Agentsheets: A Medium for Creating Domain-Oriented Visual Languages

Alexander Repenning
Tamara Sumner
University of Colorado

Customized visual representations enable end users to achieve their programming goals. Here, designers work with users to tailor visual programming languages to specific problem domains.

In the high-technology workplace, many professionals work in domains that involve the analysis of complex, dynamic systems. These systems include computer networks and user interfaces. To cope with the complexity intrinsic to these domains, environments for visualizing and interacting with dynamic processes are essential. Using current technology, creating and modifying such environments often requires traditional computer science programming skills. However, the professionals working in these domains typically are not formally trained in computer science, have no interest in learning programming skills per se, and want to use computers only to solve their specific problems.[1]

These domain professionals could benefit from visual programming languages if several barriers to programming were removed. Specifically, languages should not require users to

1. build up desired program behaviors from low-level programming constructs such as iteration and conditionals,[2] or
2. bridge the semantic gap between their conceptual model of the problem to be solved and the computational model of the program.[2,3]

We discuss how these barriers to programming can be lowered through languages with familiar, visible representations that let users express programs in terms pertinent to the problem to be solved. We characterize programming approaches along the dimensions of visualization and domain orientation[4] (Figure 1).

Visualization is a syntactic characteristic indicating *how* concepts are presented, that is, the look of concepts but not their semantics. The Prograph programming language[5] is based on the visualization of dataflow and functional programming. Its two-dimensional visual syntax simplifies program construction. However, Prograph's language components use constructs such as iteration and therefore still require users to build up problem domain behaviors by assembling lower level primitives.

Domain orientation, on the other hand, is a semantic characteristic describing *which* things are represented. Domain-oriented languages feature components directly relevant to end users' tasks and therefore narrow the semantic gap between the problem and programming domains.[3] LabView evolved from circuit design.[6] It has been enriched with general-purpose programming constructs but has preserved a domain-oriented flavor. Applications such as Mathematica and Matlab feature domain-oriented constructs, such as integration and Fourier transformations, familiar to mathematicians. The spreadsheet formula language can be viewed as being domain-oriented, since it is based on the tabular forms originally used by accountants. Many of these languages in the upper half of Figure 1 have enjoyed widespread success as end-user programming environments.

0018-9162/95/$4.00 © 1995 IEEE

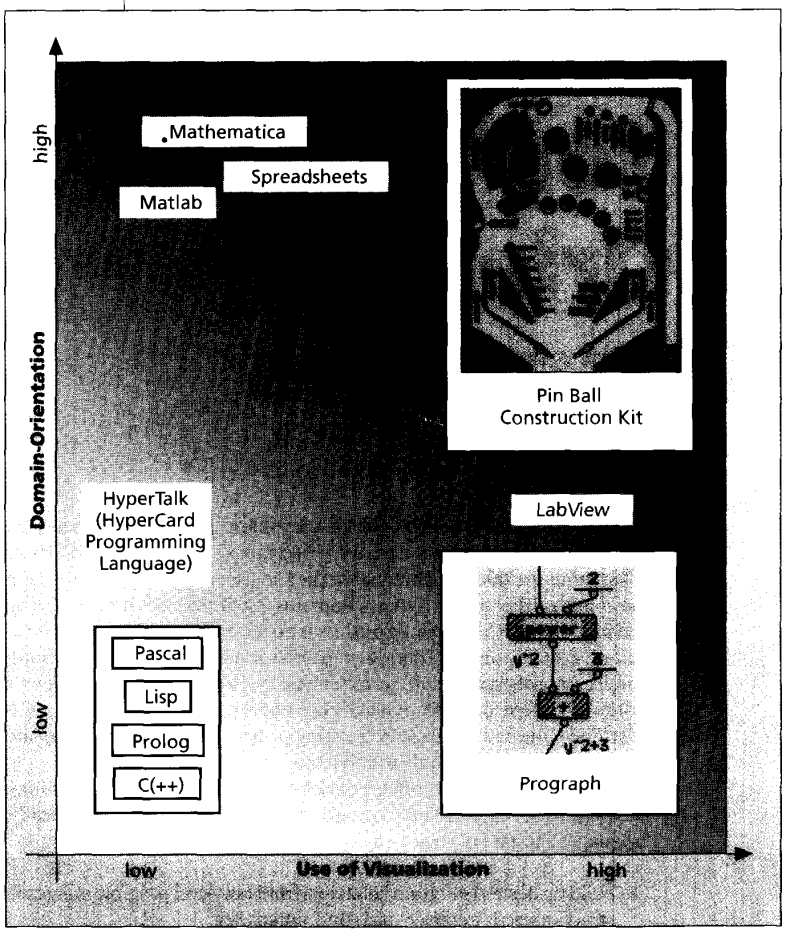

Figure 1. Languages can be characterized along the dimensions of visualization and domain orientation. Visualization is a syntactic characteristic indicating how concepts are presented. Domain orientation, a semantic characteristic describing which things are represented, can bridge the gap between problem and programming domains. The Pinball Construction Kit is a domain-oriented visual language, whereas Prograph is a general-purpose visual language.

We are interested in creating domain-oriented visual programming languages to support the activities of professionals working in domains involving dynamic systems analysis. These languages elevate the task of programming to the interaction between end users and problem-related components. For instance, using a domain-oriented visual language such as the Pinball Construction Kit, end users program by arranging and interacting with language components such as bumpers, obstacles, and flippers.[3]

We describe a design methodology and a tool for creating domain-oriented, end-user programming languages that effectively use visualization. We first describe a collaborative design methodology involving end users and designers. We then present Agentsheets, a tool for creating domain-oriented visual programming languages, and illustrate how it supports collaborative design by examining experiences from a real language-design project. Finally, we summarize the contributions of our approach and discuss its viability in industrial design projects.

REAL-TIME COLLABORATIVE DESIGN METHODOLOGY

Well-designed domain-oriented languages have a small number of high-level abstractions.[1] These abstractions should be *expressive* enough to allow users to state solutions for commonly occurring problems, yet *constrained* enough to shield users from decisions and details they don't want to be concerned with. That is, the abstraction level of the language components should match the user's conceptualization of the problem. A key challenge in designing domain-oriented languages is determining what these abstractions or language components should be. It's unlikely that language designers working alone could determine these components because this requires a deep understanding of the end users' problem domain. Conversely, it's unlikely that end users could articulate what kind of components they need because much professional knowledge is tacit and emerges only in actual practice.[7] Thus, end users and designers need to work together *to design and evolve domain-specific languages through use to create the right abstractions.*

The process

Figure 2 illustrates such a process. These collaboration sessions typically occupy a couple of hours when the end user and the designer meet and write a program to solve a real task in the problem domain. The key steps are

1. *Use.* The end user constructs a program with the language while the designer primarily observes the process, noting usability problems and answering questions as needed.
2. *Breakdown.* At some point, the end user experiences a breakdown in the language because the components it provides are insufficient to express desired concepts or are somehow inappropriate for the current situation.
3. *Negotiation.* At that point, both parties must discuss the breakdown and negotiate how to modify the language to address the shortcoming. In the case of insufficient expressiveness, both parties must decide which components to change or add to the language. In our experience, inappropriate components often result when the provided domain components are not at the appropriate abstraction level for the task being performed. In these cases, language changes often involve modifying or combining existing language components to create higher level domain abstractions.
4. *Modification.* Once the breakdown and possible lan-

guage modifications are discussed, the designer steps in and makes modifications.

5. *Testing*. After briefly testing the modifications, the designer gives control back to the end user, who continues to construct a program.

The tool

Powerful tool support is required for this real-time collaborative approach to be viable. Specifically, the tool must

- enable the designer to quickly change existing language components and add new ones without disrupting the end user's programming process, and
- allow control to switch quickly between the end user and the designer without losing each party's current work context.

These sessions are similar to other forms of collaborative prototyping. The prototypes described by Bodker and Gronbaek[8] and Madsen and Aiken[9] are built with the Hypercard program. Both projects report favorably on the extensions made possible in real time by many of Hypercard's direct manipulation features, which easily change the look and placement of interface components. However, both projects also report difficulties when real-time extensions require programming. Madsen and Aiken[9] noted that Hypercard's "missing object-oriented features made it harder to create and modify domain-specific building blocks" (page 63).

Our approach relies on the Agentsheets system, which supports the collaborative design of domain-oriented visual programming languages. Agentsheets provides powerful tools and mechanisms that enable substantive extensions, such as creating and modifying domain-specific language components, to be made in real time.

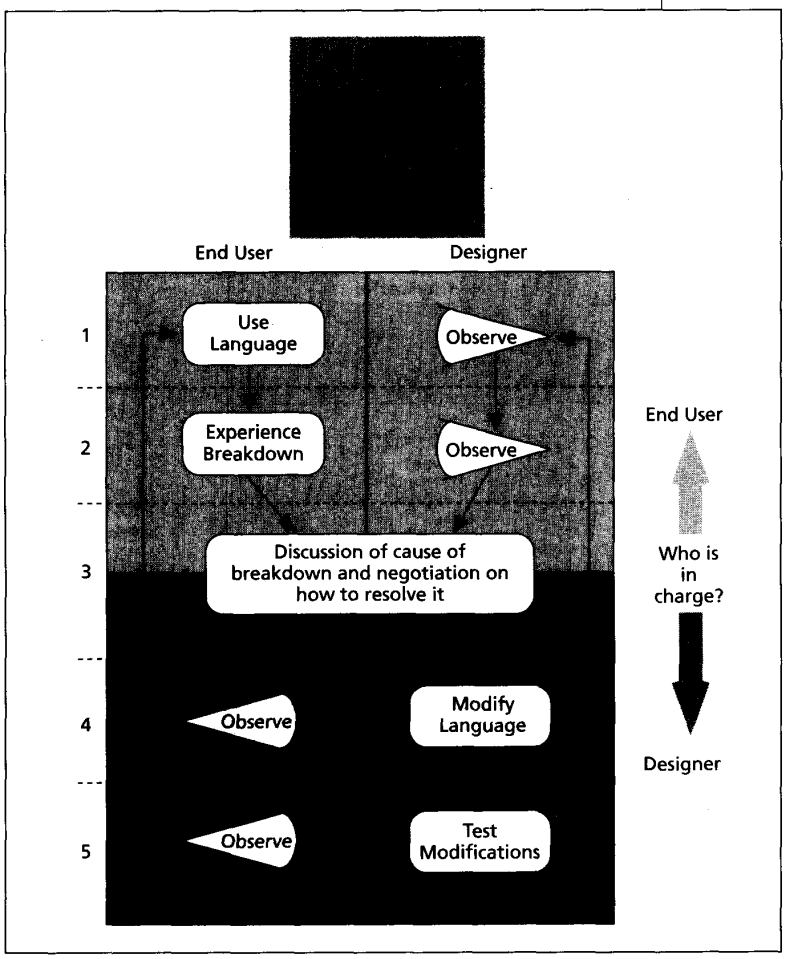

Figure 2. In a real-time collaborative design session, end users and designers work together to design and extend the visual language. The end user constructs a program representing a real problem in the domain under consideration. When a breakdown is encountered, both parties work together to understand the cause and to design language modifications in real time to overcome the breakdown.

AGENTSHEETS: A COLLABORATION MEDIUM

Agentsheets[10,11] is a Macintosh application implemented in Common Lisp. In the last four years, it has been used to create more than 40 domain-oriented visual programming languages in areas such as art, artificial life, education, and environmental design.

Using agents

Visual languages created with Agentsheets consist of autonomous, communicating agents organized in a grid similar to a spreadsheet. The grid that contains the agents is called the *agentsheet*. An agentsheet cell can contain any number of stacked agents. Users interact with agents through direct manipulation. Agents can be animated, move among cells, and play sounds.

Designers create visual programming languages by defining the look and behavior of agents. Figure 3 shows visual languages created for four unique problem domains. Each language component is an agent (a software routine that waits in the background and performs an action when a specified event occurs). To define the visual language syntax and semantics, the designer must specify how agents communicate and how they can be arranged in the grid. Agent communication can be *explicit*, using links between agents, or *implicit*, based on spatial relationships between agents such as proximity, order, and distance. For instance, the LabSheet application (Figure 3) uses explicit spatial relationships. The LabSheet application represents dataflow[6] between agents using links. Agents in LabSheet represent language components such as data-input cells, data-output cells, and functions such as multipliers.

In the ProNet application (Figure 3), agents communi-

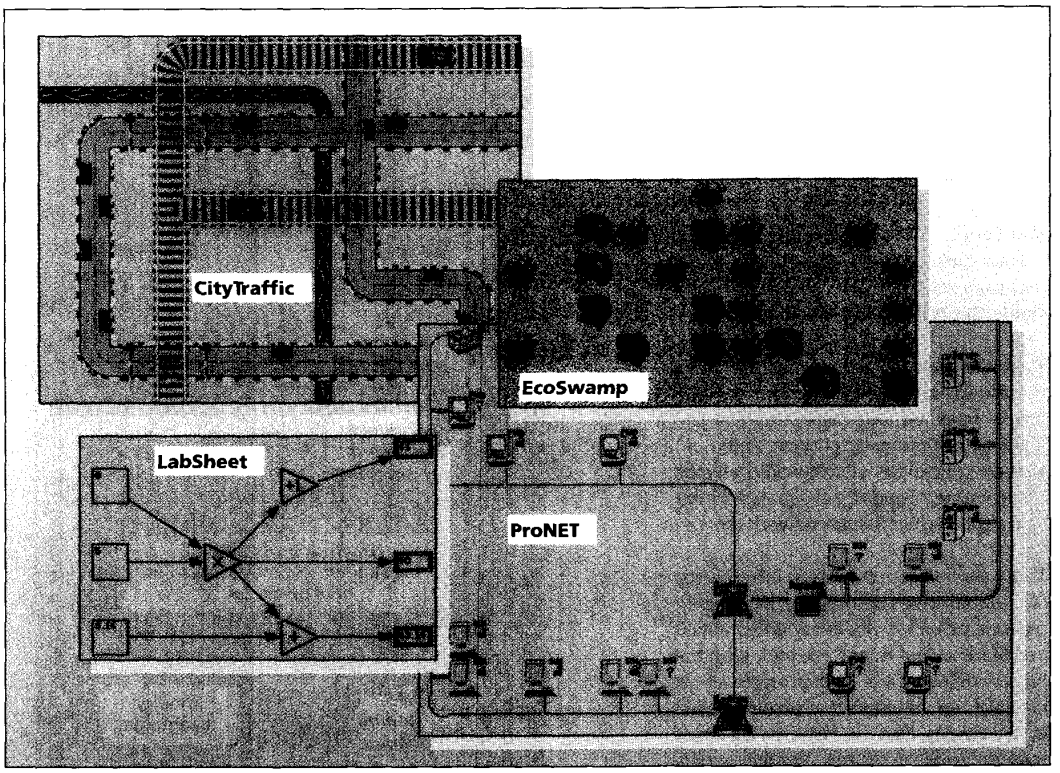

Figure 3. Example Agentsheets applications: LabSheets, ProNet, EcoSwamp, and CityTraffic.

cate implicitly based on adjacency. Adjacency relationships determine network connectivity; that is, end users connect computers such as Macintoshes and Sun workstations, using agents representing network wires. Depending on the network specifications given by end users, generic network wires proactively turn into more specific connections such as Ethernet cables or into necessary network devices such as gateways.

Agentsheets can also be used to create dynamic simulations such as the EcoSwamp and CityTraffic applications (Figure 3). Both are educational interactive simulation programs that use implicit communication between agents based on both proximity and overlap. The language components in EcoSwamp are animals such as frogs, alligators, and bugs, and environment patches such as land and water. Frogs autonomously move around, eat nearby bugs, and lay eggs only when in water. In the CityTraffic application, agents representing cars move forward when a piece of road is in front of them and stop when they are next to traffic signals.

Supporting collaboration through role-specific views

An effective collaboration medium requires that the respective roles of end users and designers be understood; that is, tools must

- support the tasks that end users and designers individually perform, and
- facilitate dialogue between end users and designers.

Agentsheets provides *role-specific views* and tools to meet the specific needs of end users and designers (Figure 4).

End Users Interact with Domain-Oriented Components. End users create programs using language components based on familiar, visible representations pertinent to their problem domain. In the CityTraffic application (Figure 4), these components are cars, trains, streets, and railway tracks. (CityTraffic is an educational simulation program used by children.) End users "program" by assembling these components and interacting with them.

- *Assemble components.* End users select components from the Gallery (see Figure 4) and assemble them in the worksheet into meaningful diagrams. For instance, they assemble individual road segments into a road system, put cars onto the roads, and install traffic signals to control traffic to create a traffic simulation.
- *Interact with components.* End users select tools from the worksheet's tool bar and apply them to rearrange, link, and query components. In the CityTraffic application, the state and frequency of traffic lights can be changed by applying the operate-on tool to traffic lights. Furthermore, while cars are moving, end users can change car parameters (such as the likelihood a car will run a traffic light), introduce additional traffic signals, and change the topology of streets and railway tracks. This interaction style extends direct manipulation schemes by allowing end users to more flexibly

interact with applications consisting of large numbers of autonomous agents. We call this interaction style participatory theater[11] because, in terms of a theatrical metaphor,[12] it lets end users direct the actors (agents) without stopping the play (running the application).

DESIGNERS DEFINE LOOK AND BEHAVIOR OF COMPONENTS. To support the dialogue between end users and designers, a collaboration medium must provide efficient, incremental mechanisms for the designer to specify language components or agents. Agentsheets lets designers

- *Incrementally define the look of agents.* An agent's look is represented by using *depictions*. These are created and stored in the Gallery using tools available in the designer view. For instance, designers can create a horizontal road depiction using the Depiction Editor (see Figure 7). However, sometimes in visual languages, many related depictions are visual variations of a common underlying theme. Many depictions are needed for one language component to represent the underlying agent's changes in state. Language components can also be placed in many different orientations with a depiction representing each one. Creating each related representation by hand with the depiction editor would be tedious and time consuming. In designer view, the Gallery provides tools that help to automate this process. Some tools help to combine existing depictions to create new ones. In the CityTraffic application, a car depiction could be combined with a gas station depiction to create a new one representing a car that is out of gas, that is, a change of state. Other tools help designers create new depictions by incrementally modifying existing ones using provided geometric transformations. In CityTraffic, road depictions for many orientations were needed; the vertical road and bent road depictions were created automatically from the horizontal road depiction using the provided transformations.
- *Incrementally define the behavior of agents.* This behavior determines how agents interact with each other and with end users through tools. Designers define agent behavior through the AgenTalk editor. Because AgenTalk is object oriented, behavior can be defined incrementally through inheritance. The class browser helps to locate functionality in the form of existing agent classes. By building on top of these classes, visual language components directly inherit many of their basic behaviors such as the ability to be linked and queried. Thus, the language designer need only augment these inherited behaviors with behaviors specific to the problem domain.

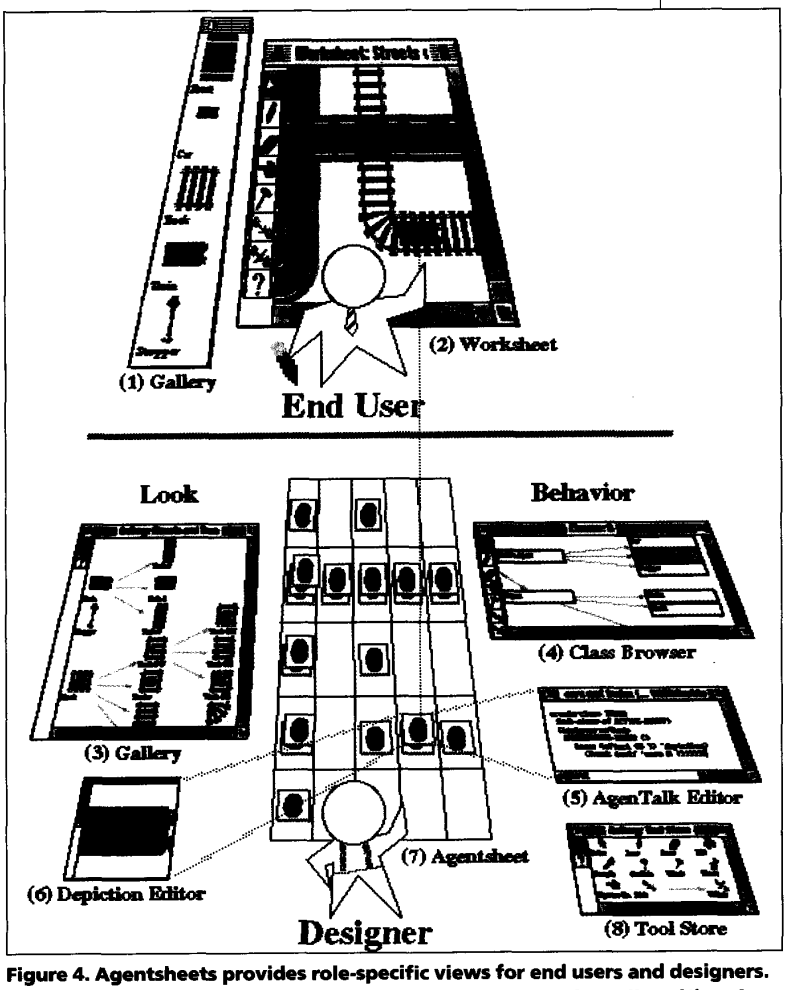

Figure 4. Agentsheets provides role-specific views for end users and designers. End users compose programs by selecting components in the gallery (1) and putting them into a worksheet (2). Designers perceive worksheets (2) as agentsheets (7), that is, agents organized in a grid. They create networks of related depictions in the expanded gallery (3), design icons with the depiction editor (6), define behavior with the AgenTalk editor (5) by reusing existing agent classes found in the class browser (4), and create or subscribe to tools in the tool store (8). Tools enable end users to interact with agents.

In the CityTraffic application, the behaviors of cars and trains must be defined. Cars have to follow roads, watch out for other cars, avoid collisions, and obey traffic signals. To define the behavior of a train to follow railway tracks, AgenTalk code could look like

```
(i)   (create-class TRAIN
(ii)      (sub-class-of ACTIVE-AGENT)
(iii)     (instance-methods
(iv)         (FOREGROUND-TASKS ()
(v)             (case (effect (ø 1) 'depiction)
(vi)                (TRACK (self 'move ø 1))))))
```

The train class (i) is defined as a subclass of ACTIVE-

AGENT. Active agents are autonomous agents receiving FOREGROUND-TASKS messages to initiate actions. The case statement (v) contains a spatial operator, (effect (ø 1) 'depiction), returning the name of the depiction to the right of the train agent. If the depiction of that agent on the right is a TRACK depiction (vi), the train moves to the right. As soon as the class is defined and the agent scheduler activated, the train begins to follow the tracks.

AgenTalk programming, which requires knowledge of Lisp and object-oriented principles, is typically done only by designers. However, end users can define simple behaviors through graphical rewrite rules,[10] which are before/after pictures that end users can edit.

- *Define interaction tools.* Designers provide mechanisms for end users to interact with agents by selectively subscribing to existing tools featured in the tool store, extending the behavior of existing tools, or creating new domain-specific tools. Agentsheets provides a default set of tools with predefined behaviors. For instance, applying the eraser tool to a car will delete it (unless the designer has specified otherwise).

In summary, Agentsheets enables real-time collaboration between end users and designers by providing

1. tools specialized to each of their respective roles,
2. the ability to quickly switch between these two roles, and
3. incremental specification mechanisms that significantly shorten the use-redevelopment loop and let language extensions be performed in real-time with end-user participation.

THE VDDE LANGUAGE: A CASE STUDY

This section illustrates how Agentsheets served as a collaboration medium during real-time design sessions, using excerpts from an actual language design project. The visual language discussed is embedded in a design environment supporting the design and simulation of phone-based user interfaces—the Voice Dialog Design Environment (VDDE). This system minimizes the time it takes to prototype phone-based interfaces by enabling user-interface designers to create their own simulations. This system is the result of a three-year collaboration between researchers at the University of Colorado and professional user-interface designers at US West Advanced Technologies.

The voice dialog design task

Voice dialog applications are an important technology for many businesses because they reduce the need for human phone operators and provide callers with direct access to information concerning business services. Voice information systems and voice messaging systems are typical applications. Designing in this domain means specifying the interface for a voice dialog application at a detailed level. Interfaces consist of a series of voice-prompted menus that request the user to perform certain actions, such as, "To listen to your messages, press 1." The user issues commands by pressing touch-tone buttons on the telephone keypad, and the system responds with appropriate voice phrases.

Interface designs are typically represented using static diagrams similar to flowcharts. These charts specify the control flow of the interface, possible user actions, and the text of all audio prompts and messages in the interface.

It is difficult for designers and end users of these applications to anticipate what the final audio design will sound like by simply looking at a static diagram. Thus, simulations are built to let designers and end users directly experience the final audio interface. Unfortunately, a simple design simulation takes a professional programmer several days to build using current software packages; a complex design simulation can take a couple of weeks. One goal of the VDDE project was to provide a design environment that shortened the time to build simulations by empowering user-interface designers to quickly create their own simulations.

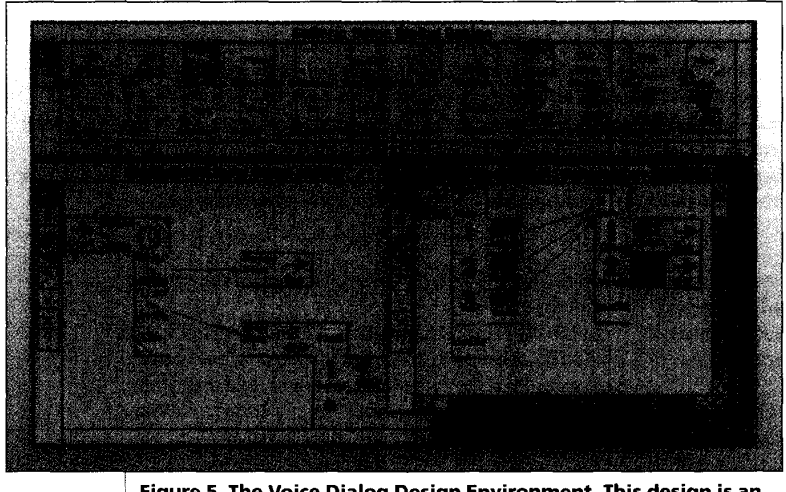

Figure 5. The Voice Dialog Design Environment. This design is an interface for a delivery service in a pizza parlor. If customers call when the business is closed, they hear a standard message. If customers call during business hours, they can navigate through a series of voice menus to specify their pizza order. The design shown consists of two programs: a subprogram processing the incoming call based on business hours and a subprogram for specifying the desired pizza order.

System overview

The VDDE system provides a gallery (top window, Figure 5) of components, such as voice menus and prompts. The lower two windows are worksheets where end users program by assembling language components according to three rules:

- *The horizontal rule.* Components placed physically adjacent to each other within a row are executed from left to right.
- *The vertical rule.* Components placed physically adjacent to each other within a column describe available options at that point in the execution sequence.
- *The arrow rule.* Linking two components also defines execution ordering

and is identical to placing components next to each other horizontally.

The interface design can be simulated at any time. Simulation consists of a visual trace of the execution path combined with an audio presentation of all prompts and messages encountered.

Analysis of a design session

In this session, the end user wanted to simulate a new product interface. She created several voice menus and then experienced a breakdown in terms of inappropriate or unexpected language components (see time period 1, Figure 6). Because the current language's undefined phone numbers were represented as zero, determining if a phone number had been specified involved testing for zero. However, in this domain, it is more appropriate to think of phone numbers and other data items as being either "present" or "absent." Using the depiction editor provided in Agentsheets' designer view, the designer modified the look of some language components to reflect this domain terminology.

After further design, the end user experienced another breakdown; this one highlighted a crucial lack of expressiveness in the language (see time period 3, Figure 6). The current language supported conditional branching on touch-tone button presses. However, features can be "on" or "off," or messages can be "new," "saved," or "unread," which demonstrates the importance of enumerated data types for this domain. For this session, the designer added language components to overcome the specific breakdown by creating an enumerated type "call forwarding" with possible values of "on" or "off," and visual components to manipulate this data type. The designer quickly created these new language components by incrementally refining the behavior of existing agent classes. The end user then used these new components to complete her program.

This session uncovered two shortcomings in the visual language design. In the first case (that is, present and absent), the functionality provided was sufficient, but its presentation did not match the user's conceptualization of the problem domain. The second case uncovered a

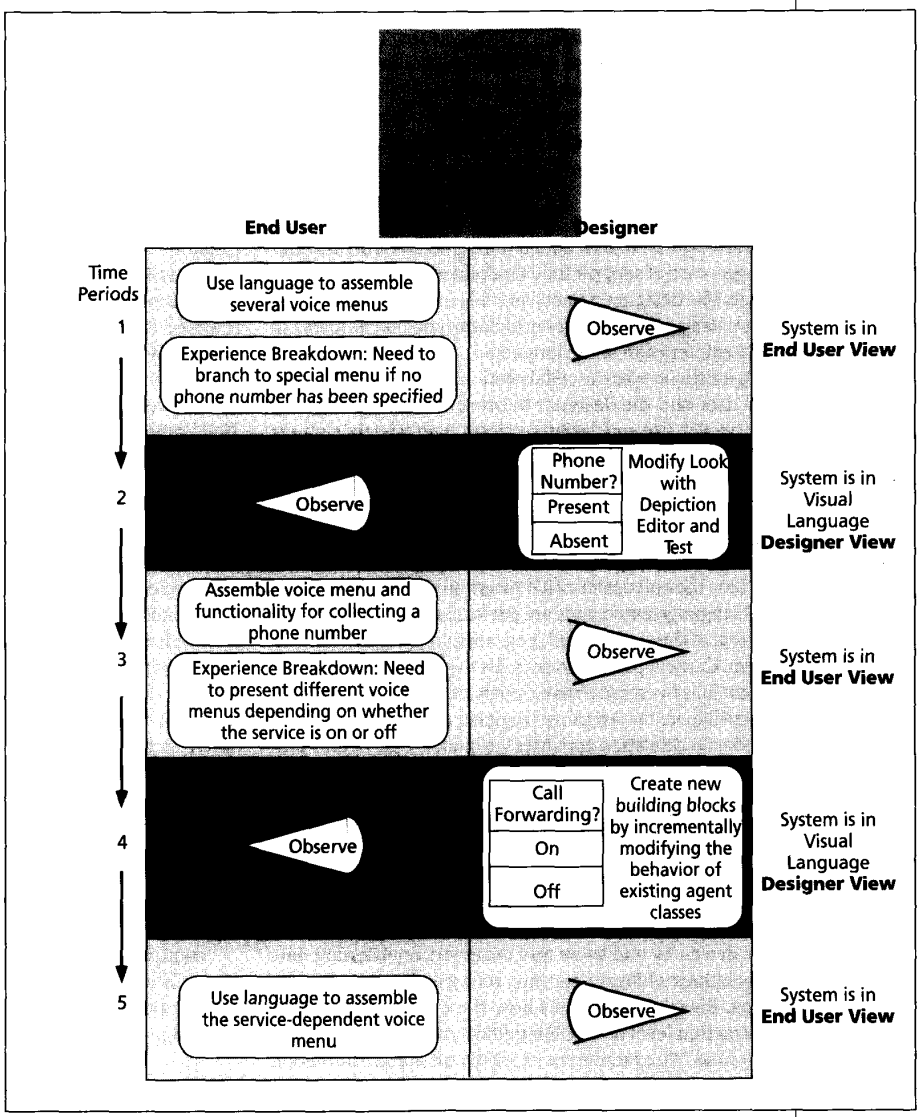

Figure 6. Language modifications in a two-hour collaborative design session. In time period 1, the end user experienced a breakdown when trying to branch to a special menu. In time period 2, the designer modified existing language components to reflect the domain terminology, "present" and "absent." In time period 3, the end user experienced another breakdown when trying to create conditional voice menus. In time period 4, the designer added new language components to reflect a particular enumerated type, "call forwarding" with possible values of "on" or "off." In time period 5, the end user completed her design.

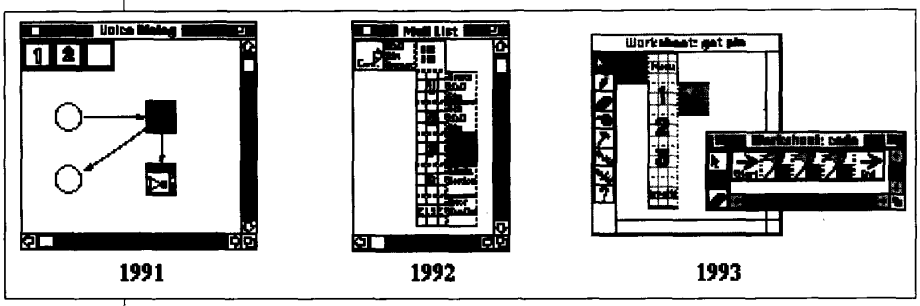

Figure 7. Evolution of the VDDE system. The first prototype consisted of six language components with no domain orientation. By the next generation shown, the three placement rules and 36 domain-oriented language components had been defined. These components were expressive, but their abstraction level was too low. In the final version shown, many of those 36 components had been combined to create higher level domain abstractions.

major expressiveness shortcoming in the current language—lack of support for user-defined, enumerated data types. The designer gained valuable information to apply when designing more general data-type mechanisms in the next iteration of the language.

Agentsheets was the collaboration medium between the end user and the designer because it supported tightly interwoven use and design sessions. Agentsheets' support for incrementally defining behavior and its tools for creating visual depictions enabled the language designer to quickly add and modify language components. Its role-specific view mechanism enabled the system to quickly switch views many times without losing context. In designer view, the end user's visual program remains on screen, and language extensions are performed and tested in the context of the user's visual program that caused the breakdown. Context preservation helps to ensure that language extensions overcome breakdowns, and it helps end users to participate in designing language extensions by situating the modification activities within their particular programming task.

Domain orientation emergence during use

The VDDE language resulted from a series of collaborative sessions over several months. The language's domain orientation emerged through repeated attempts at problem solving. In our experience, language evolution was driven by end users and designers envisioning new possibilities while directly interacting with successive versions. Figure 7 illustrates how the expressiveness and abstraction level of VDDE language components evolved over time. We examine the expressiveness and abstraction level of VDDE language components over time.

The first prototype (1991) consisted of six language components such as circles and buttons. These components could be connected by using links and, when executed, displayed a visual trace of the control flow through the network and simultaneously produced simple sounds. This simplistic node-link representation was similar to existing static design representations. By interacting with this version, end users and designers could see how the visual language paradigm could be used to create a phone-based interface design environment.

By the next generation (1992), the three rules (horizontal, vertical, and arrow) guiding language component placement were defined, based on the grid underlying the worksheet. These rules were a new way to express solutions made possible by the visual programming language environment, eliminating many links that cluttered designs in the previous version. Expressiveness also substantially increased with the enumeration of 36 domain-oriented language components representing atomic actions, such as touch-tone button presses and audio messages.

Collaborative sessions with this language version revealed that many components were not at the appropriate abstraction level. Voice dialog designers thought about their domain in terms of higher level abstractions such as voice menus and phone numbers. Thus, in the 1993 language version, many atomic components were combined to create higher level language components.

DOMAIN-ORIENTED VISUAL LANGUAGES can be valuable computational artifacts for a wide class of users who need to analyze complex, dynamic systems but are not trained in traditional computer science concepts. Creating domain-oriented visual languages requires end users familiar with the domain to collaborate with language designers familiar with the technological possibilities. Computational tools are needed to serve as a collaboration medium between these two roles during the language design process. The Agentsheets system is an effective collaboration medium providing

1. role-specific views containing tools tailored to the specific activities of end users and designers,
2. the ability to easily switch back and forth between these two views while preserving the end user's work context, and
3. tools and mechanisms that support the rapid incremental modification of the look and behavior of language components.

We have demonstrated how Agentsheets allowed us to use real-time collaborative design sessions to drive language design by analyzing experiences from a specific language project: the Voice Dialog Design Environment. Although these experiences represent events from only one case study, we see this approach as promising for fast-paced, industrial, product-development environments. Our experiences indicate that real-time collaborative language design is both useful and viable. It is useful because it provides an action-oriented methodology for simultaneously doing domain analysis, language design, and

usability testing. It is viable when coupled with proper tool support in the form of a collaboration medium such as Agentsheets. ■

Acknowledgments

We thank the members of the Center for Lifelong Learning and Design group at the University of Colorado, Gerhard Fischer, and Clayton Lewis, who contributed to the conceptual framework and the systems discussed in this article. We particularly thank Susan Davies, Josh Staller, and Mike King for their support during the VDDE project. We thank Jim Sullivan and Chris Digiano for their work on some of the applications presented here. We also thank Jonathan Ostwald, Kumiyo Nakakoji, Gerry Stahl, Jim Ambach, Loren Terveen, Francesca Iovine,and our five anonymous reviewers for their comments on various drafts of this article. This research was supported by the National Science Foundation under grant No. RED-9253425, Apple Computer Inc., and US West Advanced Technologies.

References

1. B. Nardi, *A Small Matter of Programming*, MIT Press, Cambridge, Mass., 1993.
2. C. Lewis and G.M. Olson, "Can Principles of Cognition Lower the Barriers to Programming?," *Empirical Studies of Programmers: Second Workshop*, Ablex Publishing, Norwood, N.J., 1987, pp. 248-263.
3. G. Fischer and A.C. Lemke, "Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication," *HCI*, Vol. 3, 1988, pp. 179-222.
4. G. Fischer, "Domain-Oriented Design Environments," in *Automated Software Engineering*, Kluwer Academic Publishers, Boston, Mass., 1994, pp. 177-203.
5. E.J. Golin, "Tool Review: Prograph 2.0 from TGS Systems," *J. Visual Languages and Computing*, Vol. 2, 1991, pp. 189-194.
6. D.D. Hils, "Visual Languages and Computing Survey: Dataflow Visual Programming Languages," *J. Visual Languages and Computing*, Vol. 3, 1992, pp. 69-101.
7. M. Polanyi, *The Tacit Dimension*, Doubleday, Garden City, N.Y., 1966.
8. S. Bodker and K. Gronbaek, "Design in Action: From Prototyping by Demonstration to Cooperative Prototyping," in *Design at Work: Cooperative Design of Computer Systems*, J. Greenbaum and M. Kyng, eds., Lawrence Erlbaum, Hillsdale, N.J., 1991.
9. K.H. Madsen and P.H. Aiken, "Experiences Using Cooperative Interactive Storyboard Prototyping," *Comm. ACM*, Vol. 36, June 1993, pp. 57-64.
10. A. Repenning, "Agentsheets: A Tool for Building Domain-Oriented Dynamic, Visual Environments," University of Colorado at Boulder, PhD dissertation, Dept. of Computer Science, 1993.
11. A. Repenning and T. Sumner, "Programming as Problem Solving: A Participatory Theater Approach," *Proc. Workshop on Advanced Visual Interfaces 94*, ACM Press, New York, pp. 182-191.
12. B. Laurel, *Computers as Theater*, Addison-Wesley, Reading, Mass., 1993.

Alexander Repenning *is a research assistant professor and member of the Center for Lifelong Learning and Design at the University of Colorado in Boulder. He has worked in research and development at Asea Brown Boveri, Xerox PARC, and Hewlett-Packard. Repenning has also been a consultant for Apple Computer. His research interests include education and computers, end-user programming, interactive learning and simulation environments, human-computer interaction, and artificial intelligence. He received a PhD in computer science and the certificate of cognitive science from the University of Colorado in 1993. Repenning is a member of ACM (SIGCHI) and IEEE.*

Tamara Sumner *is a research assistant and member of the Center for Lifelong Learning and Design at the University of Colorado in Boulder. She is working toward a PhD in the field of human-computer interaction. Prior to joining the University of Colorado, she worked in research and development at Hewlett-Packard. Her research interests include design processes and design environments, human-computer interaction, and interactive learning and simulation environments. Sumner received an MS in computer science from the University of Colorado in 1992 and a BS from the University of California at Santa Cruz in 1982.*

Readers can contact the authors at the Department of Computer Science and the Center for Lifelong Learning and Design, University of Colorado, Boulder, CO 80309-0430. Repenning's e-mail is ralex@cs.colorado.edu; Sumner's is sumner@cs.colorado.edu. The World Wide Web home page for the Center for Lifelong Learning and Design can be accessed at http://www.cs.colorado.edu/~hcc/.