development productivity and quality. By encapsulating usable functionality in software components (e.g. subroutines, object classes), and then reusing those components, software of greater functionality can be developed in less time, with some assurance that the overall system is correct because it is built from trusted components.

However, there are impediments to the use of software libraries. Obviously, nonprogrammers cannot use them directly. Even for programmers, there is a barrier in the form of the startup time to learn the library. The library may be inadequately documented, or, on the other hand, may be voluminously documented. Learning the conventions for using the library takes time, as well as learning which components implement the needed capabilities. Programmers who are not intimate with a particular library sometimes find it expedient to rewrite code from scratch rather than investigate thoroughly the capabilities of a given library. All of this is exacerbated because, we claim, conventional programming languages are not at a high-enough level. For example, third-generation and object-oriented languages do not allow for the specification of constraints. Components built from such programming languages will necessarily require that the user be cognizant of programming details, e.g. value-passing conventions and data representations, which are below the conceptual level of the application domain. If the language does provide a level above this, efficiency may be lost due to run-time interpretation.

The goal of the Amphion project is to provide a system with the following properties.

1. The target audience is end-users, or programmers familiar with the application domain but not necessarily the details of particular libraries.
2. The user develops a problem specification in a language that is readily assimilable by the user because it is high-level and domain-oriented. A specification-acquisition interface guides the user through the constructs of the language. Specifications, not programs, are maintained. The target of reuse will be not only software components, but specifications.
3. An automatic program synthesis system generates a program that correctly implements a solution to the specified problem. The generated program invokes components from the library to solve the problem.
4. The specification acquisition and synthesis systems are generic. They are retargeted to new domains by the addition of declarative knowledge that component developers, as opposed to program-synthesis experts, can provide. Our main application focus to date has been the particularly well-engineered FORTRAN-77 libraries SPICE, EUCLID/ESCHER, and PERCY developed by the Navigation and Ancillary Information Facility (NAIF) at NASA's Jet Propulsion Laboratory. These libraries are used e.g. by planetary scientists to plan and analyze the observation geometry for data collected during interplanetary missions. The application domains of these libraries are, respectively, solar system geometry, visualization, and search. The examples below are from this application of Amphion.

Specification Acquisition

Our current implementation of Amphion has as a specification acquisition component a generic, menu-guided GUI that enables the user to build a diagram that specifies a problem. An Amphion specification diagram bears a superficial resemblance to those used in dataflow-oriented graphical environments that enable users to select icons from a palette that represent individual subroutines, and then connect input and output ports. However, these environments only provide an alternate notation to conventional programming languages. In contrast, Amphion enables a separation between the level at which problems are specified and the level at which solutions to the problem must be programmed. The diagram provides an alternate notation to formal specifications written in mathematical logic. (The notation of mathematical logic can be formidable; that is one reason that specification-based software engineering life-cycles have not previously been adopted in practice.) The formal specification describes a set of abstract objects and their defining or constraining interrelationships. To complete the specification of a program, the user chooses which abstract objects will be input and output and what their data representations will be, chosen from those provided in the library.

The specification language defined in the declarative theory for the NAIF domain is at the level of abstract geometry. The vocabulary is basic Euclidean geometry (e.g., points, rays, ellipsoids) augmented with astronomical terms (e.g., planet and spacecraft objects, and *photon* constraints, the latter used to correct for light travel time over large distances). A specification does not need to include the myriad implementation details required for correctly calling SPICE subroutines, such as coordinate frames, units, time systems, etc.; these details are automatically deduced during program synthesis. The user does need to define the abstract problem and the desired representation conventions for the program inputs and outputs.

The diagram at the end of this note is an example of a completed specification: it specifies the problem of finding the solar incidence angle (ILLANG, the angle of the Sun from the local vertical) at the point on Jupiter (Observed-Point) pointed at by an instrument (INSTID) on the Galileo spacecraft at a particular time (TGAL). The input time TGAL has been specified to be in a time system particular to the Galileo spacecraft. The specification acquisition component performs a semantic check on the completed specification diagram, and then automatically translates it to a logical formula used by the program synthesis component.

Amphion's specification-acquisition component not only enables specifications to be developed from scratch, but also enables specification reuse and modification. It is easier and more reliable to use Amphion's graphical editor to modify the problem's high-level graphical specification and then resynthesize an implementation than it is to modify with a text editor the code of the implemented solution. Amphion also allows specifications to be encapsulated and reused in other specifications, just as programming languages provide constructs to encapsulate reusable functionality.

Program Synthesis

The current program synthesis system uses deductive synthesis [1,2] to deduce an applicative level program, which is then translated into the desired target language. The output of program synthesis for the NAIF domain is a FORTRAN-77 program consisting of straight-line calls to subroutines. (We have also output C++ and LISP for other domains.) One advantage of the deductive approach is that a synthesized program is guaranteed to be a correct implementation, with respect to the domain theory, of a user's specification. This reduces the software verification problem to a verification of the domain theory. For validating specifications, we are investigating visualizations of the situation described by the specification.

Deductive synthesis is performed by the SNARK resolution theorem prover for first-order logic developed by Mark Stickel at SRI International [3] and enhanced by Jeff Van Baalen and Steve Roach at the University of Wyoming so that it can use fast special-purpose decision procedures in general inference [4]. With suitable decision procedures specified, almost all search by the theorem prover has been removed. Amphion generated the program at the end of this paper from the specification in 16 seconds of CPU time on a Sparc 20.

Because it uses a generic architecture, Amphion can be applied to other domains and subroutine libraries by developing the appropriate domain theories. Reference [4] describes compilation of declarative domain theories into decision procedures suitable for program synthesis. This technology needs to be developed so that Property 4 above can be achieved.

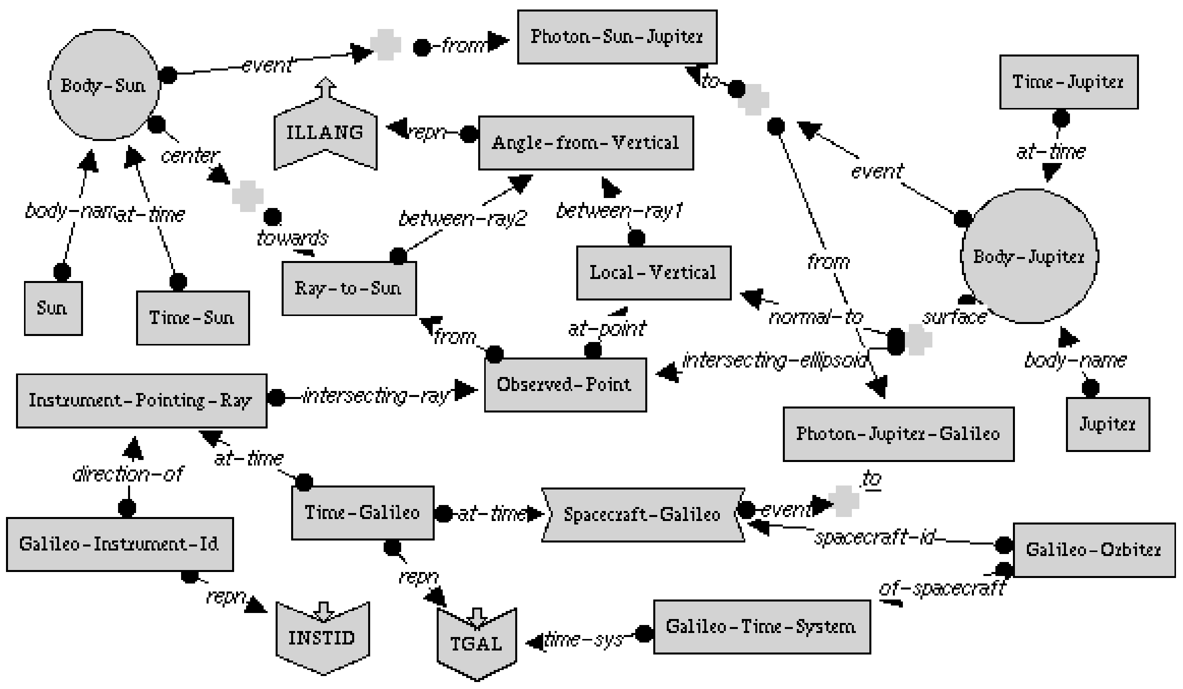
Further Work

We have also applied Amphion to the domain of solar-system visualizations by developing a theory of the rendering subroutines in the Escher/Euclid libraries. This resulted in new constructs at the specification level for *camera*, *illumination source*, and *illuminated body*. We used Amphion to specify a visualization that helped planetary scientists working on the Cassini mission to Saturn evaluate whether proposed tours could satisfy their observational requirements. It was also used by a planetary scientist at NASA Ames to generate the program that was the basis for the WWW ring-plane crossing Saturn viewer. The scientist created (admittedly with some hand-holding) an *informational* specification for calculating various desired quantities, e.g. scattering and opening angles, and a *visualization* specification of the view of Saturn from the Earth. There is a tutorial taking about two hours that teaches a new Amphion user how to create specifications.

Why the name Amphion? Amphion was the son of Zeus who used his magic lyre to charm the stones lying around Thebes into position to form the city's walls. Similarly, the Amphion system is able to charm subroutines into position to solve specified problems. We will continue to develop Amphion so that sophisticated software components can become more readily reusable.

References

1. Green, C., 1969. "Application of Theorem Proving to Problem Solving", IJCAI-69, pp. 219-239.
2. Waldinger, R., and Lee, R, 1969. "PROW: A Step Toward Automatic Program Writing", IJCAI-69.
3. Stickel, M., Waldinger, R., Lowry, M., Pressburger, T., and Underwood, I., 1994. "Deductive Composition of Astronomical Software from Subroutine Libraries", in 12th Conference on Automated Deduction.
4. Lowry, M., Van Baalen, J., "Meta-Amphion: Synthesis of Efficient Domain-Specific Program Synthesis Systems", Proc. of 10th Knowledge-Based Software Engineering Conference, Boston, Mass, Nov. 12-15, 1995, pp. 2-10.



```
;; Only the procedural statements are listed below.  "State" means position and velocity.
SUBROUTINE ILLUM ( INSTID, TGAL, ILLANG )
CALL BODVAR ( JUPITE, 'RADII', DMY1, RADJUP )   RADJUP = Jupiter's radii.
CALL SCS2E ( GALILE, TGAL, ETGALI )             ETGALI = TGAL converted to an internal time system.
X0 = I2SC ( INSTID )                            X0 = Galileo's ID.
CALL SPKSSB ( X0, ETGALI, 'J2000', PVX )        PVX = the state of Galileo at time ETGALI.
CALL SCE2T ( INSTID, ETGALI, TKINST )           TKINST = ETGALI converted to another system.
TJUPIT = SENT ( JUPITE, GALILE, ETGALI )        TJUPIT = the time when light left Jupiter.
CALL BODMAT ( JUPITE, TJUPIT, MJUPIT )          MJUPIT = the rotation matrix to Jupiter's frame.
CALL ST2POS ( PVX, PPVX )                       PPVX = extract Galileo's position.
CALL SPKSSB ( JUPITE, TJUPIT, 'J2000', PVJUPI ) PVJUPI = the state of Jupiter at time TJUPIT.
CALL CKGPAV ( INSTID, TKINST, TIKTOL, 'J2000',  C = rotation matrix to the instrument's frame.
              C, DMY2, DMY3, DMY4 )
TSUNN = SENT ( SUNNAI, JUPITE, TJUPIT )         TSUNN = the time when light left the Sun.
CALL ST2POS ( PVJUPI, PPVJUP )                  PPVJUP = extract Jupiter's position.
CALL MATROW ( C, 3, V )                         V = the instrument pointing vector.
CALL SPKSSB ( SUNNAI, TSUNN, 'J2000', PVSUN )   PVSUN = the state of the Sun at time TSUNN.
CALL VSUB ( PPVX, PPVJUP, DPPPP )               DPPPP = the vector from Jupiter to Galileo.
CALL MXV ( MJUPIT, V, XV )                      XV = V in Jupiter-centered coordinates.
CALL ST2POS ( PVSUN, PPVSUN )                   PPVSUN = extract the Sun's position.
CALL MXV ( MJUPIT, DPPPP, XDPPPP )              XDPPPP = DPPPP in Jupiter-centered coordinates.
CALL VSUB ( PPVSUN, PPVJUP, DPPPP0 )            DPPPP0 = the vector from Jupiter towards the Sun.
CALL SURFPT ( XDPPPP, XV, RADJUP(1), RADJUP(2), P = The vector from the center of Jupiter to the
              RADJUP(3), P, DMY5 )                  observed point. (SURFPT does intersection)
CALL MXV ( MJUPIT, DPPPP0, XDPPP0 )             XDPPP0 =  DPPPP0 in Jupiter-centered coordinates.
CALL SURFNM ( RADJUP(1), RADJUP(2), RADJUP(3),  PP = surface normal vector at the observed point.
              P, PP )
CALL VSUB ( XDPPP0, P, DPXDPP )                 DPXDPP = vector from observed point to the Sun.
ILLANG = VSEP ( PP, DPXDPP )                    ILLANG = the angle between PP and DPXDPP.
```