Generating Programs from Connections of Physical Models

Gordon S. Novak Jr. [*]
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
`novak@cs.utexas.edu`

October 27, 1997

This article appears in *Proceedings of the Tenth Conference on Artificial Intelligence for Applications*, San Antonio, Texas, March 1-4, 1994, pp. 224-230. IEEE Computer Society Press Order No. 5550-02.

Abstract

We describe a system that constructs a computer program from a graphical specification provided by the user. The specification consists of diagrams that represent physical and mathematical models; connections between diagram ports signify that corresponding quantities must be equal. A program (in Lisp or C) is generated from the graphical specification by data flow analysis and algebraic manipulation of equations associated with the physical models. Equations, algebraic manipulations, and unit conversions are hidden from the user and are performed automatically. This system allows more rapid generation of programs than would be possible with hand coding.

[*]Computer equipment used in this research was furnished by Hewlett Packard.

1 Introduction

Scientific programming is typically done in languages such as Fortran or C. It usually requires the programmer to select appropriate equations to model the physics and mathematics of the problem, manually manipulate the equations to compute the desired quantities, and convert units of measurement. Although powerful algebraic manipulation packages such as Mathematica [19] exist, they are not well integrated with the programming process. There are few machine checks of the correctness of equations or unit conversions, making it possible for errors to be overlooked.

In this paper, we describe a system called VIP (for View Interactive Programming). VIP allows a scientific program to be specified by means of graphical connections of diagrams that represent physical or mathematical principles. Each principle has a set of equations associated with it. When a specification is complete, a program is generated from the graphical specification by data flow and by symbolic manipulation of equations. The resulting program, in the GLISP language [11], can then be compiled into Lisp or, with an additional mechanical translation step, into readable C. The compilation process performs automatic conversion and checking of units of measurement.

The examples that have been done using VIP have shown that programs can be generated much more rapidly than is possible with conventional programming. The radar example, described below, is about 22 lines of code in C; it was produced in two minutes of user interaction using VIP. These results are an encouraging indication that this kind of programming interface and program generation may lead to increased programmer productivity.

2 Related Work

The SIGMA system [4] has a goal similar to ours: helping scientists to perform calculations based on a knowledge base of physical models. SIGMA differs from VIP in several ways. First, SIGMA is based on direct use and connection of equations. VIP is based on connections of physical principles, each of which may involve multiple equations. SIGMA, as reported in [4], did not yet have a graphical interface, although one was planned. In SIGMA, the user must specify directionality of data flow; we prefer to let the system determine data flow, removing this burden from the user. Finally, SIGMA is interpretive: it produces a queue of equations whose inputs are known and that are therefore ready for solution. VIP, in contrast, produces a program, making VIP useful both for quick calculations and for generating application programs.

The Sinapse system [3] synthesizes programs that solve differential equations by finite difference methods. This system is notable for generating programs that are moderately large (hundreds to thousands of lines) that deal with large spatial arrays and must therefore be efficient.

The use of diagrams as a means of communication and as a reasoning aid is common in science and engineering; Larkin and Simon [7] consider psycho-logical benefits of diagrams. With Bulko [13] [16], we have investigated the use of diagrams together with English text as a means of specifying physics problems. Diagrams are heavily used together with equations in books of standard formulas such as that of Gieck [2]. The use of circuit diagrams has a long history in electrical engineering. Related kinds of diagrams have also been used to specify programs. The LabView system [6] allows the creation of "virtual instrument" programs for measurement, display, and control by making graphical connections of computational modules; the connections represent data flow. Data flow graphs have also been used to specify parallel computations [10].

There have been many approaches to visual or graphical programming environments. Shu [17] contains a good selection of representative papers.

3 An Example

We begin with a simple example to illustrate the use of VIP. The problem could be stated as follows:

> An object is dropped and hits the ground with velocity `vel` (in *meters/second*). From what height was it dropped?

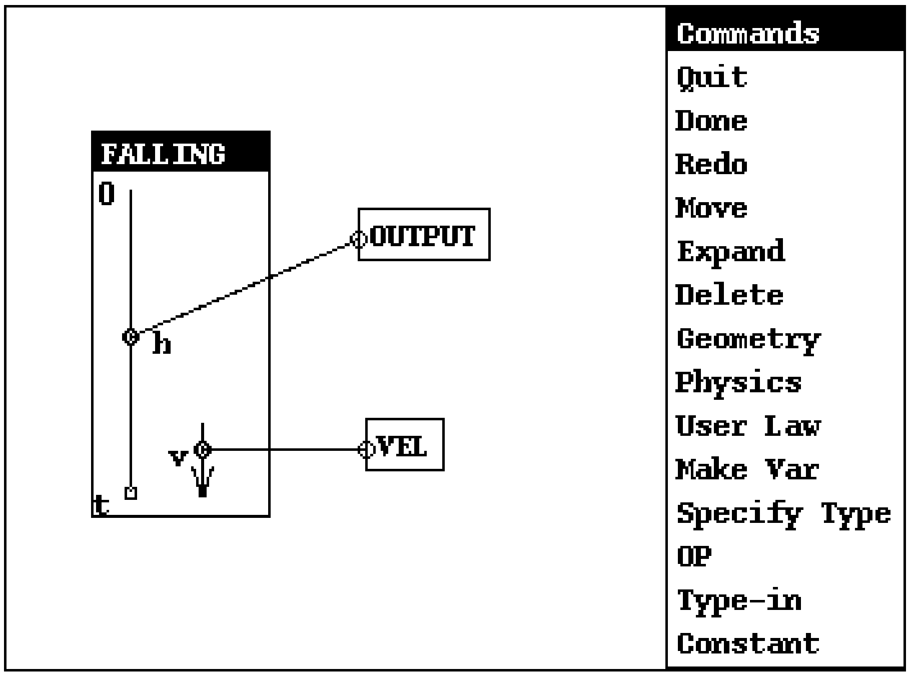

Figure 1: Height from which Object was Dropped

The user invokes VIP with a list of the input variables for the program and their types, and the name of the program that is to be produced:

```
(vip '((vel (units real (/ meter second))))
     't1)
```

In this case, there is a single `real` input variable `vel` whose unit is `(/ meter second)`. VIP responds by opening a window that contains a command menu, boxes for the input variables, and an `OUTPUT` variable (Fig. 1).

In order to model the physics of this problem, the user selects `Physics` from the command menu. A series of menus is then presented to allow the user to select the desired physical model from a hierarchical structure. The user selects `Kinematics` from the physics menu, then `Falling` from the kinematics menu. The `Falling` diagram is then added to the window. The user moves the variable boxes into appropriate positions, then connects these boxes to ports of the `Falling` diagram by clicking the mouse on the boxes and the ports to which they are connected. The system responds by drawing lines to show the connections that have been made. Finally, the user selects `Done` from the command menu. The system then produces a program in the GLISP language by data flow and by algebraic solution of equations associated with the `Falling` physical model. The GLISP program is shown below:

```
(LAMBDA (VEL: (UNITS REAL (/ METER SECOND)))
  (LET (OUTPUT)
    (OUTPUT :=
      (* (/ '(Q 9.80665 (/ M (* S S))) 2)
         (EXPT (/ VEL '(Q 9.80665
                          (/ M (* S S))))
               2)))
    OUTPUT))
```

The form `'(Q 9.80665 (/ M (* S S)))` is a numeric constant for the acceleration of gravity, including its unit of measurement. This program can be compiled into plain Lisp, as shown below:

```
>(glcp 't1)         ; compile function t1
result type: (UNITS REAL METER)
(LAMBDA (VEL)
  (LET (OUTPUT)
    (SETQ OUTPUT (* 0.0509858 (EXPT VEL 2)))
    OUTPUT))
```

The compiler has folded constant computations and has derived the units of measurement of the result. The program can also be mechanically translated into C:

```
>(gltoc 't1)    ; translate function t1 to C

/* Unit of T1 is METER */
float t1 (vel)
  float vel;
  {
    float output;
    output = 0.0509858 * square(vel);
    return output;
  }
```

This is a very simple example. Nevertheless, it required some algebraic manipulation of equations, the use of a physical constant, and derivation of the units of the result. All of these details were performed automatically by VIP; it is clear that it is much easier for the user to specify such a program by making diagrammatic connections than by manual algebra and programming. Larger examples are given later.

4 User Interface

VIP's user interface allows the user to take most of the initiative in specifying a program. The initial display consists of a command menu, a set of boxes that represent input variables, and a single `OUTPUT` variable. Variables can be data structures as well as simple numeric variables. The user builds a program specification by selecting physical or mathematical models and adding them to the workspace, and by connecting the models appropriately. The major commands that can be selected from the `Commands` menu are as follows:

1. `Geometry` and `Physics` allow the user to select mathematical and physical models and add them to the workspace. Each of these commands causes menus to be presented to the user for selection of the desired model from a tree-structured hierarchy. The top level of the Physics hierarchy includes `Kinematics`, `Dynamics`, `Force`, and `Energy`. When the user has selected the desired model, the system presents a "ghost" box so that the user can place it in the desired position. The diagram associated with the model is then drawn. (An interactive drawing utility program makes it easy to add new principles and their associated diagrams to the system.)

2. The `Make Var` command allows the user to specify the name and type of a variable, which is added to the workspace. This feature allows the user to specify intermediate variables if desired. If the type of a variable is a structured type, a new structure can be created from a set of component values.

3. The `Specify Type` command allows the user to specify the type and units of measurement of a variable.

4. The `OP` command selects arithmetic operations, such as `+`. This allows the user to specify equations that do not exist as predefined models.

5. The `Type-in` command allows the user to enter a numeric constant and its units.
6. The `Constant` command allows selection of commonly used physical constants (such as the speed of light) from a menu hierarchy.

Each diagram that represents a physical principle has "buttons" or ports that are shown on the diagram and labeled with mnemonic identifiers. Each button corresponds to a variable in the underlying set of equations for the model. In the diagram shown in Fig. 1, the `Falling` diagram has three buttons, labeled `h`, `v`, and `t`; these represent the height from which an object is dropped, the final velocity, and the time of falling, respectively. When the mouse pointer is moved close to a button, the button is highlighted by displaying a small box around it; if the mouse button is clicked while the pointer is inside the box, the button is selected. In the case of variables or constants, the entire box is treated as a single button. The user can connect two buttons by selecting them, in either order; the system responds by drawing a line connecting the selected buttons.

Once the user has made a complete specification, with the desired result connected to the `OUTPUT` box, the user selects the `Done` command. The system then converts the graph specification into a program.

5 Program Derivation

The diagrammatic specification given by the user corresponds to an undirected graph; this graph must be converted into executable code. The conversion is accomplished by data flow analysis and by algebraic manipulation. Each constant and input variable is assumed to be "solved". The value of a solved variable is propagated, in the form of a Lisp expression, to all of the ports to which it is connected that do not already have values.

When a value is propagated into a port of a model, the equations associated with the model are examined to see whether any of them can be solved. If an equation has only a single unknown, it can be algebraically rewritten to express that unknown in terms of solved variables. This makes that unknown solved, which can lead to the solution of other equations within the model itself; the solved variable will also be propagated to other models to which it is connected. If the expression that represents the solution of a variable is larger than a certain size, or is connected to multiple ports, a local temporary program variable is created, and code is output to set its value to the value of the expression; the program variable name then replaces the expression as the code for the variable. Because this process is performed only when a variable becomes solved based on previously solved values, it is guaranteed that the inputs to the expression will be available when the expression is computed. If a value is propagated into the `OUTPUT` variable, the specification of the program is complete.

In the case of the example of Fig. 1, the equations associated with `falling` are:

```
(setf (get 'falling 'equations)
      '((= g   '(q 9.80665 (/ m (* s s))))
    (= h   (* (/ g 2) (expt time 2)))
    (= v   (* g time)) ) )
```

When the input `vel` is propagated into the `falling` box as the value of the local variable `v`, this equation set is examined to see which equations can be solved. The first equation, which defines the gravitational constant, is trivially solved. The third equation then has only one unknown, and it can be solved for `time`:

```
   (= TIME (/ V G))
```

This causes the second equation to become solved in terms of `time`, so that all equations are now solved.

Variables whose types are data structures are treated differently from equations. When a value is propagated into a component of a structured variable, the value is saved. When all the components have become defined, code is produced to create an instance of the structured variable and assign it to the variable name. The structured variable itself can then be propagated.

This simple algorithm is surprisingly powerful, and it has been able to solve most of the examples we have tried. Although it is unable to solve simultaneous equations in the general case, it often is able to do so in practice. If there is a way to solve for a single variable at a time, this can allow simultaneous equations to be solved in steps. The equation handling of VIP allows redundant equations to be specified within a model, so that the model builder can predefine ways of solving for variables based on various sets of known quantities. When failure to solve simultaneous equations occurs, it is generally because multiple models are connected by multiple ports simultaneously. Clearly, VIP should be extended to be able to solve simultaneous equations.

The result of the program derivation process is a program expressed in the GLISP language. GLISP [11] is a high-level language with abstract data types that is compiled into Common Lisp; with an additional mechanical translation step, the Lisp output of GLISP can be translated into readable C. Several features of GLISP facilitate the production of programs by VIP. GLISP performs automatic conversion and checking of units of measurement. Therefore, VIP does not have to be concerned with units; GLISP inserts appropriate conversion factors automatically when needed. However, VIP probably should be modified to examine units to provide earlier detection of errors. GLISP allows overloading of operators, e.g., the operator `+` can be used with vectors as well as scalars; this allows the operators of VIP to be polymorphic. GLISP translates structure accesses and code that creates data structures into appropriate code for the implementation of the data structures; this allows VIP simply to refer to the components of structures without any concern for their implementations. We have investigated the creation of interfaces to subroutines [14] and the use of views of data as different types [15]; it would be useful to integrate these capabilities with VIP.

6 Larger Examples

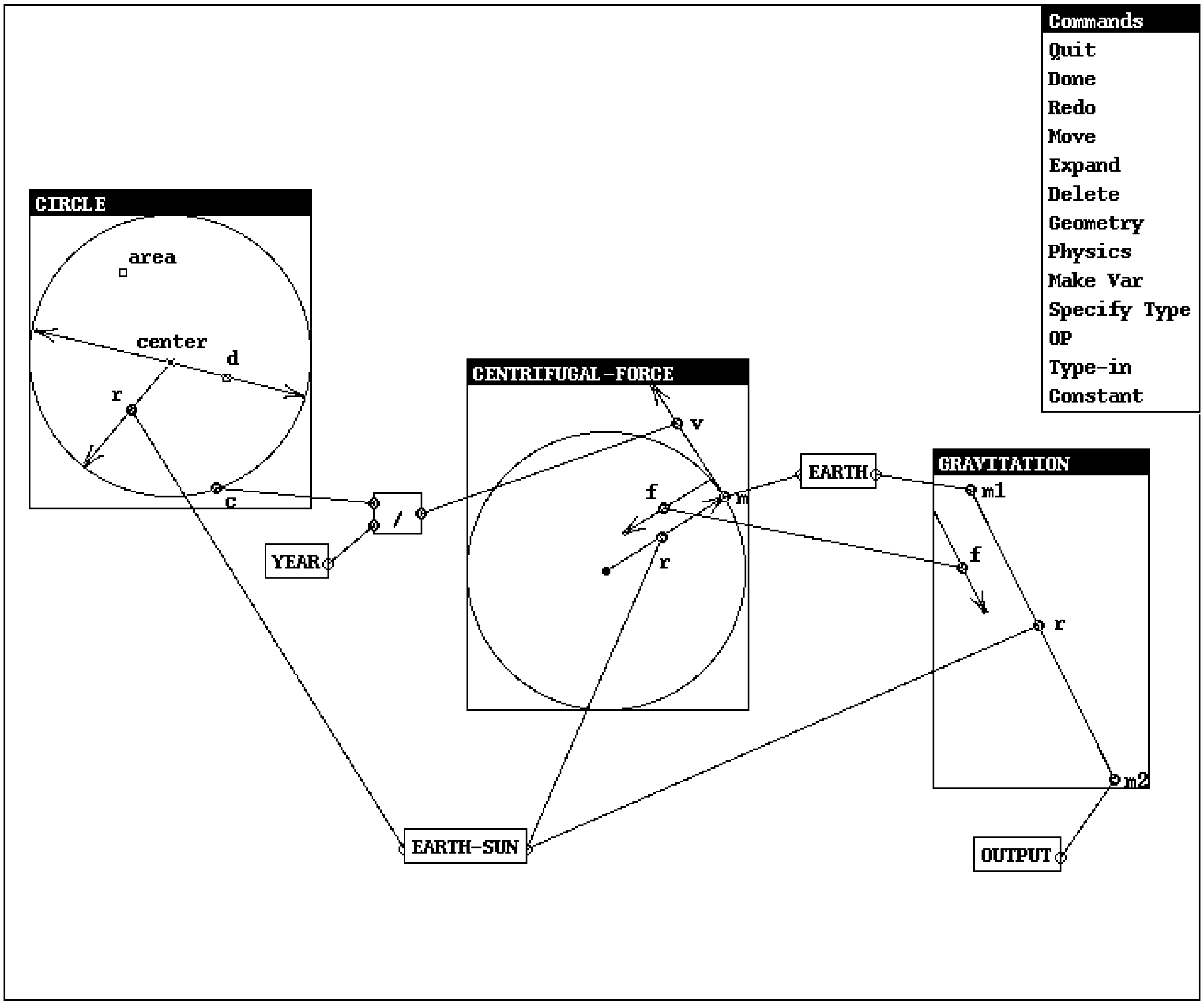

Figure 2: Calculation of the Mass of the Sun

In this section, we consider some larger examples that have been solved using VIP. The first example, shown in Fig. 2, is a physics problem: calculation of the mass of the sun. In this example, we follow Newton's reasoning: the gravitational attraction between the earth and the sun is equal to the force required to keep the earth in its (nearly) circular orbit. The user selects a `Gravitation` model and a `Centrifugal-Force` model and adds them to the workspace. The `f` buttons of these two models are connected, constraining the forces to be equal. A `Constant` for the mass of the earth is selected and connected to mass buttons in each model; likewise, the earth-sun distance is connected to the radius of each model. The second mass button in the `Gravitation` model, which will be the mass of the sun, is connected to the `OUTPUT` box. Following these actions, only the velocity of the earth in its orbit remains unspecified. This can be found by noting that the earth goes around the sun once each year. A `Circle` model is selected from the `Geometry` menu, and the earth-sun distance is connected to its radius. A division operator is then selected using the `OP` command; the circumference of the earth's orbit is connected to the numerator, and a time constant of one year is connected to the denominator. The output of this operator is then connected to the velocity in the `Centrifugal-Force` model. The resulting program has no inputs; it simply calculates a numeric result, which is `1.966E30` kilograms.

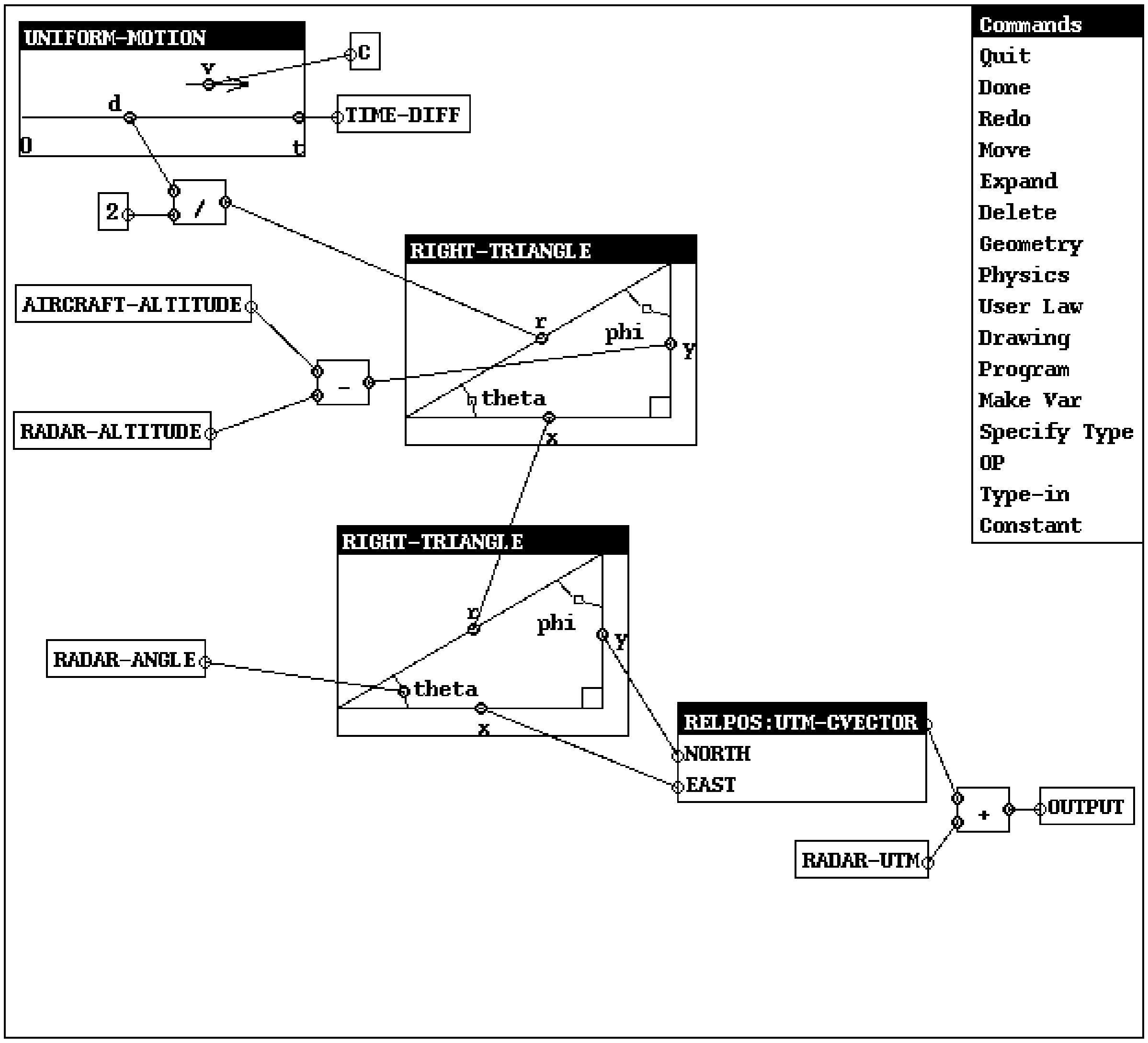

Figure 3: Position of Aircraft from Radar Data

Our second example, shown in Fig. 3, is a small but realistic version of an application problem: the calculation of the position of an aircraft from data provided by an air search radar. We assume that the radar provides as input the time difference between transmission and return of the radar pulse, as well as the angle of the radar antenna at the time the return pulse is detected. When the radar illuminates the aircraft, we assume that the aircraft transponder transmits the identity of the aircraft and its altitude. The position and altitude of the radar station are assumed to be known. These items comprise the input data provided to the program. We assume that the units of measurement of the input data are externally specified (e.g., by hardware devices), so that the program is required to use the given units.

```
(LAMBDA (TIME-DIFF:
          (UNITS INTEGER (* 100 NANOSECOND))
         AIRCRAFT-ALTITUDE:
          (UNITS INTEGER (* 10 FOOT))
         RADAR-ALTITUDE:
          (UNITS INTEGER (* 10 FOOT))
         RADAR-ANGLE: (UNITS INTEGER
                  (/ (* 2 PI RADIANS) 4096))
         RADAR-UTM: UTM-CVECTOR)
  (LET (OUT3 OUTPUT D2 OUT4 X3 Y2 X4
        RELPOS:UTM-CVECTOR)
    (OUT3 := (- AIRCRAFT-ALTITUDE
                RADAR-ALTITUDE))
    (D2 := (* '(Q 2.998E8 (/ M S)) TIME-DIFF))
    (OUT4 := (/ D2 2))
    (X3 := (SQRT (- (EXPT OUT4 2)
                    (EXPT OUT3 2))))
    (Y2 := (* X3 (SIN RADAR-ANGLE)))
    (X4 := (* X3 (COS RADAR-ANGLE)))
    (RELPOS := (A UTM-CVECTOR NORTH Y2
                              EAST X4))
    (OUTPUT := (+ RELPOS RADAR-UTM))
    OUTPUT))
```

Figure 4: GLISP Program Generated by VIP for Radar Problem

The user first decides to model the travel of the radar beam as an instance of `uniform-motion`. The user selects the `Physics` command, then `kinematics` from the `Physics` menu, then `uniform-motion` from the `kinematics` menu. The input value `TIME-DIFF` is connected to the time button `t` of the motion. Next, the user selects `Constant` and obtains the constant for the speed of light, denoted `C`, and connects it to the velocity `v` of the motion. The distance `d` of the motion then gives the total (out-and-back) distance from the radar to the aircraft; by dividing this distance by `2`, the one-way distance is obtained. This distance is connected to the hypotenuse of a `Geometry` object, `right-triangle`. The difference between the altitude of the aircraft and the altitude of the radar is connected to the `y` of this triangle. The `x` of this triangle is then the distance to a point on the ground directly underneath the aircraft. This distance and the angle of the radar give a range and bearing to the aircraft from the radar; by connecting these to another right triangle, `x` and `y` offsets of the aircraft from the radar are obtained. These are collected to form a relative position vector, `RELPOS`, which is added to the radar's UTM (universal transverse mercator) coordinates to form the output.

Note that the + operator is generic and can be applied to structured objects such as vectors so long as the operator is overloaded for the structured objects in GLISP.

```
CUTM  *tqc (time_diff, aircraft_altitude,
    radar_altitude, radar_angle, radar_utm)
  long time_diff, aircraft_altitude,
       radar_altitude, radar_angle;
  CUTM  *radar_utm;
  { long out1;
    CUTM  *output;
    float d1, out2, x1, y1, x2;
    CUTM  *relpos,  *glvar1621;
    out1 = aircraft_altitude
           - radar_altitude;
    d1 = 2.997925E8 * time_diff;
    out2 = d1 / 2;
    x1 = sqrt(square(out2)
              - 9.290304E14 * lsquare(out1));
    y1 = x1 * sin(0.00153398 * radar_angle);
    x2 = x1 * cos(0.00153398 * radar_angle);
    relpos = (CUTM*) malloc(sizeof(CUTM));
    relpos->north = 1.00000000E-7 * y1;
    relpos->east = 1.00000000E-7 * x2;
    glvar1621 = (CUTM*) malloc(sizeof(CUTM));
    glvar1621->east = relpos->east
                      + radar_utm->east;
    glvar1621->north = relpos->north
                       + radar_utm->north;
    output = glvar1621;
    return output;  }
```

Figure 5: Radar Program Converted to C

While the process described above is rather lengthy when described in words, the time taken by an experienced user to create this program using VIP was less than two minutes. Note that this problem involves several instances of conversion of units of measurement, a physical constant, and algebraic manipulation of several equations; all of these were hidden and performed automatically. The GLISP program produced by VIP and a version of the program that has been compiled and mechanically translated into C are shown in Figs. 4 and 5.

This example calculation is conceptually simple. However, for a human programmer, the requirements of performing algebraic manipulation of equations, converting units of measurement, and finding values of physical constants keep it from being a simple program.

VIP hides these sources of programming difficulty and potential error.

7 Discussion and Future Work

Equations are heavily used in science and in scientific programming. Equations are powerful in the sense that, by algebraic manipulation, a small set of equations can represent a large set of problems. However, our study of the process of solving physics problems [12] indicates that equations are not the most fundamental representation of a problem: choice of the correct equations is based on a representation of a problem in terms of physical principles. Larkin *et al.* [9] and others have investigated the *physical intuition* that underlies the choice of physical models. With Kook [5], Lee [8], and Chang [1], we have examined computational representations of physical models and their use in understanding and solving problems.

VIP allows the user to specify a problem directly in terms of physical principles, rather than in terms of the equations associated with those principles. Manipulation of equations by humans is recognized as a source of difficulty and error. By abstracting away the algebra and unit conversions normally associated with scientific programming, VIP makes it significantly faster and easier to write the kinds of programs illustrated by our examples. Other systems, notably SIGMA [4], are based directly on equations. VIP's models often involve multiple equations; packaging together all of the equations associated with a single model gives the user fewer objects to deal with and facilitates the solution of multiple equations within the same model.

It remains to be seen how well the techniques of VIP will scale up for larger problems. Graphical displays often look good when they are small, but become confusing and unwieldy as they grow larger. It is clear that there needs to be a way to modularize parts of a VIP diagram, both for possible reuse and to make the size of a display manageable. The lack of directionality in VIP diagrams is important for reuse, since reuse of a subset of a diagram might involve different data flow directions than the original use.

VIP itself needs to be improved in several ways, including more thorough type checking such as that performed by SIGMA [4] (type errors currently are caught by GLISP, but should be detected earlier by VIP) and an ability to solve simultaneous equations. We also plan to investigate the use of VIP's user interface for specification of other kinds of programs.

References

[1] Chang, Ruey-Juin, "Cliche-Based Modeling for Expert Problem-Solving Systems", Ph.D. dissertation, Department of Computer Sciences, Univ. of Texas at Austin, Dec. 1992.

[2] Gieck, Kurt, *Engineering Formulas* (5th ed.), McGraw-Hill, 1986.

[3] Kant, E., "Synthesis of Mathematical-Modeling Software", *IEEE Software*, vol. 10, no. 3 (May 1993), pp. 30-41.

[4] Keller, R. M. and M. Rimon, "A Knowledge-based Software Development Environment for Scientific Model-building", *Proc. 7th Knowledge-Based Software Engineering Conference (KBSE-92)*, McLean, VA, Sept. 1992, IEEE Computer Society Press, pp. 192-201.

[5] Kook, H. J. and Novak, G., "Representation of Models for Expert Problem Solving in Physics, *IEEE Trans. on Knowledge and Data Engineering*, **3:**1, pp. 48-54, March 1991.

[6] Jagadeesh, J. M. and Y. Wang, "LabView" (product review), *IEEE Computer*, vol. 26, no. 2 (Feb. 1993), pp. 100-103.

[7] Larkin, J. and Simon, H. A., "Why a Diagram is (Sometimes) Worth 10,000 Words", *Cognitive Science*, **11:**65-99, 1987; also in [18].

[8] Lee, Xiang-Seng, "Temporal and Spatial Analysis in Knowledge-Based Physics Problem Solving", Ph.D. Dissertation, Tech. Report AI-93-205, A.I. Lab, C.S. Dept., Univ. of Texas at Austin, 1993.

[9] Larkin, J., J. McDermott, D. Simon, and H. A. Simon, "Expert and Novice Performance in Solving Physics Problems", *Science*, vol. 208 (20 June 1980), pp. 1335-1342.

[10] Newton, P. and J. C. Browne, "The Code 2.0 Graphical Parallel Programming Language", *Proc. ACM Int. Conf. on Supercomputing*, July 1992.

[11] Novak, G., "GLISP: A LISP-Based Programming System With Data Abstraction", *AI Magazine*, vol. 4, no. 3, Fall 1983, pp. 37-47.

[12] Novak, G., "Representations of Knowledge in a Program for Solving Physics Problems", *Proc. 5th International Joint Conf. on Artificial Intelligence (IJCAI-77)*, 1977, pp. 286-291.

[13] Novak, G. and W. Bulko, "Understanding Natural Language with Diagrams", *Proc. Eighth National Conference on Artificial Intelligence (AAAI-90)*, 1990, pp. 465-470.

[14] Novak, G., F. Hill, M. Wan, and B. Sayrs, "Negotiated Interfaces for Software Reuse", *IEEE Trans. on Software Engineering*, vol. 18, no. 7 (July 1992).

[15] Novak, G., "Software Reuse through View Type Clusters", *Proc. 7th Knowledge-Based Software Engineering Conference*, McLean, VA, Sept. 1992, pp. 70-79 (IEEE Press).

[16] Novak, G. and Bulko, W., "Diagrams and Text as Computer Input", *Journal of Visual Languages and Computing*, vol. 4 (June 1993) pp. 161-175.

[17] Shu, Nan C., *Visual Programming*, New York: Van Nostrand Reinhold, 1988.

[18] Simon, H. A., *Models of Thought*, vol. 2, Yale Univ. Press, 1989.

[19] Wolfram, S., *Mathematica: a System for Doing Mathematics by Computer*, Addison Wesley, 1991.