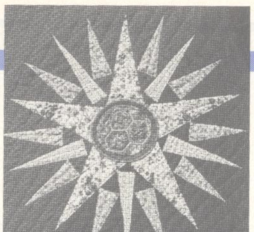


Working in the Garden Environment for Conceptual Programming

Steven P. Reiss, *Brown University*

Program designers use a variety of techniques when creating their systems. This automated design system conforms to the programmer.

One important problem in automating the software development process is providing an environment for designing software systems. Most of the approaches for automated design environments provide one or two design methods and force the programmers to design with only these methods. Examples of such approaches include automated versions of SADT,[1] SREM,[2] and dataflow-based design.

Most program designers, however, naturally use a variety of techniques when designing their systems.[3] They draw pictures of their data structures, describe a control-oriented module with an automaton or a decision table, and illustrate the system structure with a module-interconnection diagram and a dataflow diagram. These techniques are selected to closely fit the system being designed. Moreover, designers modify the techniques to fit the problem better and develop new strategies or languages to simplify the description of an otherwise complex design.

Rather than force the programmer to conform to particular design methods, an automated design system should provide an environment that conforms to the programmer. Such an environment must provide many design paradigms. These range from traditional dataflow diagrams, pseudocode, and finite-state automata to logical specifications, object-oriented programming, or whatever language the designer developed to best describe the problem approach. These paradigms must be presented to the programmer in their most natural form. This form can be textual for pseudocode or logical specifications, a standard type of diagram for dataflow or automata, or something designed by the developer (textual or graphical or some combination of the two).

As well as providing a multiparadigm environment for program design, a good design-automation system should provide a framework for evaluating the resulting design. The simplest approach is to allow design-level prototyping. The system

0740-7459/87/1100/0016/$01.00 ©1987 IEEE

should let the design itself act as the prototype program and let the developer evaluate the design by running it. The eventual goal is to make the design itself be the program, thus allowing system debugging and especially maintenance to be done directly in terms of the design. This goal is not realistic with current technology, but faster machines and better compilers could make it a reality in the future.

Finally, an automated design system should provide more than just a variety of design languages that can be consistently combined. It must give the designer a complete environment, including facilities for creating, modifying, recalling, and displaying designs. The system must also support browsing and automated analysis of the designs, plus cooperative design within a team of designers.

Garden is such a programming system. It is designed to support the concurrent use of a variety of languages that represent different programming paradigms. It tries to provide equal support to both textual and graphical languages and to support a wide spectrum of programming paradigms. In Garden, you define a conceptual language by giving its visual and textual syntax and its semantics in terms of an object basis. The language is then used through views of the objects that represent its programs.

Designers developing systems typically build system models in their heads. These models consist of the languages or paradigms used in the design. This is the conceptual system model that the programmers must understand and work with while implementing, debugging, and maintaining the system. The type of automated design system outlined above lets designers work directly with the model. Moreover, it turns the model into the program. A system that supports this type of effort — called conceptual programming — with an automated environment is a conceptual programming environment.

Requirements

A system that supports conceptual programming must be both flexible and powerful. The requirements for such a system are divided between those supporting multiple paradigms and those providing an appropriate environment.

A conceptual programming environment must simplify the definition and use of new languages. These languages can be visual or textual and can be completely new or derived from an existing language. The requirements here include:

• A consistent support framework. The environment's framework must let languages be freely mixed to form the proper conceptual model. This mixing will generally follow a hierarchy, for example, either with dataflow actions referenced by the nodes of a control-flow graph or with control-flow actions assigned to the arcs of a finite-state automaton. More sophisticated mixing would not be restricted to such a hierarchy. For example, a single framework would let a piece of a program be viewed and edited as both a data diagram and a control-flow diagram.

A system that supports conceptual programming must be both flexible and powerful.

• Equal support for visual and textual languages. Many languages used for design are visual languages. The environment should offer visual languages all the facilities normally offered to textual languages, including syntax definition, editing, file-based storage, program sharing, and browsing. This prevents worthwhile languages from being ignored because of inadequate support.

• Multiple views of a single language form. The system should let programmers view a complex design many ways. The environment should support this (by allowing multiple views of the design languages) to give programmers different perspectives on the underlying design. For example, one view of a design diagram might show only the dataflow information while another might show both the dataflow and the control flow. Such views can provide different levels of abstraction and accommodate slight variations in formatting (especially graphical formatting) that different users might want.

• Facilities to simplify language definition. The environment must provide a rich set of support functions so the semantics of new languages can be defined with minimal effort. There should be a rich set of built-in data types and corresponding operations. There should also be specialized support for common but difficult-to-implement design-language features, such as dataflow, concurrency, and constraints. Finally, it should be easy to reuse and modify existing language definitions.

As well as providing a usable framework for incorporating multiple languages, a conceptual programming environment must provide substantial environmental support for design. In particular, it should provide:

• Prototype evaluation. One key idea of conceptual programming is that the design should be an executable prototype so designers can experiment directly with designs as they work on it. The environment must support this by providing a general interpreter that can evaluate programs in any language that can be defined in the system. Additional support can be provided by letting these languages be compiled to yield a more efficient evaluation.

• An experimental framework. Design prototyping should be encouraged by providing an interactive environment with immediate feedback. The advantages of such an environment are shown by the success of such systems as Lisp and Smalltalk.

The environment should have facilities that let designers understand the program as it executes. These facilities might include various execution views, profiling tools, debugging aids, and dynamic display of the program's data structures.

• A multiwindow, user-oriented front end. The environment should run on a workstation where different views and different languages can be displayed with multiple windows. These views should provide the functionality needed to do the design: They should be editors for the underlying design that support browsing and documentation as it is written. Multiple windows should also give programmers system-control functions such as the ability to execute and interact with the prototype designs.

• General environmental support. The environment should provide the tools needed for software design in a moderate-sized project, including design storage and retrieval with version control. This lets new ideas be tried without modifying a stable system. There should be support for cooperative design so a group of programmers sharing a common design can safely work on different pieces of it. While the system should provide a good interactive environment, it should also be able to produce a readable printout of the resulting design.

Garden overview

Garden is an attempt to meet many of these requirements for a conceptual programming environment. It consists of a programming system designed to support multiple languages, a set of tools that provide a multiwindow user interface, and an underlying database system to provide environmental support.

Object-oriented framework. Garden uses an object-oriented programming system to provide the necessary control abstraction.[4] Object-oriented systems view all their data in terms of objects: data blocks that are instances of a particular class or type. Associated with each class is a set of operations that can be applied to the object. The classes are arranged in a hierarchy so subclasses can inherit the properties, data, and operations of their parents. Experience with Smalltalk[5] has shown that object-oriented systems are good for prototyping because they provide a high degree of reusability and encourage the use of data abstraction.

Garden uses objects to represent programs as well as data. The result is a system that is good for prototyping and encourages the use of both data and control abstraction. You build programs by putting together collections of objects, define new languages by defining new types of objects that represent programs, and use the class hierarchy to reuse existing languages when defining new ones.

The object-oriented Garden system is good for prototyping and encourages the use of both data and control abstraction.

Objects form a consistent basis for supporting multiple languages — any language can be defined in terms of its underlying constructs. In Garden, the differing constructs are represented by different classes of objects; the relationships among the constructs are represented as other objects referred to by the objects. For example, an automaton is represented as an object of class Fsa and includes objects of class State for each state of the automaton and each object of class Arc for each arc.

Garden uses these object constructs as the actual program. One operation it provides for an object class defines what it means to evaluate an object of that class. For example, evaluating an Fsa object with a value causes the automaton to move to the next state using that value. Since this operation is defined for all program objects, different program abstractions and hence different languages can be freely mixed hierarchically. Garden lets you define an operation for any object by providing another object that describes the operation. Thus you can use any currently defined language when describing the evaluation semantics of a new language. Garden provides an interpreter and a corresponding compiler to yield efficient execution of object-based programs.

Because objects represent programs, Garden has no bias toward any syntactic form; it lets the syntax of an object-based language be defined textually or graphically, or both ways. The current implementation allows wide latitude in the selection of graphical displays for such languages. While Garden's goal is to let the system provide a natural representation using the underlying objects of most languages, the system now provides only a single textual, Lisp-like representation that is not suitable for all languages, particularly those with cyclic underlying structures.

Garden supports the definition of semantics for object-based languages by providing a rich underlying set of programming primitives, including strings, lists, and tables (indexed relations in the database sense) with a full range of operations. It provides primitives for concurrent processing using lightweight processes, including monitors and semaphores.

A general dependency mechanism can handle constraint-based programming as well as event-triggered demons. It gives full access to the system's underlying naming, typing, and evaluation mechanisms. These facilities can be used with any defined languages to define the semantics of a new language.

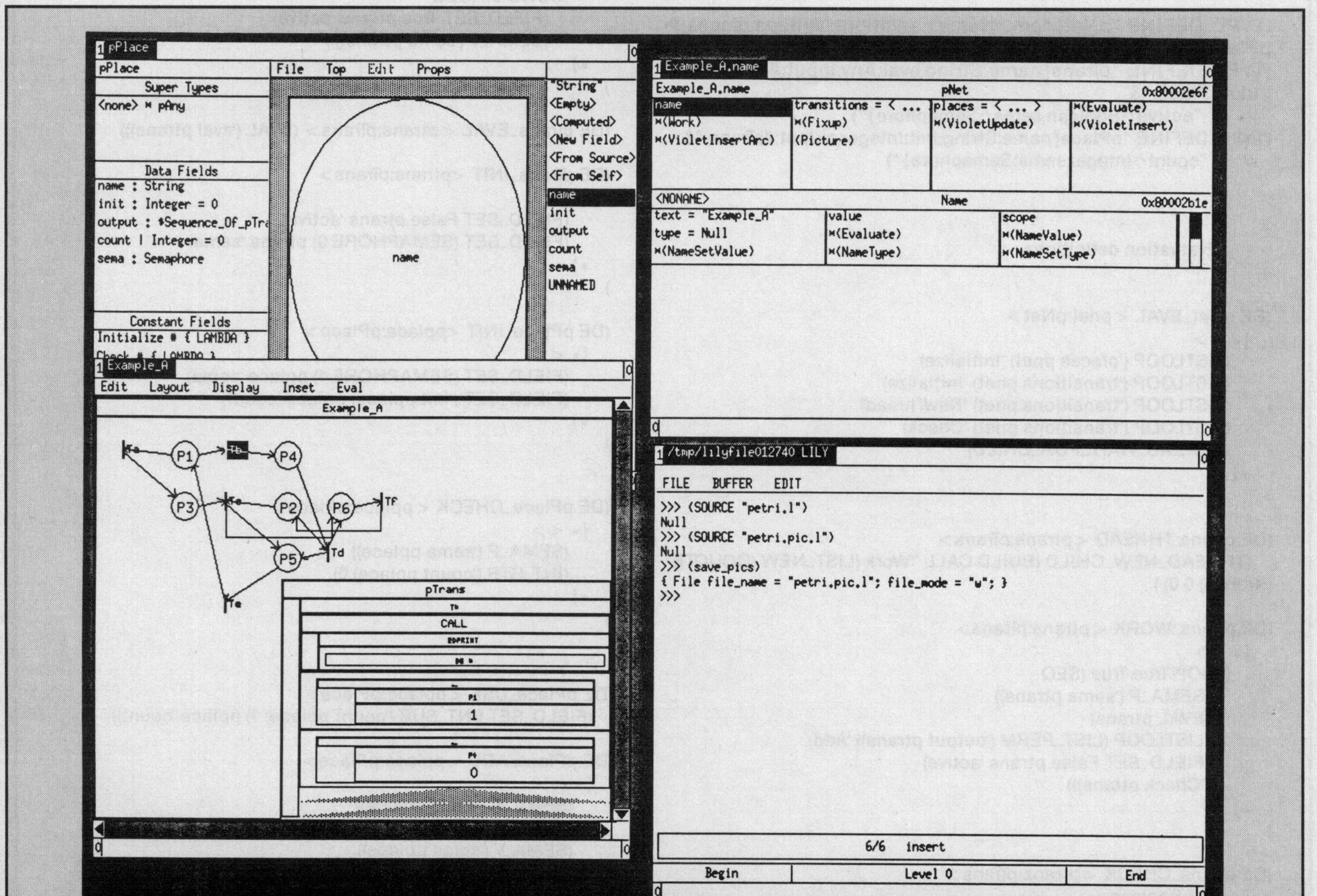
Multiwindow environment. Garden's programming environment lets you create and modify the objects that represent their programs and data. Objects can be viewed or modified in any of three editors. One editor displays the textual form of the object and allows normal text editing. The second provides an object-based browser on the object, letting you select, view, and modify an object's contents or one of its component objects on a field-by-field basis. The third displays a visual representation of the objects and lets you interact directly with this form.

The system coordinates these different editors with a multiwindow display on a Unix-based workstation. Each editor runs in a window on the display; you can set up these windows with a window manager or nest them in another editor or tool. Garden lets multiple instances of each editor be active simultaneously. It also lets you put up multiple editors, of the same or different types, on the same objects simultaneously. In this multiple view, the system keeps the various windows consistent: When you change the underlying object with any editor, all the other views update automatically.

The object editors can view data as well as enter and work on programs. The various editors can be brought up under program control as sophisticated input mechanisms or output displays. You can put up views of the data structures the program is working on. Garden automatically updates such views as the program changes the underlying objects at a user-controllable granularity.

The multiwindow display provides other programming tools. One or more interactive windows can provide a read-eval-print loop with the textual language interface. A variety of system browsers can be defined that let you select an object with a variety of criteria such as the scope in which it is defined, its class, its name, and its fields. A documentation editor lets you quickly find and create textual documentation for any object in the system. In addition, windows can be defined for both graphical and textual program input and output.

Figure 1 shows a sample of a complete Garden screen. The sequence of Gothic letters and icons at the bottom represents the windows available. The windows displayed include a type editor in the upper left, a browser editor in the upper right, a read-eval-print loop window in the lower left, and a graphical editor in the lower right. Each window contains a title bar, move and resize icons in the corners, and a blank bar at the bottom to move the window. Hidden beneath the title bars are buttons to remove the window and to pop



```
name : String
init : Integer = 0
output : *Sequence_Of_pTra
count | Integer = 0
sema : Semaphore
```

```
Initialize # { LAMBDA }
Check # { LAMBDA }
```

```
name                 transitions = < ... >   places = < ... >    ¤(Evaluate)
¤(Work)              ¤(Fixup)                ¤(VioletUpdate)     ¤(VioletInsert)
¤(VioletInsertArc)   ¤(Picture)
```

```
text = "Example_A"   value          scope
type = Null          ¤(Evaluate)    ¤(NameValue)
¤(NameSetValue)      ¤(NameType)    ¤(NameSetType)
```

```
>>> (SOURCE "petri.l")
Null
>>> (SOURCE "petri.pic.l")
Null
>>> (save_pics)
{ File file_name = "petri.pic.l"; file_mode = "w"; }
>>>
```

Figure 1. Garden screen.

Petri net example

The example in Figure A is specified in the textual language currently provided by Garden. The language is defined to look and feel like Lisp so first-time users have a degree of familiarity when they start using Garden. However, it is only a Lisp-like syntax that Garden uses to define objects. The basic form of a Lisp S-expression serves several purposes. If the first component is a type name, it is a definition of an object of that type where the latter components contain the field values for the object. If the first component is a field access object (identified by an initial open single quote [']), it is an invocation of the corresponding message applied to the second component; if the field object refers to a data field, the corresponding message just returns the field's value. If the first component is neither a type or a field access object, the S-expression is translated into a Call object where the first component is the object to evaluate and the latter components are composed into an argument list.

The language provides other extensions to simplify textual Garden programming. Objects can be defined with a named representation rather than the positional representation inherent in S-expressions. Quoting occurs as in Lisp, with a quotation mark (") followed by an expression actually yielding a Quote object containing that expression. Local scopes and local names can be defined with the separators {* and }* containing a list of names and one or more evaluatable objects. The result is either a Block object or an appropriate type of Lambda object with the initial list of names as local variables defined in a new scope and a Seq object containing the sequence of actions as the body.

The first part of the example in Figure A defines the type structure used for Petri nets. Each type is defined as a string containing its name and then a list of its fields inside braces ({ }). The fields are defined with a name, a separator, and a type. The separator can be a colon (:) indicating a data field, a vertical bar (|) indicating a structural field, or a caret (^) indicating a dynamic field. In general, data fields are descriptive; structural fields include the other components of the language; and dynamic fields are used for runtime values.

The second portion contains the definition of the Lambda objects associated with the messages used in evaluating the objects composing a Petri net. The function De takes three arguments: a name, a list of parameters, and a body. It creates a Lambda object with the list of parameters and the body and binds this to the given name. The parameters are defined in a local scope for this particular definition.

```
--
--    Type definitions
--
(TYPE_DEFINE "pNet{name:Name,transitions|#pTrans,places|#-
pPlace}")
(TYPE_DEFINE "pTrans{name:String,eval:Any,input:#pPlace,out-
put:#pPlace,"&
        "active^Boolean,sema^Semaphore}")
(TYPE_DEFINE "pPlace{name:String,init:Integer,output:#pTrans,"&
        "count^Integer,sema:Semaphore}")


--
--    Evaluation definitions
--

(DE pNet_EVAL <pnet:pNet>
  {* <>
    (LISTLOOP ('places pnet) 'Initialize)
    (LISTLOOP ('transitions pnet) 'Initialize)
    (LISTLOOP ('transitions pnet) 'NewThread)
    (LISTLOOP ('transitions pnet) 'Check)
    (THREAD_WAIT_FOR_CHILD)
  *}
)

(DE pTrans_THREAD <ptrans:pTrans>
  (THREAD_NEW_CHILD (BUILD CALL "Work (LIST_NEW (QQUOTE
ptrans))) 0 0) )

(DE pTrans_WORK <ptrans:pTrans>
  {* <>
    (LOOP True True (SEQ
      (SEMA_P ('sema ptrans))
      (EVAL ptrans)
      (LISTLOOP (LIST_PERM ('output ptrans)) 'Add)
      (FIELD_SET False ptrans 'active)
      ('Check ptrans)))
  *}
)

(DE pTrans_CHECK <ptrans:pTrans>
  {* <ok:Boolean>
    (COND ('active ptrans) (RETURN Null))
    (SETQ ok (LISTLOOP ('input ptrans) 'Check BOOL_AND True))
    (COND ok (LISTLOOP ('input ptrans) 'Use))
    (LISTLOOP ('input ptrans) 'Release)
    (COND ok (SEQ
      (FIELD_SET True ptrans 'active)
      (SEMA_V ('sema ptrans))))
  *}
)

(DE pTrans_EVAL <ptrans:pTrans> (EVAL ('eval ptrans)))

(DE pTrans_INIT <ptrans:pTrans>
  {* <>
    (FIELD_SET False ptrans 'active)
    (FIELD_SET (SEMAPHORE 0) ptrans 'sema)
  *}
)

(DE pPlace_INIT <pplace:pPlace>
  {* <>
    (FIELD_SET (SEMAPHORE 1) pplace 'sema)
    (FIELD_SET ('init pplace) pplace 'count)
  *}
)

(DE pPlace_CHECK <pplace:pPlace>
  {* <>
    (SEMA_P ('sema pplace))
    (INT_GTR ('count pplace) 0)
  *}
)


(DE pPlace_USE <pplace:pPlace>
  (FIELD_SET (INT_SUB ('count pplace) 1) pplace 'count))

(DE pPlace_ADD <pplace:pPlace>
  {* <>
    (SEMA_P ('sema pplace))
    (FIELD_SET (INT_ADD ('count pplace) 1) pplace 'count)
    (SEMA_V ('sema pplace))
    (LISTLOOP (LIST_PERM ('output pplace)) 'Check)
  *}
)

(DE pPlace_RELEASE <pplace:pPlace> (SEMA_V ('sema pplace)))
```

Figure A. Example of a Petri net in the Garden language.

(uncover) and push (cover) it on the display. General window-management commands and a prompt window are at the very bottom of the screen.

Garden provides a consistent menu and mouse-oriented interface to most windows. Apple Macintosh-style pull-down menus provide editing and control-oriented options in the editors. Complex parameters are entered in each window via dialogue boxes. A base editor with a common set of editing operations is used for all textual displays and editing. It can cut and paste text among displays. All these facilities are available for user programs from within the Garden system. They are implemented using the tools of the Brown Workstation Environment.[6]

Garden also provides a consistent set of mouse utilities to select and define objects. A common routine provides a series of dialogue boxes that prompt you to define an object of a given type. This dialogue is modified on the basis of the expected type to simplify your task as much as possible. For example, when a string object is required, you must type only the contents of a string. In general, this dialogue prompts you for access to an object or for the type of a new object; in the latter case, it prompts you for the various fields. You can control the dialogue on newly created objects of your own types by identifying the expected field types and noting which fields should not be prompted for.

In addition, Garden provides a common facility for selecting objects and reusing them anywhere on the display. All editors let an object be selected by pressing the right mouse button. The selected object is stored in a common buffer as the current object. All dialogue boxes let you choose this current object as an alternative to entering all the box's fields with generic types for the object.

Environmental support. Garden's object-oriented database system provides many environment facilities.[7] The database system is designed to store all objects in use. Because objects are used to represent programs and data — as well as the semantics for evaluating object-based programs and the syntax rules for drawing and editing objects textually and visually — this facility saves your complete environment.

You can use the external database several ways. In its pure form, Garden provides a persistent environment where everything you do is stored in the database. Because the underlying database system is a real database system, many programmers can share the same object space with appropriate consistency checking and access control. The database system also provides version control, letting you create and restore versions of your whole environment.

You can access the database as if it were a workspace facility — taking an APL-like or Lisp-like approach to development, independent of whether it is done on one or several databases. You can open a database as read-only to access a set of languages, an environment, and the current system state. Then, as new facilities are added to this workspace, they can be saved in the workspace in a separate database that can in turn be used as the starting point for a later run.

A conceptual environment first requires a workable underlying model that can support a wide variety of languages.

Partitioning the system this way uses the inherent similarities between environmental support and database technology, providing such features as version and access control consistently across systems and among users. However, it does place an efficiency burden on both Garden and the database system since an environment will consist of many (typically 20,000 to 100,000) relatively small (40-byte average) objects that must be rapidly accessed when evaluating the program and displaying object structures.

Garden attacks this problem by providing an in-core database system to interface to the external database and by having the in-core system cache as many objects as possible. The in-core system provides additional environmental support, including background garbage collection over all objects not known to the database system in an effort to eliminate as many temporary objects as possible. It also provides a nested transaction mechanism with both fast and abortable transactions. This mechanism provides the basis for the general dependency mechanism that Garden offers; it could also be used as the basis for an undo facility because it lets you set marks in a transaction and partially abort the current transaction back to a previously defined mark.

Defining languages

The first problem in building a conceptual programming environment is to develop a workable underlying model that can support a wide variety of languages. This variety must include textual languages, visual languages, nonexecutable design languages, and languages that are now only a figment of someone's imagination. The key to solving this problem is the choice of an underlying representation through which languages can be defined.

Objects as a basis. Today's languages are usually defined formally in terms of their abstract syntax. This abstract syntax is represented as a tree where the internal nodes represent constructs such as statements and subprograms and where the leaf nodes represent semantically relevant terminals such as names and constants. A mapping from the concrete syntax (the textual form of the language) to the abstract syntax is provided either formally (by using a context-free grammar) or informally (by stating what the concrete form of each abstract construct is). The semantics of such languages are defined as mappings from the abstract syntax trees to some semantic form. This can be a program for operational semantics, a set of mathematical functions that show how the state changes for denotational semantics, or a set of logical rules for axiomatic semantics.

While abstract syntax trees work well for hierarchical, textual languages, they are not a natural representation for the nonhierarchical languages that arise in conceptual programming. In particular, two-dimensional languages such as finite-state automata and dataflow diagrams have a natural representation that is a general cyclic graph rather than a simple tree. Using an unnatural representation here would complicate language definition beyond what is desirable in conceptual programming.

Garden addresses this problem by generalizing the abstract syntax tree model of semantics into an object-based model. Objects represent programs directly. The instance data (fields) associated with an object are used three ways in specifying a program: (1) Some fields are structural, specifying an underlying graph of objects that replaces the abstract syntax tree. (2) Other fields are static attributes, containing data about the program instance relevant to the static semantics and corresponding to the attributes that would be attached to the abstract syntax tree to store the static semantics. (3) Still other fields are dynamic values, those that reflect the execution semantics, since Garden actually runs object-based programs.

Defining a language in Garden is a three-step process: (1) The type structure that serves as the object basis is developed. (2) The semantics for these types is defined. (3) The syntax, both textual and visual, is specified.

Defining the object basis. The first step is to define the set of types that describe program objects in the new language. This requires that you understand and characterize the language's components. You should create a type for each component. The type should have fields to contain both the structural information needed to describe the corresponding program structure and any state information needed to evaluate this object.

For example, suppose you wanted to develop a language based on Petri nets.[8] A Petri net is composed of places that can store markers and transitions that use markers and generate new ones. Typically, the markers are used for concurrency control while actions are associated with the transitions. Figure 2 shows an example Petri net. The object basis for Petri nets contains three types of objects: pNet objects represent complete Petri nets, pTrans objects represent transitions, and pPlace objects represent places.

A pNet object contains three fields, one holding the name of the Petri net and the others lists of transitions and places.

A pTrans object contains six fields. The first holds an identifying string that names the transition. The second field holds the associated action, an arbitrary object that Garden will evaluate when the transition is triggered so this Petri-net language can be associated naturally with any other Garden language. The next two fields contain the list of input places and the list of output places for the transition. The remaining fields hold a flag during execution indicating that the transition is currently active and a semaphore associated with the transition.

A pPlace object contains fields with the identifying name and the number of markers that should initially be at the place. It also contains a field with the list of transitions it is connected to, a field for the current count of the number of markers stored at the place, and a field containing the semaphore controlling access to the place.

Defining the semantics. The second step in defining a Garden language is to define the semantics of the types. You do this by associating evaluation functions with the types. Sometimes an evaluation function is defined only for the top-level type (for example, a flowchart or an automaton). In other cases, it is convenient to define the evaluation of an object of this type with the evaluation of its component objects, and it is thus necessary to define semantics for the component types.

Evaluation of a Petri net occurs when the pNet object is evaluated. The code associated with evaluating a pNet object first initializes the net and starts a control thread for each transition. It then tries to execute each transition by sending each one a Check message. After this, the Petri net will continue executing on its own until one of the transitions terminates.

Each transition evaluates in its own control thread. The message NewThread starts the control thread and sends a Work message to the transition. The code associated with this message is a simple loop that waits for the transition to be activated by doing a P operation on the transition's semaphore; it then evaluates the transition object itself to evaluate the associated action. When the action completes, an Add message is sent to each output place and the transition sends itself a Check message to see if it should fire again.

When a transition gets a Check message, it must check if it should fire and, if so, enable its semaphore. It first checks if the transition is active and, if so, returns immediately. It then loops through the input places, sending each a Check message. This message does a P operation on

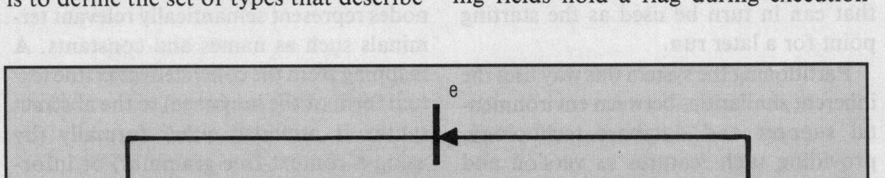


Figure 2. Petri net example.

the corresponding place's semaphore and returns a Boolean value indicating whether there is a marker at the place. If all markers are present, each input place is sent a Use message to decrement its marker count. Next, each input place is sent a Release message to do a V operation on its semaphore. Finally, if the transition should fire, it does so by setting its active flag and doing a V operation on its semaphore. (This example is simplified so that multiple connections from a place to a transition are not handled. In a more practical framework, a link object would represent the connection between a place and a transition.)

pPlace objects must respond to five messages: (1) Initialize, (2) Check to lock and check if there is a marker, (3) Use to remove a marker, (4) Add to add a marker, and (5) Release to unlock their semaphore. The only complexity here involves adding a marker to a place when all transitions connected to the place must be sent a Check message.

Defining the language syntax. The final step in developing a conceptual language in Garden is providing a syntax for the language. Garden offers several syntactic forms that can be defined for the new language. These reflect the different ways that the language can be displayed and input.

Garden provides a textual interface to objects (this interface is a Lisp-like language). In the language, objects are defined by putting the object type and values for the fields in braces ({ }). The field values can be named or be provided positionally, where the first value is the first field. Omitted fields are initialized to a default value. In addition, parentheses can be used instead of braces when only positional fields are given.

Currently, you can make only small variations in the textual forms allowed for an object. For input, you can define reasonable default values for fields and can determine the fields' order. If you want more complex manipulations, you can define an explicit Build or Instance operator for the type. The Build operator takes the type and the set of initial field values and builds the correct instance; the Instance operator must build an initial instance with no defined fields.

You have some control over the textual output form used to display the object. You can control the basic display form (whether to use the braced form with explicit field names or the parenthesized form).

Garden also gives you control over which fields are normally displayed. You can indicate fields that should not be displayed when showing a particular object type textually. Different fields can be indicated for objects that are displayed explicitly versus objects that are components of another object, a facility that provides readable displays of cyclic structures. In this case, all fields are displayed at the top level, but only identifying fields are displayed at lower levels.

Finally, if you need more control over the object's textual display, you can provide a message handler that is invoked when the object is to be displayed that will display a substitute object. This message handler can differ for objects displayed at the top level and those displayed inside other objects.

In addition to a textual syntax, Garden provides a general facility for interactively

Visual programming environments

The Garden effort draws on past research in several areas. Garden is a general-purpose visual programming system, providing many of the capabilities of visual programming languages. At the same time, Garden is an interactive programming environment in the flavor of the many versions of Lisp. Garden, being an object-oriented system, is closely related to previous work in this area, in particular to interactive systems such as Smalltalk and Lisp with types. And Garden is a programming environment designed for use with multiple windows on a powerful personal workstation and shares features of other such environments.

A multilanguage approach to conceptual programming such as Garden's differs considerably from the extensive body of work in visual languages.[1] Much of this work uses a single visual representation as a programming language. This work extends from flowchart programming to simulating finite-state automata, graphical dataflow representations, graphical data-structure representations, graphical programming-by-demonstration, and functional programming. Other work has concentrated on using visual representations to support design languages such as SADT, SREM, and the Yourdon method, or to provide machine-checkable documentation as a design aid. All these efforts are single-view systems. They do not support the wide range of views necessary for conceptual programming to be a practical approach to large-scale programming.

While Garden has a lot in common with interactive object-oriented environments such as Smalltalk, there are significant differences. Some of these are apparent at a finer level of detail than is presented in the main article, notably in the underlying model of objects. The basic difference, however, is the use of objects throughout the system to represent both programs and data. This, along with the lack of any preferred programming language or methodology, makes Garden an ideal testbed for working with several paradigms concurrently. Moreover, the heavy emphasis on the graphical display of objects provides a visual component that would have to programmed explicitly for each structure in other systems.

Garden's approach to producing a complete programming environment also differs from the multiview program-development systems for workstations developed over the past 10 years. These include the Cedar Mesa environment from Xerox PARC, the Magpie system from Tektronics, and the Pecan program-development system at Brown University. These systems are based on a single textual programming language. The Pecan system and PV try to provide alternative graphical representations to the textual programming language, but the experience with Pecan has shown that such graphical views have limited power and usefulness when they are tied to syntax. The syntactic basis forces you to treat these two-dimensional representations one-dimensionally, and the graphics provide no significant advantage over text. Because the wide range of graphical views that people use do not conveniently fall in the confines of a single language, it seems unlikely that a system based on a single programming language can effectively support them all.

Reference

1. G. Raeder, "A Survey of Current Graphical Programming Techniques," *Computer*, Aug. 1985, pp. 11-25.

requesting an object definition. This facility builds an instance of an object of the designated type and then uses a sequence of dialogue boxes to get field values for the component fields. You can customize this process to some extent for each object type you define. For example, you can order the fields so values are requested in a given order and abort the sequence of dialogue boxes to use default values for the remaining fields, or you can designate fields for which the system will not request values (these fields thus will hold the default value).

Additional control is provided through the typing of the fields, since values of different types are prompted for differently. For example, if a string is expected, you can enter the string without having to type quotes around it; if a type is expected, you are given dialogue-box buttons for the most common types.

You can also provide a message handler for the type to be invoked when you finish providing the fields. This function is then responsible for cleaning up the initial object and maintaining consistency of the data structures. Such functions are normally used to set up complex structural fields of objects with minimal user information. For a doubly linked list, for example, this function could set up the proper back links once you defined the forward links.

Defining the visual syntax. Defining the visual syntax of the new language is the final part of the language specification. This process has two parts: defining the visual display of the language's objects and defining the interpretation of graphical editing operations on this display. Garden tries to make this complex process as simple as possible by using a graphics package explicitly designed for drawing data structures. This package includes an editor that lets you interactively describe different ways to draw objects of a given type and a graphical editor that lets you modify the objects being displayed.[9]

Garden's general layout package, Gelo, is designed to be a flexible and powerful interface for drawing pictures of data structures or, in Garden, structures of objects representing programs and data. It provides a simple interface for designing the graphical representation for the structure and can lay out arbitrarily complex structures without programming the layouts.

Gelo's layouts are based on a hierarchy of graphical objects that are loosely related to the various components of the original data structure. Initially, four different types of objects are provided:

• Basic objects. These display simple user data, fields of more complex user data, and constants such as the name of the data type. They contain a text string representing their value. This string can be enclosed inside a rectangle, circle, or other figure, which can be filled and colored as desired.

• Tiled objects. These display fixed composites and recursive structures. They consist of a rectangle split into tiled regions. Each region contains another type of object to be drawn. You can impose some constraints on the regions' sizes and have some control on the tiles' sizes.

Defining the visual syntax of the new languages is the final part of the language specification.

• Layout objects. These display more complex or variable structures. They consist of a rectangular region into which nodes and arcs are placed. Gelo uses heuristics to automatically lay out the resulting graph. It chooses the amount of space between the nodes, the method for laying out the nodes, and the method for routing the arcs between the nodes.

• Arc objects. These represent arcs in layouts. They allow a characterization of how the arc is drawn and of the labels that can be placed on the arc.

This simple framework is powerful enough to handle a wide variety of data-structure displays. Moreover, the system supports a general mechanism for these blocks, letting new block types be added as needed.

Visual languages in Garden are characterized by mappings from the Garden object types of the language to Gelo objects. These mapping are defined through a set of stylized examples of the display for the types and are used to build the hierarchy of Gelo objects that correspond to a particular Garden structure. You can display and edit this hierarchy.

Figure 3 shows the stylized mappings for the Petri-net language. A pNet object, the top level of the net, is drawn as a simple tiling with two tiles, one on top to hold the name and one below to hold the Petri net's picture. The layout used in the editor for this mapping is shown in Figure 3a.

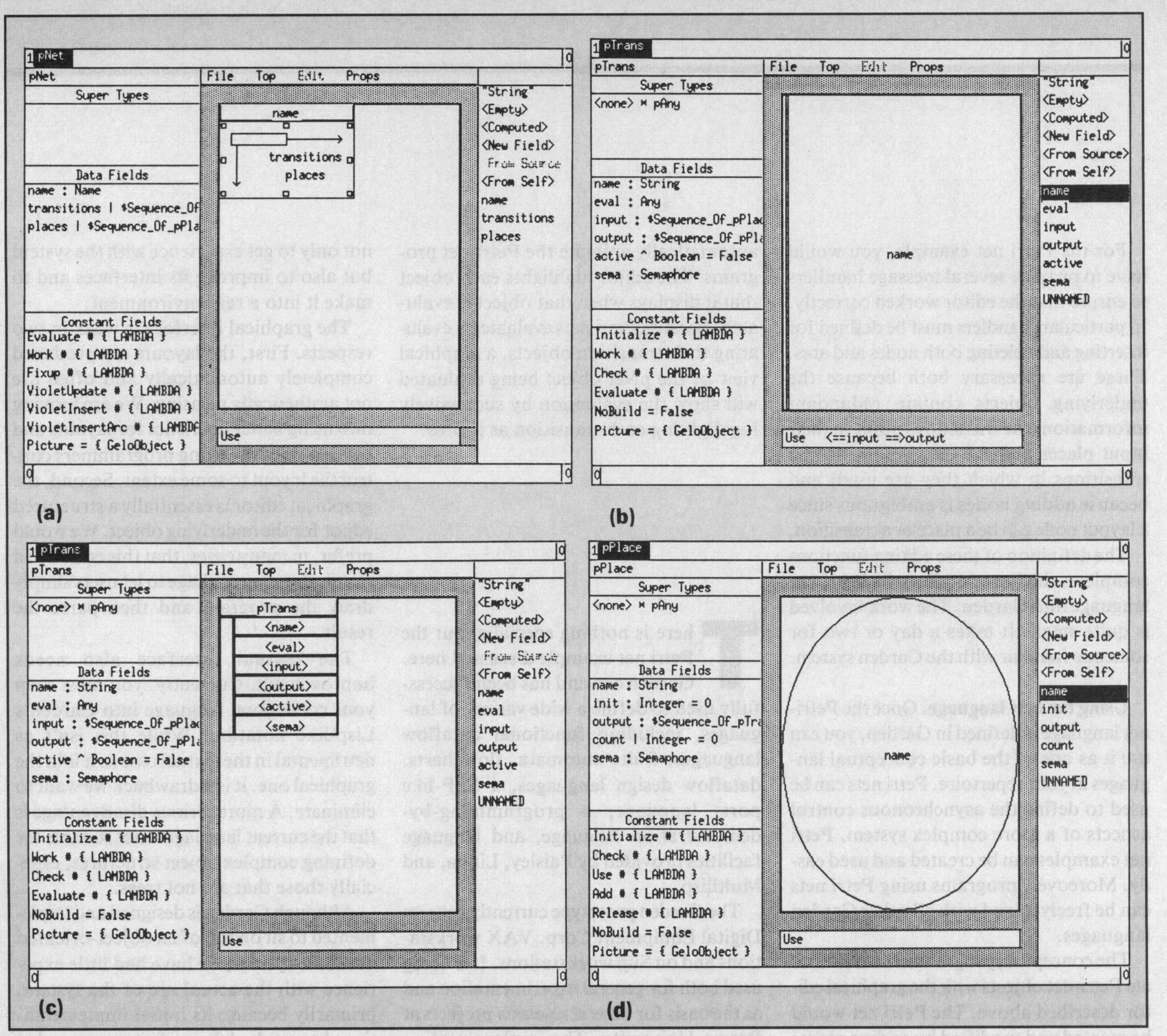
Two mappings are shown for pTrans objects.

The first is used when the object is drawn inside a layout. This is the form used in the picture of the Petri net; it is a vertical line labeled with the identifying string of the transition. The mapping itself, shown in Figure 3b, specifies that both the input and output fields should be used, with arcs drawn from the objects denoted by the input fields and arcs drawn to the objects denoted by the output fields.

The second mapping, the default mapping for pTrans objects shown in Figure 3c, is used when the object occurs outside a layout. This is a tiling containing the type name on top, an indentation bar on the left, and then the fields in the order in which they occur in the type definition. The mapping for pPlace objects in the picture of a Petri net (Figure 3d) is a basic type shaped as a circle with the identifying string displayed.

The result of these mappings is the Petri net example of Figure 2 displayed by Garden in Figure 4. Garden lets you use this visual form to create and edit objects as well as for simple display. The editor uses Gelo to put up a display and then lets you apply graphical editing operations to modify it. Each graphical editing operation must map to a change of the underlying object. This change is made and the display is then updated for the modified object.

To support consistent graphical editing for a variety of structures, the graphical structure editor translates your editing request into requests to change values or



```
pNet
Super Types

Data Fields
name : Name
transitions | $Sequence_Of
places | $Sequence_Of_pPla

Constant Fields
Evaluate # { LAMBDA }
Work # { LAMBDA }
Fixup # { LAMBDA }
VioletUpdate # { LAMBDA }
VioletInsert # { LAMBDA }
VioletInsertArc # { LAMBDA
Picture = { GeloObject }

"String"
<Empty>
<Computed>
<New Field>
From Source
<From Self>
name
transitions
places
```

```
pTrans
Super Types
<none> × pAny

Data Fields
name : String
eval : Any
input : $Sequence_Of_pPlac
output : $Sequence_Of_pPla
active ^ Boolean = False
sema : Semaphore

Constant Fields
Initialize # { LAMBDA }
Work # { LAMBDA }
Check # { LAMBDA }
Evaluate # { LAMBDA }
NoBuild = False
Picture = { GeloObject }

"String"
<Empty>
<Computed>
<New Field>
<From Source>
<From Self>
name
eval
input
output
active
sema
UNNAMED
```

```
pPlace
Super Types
<none> × pAny

Data Fields
name : String
init : Integer = 0
output : $Sequence_Of_pTra
count | Integer = 0
sema : Semaphore

Constant Fields
Initialize # { LAMBDA }
Check # { LAMBDA }
Use # { LAMBDA }
Add # { LAMBDA }
Release # { LAMBDA }
NoBuild = False
Picture = { GeloObject }

"String"
<Empty>
<Computed>
<New Field>
<From Source>
<From Self>
name
init
output
count
sema
UNNAMED
```

Figure 3. Stylized mappings for Petri-net objects: **(a)** layout for pNet object, **(b)** pTrans object mapping when drawn in a layout, **(c)** default pTrans object mapping, and **(d)** pPlace object mapping.

to add or delete elements from a layout. These requests are mapped by Garden into messages sent to the displayed objects. Garden's default facility for handling these messages is usually sufficient.

But one instance where it is insufficient is when the mapping from Garden objects to Gelo types is so complex that the facility cannot determine where a node or arc can be added or should be deleted. It also fails when nodes or arcs are added several times but actually appear only once. And it will succeed but yield inconsistent object structures if the Garden objects contain redundant information that was not used in the drawing, such as doubly linked lists. In these cases, you must provide message handlers that perform the correct operation on the particular data types.

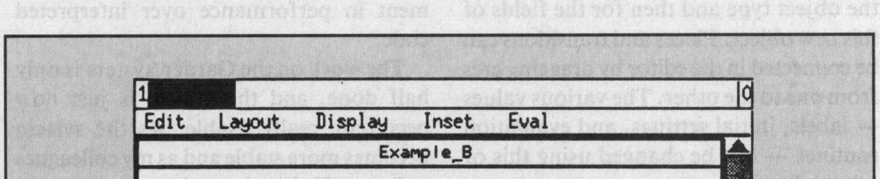


Figure 4. Garden's Petri-net display.

For the Petri net example, you would have to provide several message handlers to ensure that the editor worked correctly. In particular, handlers must be defined for inserting and deleting both nodes and arcs. These are necessary both because the underlying objects contain redundant information (the transitions point to their input places and the places point to the transitions in which they are used) and because adding nodes is ambiguous since a layout node can be a place or a transition.

The definition of these editing functions completes the introduction of a Petri-net language into Garden. The work involved is quite small: It takes a day or two for someone familiar with the Garden system.

Using the new language. Once the Petri-net language is defined in Garden, you can use it as one of the basic conceptual languages in your repertoire. Petri nets can be used to define the asynchronous control aspects of a more complex system. Petri net examples can be created and used easily. Moreover, programs using Petri nets can be freely mixed with all other Garden languages.

The conceptual programmer would create Petri-net objects with the graphical editor described above. The Petri net would be created and modified by adding places and transitions to the drawing of the net. When a new object is to be inserted, the system uses dialogue boxes to prompt for the object type and then for the fields of this new object. Places and transitions can be connected in the editor by dragging arcs from one to the other. The various values — labels, initial settings, and evaluation routines — can be changed using this or other editors.

Evaluation of the Petri net begins with the top-level pNet object. Garden's general debugging and monitoring facilities are available for the new language, and, if there is an error during evaluation, you would be placed in a read-eval-print loop at the point of the error so the variables can be queried and execution can be continued if requested. Similarly, you could trace and suspend execution of any object composing the Petri net.

In addition to these standard debugging facilities, the Garden graphical editor can automatically animate the Petri-net programs. The editor highlights each object that it displays when that object is evaluated. Because Petri nets evaluate by evaluating their transition objects, a graphical view of the pNet object being evaluated will show the evaluation by successively highlighting each transition as it fires.

There is nothing special about the Petri net example discussed here. Garden can and has been successfully used to define a wide variety of languages, including functional dataflow languages, finite automata, flowcharts, dataflow design languages, a CSP-like ports language, a programming-by-demonstration language, and language facilities provided by Paisley, Linda, and Multilisp.

The Garden prototype currently runs on Digital Equipment Corp. VAX workstations and on Sun workstations. It is being used both for general experimentation and as the basis for several research projects at Brown University. The implementation runs at about the speed of a Lisp interpreter and includes a compiler that produces C code to give a tenfold improvement in performance over interpreted code.

The work on the Garden system is only half done, and the system is just now becoming really usable. As the system becomes more stable and as my colleagues at Brown University and I get more experience with it, we hope to test out the promises of conceptual programming and see if this is a viable approach to design that improves programmer productivity as we hope it will. Such experiments will at first be informal, based on the experience of the initial users, but we hope be more rigorous later when the system can handle a controlled experiment.

We are working in several areas related to this system and its concepts because Garden is by no means perfect or complete. Significant work remains to be done not only to get experience with the system but also to improve its interfaces and to make it into a real environment.

The graphical interface is weak in two respects. First, the layouts are produced completely automatically and often are not aesthetically pleasing. We are looking into using better heuristics for layouts and finding ways of letting programmers control the layout to some extent. Second, the graphical editor is essentially a structured editor for the underlying object. We would prefer, in many cases, that this editor be a simple drawing package to let users simply draw the program and then parse the result.

The textual interface also needs improvement. Currently, you must map your conceptual language into Garden's Lisp-like notation. While this isn't as detrimental in the textual case as it is in the graphical one, it is a drawback we want to eliminate. A more serious disadvantage is that the current language is not suitable for defining complex object structures, especially those that are not trees.

Although Garden is designed and implemented to sit on top of an object-oriented database system, we have had little experience with the actual use of the system, primarily because its initial implementations has not had the performance needed for an interactive system. The performance problem is being addressed and may be resolved in the next year.

We are also looking at the very difficult problem of semantics that the current implementation of Garden tries to avoid. Ideally, a multiparadigm system should have a consistent internal semantics. Such a semantics provides a basis for compilation, for consistency checking between views, and for view mapping. View mapping is important if the several programming paradigms are to be applied to the same portion of the program, a situation that arises when you want to see and use both a dataflow and a control-flow view of the same module. In this case, the system must be able to map changes in either view to appropriate changes in the other. With a broad spectrum of views, the simplest way of allowing such mappings is to have a common semantic basis and to generate the views from this basis. -ϕ-

Acknowledgments

This research was supported in part by the Office of Naval Research and the Defense Dept.'s Advanced Research Projects Agency under contract N00014-83-K-0146 and ARPA Order No. 4786, by National Science Foundation grant SER80-04974, by a contract with IBM, by a grant from AT&T Foundation, and by a grant from Digital Equipment Corp. Partial equipment support was provided by Apollo Computer, Inc.

References

1. D.T. Ross, "Applications and Extensions of SADT," *Computer*, April 1985, pp. 25-35.
2. M. Alford, "SREM at the Age of Eight: The Distributed Computing Design System," *Computer*, April 1985, pp. 36-46.
3. F.P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, April 1987, pp. 10-19.
4. S.P. Reiss, "An Object-Oriented Framework for Graphical Programming," *SIGPlan Notices*, Oct. 1986, pp. 49-57.
5. A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, Mass., 1983.
6. J.N. Pato, S.P. Reiss, and M.H. Brown, "An Environment for Workstations," *Proc. IEEE Conf. Software Tools*, CS Press, Los Alamitos, Calif., 1985, pp. 112-117.
7. A.H. Skarra, S.B. Zdonik, and S.P. Reiss, "An Object Server for an Object-Oriented Database System," *Proc. Workshop Object-Oriented Database Systems*, CS Press, Los Alamitos, Calif., 1986, pp. 196-204.
8. J.L. Peterson, "Petri Nets," *Computing Surveys*, Sept. 1977, pp. 223-252.
9. S.P. Reiss and J.N. Pato, "Displaying Programs and Data Structures," *Proc. 20th Hawaii Int'l Conf. System Sciences*, CS Press, Los Alamitos, Calif., 1987

Steven P. Reiss is an associate professor of computer science at Brown University. His research interests include programming environments, graphical programming, database implementation, statistical database security, and computational geometry. Before working on Garden, he developed the Pecan program-development system.

Reiss received a BA in mathematics from Dartmouth College and a PhD in computer science from Yale University.

Address questions about this article to Reiss at Computer Science Dept., Brown University, Providence, RI 02912.

Reader Service Number 2