Prototyping a Process-Centered Environment

Debra J. Richardson
Stephanie Leif Aha
Harry E. Yessayan
Leon J. Osterweil

April 23, 1990

Department of Information and Computer Science
University of California
Irvine, CA 92717

Technical Report 90-28

Abstract

This paper describes an experimental system developed and used as a vehicle for prototyping the Arcadia-1 software development environment. Prototyping is viewed as a knowledge acquisition process and is used to reduce risks in software development by gaining rapid feedback about the suitability of a production system before the system is completed. Prototyping a software development environment is particularly important due to the lack of experience with them. There is an acute need to acquire knowledge about user interaction requirements for software environments. These needs are especially important for the Arcadia project, as it is one of the first attempts to construct a process-centered environment. Our prototyping effort addresses questions about effective interaction with a process-centered environment by simulating how Arcadia-1 would interact with users in a representative range of usage scenarios. We built a prototyping system, called PRODUCER, and used it to generate a variety of prototypes simulating user interactions with Arcadia-1 process programs.

Experience with PRODUCER indicates that our approach is effective at risk reduction. The prototypes greatly improved communication with our customer. They confirmed some of our design decisions but also redirected our research efforts as a result of unexpected insight. We also found that prototyping usage scenarios provides conceptual guides and design information for process programmers. Most of the benefits of our prototyping effort derive from developing and interacting with usage scenarios, so our approach is generalizable to other prototyping systems. This paper reports on our prototyping approach and our experience in prototyping a process-centered environment.

Supporting Agencies

This work was supported in part by the National Science Foundation under grant CCR-8996102, with cooperation from the Defense Advanced Research Projects Agency (ARPA Orders 6100, Program Code 7T10) and by the National Science Foundation under grant CCR-8521398-04.

Trademarks

OpenLook is a trademark of AT&T Bell Laboratories.
Sun and XView are trademarks of Sun Microsystems, Inc.
The X Window System is a trademark of MIT.
Unix is a registered trademark of AT&T Bell Laboratories.
SuperCard and SuperTalk are trademarks of Silicon Beach Software, Inc.
HyperCard and Macintosh are registered trademarks of Apple Computer, Inc.
NeXT and NeXTStep are trademarks of NeXT, Inc.

1 Introduction

This paper describes an experimental system developed and used as a vehicle for prototyping the Arcadia-1 software development environment. Prototyping is used to reduce risks in software development by gaining rapid feedback from users, customers and designers about the suitability of a production system before the system is completed. It is rarely possible to define firm software requirements that will remain unchanged throughout the software development and usage cycle. Indeed many observers have noted that users' experiences with a system often change their ways and perceptions of doing their jobs, thereby altering the requirements for the system itself. This is particularly true for systems that are innovative or being developed to address problems in a new domain. In these cases, it is particularly useful to create prototypes to portray accurate impressions of the eventual system. Such prototypes can be used by customers and prospective users in evaluating both existing requirements and proposed changes to the requirements [Mou90]. This evaluation may identify the need for significant changes in requirements before coding and design are completed (or begun), leading to significant savings due to the avoidance of rework.

Prototyping a software development environment is particularly important. Environments are relatively new types of software systems and few software developers have had significant firsthand experience with them. It is unrealistic to expect to definitively establish a firm set of requirements for an environment at the beginning of developing the environment. Indeed, it should be expected that continuing contact with an environment will continually migrate user perceptions of its requirements. Because software environments are such large and expensive software products, prototyping should be used whenever possible to help stabilize requirements, thereby reducing effort and cost of rework.

While the preceding rationale for prototyping seems compelling and is widely accepted, it is far less clear how to prototype effectively. There is virtually no limit to the amount and variety of feedback one might wish to have on a proposed system. Thus quite a large and complex prototype might be built to project a faithful image of the eventual system. If the prototype is very large, the cost of its own development might approach the cost of the final system, negating the rationale for prototyping. Indeed, there is a considerable history of aborting the development of production systems in favor of converting elaborate prototypes into production systems. There is considerable risk in doing this. Prototypes are usually built with less care and attention to design than is appropriate for full scale system development. Thus, prototypes often suffer from such ills as poor modularity, inefficiency, and lack of robustness. These may be acceptable in prototypes but are disastrous in production systems.

In recognition of these problems, efforts are currently underway to create prototyping support systems (e.g. see [Bal89]) that are intended to facilitate rapid development of inexpensive prototypes that can be used to explore underlying requirements and be readily migrated into high quality production systems. Our project was undertaken very much in this spirit. We have developed a technology to support cost effective experimentation with software environment prototypes. This technology has been used to prototype the Arcadia-1

environment. The technology exploits certain characteristics of process-centered software environments, such as Arcadia-1, to smooth the migration path from prototype to high quality production system.

1.1 General Approach

Prototyping should be viewed as a knowledge acquisition process. Prototype system development is software development where the requirement is effective acquisition of particular knowledge or information that has been identified beforehand. We believe that many prototyping activities have gone awry because they fail to identify knowledge acquisition objectives clearly, leading to increasingly ambitious and uncontrolled prototype development. If, on the other hand, the goals and requirements for a prototype have been identified clearly at the outset, it is possible to develop a prototype to address those goals directly and effectively.

Historically, people have used prototypes to acquire a broad range of understanding. Many prototypes have been aimed at understanding design alternatives (e.g., helping to evaluate relative merits of competing system architectures and modularization). Many others have been aimed at understanding system requirements — especially performance requirements and user interface requirements.

There is an acute need to acquire knowledge about user interface requirements for complex, innovative systems, such as software environments. These needs are particularly pressing for the Arcadia project [TBC+88]. Arcadia is one of the first attempts to construct a process-centered environment [Ost87]. In such environments, the software development process to be followed is expressed in terms of actual executable code and the environment is charged with executing the code to invoke tools proactively and assign tasks to humans to carry out the process. If this is to be done successfully, the system must have a great deal of understanding about the ways in which humans will interact with the environment effectively. This knowledge ranges from low level details about the appearances of windows and menus to more difficult questions about how humans receive work assignments from the environment, how they negotiate flexibilities and support in carrying them out, and how they feel about interacting with an environment in this way. These questions are being addressed through interactions with a prototype of Arcadia-1 that simulates how Arcadia-1 would interact with users in a representative range of usage scenarios. Thus, the need to contrive realistic interaction scenarios easily, driven by realistic process programs, was taken as the basic requirement for a prototype generation system. This system, called PRODUCER, was designed, implemented, and used to generate a variety of prototypes simulating human interactions with Arcadia-1 process programs.

As shall be described subsequently, our experiments with PRODUCER indicate that this approach is effective in risk reduction. The scenarios that we constructed helped communicate goals and directions among project members and customers. They confirmed some of our tentative design directions but also helped us to identify some unexpected problems.

One major unexpected benefit was identifying the need for sharply accelerated research efforts on process visualization. We expected that the availability of an explicit process

representation would improve user effectiveness through clarification of the user's role in the process. We expected that displaying process code would provide that benefit. We found that far more powerful process visualizations are necessary. This has led to a significant new Arcadia research thrust.

Another major benefit was the realization that scenario writing is an important step in the process program development process. We found that developers faced with creating process programs often had trouble getting started. Our experience shows that developing scenarios can be an effective way to focus thoughts and begin process program development.

In fact, we also found that scenarios prototyped through the use of PRODUCER include module interfaces, design and code useful in the production of the final process programs. Whereas we believe that it is acceptable for a prototype's primary contribution to be knowledge about requirements, it is clearly beneficial when prototypes also contribute modules to the final implementation. We found that PRODUCER helps achieve this secondary goal as well as the primary knowledge acquisition goal.

We achieve this secondary goal by considering scenario development to be akin to software development. Scenario development should begin with the formulation of scenario requirements (knowledge acquisition objectives), proceed to scenario design, and then on to implementation in an executable programming language. This view of scenario development and the resulting advantages derive as much from the design of PRODUCER as from the inherent nature of process-centered environments.

Finally, it is important to observe that most of the benefits of our scenario prototyping activities derive from developing and interacting with scenarios rather than from developing the PRODUCER system itself. Our decision to develop PRODUCER was a pragmatic one. The benefits of scenario prototyping are not specific to this tool but are generalizable to other tools supporting this approach.

These conclusions will be explored and motivated in detail in the latter sections of this paper. The next section discusses other options for environment prototyping and why we developed our own prototyping system. Section 3 discusses the process of using PRODUCER and provides a detailed description of PRODUCER, while Section 4 illustrates its use. In Section 5, we describe our experience with PRODUCER. In conclusion, we summarize the work.

2 Options for Prototyping an Environment

Before deciding to build our own prototyping system, we evaluated a number of existing tools. We hoped we would be able to use one of these tools in our prototyping effort. After comparing the capabilities of all the tools to our requirements, however, we recognized deficiencies in each and decided to build our own system. Looking closely at existing prototyping tools was worthwhile, because it provided ideas about useful capabilities for PRODUCER.

2.1 Requirements

We identified the following basic requirements for our prototyping system. PRODUCER should provide:

- maximum flexibility of screen layout and of graphical representation of objects within specified guidelines;
- quick turn-around time in the evolution of a prototype, allowing the prototyper to easily incorporate changes and make enhancements;
- low entry barrier for novice programmers/prototypers;
- facilities for representing a standard look and feel such as Open Look [Mic89], which we have adopted as the user interface guidelines for the Arcadia-1 environment;
- capabilities for incorporating bitmaps created on any medium;
- capabilities for simultaneously utilizing multiple displays;
- capabilities for activating external software processes, independent of the prototype, so that actual software environment tools implemented elsewhere could be incrementally included with the prototype as part of the prototype development;
- extensibility such that more support for generating prototypes could be added as needed.

2.2 Available Prototyping Tools

We evaluated several prototyping tools and systems, most notably SuperCard for the Macintosh, Interface Builder for the NeXT, and GUIDE for the Sun workstation. After evaluating these tools, we decided that building our own prototyping system would provide the greatest amount of flexibility, an advantage that might be critically important in the evolution and enhancement of the prototypes. Here, we describe the features of these products and our evaluation.

2.2.1 SuperCard

SuperCard for the Macintosh [Sil89], which is based on Apple's HyperCard program, has many features that make it an appropriate tool for building prototypes. SuperCard provides a full complement of drawing primitives that are useful for designing windows containing collections of graphical objects. It operates in a window-based environment that includes easily customizable, standard user interface mechanisms such as buttons, menus, and text fields. The scripting language, SuperTalk, manipulates objects and data in the SuperCard environment on an event-driven basis. Because SuperTalk was designed as an intuitive, English-like language, a novice could easily create objects through the use of a graphic editor and could

associate operations with these objects. SuperTalk scripts can do three fundamental things: perform actions, get information, and change properties or contents of objects. A script is executed by an event usually provided by the user. A SuperCard programmer combines objects and scripts to create "projects", which are executable within the SuperCard run time environment.

2.2.2 Interface Builder

NeXT's Interface Builder [NeX89] is centered around a graphic editor that provides a collection of user interface building blocks and the capability for connecting graphical objects so they can communicate with one another. Construction and modification of the window layout, buttons, and menus is considerably facilitated by support for selecting and dragging graphical objects. Interface Builder provides an even richer set of predefined graphical objects than SuperCard. The programmer constructs a complete interface by selecting objects from a palette, placing and sizing them with the mouse, and defining appropriate object attributes through graphically oriented dialogs. The programmer uses facilities in Interface Builder to define relationships between the graphical objects. Interface Builder is a part of the NextStep programming environment which also includes the Objective-C based Application Kit. Objective-C and the Application Kit can be used further to link application objects and complete the application program. Applications created through Interface Builder also have access to the capabilities of the Mach operating system.

2.2.3 GUIDE

GUIDE [Sun89], an interface building tool which uses the XView toolkit on Sun workstations, seems to bring many features of NeXT's Interface Builder to the Sun platform. Like Interface Builder, GUIDE provides a graphic editor for the design and construction of the user interface portion of a program. The interface can be tested independently of the application code within the GUIDE environment. GUIDE takes advantage of the XView toolkit which, in turn, utilizes the X library and the X Window system for lower level graphical and windowing support. GUIDE supports the Open Look user interface guidelines as the standard look and feel for its graphical specifications.

2.3 Evaluation

These three prototyping tools meet some of our requirements better than others. All three systems provide a great degree of flexibility in manipulating the graphical objects on the screen. Because screen layouts are maintained as collections of graphical objects, modification of the screen layouts can be done quickly. The object oriented nature of collections of graphical objects supports the representation of a standard look and feel by creating object classes for each window style and inheriting the look and feel for actual windows. All of these systems have a fairly low user entry barrier because they are graphically oriented systems.

In many ways, SuperCard is an ideal tool for providing support for early stages of prototyping [Mou90]; the graphical manipulation facilities and the scripting language allow easy transformation of rough ideas, in the form of sketches and simple scenarios describing the behavior of the software, to an executing prototype. In later stages of prototype evolution, however, more powerful facilities are essential for environment prototyping. In particular, the scripting language supported by SuperCard is inadequate for meeting the needs of a process-centered environment. The data typing capabilities in SuperTalk are not sufficient for describing the software artifacts and relationships between them, both of which are created by software processes, as first class objects. Since we are prototyping software processes, we found it important to ensure that the prototyping system support the expressiveness of process programming languages such as APPL/A [Sut89].

Although GUIDE and NeXTStep provide more powerful language capabilities, their dependence on C or Objective-C, respectively, as the native language for programming introduces other problems. Specifically, the need to integrate tools and processes expected to inhabit the final environment is difficult to address, since many of our tools and processes are Ada programs. Although we have mechanisms within Arcadia-1 to bridge the Ada/C interface [MS89], dependence on such an interface at an early stage causes unnecessary overhead and complicates the design of the prototyping system. This problem is even more paramount for SuperCard, whose scripting language is more difficult to interface with.

We also found that having the capability to display windows on multiple workstations was critical. Although multi-display capabilities are technically feasible with Macintosh and NeXT systems, the networking display capabilities available through X Windows, which works on a client/server basis, are more adaptable to our requirements. Moreover, X Windows has been widely used on networks of workstations and can be expected to be more reliable. Because of its reliance on the X Window system, GUIDE is a more suitable prototyping support tool for our multiple display requirement.

An effective prototype should emulate the look and feel of the actual product as closely as possible. Arcadia-1 is targeted to use Open Look user interface guidelines and be built on a platform of networked workstations. SuperCard and Interface Builder are hardware dependent programs, requiring Macintosh and NeXT computers, respectively; effort must be put into following the Open Look guidelines on these machines. GUIDE was developed for the Sun workstations and its default window style is Open Look. Thus, GUIDE is more closely aligned with the look and feel we intend for our final product.

Many of the features of GUIDE are useful for our purposes, especially GUIDE's use of Open Look and X Windows. Having the capabilities of automatically constructing an interface with the same look and feel of the actual product would not only speed up the prototyping activity but would facilitate reuse of the prototyped interface for preliminary versions of the actual product. GUIDE, however, had not yet been released at the time of developing our prototyping system. Once GUIDE is more readily available, it might behoove us to use it as our standard means of drawing windows, but continue to employ PRODUCER to prototype the script actions.

The inadequacies of SuperCard and Interface Builder and GUIDE's unavailability brought us to the decision to design and build our own prototyping system. We strove for a system that had minimal functionality in the beginning, that could be built on a Unix workstation platform, that would not be hardware dependent, and that could be enhanced as needed. This enabled us to rapidly construct rough prototypes focusing mostly on the user interaction with processes, without investing too much effort in the prototyping process and without sacrificing an open-ended, upward migration path. PRODUCER was designed so that it could be incrementally enchanced to a system that would integrate actual parts of the Arcadia-1 product.

3 The PRODUCER System

We call our system "PRODUCER" because it provides the means to make a "moving picture" of the Arcadia-1 process-centered environment. We refer to the code for a scenario as the scenario *script*. Each script created through the PRODUCER system contains a collection of windows that portray using the software environment. A script also defines the actions that should occur in response to user inputs, thereby controlling when, where, and how the windows are displayed. The scenario script is accompanied by English language narrative, referred to as the *commentary*. The commentary describes the scenario and provides the reasoning and rationale behind the scenario. The commentary also guides a specific script execution, which is referred to as a *screenplay*. By showing specific screenplays accompanied by commentary, our approach to scenario development facilitates knowledge acquisition.

PRODUCER serves three types of users:

Audience: views screenplays (the audience only watches a screenplay, but does not directly guide the flow of control);

Director: controls a given screenplay by directing the action with the mouse and keyboard (the commentary tells the director what a given window represents, what data is passed to that window, and the response to each possible window action);

Scriptwriter: writes the commentary and script.

Figure 1 describes the scenario development process, which is similar to the requirements, design, and implementation processes for producing software. The scenario requirements should define the process to be prototyped and the objectives of portraying that process. The design shows how the scriptwriter plans to satisfy the requirements and should describe the progression of windows and the look and feel of the screenplay. The implementation is a script written using facilities provided by PRODUCER. Note that this is an iterative process, both within and between phases. Screenplay execution or demonstration may prompt changes in requirements or design. This process, specifically those portions that interact with PRODUCER, will be described further below.

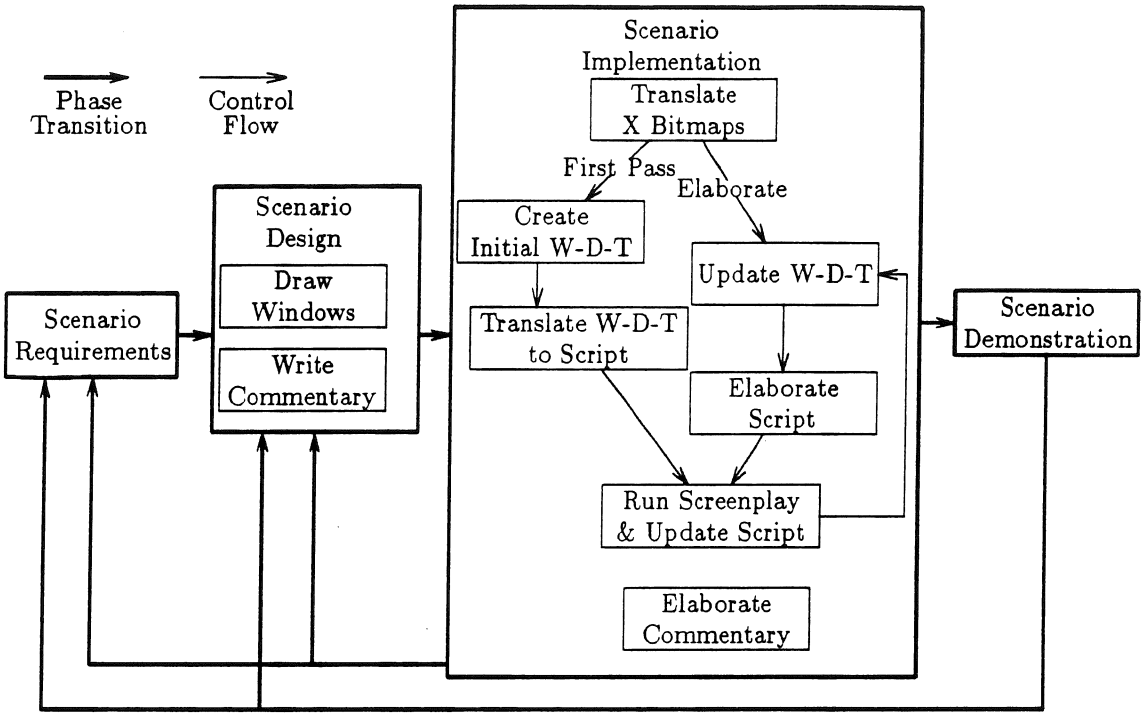

Figure 1: Scenario Development Process

3.1 Writing a Script

The scriptwriter is, for the most part, the scenario developer, although just like customers and users have a part in analyzing and modifying software requirements and design, so too might a director or audience have input to scenario requirements and design.

Scenario design consists mostly of developing an initial commentary for the scenario and obtaining windows corresponding to the tools invoked in the scenario. Windows are drawn using standard drawing tools or clipped from executing tools. These tools might be provided on the same platform as PRODUCER or any other platform.

Scenario implementation consists of creating and elaborating the script as well as elaborating the commentary where needed. The scenario script consists of windows that have been translated to X bitmap format and actions in response to user input. We provide translators from several common window formats into X bitmap format. The PRODUCER system includes capabilities to create an initial script and then interactively elaborate the script while it executes. The commentary explains what different options are present in each window and where the data in a window came from. Execution of the screenplay provides feedback, which may lead to modification of the windows and script.

Most script actions are window updates (e.g., remove and raise) and control flow between windows (they may also be process activations, but this is generally only done in highly refined scenarios). The control flow can be viewed as a function: $window \times button \times state \rightarrow window$, where *window* is the name of a window, *button* is the name of a button within that window, and *state* is the internal state of the screenplay.

Scripts include a *window-definition-table (W-D-T)*, which consists of the following fields:

key: the name of a window to be used within the PRODUCER system;

file name: the name of the file on disk that contains the bitmap representation of the window;

window location: the (x,y) screen coordinates for this window;

parent window: enclosing window for a button;

display-id: the identification of the display on which this window should appear.

Scripts are initially derived from the window-definition-table. As a first pass, the screenwriter need not worry about window locations on the screen (she or he might approximate locations or assign all windows to the origin) and should define a basic window sequence. PRODUCER automatically translates this window-definition-table into a simple script, which has linear control flow — that is, a mouse click anywhere in a window would cause display of the next window in the sequence. Most scenario requirements, however, call for screenplays that do not simply step through the windows. To create these more sophisticated screenplays, a scriptwriter elaborates the script.

PRODUCER provides some facilities for elaborating a script interactively while executing the screenplay. On the first pass, the script with linear control flow can be executed. The

scriptwriter can reposition a window on the screen and save its new coordinates in the window-definition-table (move is actually a director function, see section 3.4). On the next execution of this screenplay, the window will appear in the new location. The scriptwriter can also define buttons, adding them to the table so that actions can be assigned to their selection.

The scriptwriter may also obtain new windows (by drawing or clipping) to modify the script. These windows must be translated to X bitmaps and the information added to the window-definition-table.

Screenplay execution is organized around an action table of stimulus-response pairs. Each stimulus is a user event, and each action is one or more PRODUCER Virtual Machine (PVM) instructions (see section 3.2). PRODUCER dispatches the action in response to a user event as specified in the action table. Initially, the action table simply indicates a sequence of window displays. The screenwriter elaborates the script by adding stimulus-response pairs to the action table as well as by adding windows and buttons.

3.2 PRODUCER Virtual Machine

A script should be viewed as executable code written in terms of a virtual machine instruction set. The actions of the script are virtual machine instructions so that the same level of functionality is provided to all screenplays, X dependent code is hidden, and the migration path from a screenplay to a system is as straight-forward as possible.

The PVM instructions are

display window: displays the window associated with a given key;

remove window: removes the window associated with a given key;

raise window: redraws a window so that it is in front of any overlapping windows on the screen;

clear all: removes all windows;

display message: displays text in the message window;

wait: delays for a given amount of time;

fork a process: starts any given process;

spawn a process: the same as fork a process except that spawning a process forces the screenplay to wait for a return status from the child process;

deactivate: clears any changes made to the appearance of a window, and is used to simulate user interaction.

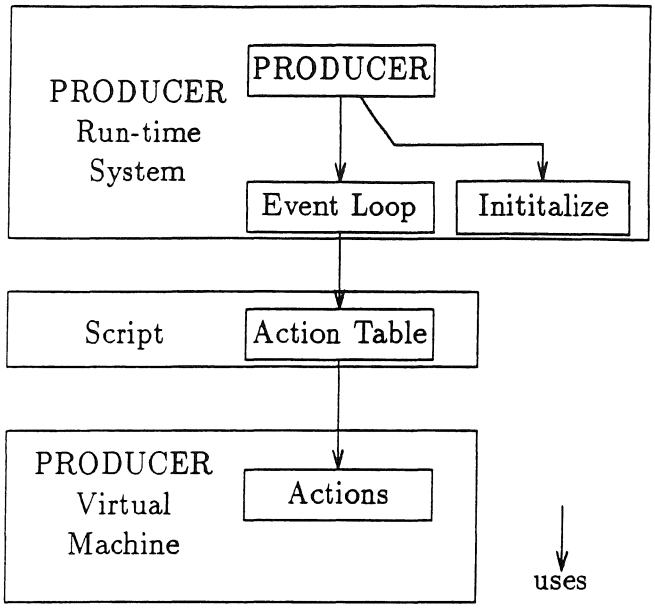

Figure 2: Screenplay Architecture

3.3 Screenplay Architecture

The PRODUCER runtime environment, which supports screenplay execution, is based on a main event loop that detects mouse button presses and performs actions as specified in the action table by the scriptwriter.

Figure 2 shows a "uses" hierarchy for a screenplay. Each outer box is a major functional component, described by the label on the left of the box. PRODUCER interprets scripts as screenplays. We do not discuss the actual process of interpretation here, only the architecture of the screenplays. A screenplay is an Ada program. The Ada code can be broken down into four different components:

initialization: sets up the displays and assigns attributes to windows and buttons;

event loop: calls the action table to interpret each user event;

action table: specifies the actions in response to each user event;

actions: provide Ada implementation of the virtual machine instructions.

The screenplay architecture was designed so the scriptwriter need only provide some of the initialization attributes and the action table.

3.4 Directing a Screenplay

The director controls a screenplay execution by selecting window buttons with the mouse. The window buttons stimulate actions as specified in the action table. Besides selecting window buttons, the following functions are also available to the director:

quit: terminates screenplay, removing all screenplay windows from all displays;

restart: clears all displays and brings back the welcome window;

move: moves a window to any position on the screen and updates window-definition-table;

clip button: clips a button, associates it with enclosing window, and updates the window-definition-table;

raise: redraws a window so that it is in front of any overlapping windows on the screen;

remove: deletes a window from the screen.

A scenario script may produce many possible screenplays depending on the director's choice of events.

4 Example Scenario

In this section, we show artifacts of developing a scenario through the process given in Figure 1. We refer to this scenario as FIB/FAB or FInd-a-Bug/Fix-A-Bug. FIB/FAB is the first scenario that we developed with PRODUCER. The main objective of FIB/FAB is to illustrate the testing, analysis and debugging tools that are being developed as a major effort within the Arcadia project. Other objectives are to portray tool integration capabilities provided by the Arcadia-1 environment, the proactive nature of the environment, and the ability to trigger activities when stored relations (possibly relating objects created by different tools) becomes inconsistent. The portion of the screenplay shown here is during test execution and after an inconsistency between execution results and a module specification is detected. The user employs debugging tools to locate the fault and modifies the source code. The process automatically begins update analysis. The artifacts shown here include screen shots containing sample windows, a portion of the window-definition-table, and parts of the corresponding script and commentary.

4.1 Windows

Figures 3 and 4 show two screen shots of a FIB/FAB screenplay. Both are of the screen that corresponds to the user interaction with the tools and the FIB/FAB process. In Figure 3, The Ada Source Browser is the active window. The DEBUS (DEsign BUilding System) window shows that execution for the test case, which is shown in the Test Execution window,

is inconsistent with a module specification, which was part of a DEBUS design. The user selects Debug and then chooses the Information Flow Analyzer to locate the source of the inconsistency. The Information Flow Analyzer highlights the dependencies that lead up to the inconsistent value in the Ada Source Browser window. Figure 4 shows that the user has completed modifying the source code. The Debug and Information Flow windows are removed and the FIB/FAB process automatically begins update analysis.

4.2 Window-Definition-Table

Recall that the scriptwriter creates a simple window-definition-table, which doesn't worry about window coordinates or buttons, before writing the script. Figure 5 shows part of the initial window-definition-table, which includes some of the windows shown in Figures 3 and 4.

This window-definition-table is translated into a simple script, which can be executed. During the screenplay execution, the windows are moved, buttons are defined, and the window-definition-table is updated. We also added process code bitmaps to be displayed on the second screen. Part of the updated window-definition-table is also shown in Figure 5.

4.3 The Script

The script generated from the window-definition-table consists only of linear control flow. We next elaborated the script by adding more complex control flow to the actions. Each action can be any combination of PVM instructions.

The script shown in Figure 6 contains a portion of the refined script, which is grouped into four sections. The first section shows the test execution process code and the detection of the DEBUS inconsistency. The second section shows responses to selecting the Debug button in the DEBUS window and to buttons in the Debug window. Note here that actions are not assigned to all buttons; this illustrates one form of incomplete prototyping allowed by PRODUCER. The third section shows the response to selecting the browse button in the Information Flow Analyzer window (Figure 3) and to clicking on the Ada Source Browser window, where the fault is corrected magically in screenplay. The fourth section shows the actions that result from clicking on the Complete button in the Fix/Edit window (Figure 4); the process activates update analysis and compilation.

4.4 Commentary

The commentary is developed in parallel with the script. It is elaborated as extra windows are added and buttons are defined. The portion of commentary presented in Figure 7 refers to the windows and script discussed above.

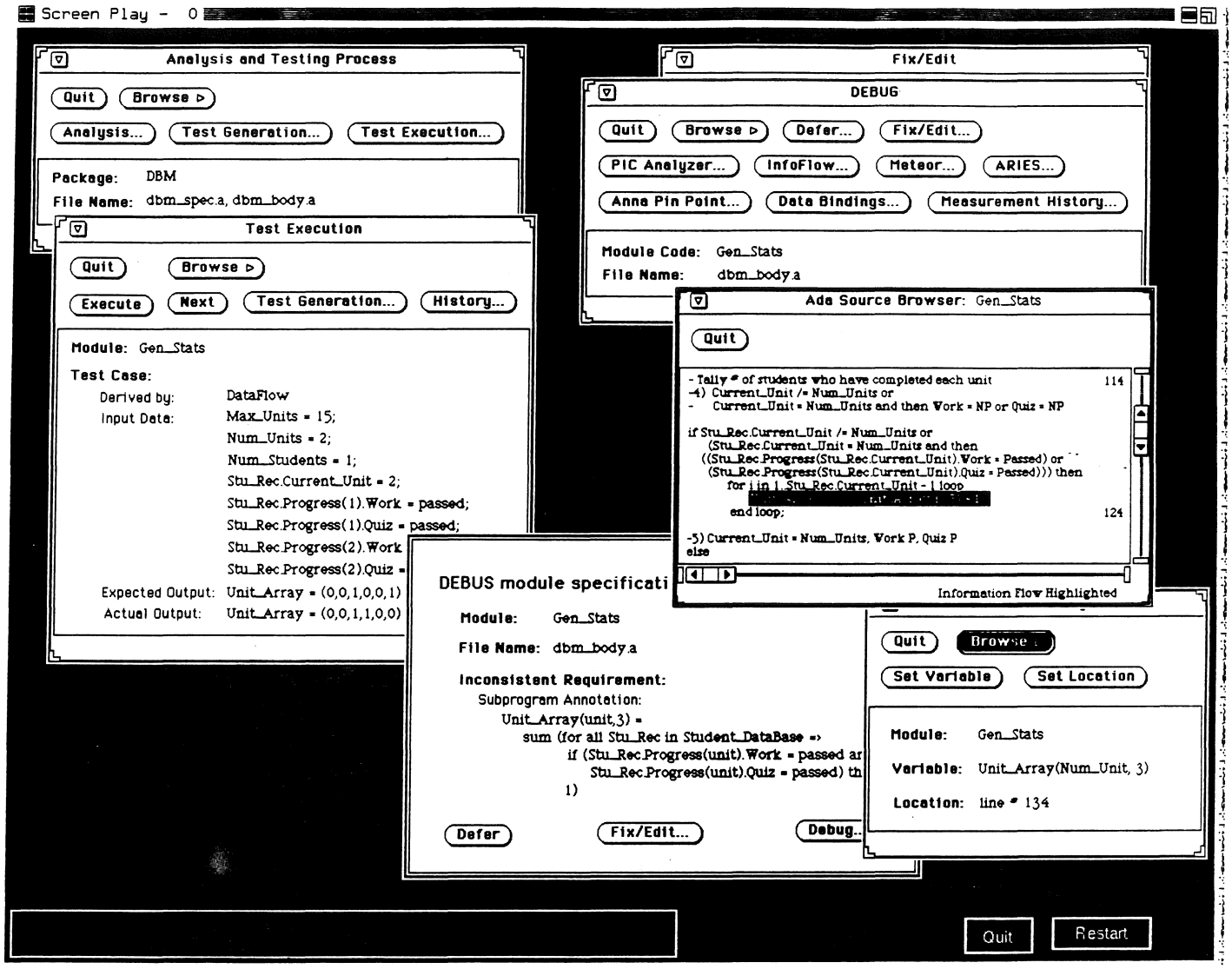

Figure 3: Sample Screen Shots and Windows

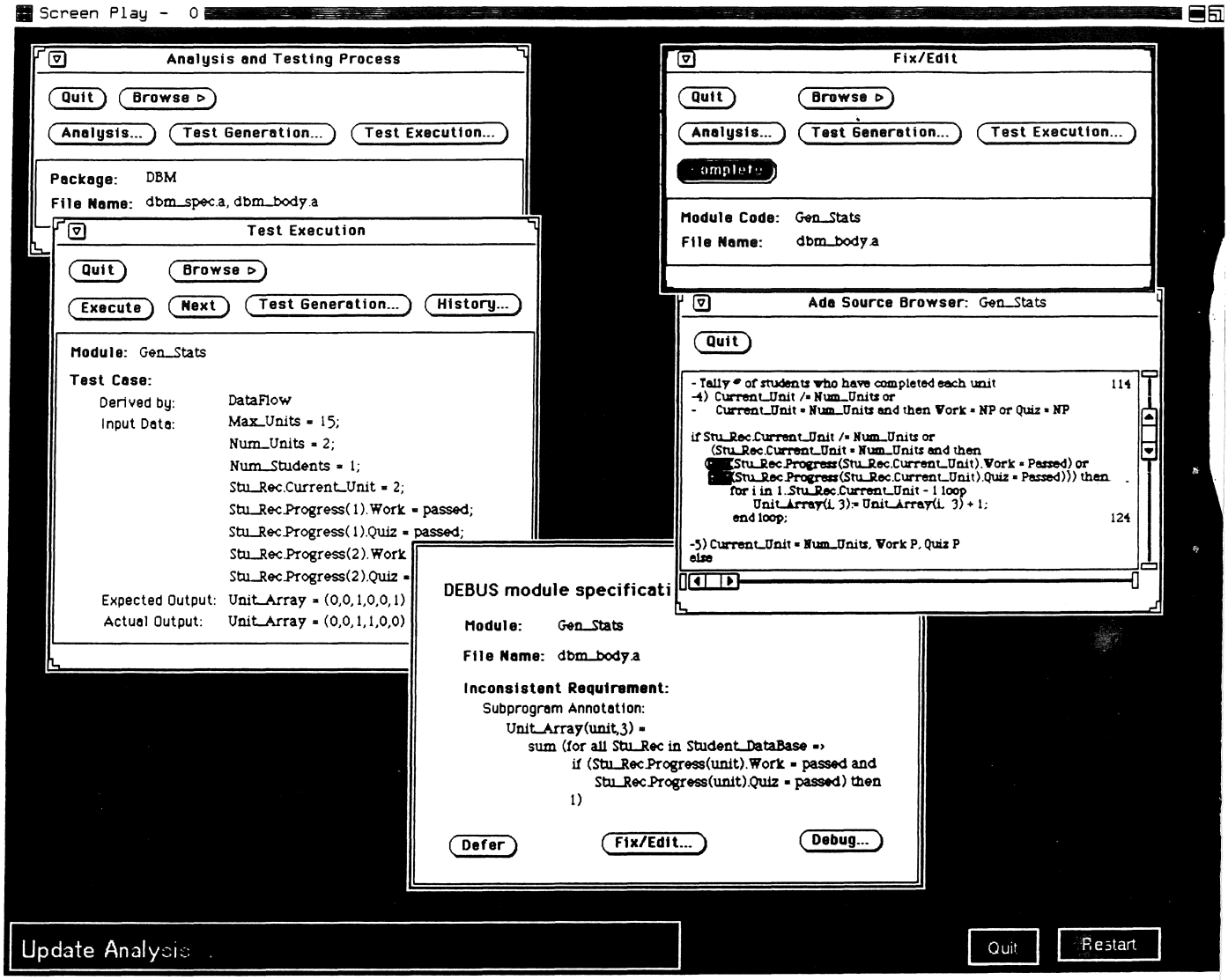

Figure 4: Sample Screen Shots and Windows

Initial Window-Definition-Table				
Key	File Name	X	Y	Display
Analysis_Testing	"analysis-testing"	0	0	0
Test_Execution	"test-exec"	0	0	0
Debus	"debus"	0	0	0
Fix_Edit	"fix-edit"	0	0	0
Debug	"debug"	0	0	0
Ada_Source	"ada-source"	0	0	0
Info_Flow	"info-flow"	0	0	0

Expanded Window-Definition-Table				
Key	File Name	X	Y	Display
Analysis_Testing	"analysis-testing"	15	15	0
Test_Execution	"test-exec"	35	170	0
Debus	"debus"	360	460	0
Fix_Edit	"fix-edit"	586	15	0
Debug	"debug"	515	45	0
Aries_Debug	"aries-debug"	105	45	0
Data_Bindings	"data-bindings"	620	510	0
Debug_Ada_Source	"debug-ada-source"	600	235	0
Ada_Source	"ada-source"	600	235	0
Info_Flow	"info-flow"	775	510	0
Info_Flow_Ada_Source	"info-flow-ada-source"	600	235	0
Cesar_Ada_Source	"cesar-ada-source"	600	235	0
A_T_Browse_Button	"a-t-browse.b"	74	36	0
A_T_Analysis_Button	"a-t-analysis.b"	14	69	0
A_T_Test_Generation_Button	"a-t-test-generation.b"	123	69	0
A_T_Test_Execution_Button	"a-t-test-execution.b"	286	69	0
Debug_Info_Flow_Button	"debug-info-flow.b"	155	69	0
Info_Flow_Browse_Button	"info-flow-browse.b"	84	37	0
Debug_Aries_Button	"debug-aries.b"	364	69	0
Debug_Data_Bindings_Button	"debug-data-bindings.b"	166	102	0
Debug_Quit_Button	"debug-quit.b"	870	810	0
Hierarchy_Button	"hierarchy.b"	227	69	0
p_Assign	"assign.p"	20	20	1
p_TestCase	"testcase.p"	20	20	1
p_RPC	"RPC.p"	20	20	1
p_If_Oracle	"if-oracle.p"	20	20	1
p_If_Rebus	"if-rebus.p"	20	20	1
p_If_Debus	"if-debus.p"	20	20	1
p_Debus	"debus.p"	20	20	1

Figure 5: Sample (partial) Window-Definition-Tables

```
when Quit => Done := TRUE;
-- the director's quit the screenplay action.
--** Group 1: Test Execution
when Test_Execution_Button =>
-- show the process code for checking the execution of a test on screen 2
   Display_Bitmap (Screen_List(p_Assign), Display_Status);    Wait (1.0);
   Display_Bitmap (Screen_List(p_TestCase), Display_Status); Wait (1.0);
   Display_Bitmap (Screen_List(p_RPC), Display_Status);      Wait (1.0);
   Display_Bitmap (Screen_List(p_If_Oracle), Display_Status); Wait (1.0);
   Display_Bitmap (Screen_List(p_If_Rebus), Display_Status); Wait (1.0);
   Display_Bitmap (Screen_List(p_If_Debus), Display_Status); Wait (1.0);
-- detect Debus error, show the error and related windows on screen 1
   Display_Bitmap (Screen_List(p_Debus), Display_Status);
   Display_Bitmap (Screen_List(Debus), Display_Status);
-- unhighlight the execution button because execution has stopped
   Deactivate (Screen_List(a_t_test_execution_button));
--** end Group 1
--** Group 2: Debug
when DEBUS_Debug_Button =>
-- show the debug tool
   Display_Bitmap (Screen_List(Debug), Display_Status);
-- The user can debug using any of the tools below
when Debug_PIC_Button =>
   Display_Bitmap (Screen_List(PIC), Display_Status);
when Debug_Info_Flow_Button =>
   Display_Bitmap (Screen_List(Info_Flow), Display_Status);
when Debug_Aries_Button =>
   Display_Bitmap (Screen_List(Aries_Debug), Display_Status);
when Debug_Data_Bindings_Button =>
   Display_Bitmap (Screen_List(Data_Bindings), Display_Status);
when Debug_Quit_Button =>
   Remove_Bitmap (Screen_List(Debug));
   Remove_Bitmap (Screen_List(Debug_Ada_Source));
--** end Group 2
--** Group 3: Info Flow
-- the user chose to invoke the info flow tool
when Info_Flow_Browse_Button =>
-- display the slice highlighted by the info flow tool
   Display_Bitmap (Screen_List(Info_Flow_Ada_Source), Display_Status);
   Remove_Bitmap (Screen_List(Debug_Ada_Source));
when Ada_Source_Browser =>
-- show the corrected source code
   Display_Bitmap (Screen_List(Ada_Source), Display_Status);
--** end Group 3
--** Group 4: Fix/Edit
when Fix_Edit_Complete_Button =>
-- the user has finished making corrections
   Display_Message ("Update Analysis ...");
-- remove debugging windows
   Remove_Bitmap (Screen_List(Info_Flow_Ada_Source));
   Remove_Bitmap (Screen_List(Debug));
--** end Group 4
```

Figure 6: Sample (partial) Script

Detect failure

- the relationship between a REBUS-recorded performance requirement and the module execution is inconsistent at run-time

↪ trigger REBUS Inconsistency window

- user selects **Fix/Edit**

↪ call FIX/EDIT

Debug fault(s)

- *click left on* DEBUG

↪ call DEBUG

- user's debugging preferences automatically execute in the background
- Debug window display with buttons corresponding to preferences highlighted
- user request help information by *shift click left on* **InfoFlow**

↪ help information on **InfoFlow** is overlayed

- *click left on* **InfoFlow**

↪ call INFO-FLOW-ANALYZER

- Information Flow Analyzer window display
- *click left on* **Browse**

↪ call IF-SOURCE-BROWSER

- IF Source window display highlights the slice of information flow dependencies in the Ada source
  - user scrolls through IF Source window

Fix/Edit

- user edits in IF Source window (*click left on* fault in **IF Source** modifying conditional in Gen_Stats at lines 120, 121)

↪ call FIX/EDIT

- Fix/Edit window raised

Update analysis

- *click left on* **Complete**

↪ Debug window and all spawned windows removed

↪ trigger addition of change information to UPDATE LOG

↪ trigger addition of change information to MEASUREMENT HISTORY

↪ call COMPILATION

↪ user's analysis preferences automatically execute in the background

- process code animated on other display

Figure 7: Sample (partial) Commentary

5 Our Prototyping Experience

The development of a software development environment of Arcadia-1's scale is a long process; many components must come together before we can display any part of the environment. We felt it important to provide some preview of a working Arcadia-1 environment and representative software development processes to elicit useful feedback from potential users. This motivated our prototyping effort. We did indeed receive feedback. We also generated many new ideas and in some cases redirected the Arcadia project research and development efforts.

We began our prototyping effort by developing scenarios that portrayed Arcadia-1's diverse capabilities, its process programming concept, and its look and feel. Arcadia is, on the one hand, an environment architecture project but also incorporates many tool development efforts. These tools are composed in process programs and interact through the capabilities provided by the environment infrastructure but also must present a smooth and integrated view to the product developer. Thus, scenarios were developed to highlight the capabilities available to all potential Arcadia-1 users: the environment builder, the process programmer, and the product developer.

We began our experiment with three draft scenarios that highlighted the Arcadia project. The FIB/FAB (FInd-a-Bug/Fix-A-Bug) scenario portrays the testing, analysis and debugging process and focuses on the interaction among various analysis and testing tools with the development tools. The DEBUS (DEsign BUilding System) scenario demonstrates cooperative work between multiple designers using different design methodologies on the same design effort. The AAT/AAT (Add-A-Type/Add-A-Tool) scenario shows how a process programmer or environment builder might add a new tool to an environment. These scenarios were refined throughout the experiment as PRODUCER's capabilities were expanded. Their current status is mentioned at the end of this section. The screenplays were periodically demonstrated to potential product developers, our sponsor, and other Arcadia project personnel. This interaction provided invaluable feedback on both Arcadia-1's functional capabilities and its look and feel.

Our first lessons involved discovering the minimal requirements for PRODUCER as a prototyping vehicle. We began with very simple PRODUCER requirements that allowed a prototyper to step through a sequence of bitmaps that represent tool and process windows for a scenario. Thinking that these requirements would enable us to get a prototype running quickly and that the capabilities were sufficient to portray the look and feel of Arcadia-1, we developed this simplistic PRODUCER. We then developed a simple script for each draft scenario, which provided immediate feedback to the prototyping process. We quickly recognized that such short cuts did not give a potential user a realistic view of the future Arcadia-1. For instance, our initial PRODUCER provided no functional buttons, but rather a mouse click anywhere in an active window would pop up the next window defined by the script. This was determined inadequate, so we added capabilities to define buttons that are highlighted by displaying them in reverse video when they are selected. Moreover, our initial PRODUCER enabled several windows to appear on the screen at any one time, as is required

by most scenarios, but there was no means of focusing the attention of the user. To avoid the confusion of a cluttered screen, we added a capability to highlight the currently active window. Thus, we found we needed a much more complex prototyping vehicle and went on to develop PRODUCER as described in the previous section.

Our next insight was into the look and feel to the product developer. There was concern about the tool-driven style of user interaction. Given the wide variety of tools and processes that we expect to inhabit Arcadia-1, it may be difficult for a project programmer to navigate through a process. A product developer who might not be aware of specific capabilities of a tool by its name might be better guided through Arcadia-1 in a goal-driven fashion. This lesson has prompted research into how one might provide a goal-driven user model.

Concern about the difficulty a user might have in determining precisely what role she or he is expected to play in a process also surfaced quickly. Arcadia-1 will provide a complex software development environment consisting of many tools. A software process may activate several tools at any one time. PRODUCER provides a window highlighting capability to focus the attention of the user, but it can be difficult to determine why this window is active. We identified the need for a process tracing facility that tells the user what portion of the process is active and what sequence of events led up to that point. This can be accomplished in part by viewing the process program as it is executing. We also realized that the user interaction within a process-centered environment requires some view of the process. A complementary feature would be a "you are here map", which would graphically represent the process history. Effective approaches to both facilities are under investigation.

We further recognized that all Arcadia-1 users need a view of the executing process, since Arcadia-1 is an environment with multiple types of users who interact with very different levels of the environment. Product programmers need to keep track of their progress in the development process, interact with tools, the see the state of the developing product. Process programmers need to visualize the process code and analyze the process in execution. Environment builders need a "behind the scenes" view that shows the underlying infrastructure — e.g., object management and message passing. We altered PRODUCER to enable it to display information relevant to multiple users types on separate workstation displays. By this mechanism, tool interaction appears on one display, the executing process program appears on another display, and the underlying infrastructure is displayed on yet another. The need to support each view has prompted research efforts into new visualization mechanisms for the information relevant to the various Arcadia-1 user types. We currently support the process program view simply by stepping through the process source code, which is in APPL/A, as each APPL/A statement executes. Infrastructure views are supported by graphical drawings of background activity. We realize, however, that new process and infrastructure visualization mechanisms are absolutely essential.

The need to display windows on multiple workstation displays was also arrived at for other reasons. The complexity of interactions among users of a software development environment could not be explored fully by a single display interface. Many of the scenarios we envisioned comprised multi-user processes. In addition, the vast amount of information that must be

conveyed even to a single user is inadequately handled by a single workstation display. In the course of building PRODUCER, we explored the options of creating a multi-display environment.

A welcome insight into the benefits of using PRODUCER was that creating scripts embodied the design and partial coding of Arcadia-1 process programs. Indeed, we found that developing scenario scripts parallels the development of production process programs, offering hope that software objects produced in the course of script development can be used in the final process program. PRODUCER thereby facilitates migration to a process-centered environment. This benefit results as much from the design of PRODUCER as from the inherent nature of process-centered environments. PRODUCER facilitated prototyping a scenario as a process, which can evolve to a production process program.

In our original plan, PRODUCER was to display drawings of screen shots developed with diverse other media as windows of a scenario. Some Arcadia-1 tools, however, have already been implemented and have a running user interaction. Two approaches were taken with such tools. Screen shots were taken of the tool in action and these were included in scripts. This first option allows us to provide an image of the tool in action supported by the fact that the windows are actual tool output. Another option is forking a process that invokes the tool. This latter option allows us to substitute actual tools and subprocesses for a series of windows, but maintains independence between production system code and prototype code. These two options provide a natural migration path from screenplays to running Arcadia-1 processes.

Our selection of the three initial scenarios was by no means meant to be complete. Upon showing the screenplays to potential users and our sponsors, we heard what they really wanted. They liked what they saw, but quite naturally asked questions regarding capabilities they did not see. To some questions, we could safely say the capability will be provided. Other questions, however, addressed functionality we had not intended but are now considering. For instance, no explicit process programs were planned to support reverse engineering, but this has become a new functional requirement of Arcadia-1. PRODUCER, therefore, enables better communication between Arcadia project personnel and our customers — potential Arcadia-1 users and our sponsor. Like [Mou90], we found that showing screenplays to our customers reduces the risk that their expectations will not be met by Arcadia-1.

Scenario development in our prototyping experiment was an iterative process. From the original draft scenarios, we identified basic prototyping requirements. As we developed scripts for scenarios and showed resulting screenplays to customers, we realized the potential of our activity. Each new script helped pinpoint additional features that would enhance PRODUCER's capabilities to prototype the Arcadia-1 environment and software development processes. New functionality spawned new scripts, etc. Table 1 summarizes the complexity of the current three scripts. FIB/FAB uses the multiple screen capability. One screen displays user interaction with various analysis and testing tools and the triggers resulting from relations that exist between objects created by development tools and the analysis and testing process. The other screen displays the process code as it executes. We have immediate plans

Script	windows	displays
FIB/FAB	42	2 (user & process)
DEBUS	83	3 (two users & process)
AAT/AAT	27	3 (user, process & background)

Table 1: Script Complexity

to invoke actual tools from this scenario. DEBUS also uses the multiple screen capability to show cooperation between two users working on the same product development effort. The users employ different design methodologies, but interact through the product requirements. AAT/AAT uses three screens to portray the environment from the three perspectives of product developer, process programmer, and environment builder. This scenario requires the greatest enhancements to alternative visualization mechanisms.

6 Conclusion

We believe that PRODUCER has served as a highly cost-effective risk reduction vehicle. Important confirmations of key Arcadia objectives and architectural decisions were obtained. In addition, the need for unexpected major new research initiatives (most notably in the area of visualization technology) were identified.

Our costs in doing this were relatively modest. PRODUCER and associated support tools have been developed and iteratively enhanced at a cost of somewhat less than ten person months. Useful screenplays were running within three or four months of project inception. We believe they have significantly reduced the risk that Arcadia-1 will not fulfill expectations.

In addition, we have found that the scripts themselves have been very helpful to process programmers as conceptual guides in designing the process programs that are to ultimately give substance to the scenarios. In a real sense, scenario scripts are themselves process programs. The design information they embody turns out to be reusable as process program design information as well. Scripts function as executable designs. Thus, scenario development has become an integral part of our process program development process.

Finally, we should repeat our earlier observation that scenario and script development and screenplay executions provided the primary benefits of this activity. PRODUCER was valuable primarily as a vehicle for enabling this development and execution. In that other systems might serve as similar vehicles, we would expect them to be similarly useful.

Scenario development and PRODUCER enhancements are continuing. We increasingly view this technology as being essential to effective risk reduction in software environment development.

Acknowledgments

We would like to acknowledge Billie Bozarth and Xiping Song for their invaluable feedback throughout their processes of developing scenarios and using PRODUCER. We also appreciate our audiences — our sponsor, potential Arcadia-1 users, and other Arcadia project personnel — who have watched screenplays in action and given us useful feedback.

References

[Bal89] Robert Balzer. Draft Report on Requirements for a Common Prototyping System. *ACM SIGSOFT/SIGPLAN*, 24(3), March 1989. Robert Balzer, Chairman; Richard P. Gabriel, Editor.

[Mic89] Sun Microsystems. Open Look graphical user interface functional specification release 1.0. Technical report, Sun Microsystems, May 1989.

[Mou90] S. Joy Mountford. Designers: Meet your users (panel). In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 439–442, Seattle, April 1990. Association for Computing Machinery.

[MS89] Mark Maybee and Stephen D. Sykes. Q: Towards a multi-lingual interprocess communications model. Arcadia Technical Report UCI-89-06, University of California, Irvine, February 1989.

[NeX89] NeXT, Inc., Palo Alto, CA. *Interface Builder Reference Manual*, 1989.

[Ost87] Leon J. Osterweil. Software processes are software too. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 2–13, Monterey, CA, March 1987.

[Sil89] Silicon Beach Software, San Diego, CA. *SuperCard User Manual*, 1989.

[Sun89] Sun Microsystems, Inc., Palo Alto, CA. *Open Windows Developer's GUIDE, Beta Version User's Manual*, 1989.

[Sut89] Stanley M. Sutton, Jr. Working report on the revised definition for the appl/a programming language. Arcadia Document CU-89-05, Department of Computer Science, University of Colorado, Boulder, Colorado 80309, June 1989.

[TBC+88] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michal Young. Foundations for the Arcadia environment architecture. In *Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments*, pages 1–13, Boston, November 1988. Appeared as *Sigplan Notices 24*(2) and *Software Engineering Notes 13*(5).