Experiences with Visual Programming Languages for End-Users and Specific Domains

Philip T. Cox *Trevor J. Smedley*

Dalhousie University, Halifax, Nova Scotia, Canada

1 Introduction

The introduction of graphics-based operating systems and user interfaces in the early 80's has contributed significantly to the proliferation of microcomputers and other personal computing devices, and their widespread use by "end-users", individuals with no detailed technical knowledge of computers or programming. This led to the study and development, both academic and commercial, of visual languages and environments for programming, and more recently, to the study of visual programming languages designed specifically for end-users or for specific domains. For example, at the *Visual End-User Workshop* held at the 2000 *IEEE Visual Languages Symposium*, topics included constraint-based visual programming for architectural CAD systems, learning support for end-users in visual programming systems, robot control and business modelling [7].

In [10], Green and Petre propose several "cognitive dimensions" within which to assess visual programming languages. One of these, "closeness of mapping", deals with the directness with which the language represents the constructs of the problem. Although the exact meaning of "closeness of mapping" depends to a large extent on the type of user, there is evidence to support the notion that a visual programming language is more likely to be effective if it concretely represents constructs that the programmer would otherwise have to code in some textual syntax [16,17]. For example Prograph [12], a general-purpose visual programming language for industrial application development provides concrete representations of abstract structures (algorithms and data). According to anecdotal evidence, this boosts the productivity of professional programmers, who are accustomed to building their own mental pictures of such things. LabVIEW, a tool for building process control systems, uses the "instrument" and "wiring diagram" metaphors to appeal to its primary users, electronic and process-control engineers. It has enjoyed great commercial success because of the productivity benefits it provides to such users [1].

Convinced that "closeness of mapping" is the key to success, we have been exploring the applications of visual programming to end-user and domain specific programming for some time. We will briefly summarise the state of three projects that lie at various points on the continuum between visualisation of concrete domain entities and visualisation of abstract structures. The project at the concrete end is *robot control*, where we have been exploring the possibilities of programming autonomous robots by demonstration. Somewhere in the middle lies the *design of structured objects*, where visualisations of design components and program structures are combined in a language for designing parametrised structures, electronic, mechanical or architectural, for example. At the more abstract end of the scale, we have been investigating end-user visual programming for hand-held computers. In our proposed language, *PDA-Graph*, the user/programmer builds visual scripts that include representations of "components" which may be plug-in Springboard hardware modules or modules implemented entirely in software. Since this paper is a summary of work to date, much of the material is taken directly from previously published works.

2 Robot Control

We have been exploring the use of direct manipulation of well-understood, concrete representations of domain-specific entities in our recent work on visual languages for programming robot control [3, 6]. The robot used as the target consists of a Programmable Brick [14], also referred to as a "handyboard" embedded in a LEGO car. The handyboard reads signals from two touch sensors mounted on bumpers at the front of the car, and five infrared sensors mounted underneath. The handyboard controls two motors at either side of the car. The vehicle runs on a LEGO track suitably coded with black tape to be read by the sensors.

Our earlier system, reported in [3], Visual Behaviour-Based Language (VBBL), is based on the subsumption architecture for robot control due to Brooks [2]. The top level of a subsumption model control specification consists of a network of behaviours connected by message-flow links, where behaviours are defined by finite state machines (FSM). In VBBL these message-flow diagrams and FSMs are explicitly represented and edited. However, even though these constructs are directly related to robot control, for programming a *particular* robot VBBL is still too abstract since it supplies no representations of the actual robot.

As a result of our experiences with VBBL, our current goal is a robot control system which is not only general enough to apply to any robot, but allows as much of the programming as possible to be done by directly manipulating representations of the robot under consideration. Clearly, in order to satisfy these apparently contradictory criteria, we will need more than just the traditional programming language embedded in a software development system. One possibility is a system consisting of two parts: first, a hardware definition module (HDM) for defining the structure and function of a robot, and a simulated environment; and second, a software definition module (SDM) which uses the model constructed in HDM to provide various capabilities, including the building and executing of robot control programs. We discuss this possibility in depth in [6], and summarise those results here.

2.1 The robot environment

In building a robot control program, a programmer specifies procedures which use information about the robot's environment to compute values used to change the environment. Since our aim is to make it possible to do at least part of this programming by direct manipulation of a representation of the robot, we must also provide a representation of the environment in which the robot operates. What we need is a way to define values at points in space for properties the robot will measure or attempt to change. To this end, we consider that the environment consists of a set of *tiles*. A tile has an icon of any shape, and has associated with it a *property* and a function that maps any point on the tile to a value for the property. Figure 1 shows an example of a tile created using HDM which corresponds to a straight piece of track on which the Lego car can run. Different regions of the tile have different reflectivity values assigned, which can then be detected by the infrared sensors which are used to guide the car along the track.

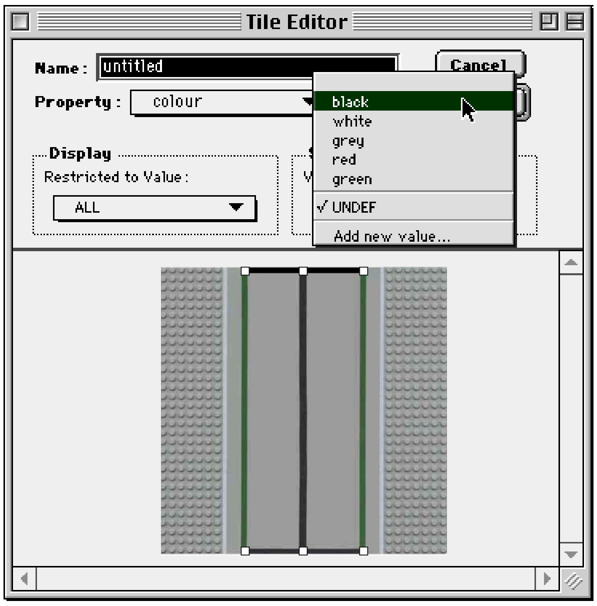
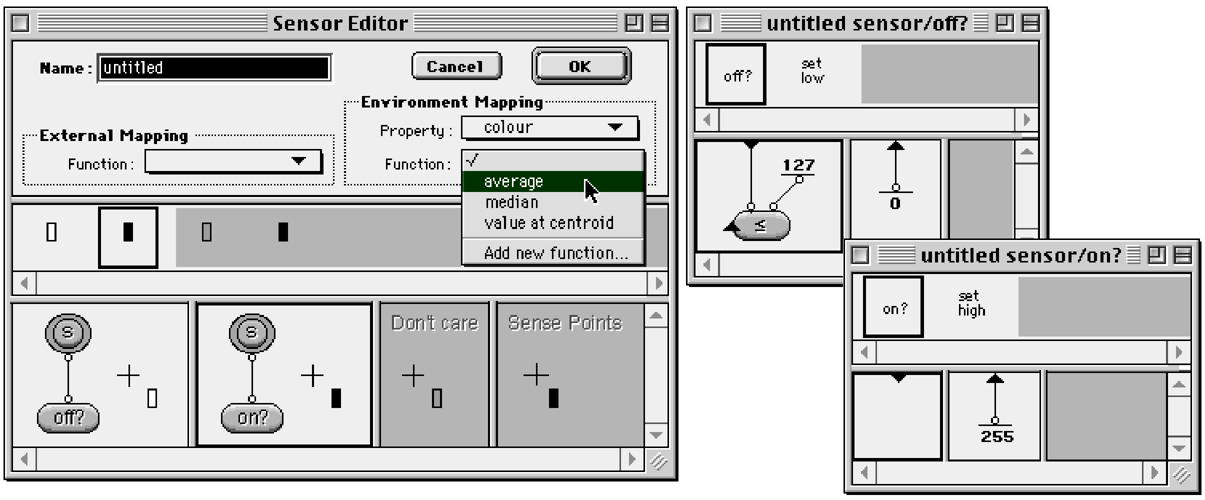


Figure 1: A simple tile in HDM

2.2 Sensors

HDM is used to build simulations of sensors which are capable of reading the values defined on the tiles which comprise the environment. The **Sensor Editor** window (shown in Figure 2) has three panels; from top to bottom, the *specifications*, *head* and *body* panels. The specifications panel is used to define the functions which read values from the physical sensors, and also the functions which are used to compute values from the simulated environment. The head and body panels are used to define two mappings; a function from the sensor value received from the hardware sensor to the visual representation (used when the sensor is being "observed" by SDM), and an inverse (used when the sensor is being "controlled" by SDM). These functions are defined by a sequence of


Sensor Editor

Name : untitled

Environment Mapping — Property : colour

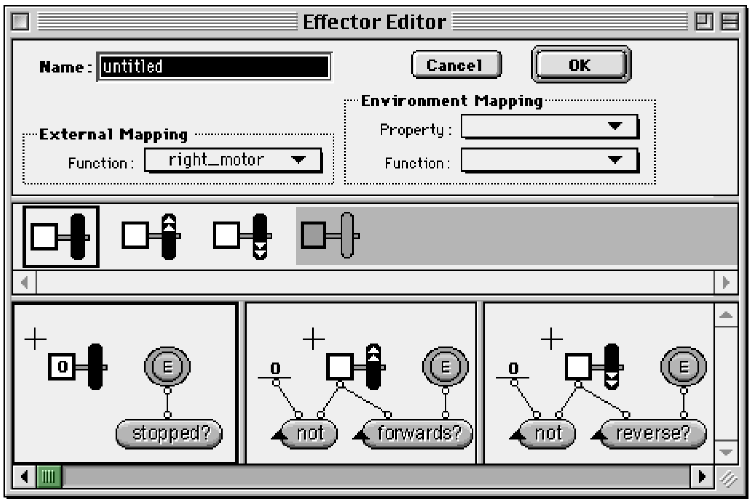
Figure 2: Completed sensor definition and associated icon definitions

cases, similar to the definition of a method in Prograph [12], arranged horizontally in each of the panels. The head panel shows “thumbnails” consisting of an icon for each case, while the body panel consists of panes containing the “bodies” of the cases.

The semantics of sensor definitions and local operations is a logic/dataflow hybrid. Links normally represent unification, but may be marked with an arrow to indicate dataflow. The direction of the flow of data is determined by the *mode* of the sensor: *observed* or *controlled*.To illustrate, let us suppose that a sensor is in observed mode and a value of 200 is being mapped to an icon by the definition in Figure 2. The first case is tried, which leads to execution of the local off?. The first case of off? applies since the arrow on its input indicates that data must flow in. The operation ≤ fails, however, so the second case is attempted, but also fails because data is expected to flow out. This leads to the failure of the first case of the sensor, and thence to the execution of the second case. The first case of the local on? succeeds immediately because it requires only that data flows in. As a result, the icon ▮ is selected.

2.3 Effectors

An effector is similar to a sensor in that it has an iconic representation defined as a collection of graphic items that varies depending on an underlying value (*effector value*), and is associated with an external function. Not surprisingly, editing an effector is similar to editing a simple sensor as described in Section 2.2. The **Effector Editor** window in Figure 3 shows the definition for the right drive effector of our LEGO car. In this example, the icon consists of three parts; a text box representing the motor, a rectangle for the driveshaft, and a rounded rectangle for the wheel. The effector value is represented by the icon Ⓔ.The exact representations of the motor and wheel depend on the speed and direction components of the effector value. The first case defines the correspondence for a stopped drive, while the other cases define it for forward and reverse movement. Each of the locals


Effector Editor

Name : untitled

External Mapping — Function : right_motor

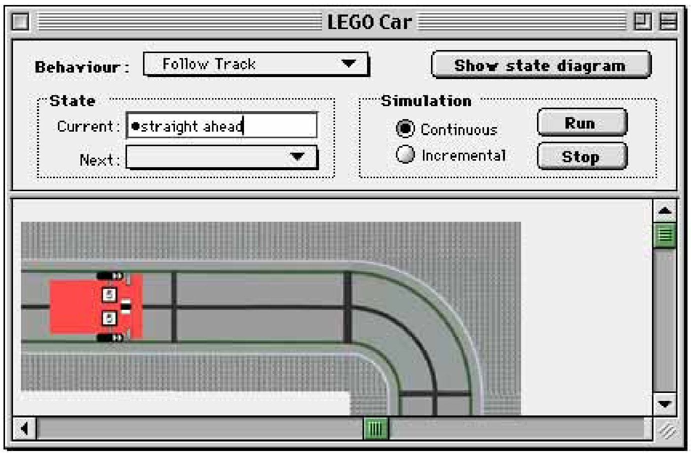
Figure 3: The Effector Editor

stopped?, forwards? and reverse? either checks that the direction component of the effector value is correct, or sets it correctly, depending on whether the definition is used to choose a graphic representation for an effector value or *vice versa*. These locals also ensure that the speed component of the effector value matches the display in the motor text box.

Further tools are available for defining the mappings for the effectors, defining the effect that they produce in the simulated or real environment, similar to the manner in which this is done for Sensors.

2.4 Programming the Robot

In this section we give a simple example to illustrate how the model produced by HDM can be used to program the robot. Figure 4 shows the SDM environment with the LEGO car on a short section of track. In this window we will define a behaviour for our robot as a finite state machine (FSM), following the subsumption architecture [2]. The current state of the behaviour is named straight ahead and the character • is added to the beginning of the name to indicate that this is the start state for the behaviour. We then set the effectors to the values they should have for straight ahead state, and select the sensors significant in implementing the behaviour. We set the two motors to "forwards at speed 5," and set to "don't care" any sensors we are not interested in.


LEGO Car

Behaviour: Follow Track

State

Current: •straight ahead

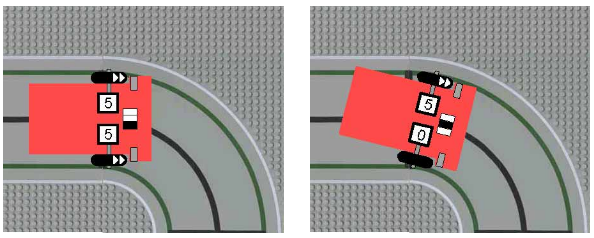
Next:

Simulation

[✓ Continuous] Incremental

Figure 4: Environment for programming the LEGO car.

To begin building the FSM for the behaviour, we click the **Run** button. Simulation begins and the robot moves forwards at a constant speed 5 until the value of an active sensor changes, which halts the simulation. In our example, this will occur when the robot reaches the curved track section as in Figure 5(a). The robot should now turn right, so we create a new state veer right. For this state we set the left and right drives to "forwards at speed 5" and "stopped," respectively.



(a) (b)

Figure 5: Programming a right turn.

Now we resume the simulation, which stops again as soon as the sensors change, as shown in Figure 5(b), at which time we select the existing state straight ahead from the Next popup, which sets the drive values defined for that state, causing the robot to continue straight.

The next change in sensor values is to the configuration □■ (left to right). Because this configuration was previously encountered in the state straight ahead, there is already a transition for it in the FSM. Therefore, since the simulation mode is set to "continuous," the simulation will not halt, but continue with a transition to the state veer right.

Programming proceeds in this way until all necessary states and transitions have been defined, at which time the simulation runs without stopping. At any time during the programming process, clicking the button **Show state diagram** opens a window depicting the FSM for the behaviour under construction. For example, if this button is clicked at the point in the above description when the robot halts in the state veer right, the window shown in Figure 6 appears. The start state of the FSM is outlined. The label on a transition is the combination of sensor values that triggers the transition.

In the subsumption architecture, behaviours are autonomous and concurrent, making it possible to build complex reactive systems in an incremental manner. For example, once the Follow Track behaviour has been defined for the LEGO car, we can expand the control program's capability by adding a Back Up behaviour that reverses the motors if the touch sensors at the front of the car are activated. Clearly motor settings produced by Back Up should take precedence over those from Follow Track. The subsumption architecture provides message-flow diagrams to specify the precedence relationships between behaviours.

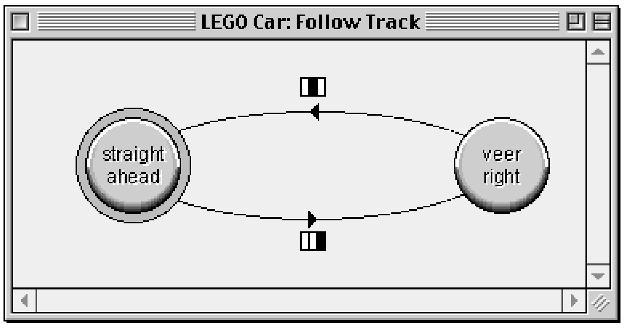


Figure 6: FSM for turning right.

3 A Declarative Language for Structured Design (LSD)

The process of designing and building computer software provides a general model for a large class of processes which aim to produce artifacts with some required form and function. Some other examples of this class are the design of digital circuits [8], building structures [13], and parameterised devices [11]. The analogy with computer programming is stronger with some domains than others, but generally the process involves building a design for some object using some descriptive language: testing the design by simulating its behaviour, which should be a consequence of the design process, and finally using the design to generate an artifact which exhibits the same behaviour as the design under simulation. These activities correspond to programming, testing using a debugging interpreter and compiling to stand-alone code. In VLSI design, the steps are, coding the design of an electronic device, testing it via simulation, and compiling low-level chip fabrication code.

Visual software tools for some design tasks, CAD/CAM systems for example, have been in widespread use for many years. Systems such as ArchiCAD [9] provide sophisticated general-purpose and special-purpose tools for drawing and solid modelling. Support for parameterised designs is also provided, but it is either quite rudimentary, or requires the use of a textual language very much like a programming language.

In contrast to tools for industrial design, tools for software design have always been dominated by textual means of expression. It is only recently that visual languages have been developed that combine the symbolic manipulation capabilities of textual programming languages with the expressiveness and ease of use of visual tools. It would appear then, that visual languages might be able to provide the programming capabilities required for building parameterised designs, while at the same time integrating more closely with the drafting and solid modelling aspects of an industrial design system.

In an attempt to more closely integrate the drawing/drafting and programming aspects of

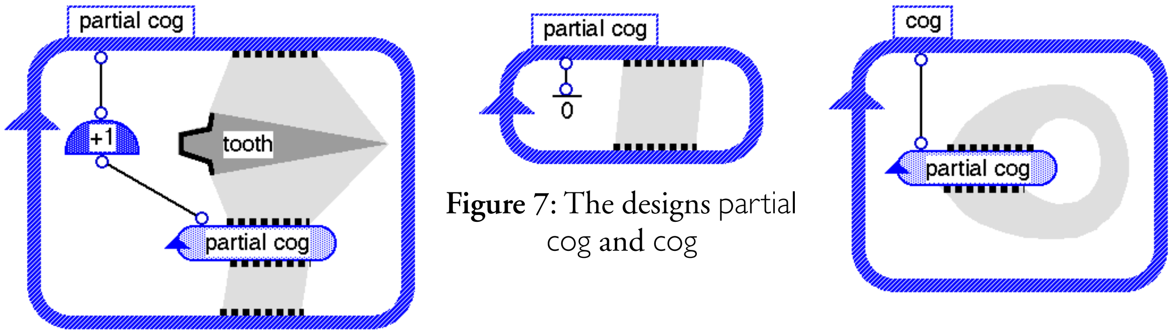


Figure 7: The designs partial cog and cog

design, in [4, 5] we proposed a *Language for Structured Design* (LSD), which we will informally describe via an example. Figure 7 depicts an LSD *program* consisting of two designs, **partial cog** and **cog**. A *design* is a declarative specification for a type of component, instances of which are represented in other designs by *implicit components* or *i-components* for short. For example, **partial cog** defines an incomplete cog with a particular number of teeth as consisting of an extra tooth bonded on to an incomplete cog with one less tooth, represented by the i-component partial cog. The case of **partial cog** in the centre of Figure 7 corresponds to an incomplete cog with no teeth, providing the base case of this recursive design. The other design in this program, **cog**, consists of a single case defining a complete cog as the component that results from bonding together the open edges of a partial cog.

The component tooth in the recursive case of partial cog of Figure 7 is an example of an *explicit component*, or *e-component* for short, and explicitly represents a single tooth of a cog. An *open edge* is a line along which one e-component can be "bonded" to another to form a new e-component; for example, the radial boundaries of tooth in Figure 7. A component is said to be *open* or *closed* depending on whether or not it has any open edges. An open component is considered to be "unfinished" in the sense that it does not represent a real object, but can exist only as a part of a larger object.

An e-component has a parameterised internal state, and a customised visual representation that may vary depending on the values of its parameters, which may correspond to characteristics such as size, position, orientation in the plane or colour. Because of this parameterisation, an e-component represents a family of components, and its appearance is actually a "typical" one chosen to be representative of that component family. Substituting values for parameters produces an

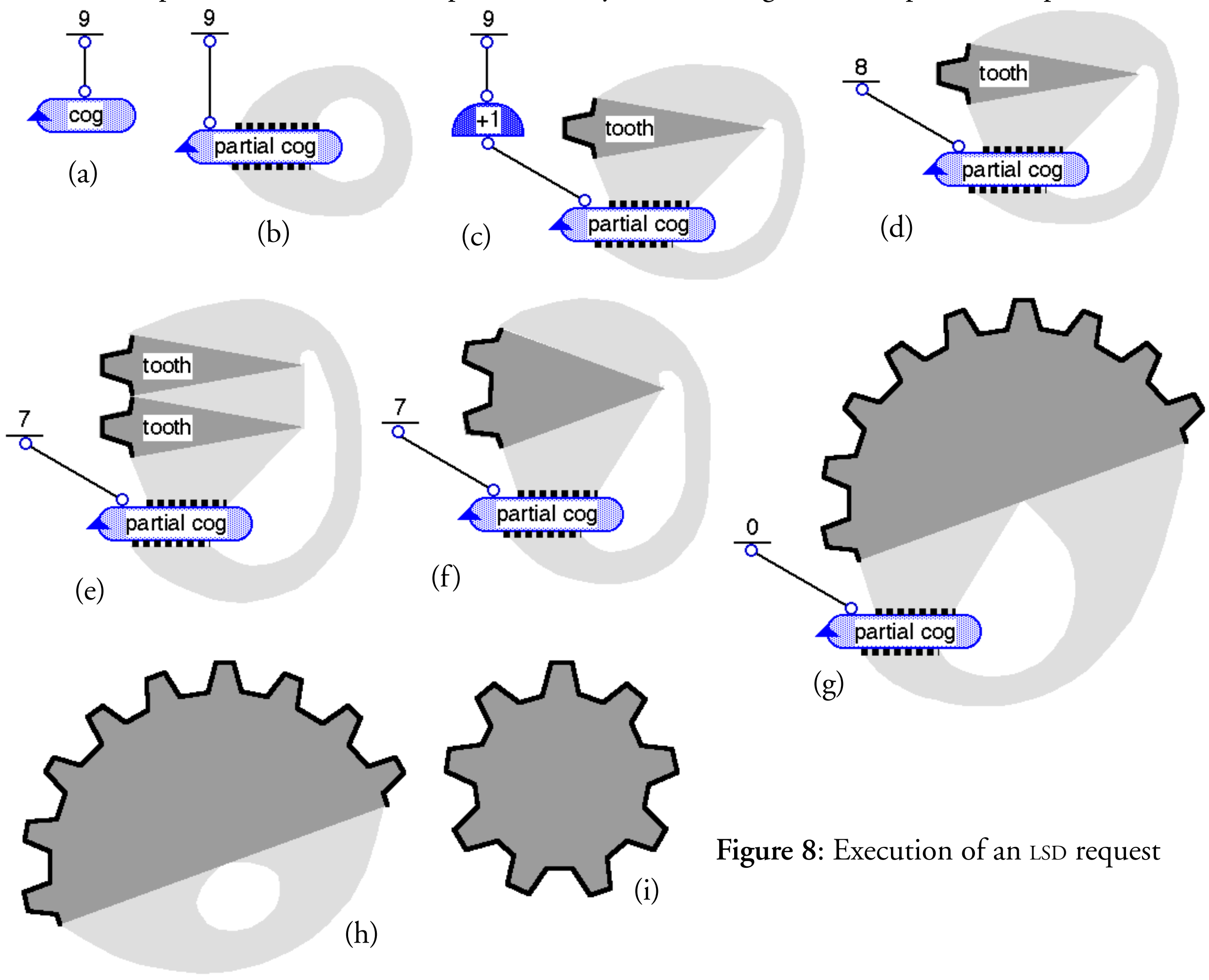


Figure 8: Execution of an LSD request

e-component defining a more constrained family of e-components.

The grey bands in the program in Figure 7 above are *bonds,* which are of two types, *external* or *internal.* An internal bond indicates open edges to be joined in defining a new component. An external bond indicates an open edge to be "exported" from the design.

Execution of an LSD program involves applying transformation rules to a *request* until an untransformable graph is obtained. We call this execution process *assembly.* To illustrate, we consider the assembly of a nine-toothed cog starting with the request in Figure 8(a). This is transformed into the request in Figure 8(b) by a rule called *replacement,* which substitutes the body of the case cog. for the implicit component partial cog.

Two rules *merge* followed by *deletion* applied to the request in Figure 8(c) results in 8(d). For brevity we will not describe the details of these two rules; however, the meaning of the transformation they perform in this example is obvious.

Applying replacement with the recursive case of **partial cog** to the i-component in Figure 8(d), followed by merge and deletion, results in Figure 8(e) where an internal bond connects two open edges. Application of a rule called *bonding* to this internal bond creates one new e-component by joining the two e-components tooth along their open edges, resulting in Figure 8(f). The internal state and parameters of the new e-component are determined from the states and parameters of the constituent e-components. Clearly, certain parameters of the two components will become mutually constrained as a result. Here for example, the two components are rotated relative to each other in order to accomplish the bonding.

Repeating this pattern of execution steps eventually leads to Figure 8(g). Replacing the i-component in this diagram with the base case of partial cog, followed by merge and deletion produces Figure 8(h), from which bonding creates Figure 8(i), consisting of one closed explicit component.

When the bonding rule is applied, it may happen that the resulting e-component represents an object which for some reason cannot physically exist. For example, suppose we attempted to assemble a cog with one tooth. Obviously this could be prevented by appropriate constraints on the extent to which a tooth can be deformed.

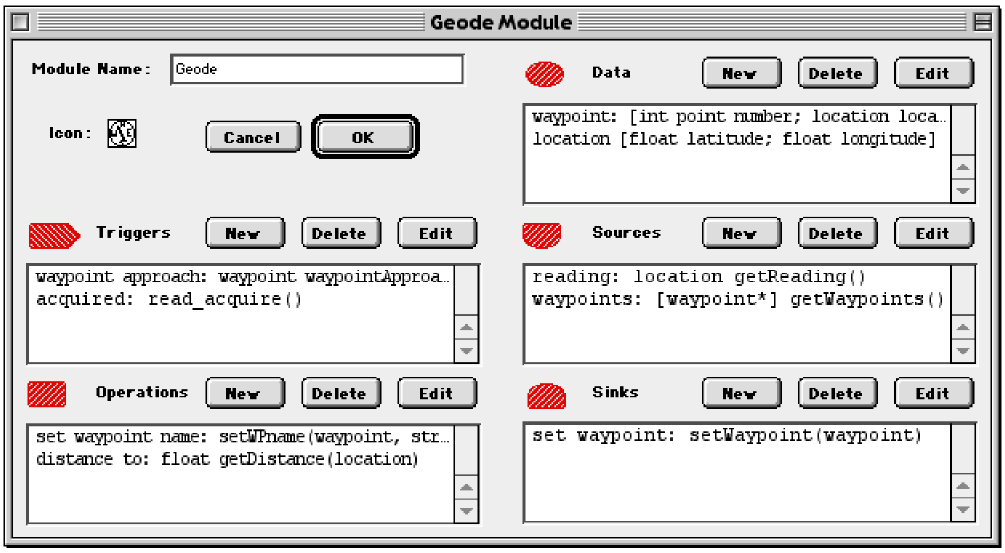
4 PDA Graph

With the ability to put more and more computing power into a smaller package, computer hardware companies started in the early 1990's to develop handheld computers, such as Apple's Newton and Palm's Pilot. These devices are now commonplace, and are expected to become even more widely used than desktop computers. The widespread use of desktop computers created a demand from end users for customisability and programmability: it is clear that there is a similar demand from users of handheld devices. Already, there are simple tools such as spreadsheets and database products for handhelds, that allow some level of end user programmability for this specific domain.

We are particularly interested in investigating end user programmability of the Handspring Visor. This device uses the PalmOS, but includes a hardware expansion slot which accepts Springboard modules. Various modules exist today for data acquisition, communications, and a variety of other functions. We feel that with this flexibility of use will come an increased demand for user programmability. We discuss here the approach we are taking in researching the provision for end-user programmability of these devices

4.1 Overall Approach

We are using an approach based on a two 'module' system: one to define the interface between the programming environment and the objects (such as Springboard modules) to be scripted (this would run on a desktop computer); and then the scripting environment itself (which would run on the handheld computer, although we anticipate investigating adaptations and enhancements that could be added to a version for a desktop environment). The first of these modules would be



Module Name: Geode

Data:
```
waypoint: [int point number; location loca…
location [float latitude; float longitude]
```

Triggers:
```
waypoint approach: waypoint waypointApproa…
acquired: read_acquire()
```

Sources:
```
reading: location getReading()
waypoints: [waypoint*] getWaypoints()
```

Operations:
```
set waypoint name: setWPname(waypoint, str…
distance to: float getDistance(location)
```

Sinks:
```
set waypoint: setWaypoint(waypoint)
```

Figure 9: Mock-up of Unit Definition Module Interface

used to define the concrete representations of the problem domain objects — representations of the data and functions available from a bar code reader, for example. Then the second module would allow the user/programmer to combine these representations with those of the programming concepts needed to complete their task. This is similar to the approach we have taken to robot programming, described briefly in Section 2 and fully in [6].

4.2 Unit Definition Module

Because we want to be able to write scripts that deal with both hardware and software units, we refer to this module, which is the analog of the HDM for robot programming, as the Unit Definition Module (UDM). The UDM would not be used by end-user programmers, but rather by someone such as the developers of the software applications or hardware expansion modules that are to be used in conjunction with the scripting environment.

Figure 9 is an interface mock-up showing the sorts of capabilities that would be provided by the UDM, illustrating some of the definitions that could be made for a GPS module. An icon must be defined for the unit which will be used throughout the scripting module to refer to anything defined in this unit. Also, any functions or data structures which are to be made available to scripting environment users must be defined. These definitions would be made in conjunction with an API for the software application or hardware module, and would provide the person using the UDM with the ability to make functions and data from the hardware or software unit available to the scripting environment user.

We have five types of definitions which can be made for a unit:

Data. This provides the ability of have data structures appropriate for the unit made available in the scripting environment.

Operations. Any functions which take input and provide output are defined as operations.

Triggers. Triggers are functions which are used to initiate the execution of scripts. They correspond to events such as mouse clicks or key presses in a traditional interface, which may cause certain portions of code to be executed.

Sources. Sources are operations which take no inputs, but do provide outputs. They correspond to data available from the unit.

Sinks. Sinks are operations which have no outputs, but do provide inputs. They correspond to data which can be sent to the unit.

4.3 Scripting Module

The scripting module (SM) would provide facilities for creating, editing, deleting, running testing and debugging of scripts (and perhaps beaming, so they could be shared with other handheld users). There would also be provision for adding new plug-in units from the UDM (probably through installation via a desktop computer).

We are initially basing our scripting language on Prograph, a general purpose, visual, object-oriented, dataflow language, with which we are quite familiar [12]. Due to space restrictions, we will not provide a description of Prograph here, but refer the reader to [14], for example, where we give a brief overview of the language.

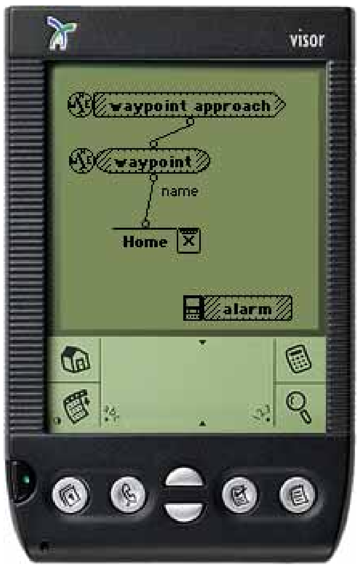


Figure 10: Example Script

The example in Figure 10 shows a script which uses a GPS module to alert the user with an alarm when they are approaching home. When the GPS senses that it is approaching a waypoint, execution of the script shown will be triggered through the GPS trigger waypoint approach. The output of this is a waypoint data structure, and the GPS data operation waypoint is then used to extract the name field from this structure. The match operation Home compares this to the value “Home”, and execution of the script is terminated if it does not match (the annotation means “terminate on failure”). If execution is not terminated, then the system operation alarm is executed, causing the alarm to sound.

Besides adding the additional operation types as described above, the most significant change we are making to standard Prograph is in the way objects are treated. In Prograph, objects flow along lines, and the leftmost input to a method call is the object which receives the message. Here, we are considering the units defined by the UDM as objects, and which object receives the message is indicated by an icon attached to the left side of the operation. In our example, we use the GPS unit and the system unit (containing any system level functions and data).

5 Concluding Remarks

Our experience has convinced us that the future of visual programming lies in its application to specific domains. Here we have briefly reported on three projects in which the visualisations used in programming range from concrete to more abstract domains. The common thread is that users are provided with representations of structures that they would otherwise have to build mentally from textual encodings.

None of these projects has progressed to the point of producing the robust and full-featured implementations which will be needed in order to conduct experiments with users to test our hypothesis about the usability of end-user and domain-specific systems. We expect that the pda project will be the first to reach that stage for the simple reason that it is the most abstract, so does not involve the hard issues that are raised by implementing “real-world” look and feel.

6 Acknowledgements

The work reported here was supported by Individual Research Grants from the Natural Sciences and Engineering Research Council of Canada.

7 References

1. E. Baroth & C. Hartsough (1995) Visual programming in the real world. in *Visual Object-Oriented Programming: Concepts and Environments*, M. Burnett, A. Goldberg & T. Lewis, eds, Manning Publications Co., Greenwich, CT, pp. 21-42.

2. R.A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, 2 (1): 14-23, 1986.
3. P.T. Cox, C.C. Risley, T. Smedley, Toward Concrete Representation in Visual Languages for Robot Control, *Journal of Visual Languages and Computing*, 9 (2):211-239, 1998.
4. P.T. Cox & T.J. Smedley (1998) LSD: A Logic Based Visual Language for Designing Structured Object, *Journal of Visual Languages & Computing*, 9(5), Academic Press, 509-534.
5. P.T. Cox & T.J. Smedley (2000) A Formal Model for Parametrised Solids in a Visual Design Language, *Journal of Visual Languages and Computing,* 6(6), Academic Press, 687-710.
6. P.T. Cox, T.J. Smedley. Building Environments for Visual Programming of Robots by Demonstration. *Journal of Visual Languages and Computing* (2000) **11**, 549-571.
7. P.T. Cox, T.J. Smedley. Guest Editor's Introduction, Special Issue on Visual Languages for End-User and Domain-Specific Programming. *Journal of Visual Languages and Computing* (2001) vol 12.
8. E.J. Golin, M.J. Haney, E. Huges, D. Miller-Karlow & G. Tharakan (1992) *Visual design with vVHDL.* University of Illinois at Urbana-Champaign, Department of Computer Science Technical Report #1745.
9. Graphisoft R&D Rt. (1996) *ArchiCAD 5.0: GDL Reference Manual.*
10. T.R.G. Green & M. Petre (1996) Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing* 7(2), 131-174.
11. A. Paoluzzi & C. Sansoni, Programming Language for solid variational geometry. *Computer Aided Design* 24, (1992), 349-366.
12. Pictorius Incorporated. (1993) *Prograph CPX User's Guide.*
13. A. Rau-Chaplin, B. MacKay-Lyons & P. Spierenburg (1996) The LaHave House Project: Towards an Automated Architectural Design Service. *Proceedings of the International Conference on Computer-Aided Design (CADEX'96)*, IEEE Computer Society Press, pp 25-31.
14. M. Resnick. Behavior Construction Kits. *Communications of the ACM*, 37(7):65-71, 1993.
15. T.J. Smedley and P.T. Cox. Visual languages for the design and development of structured objects. *Journal of Visual Languages and Computing.* vol. 8, pp. 57-84, 1997.
16. K.N. Whitley (1997) Visual Programming Languages and the Empirical Evidence For and Against, *Journal of Visual Languages and Computing*, 8(1), Academic Press, 109-142.
17. K.N. Whitley & A.F. Blackwell (2001) Visual Programming in the Wild: A Survey of LabVIEW Programmers, *Journal of Visual Languages & Computing*, 12(4), 435-472.