Creatr: A Generic Graphical Distributed Debugger with Language Support for Application Interfacing

by

Shakil A. Chunawala

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Science

and

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1994

Author .....
Department of Electrical Engineering and Computer Science
June 10, 1994

Certified by .....
John Wroclawski
Research Scientist, MIT Department of EECS

Certified by .....
Jay Thomas
Hardware Tools and CBX Hardware/Firmware Manager, ROLM

Accepted by .....
Frederic R. Morgenthaler
Chair, Department Committee on Graduate Students

Creatr: A Generic Graphical Distributed Debugger with Language Support for Application Interfacing

by

Shakil A. Chunawala

Submitted to the Department of Electrical Engineering and Computer Science
on June 10, 1994, in partial fulfillment of the
requirements for the degrees of
Master of Science
and
Bachelor of Science in Computer Science and Engineering

Abstract

High level debugging tools which can model system-level views of programs have proven very useful to developers of distributed systems. Unlike source-level debuggers, system-level debugger allow programmers to quickly gain an understanding of how the various parts of a system are interacting by displaying information at a higher granularity. Traditionally, such tools have received limited use due to target dependencies. When the application changed, existing tools were unable to handle the new context, thus requiring the development of new tools, or major rewrites of existing tools.

This thesis introduces Creatr, a tool for debugging EventFlow systems. Creatr targets a specific programming model, EventFlow, but makes limited assumptions about how various systems make use of this model. Instead, Creatr provides linguistic support to allow the user to define how a given target application maps to the general model. By providing a hook library, users can quickly interface Creatr to a wide range of applications. Creatr also provides support for user-defined data visualization and information filtering. By abstracting out target dependencies, Creatr provides a more flexible and reusable solution to system-level debugging than has been previously available.

Thesis Supervisor: John Wroclawski
Title: Research Scientist, MIT Department of EECS

Thesis Supervisor: Jay Thomas
Title: Hardware Tools and CBX Hardware/Firmware Manager, ROLM

Acknowledgments

Many people have been instrumental in the development of this thesis. Foremost, I would like to thank Ken Duda. He provided interesting insight which helped guide the development of the ideas presented in this thesis and assisted in resolving various implementation problems. Kevin Raper, Mark Clark, and Monica Seles also deserve thanks. Creatr's success is attributable to the valuable end-user input they provided. I would also like to thank my advisor, John Wroclawski for his unbelievable patience with me. Credit goes to Jerry Saltzer and Jay Thomas for acting as go-betweens when company policies were in conflict with MIT requirements.

Finally, I need to thank my family. My education would not have been possible without the moral and financial support of my parents, Jamil and Shaista. Every day, I read the paper or look at the job market and realize that they knew what they were talking about: education does what no other investment can - it reduces risk while increasing expected returns.

Contents

List of Figures

Chapter 1

Introduction

The basis of this thesis is that good debugging tools are critical to reducing software development cycles. Increasing program complexities and performance demands have forced developers to adopt new languages and programming methodologies. However, debuggers have not kept pace with this migration. This thesis studies the problem of extending debugging capabilities to meet the demands of the latest development processes. In particular, a generic debugger is introduced. This debugger addresses some of the problems associated with a recent class of debuggers: system level debuggers.

Currently there are three general classes of software debugging tools: object-level, source-level, and system-level. Object-level debuggers are used to debug programs written in a machine's native language. They have an instruction-set level control granularity. They allow users to view and/or modify the stack, registers, and memory. While a native language is useful and often required to write programs such as device drivers or to make performance optimizations specific to a given machine, it is very difficult to develop software of any complexity at this level.

High-level languages mask machine-specific dependencies and allow programmers to deal with computation issues instead. Such issues include control flow and data manipulation. Furthermore, such languages provide a barrier between the programmer and the machine, allowing for portable code development. Debuggers which provide control granularity consistent with high-level languages are often called source-level debuggers. They allow developers to single-step through source code (code written

in the high level language), view/modify variables (instead of specific memory locations), set source-level breakpoints, etc. User control of programs shifts from an object-level to a source-level viewpoint.

High-level languages increase the application domain by abstracting out certain object-level complexities. However, large software systems continue to be unmanageable when written in such languages. Recently, various programming methodologies have gained acceptance as a way to further reduce complexity and increase software manageability. Instead of developing even higher level languages (which are being explored as well), program complexities can be contained by structuring programs to follow some well understood paradigm. Examples of such methodologies include layered development, object-oriented programming, and distributed programming. While source-level debuggers remain useful in the context of such programming methodologies, tools which allow users to visualize the system-level behavior of programs have become important. Such system-level debuggers have only recently received attention in the research community and have not adequately addressed issues such as portability/reuseability and visualization.

This thesis describes a debugging tool called Creatr. Creatr allows users to debug programs which conform to the EventFlow system model. EventFlow is a high-level abstraction used to characterize systems which communicate through message-passing[1]. Creatr makes limited assumptions about how a target conforms to the EventFlow model. Thus, Creatr is more target-independent than past debuggers. Linguistic support for target interfacing and an API library make Creatr reusable, while user defined data visualization and filtering provide a powerful framework for customization.

The remainder of this thesis is structured as follows. In Chapter 2 I examine other existing system-level debuggers. Chapter 3 develops an overview of the Creatr model for system debugging and presents the goals of Creatr. Chapter 4 outlines the design of Creatr while Chapter 5 details the capabilities and syntax of Creatr's languages. In Chapter 6 I describe a particular usage scenario. Evaluation of the Creatr system is

[1] A detailed account of the EventFlow system model is presented in Chapter 3.

covered in Chapter 7 and Chapter 8 concludes this thesis with a discussion of possible future improvements.

Chapter 2

Background

This chapter elaborates on previous work in system-level debugging. While system-level debugging is a relatively new field, a significant amount of groundwork has been laid. Important instances of such debuggers are described in the first half of this chapter. The second half discusses common problems associated with these tools. This chapter concludes by outlining the differences between Creatr and existing system-level debuggers.

2.1 Related Work

2.1.1 Behavioral Abstraction

System-level debugging was first explored by Peter Bates, who developed an approach called behavioral modeling[2]. Behavioral modeling is built on an event driven programming model - events define the computational behavior of a program. Bates' approach assumes a base class of events, and allows developers to define higher level abstractions based on these primitive events. Linguistic support for abstraction definitions is provided through a language called EDL (Event Definition Language). During run-time monitoring of a system, Bates' tool will receive primitive events from the target. Based on the abstractions provided in an EDL configuration, the tool will then synthesize new events. Users can view both primitive and synthesized

events. Although this debugging system provides a powerful construct for viewing system-level events, it does not provide an interactive mechanism through which user can affect the state of the target application. Furthermore, the notion of primitive events is embedded into the tool. This requires recoding and rebuilding the tool when interfacing to different systems. Migration requires users to be intimately familiar with the tool's implementation and the target environment. Finally, viewing of events is textual.

2.1.2 Interactive Debugging

Examples of interactive system-level debuggers include IDD, Instant Replay, and DPD[9, 11, 15]. Unlike Bates' tool, these debuggers allow users to view and modify the state of software at run-time. However, flexibility in defining the event structure has been lost in such tools. Each tool has a predefined set of system events. Only the contents of these events can be modified at run-time. Generally, such tools have been targeted for distributed systems, and thus interprocess communications (IPC) have been the events of choice.

While IDD, Instant Replay, and DPD have limited target domains, they are important in other aspects. Foremost, they are interactive. IDD was also one of the first such debuggers to introduce a graphical front-end. Different processes are represented as vertically-stacked horizontal lines in a time-process graph space. IPCs are represented as connecting lines between processes, from "send" to "receive" time. IDD also allows users to intercept IPCs and modify them before completing delivery. Instant Replay allows users to record histories of events (IPC flow), and then re-execute parts of a program against the saved history. Finally, DPD incorporates the graphical capabilities of IDD and the functionality of Instant Replay.

While IDD, Instant Replay, and DPD represent three different approaches to interactive debugging, there have also been a number of other debuggers built on the same ideas[3, 5, 7, 8, 14, 16, 17, 18, 19].

2.2 Common Problems

Problems associated with past system-level debuggers can be classified into two main categories: target scope and visualization.

Most past debuggers have been developed as part of a larger system. Once a system is in place, an appropriate debugger is needed to make software development easier. As a result, past debuggers have often been limited in their scope. Bates' behavioral abstraction paradigm has been built into a specific distributed system simulation. The system, called VMT, is being used to address cooperative distributed problem solving[2]. Bates' tool can only accept a certain class of primitive events and the event injection mechanism is directly embedded into the simulation. IDD, while not embedded into any operating system, is intended to monitor process level communications. It provides a specialized library for standard Unix IPC functionality such as forking, sockets, pipes, etc.[9]. Instant Replay is built for the BBN Butterfly Parallel Processor project and has IPC monitoring embedded into the Chrysalis operating system[11]. Likewise, DPD is built for, and embedded into, the REM environment[15]. Because of the reduced target scope of these debuggers, usage within other systems entails high migration costs. The tools must be modified at a source level to communicate with the new system and then rebuilt. Hooks must be added to the new target as well.

Existing debuggers' visualization facilities have commonly been text based. Those which do provide graphical front-ends hardcode the display of information. Debuggers intended for IPC monitoring assume that messages are single-sender/single-receiver and visualize accordingly. With new communication protocols opening possibilities for other types of IPCs (such as broadcast messages and messages destined for a subset of the active processes), such debuggers cannot be used to capture and display all message flow without significant changes made to the infrastructure of the tools themselves. Furthermore, past visualization of system events has been homogeneous, regardless of the type or content of a given event. In such systems, users cannot passively distinguish events based on graphical visualization alone. To realize the

significance of an event, users need to actively view event content.

2.3 Differences in Creatr

There are several features which distinguish Creatr from other system-level debuggers.

- System-level debugging is accomplished via EventFlow modeling. The type of debugging information which is desired within a given target application must map to this model.
- Creatr does not have a predefined set of events. It will work with any set of events that conforms to the EventFlow model. Primitive events are user-defined and facilities provided for mapping them into Creatr. Consequently, event granularity is not an embedded feature.
- Graphical and textual presentation of events are available. Creatr allows users to customize both presentational forms.
- Filtering capabilities are provided at the event level. Users can filter events based on type and/or content.

Chapter 3

Goals and Overview

Creatr is a software debugging tool intended to aid programmers during code development and reduce development costs. Currently, many such tools exist. Creatr is designed to perform a specific task while addressing problems associated with existing debuggers of its class. The first half of this chapter details the high level goals of Creatr and explains why they are important. The latter half presents a system-level overview of the tool and provides context to how the goals are addressed.

3.1 Goals

Creatr has five main goals. The first goal is intended to classify the debugging functionality of Creatr, while the remaining four address common problems associated with existing debuggers.

3.1.1 System-Level Debugger

Most debuggers are event driven. Events happening in a target environment are made available to the debugging tool, which can then manage and present the information to the user. Object-level debuggers have an event granularity defined by machine instructions, while each line of source code comprises an event in source-level debuggers.

System-level debuggers have an event granularity defined by the system model that

is assumed by the tool. Examples of system models include structured programming, object-oriented programming, and distributed programming. A tool which assumes a structured model may have an event granularity of procedure calls. Such a tool could allow users to visualize the run-time call graph of a program and ensure that the correct procedures are being called at the right time. An object-oriented debugger may allow users to view the creation/destruction of objects, composition of objects, and method calls. Finally, a distributed debugger assumes multiple, co-existing processes. In this model, users are interested in seeing the interactions between the different processes. Is the message flow among processes happening in the correct time sequence? Is the information contained in such messages correct?

Creatr should be a system-level debugger. While object-level and source-level debuggers are important, they are quite prolific and well understood. System-level debugging is a relatively new field and issues associated with them need to be better explored. As mentioned above, system-level debuggers need to model a computational environment. Creatr should models the EventFlow environment. EventFlow will be explained in more detail in the second half of this chapter. The major difference between the models described above and the EventFlow model is that the EventFlow model is less restrictive and can be used to model a larger class of programs.

3.1.2 User-Defined Event Granularity/Target Independence

Past system-level debuggers have had a limited target domain because they predefined the events they understood. For example, consider the DPD distributed debugger[15]. This tool models a distributed environment but is limited to use in the REM environment because the event types and composition are hardcoded into the tool. In particular, REM provides point-to-point message delivery between processes. While it is not known exactly how REM implements this, assume that this is an unreliable datagram protocol in which processes are identified by process identifiers (PIDs) that are globally unique integers. A message is composed of two such PIDs (sender/receiver) and a variable amount of data. In this situation, the event which REM delivers to DPD is a packet which consists of two integers followed by some data. DPD is thus

built to receive data packets of this format only. Now consider the following three situations. First, another environment models point-to-point messages using string based process identifiers. Because DPD has hardcoded its packet protocol, it cannot be used to debug in such an environment without changing the tool itself. Second, a new point-to-point delivery mechanism, reliable datagrams, is defined on top of the existing delivery mechanism. Event injection of such a message must be done is exactly the same way as it was done previously. Unfortunately, the debugger will have no way of distinguishing between an event which signifies an unreliable datagram and one which signifies a reliable datagram. Finally, suppose new functionality is added to REM which allows for messages to be delivered to multiple receivers (broadcast messages). An event signifying such a message cannot be injected into DPD because DPD's packet protocol cannot adequately transmit information about multiple receivers. In all three situations, the fundamental event which DPD understands must be modified.

To deal with situations where event protocols will change, Creatr should place a premium on target independence. Instead of making assumptions about events that may come from a target and what those events are composed of, Creatr must allow users to specify the type and makeup of events. In other words, Creatr should let users define the meaning of "system-level events" in the context of the target.

3.1.3 Portability/Reuseability

As mentioned in the previous chapter, most existing system-level debuggers are tightly coupled to the target environment. Event injection is embedded within the target environment and the tool is built to only deal with those events and entry points. To migrate existing debuggers to other targets requires two steps. First, hooks must be added to the target. This requires the user to be familiar with the system model the debugger is built for. The user must also know where and how to add the hooks to the target such that it can communicate with the debugger. Second, the debugger needs to be modified to deal with the new event definitions. Debugger sources must be available to the user. The user must also have a strong understanding of the internal

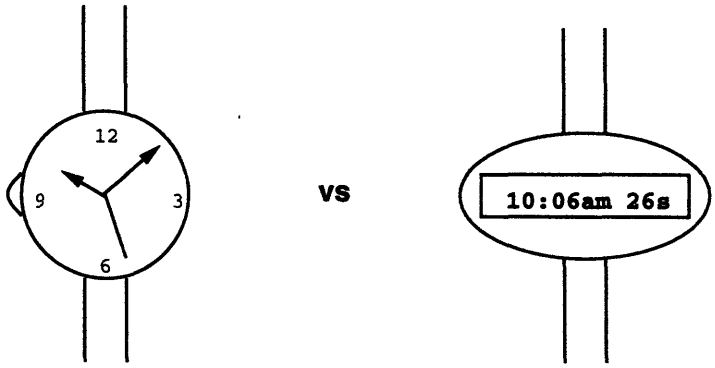
workings of the tool, as he/she will need to modify the code to deal with the events defined by the new target. The tool must then be rebuilt and tested.

The first assumption about the user is reasonable. Any user who wants to interface the debugger to a target will probably be familiar with the target. Knowledge about the system model is necessary to guarantee target conformity. However, the second assumption is unreasonable. The debugger may have been built by a third party. The tool developer may not want to release the sources for proprietary reasons. Furthermore, if the person who is interfacing the tool to the target did not write the tool, learning the internal workings of the tool can have extremely high start-up costs. As a result, migration of existing tools to new targets is both difficult and time-consuming.

To encourage reuse, interfacing Creatr to targets should be easy and quick. Users should not have to know implementation details about the tool itself, but rather the tool must provide facilities through which users can interface new targets without having to change tool sources.

3.1.4 Graphical and Textual Display

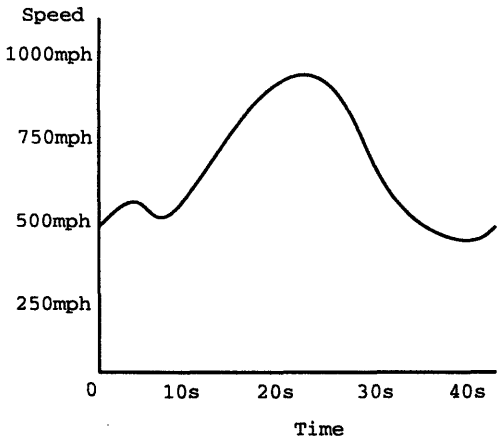
Existing debuggers present information to users in two ways: textually and graphically. Both Peter Bates' system and Instant Replay are text based[2, 11]. They allow users to view events as sequences of text data. Newer systems like DPD have shifted event visualization to a graphical basis[15]. Based on experience with existing debuggers, text is useful for showing the specific value of data associated with any given state during run-time, but graphical display is better suited for showing the dynamic movement of data as the state of the program progresses. For example, consider a source-level debugger. In such a tool, it is appropriate to show the values that variables hold in text form. Trying to convey the value of an integer variable via a bar graph is difficult when the range is large. On the other hand, it makes sense to display how the variable has changed over time in a graphical format. A line graph which maps the values over time is more useful than a spreadsheet listing the values and the times at which they happened. Figure 3-1 shows two different scenarios - the



A Case for Textual Display

Time	Speed
0s	502mph
2s	512mph
5s	549mph
7s	525mph
10s	516mph
12s	617mph
15s	749mph
17s	779mph
20s	883mph
22s	912mph
25s	735mph
27s	710mph
30s	690mph
32s	500mph
35s	493mph
37s	480mph
40s	476mph

vs



A Case for Graphical Display

Figure 3-1: Graphical vs. Textual Display

first an argument for textual display and the second an argument for graphical display. Based on the philosophy that both graphical and textual display are useful for different classes of visualizations, Creatr should display event-flow graphically while event content should be text based.

3.1.5 Customization

Past debuggers have been very limited in their scope of customization. While Peter Bates' system allows users to define higher level events based on lower ones, during presentation time users cannot selectively filter event content. The same holds for

other existing debuggers. Likewise, most graphical system-level debuggers hardcode the way in which they display event-flow. This limits the usefulness of the graphical display.

To encourage use, Creatr should export three levels of user-defined customizations. First, users must be allowed to customize the graphical presentation of event-flow. Second, users should be able to filter events and selectively display only the interesting ones. Finally, the textual presentation of event data needs to be user-defined.

3.2 Overview

The design of event driven, system-level debuggers can generally be broken down into three parts: a back-end, an engine, and a front-end. The back-end is responsible for communicating with the target software. This means receiving event packets from the target and, for interactive debuggers, injecting synthesized events back into the target. The engine acts as a mapping agent and database for the events. It maps the incoming events into a format consistent with the system model, and then stores the events in an internal database. The engine also services database queries. Some engines are passive - they pass data to the client only when queried. Others are active - information is passed upon availability. Finally, the front-end displays the system-level view of the target environment to the user. Information about the target environment is queried from the engine.

The back-end/engine/front-end model can also be thought of as a client-server system. The back-end services the engine by collecting primitive events from the target and passing them to the engine. The engine provides the front-end with a high-level view of the target. Finally, the front-end services the user by visualizing the system-level view of the target execution.

In past debuggers, all three sections have been strictly defined. Back-ends received predefined events, engines stored them, and front-ends displayed them.

Creatr uses a similar approach with two major differences. First, the engine models a less restrictive environment. Second, both the back-end and the front-

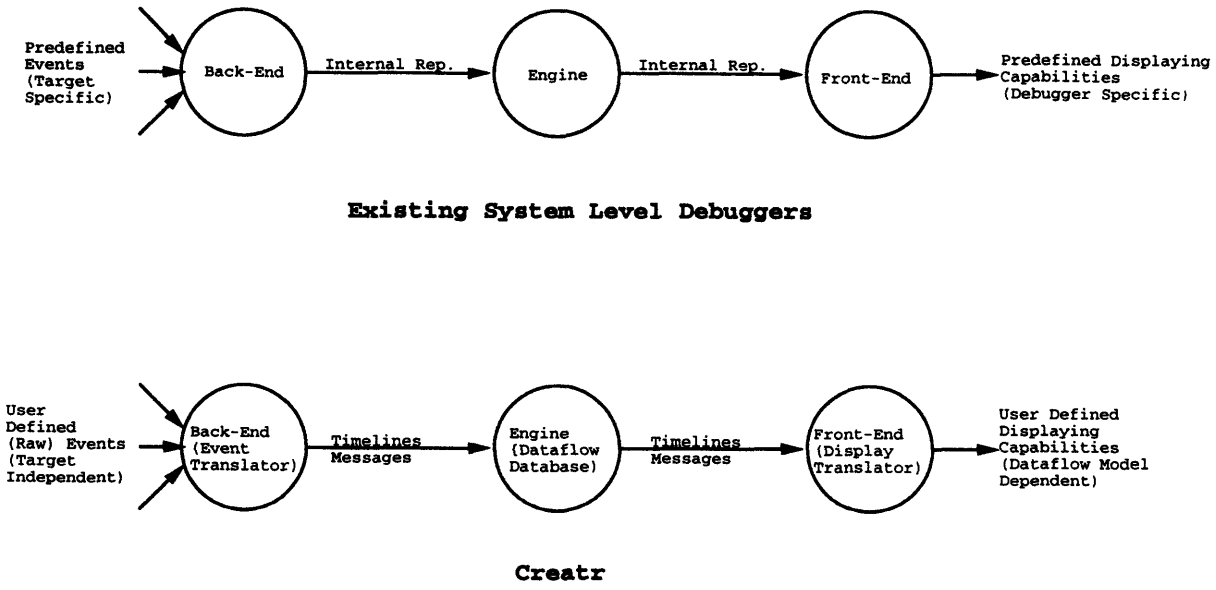


Figure 3-2: System Level Views

end are user configurable. This allows Creatr to be used against multiple targets. Figure 3-2 outlines the major differences in the system design of Creatr from that of past debuggers.

3.2.1 EventFlow Modeling

Creatr's engine is based on the EventFlow model. EventFlow is used to model high-level systems in which separate, loosely coupled execution threads have different tasks. System integration is accomplished through message passing. An abstract view of such an environment is shown in Figure 3-3. In this figure, each thread is just some piece of software responsible for a specific subtask. Through communications, the subtasks are combined to build a larger system.

The EventFlow model was chosen as the appropriate model for Creatr's engine because it represents a more flexible model than previous debuggers have used. In fact, some of the models used previously can be viewed as subsets of the more general EventFlow model. In particular, the distributed programming model can be interpreted as a EventFlow model in which the computing entities are the separate processes and the event-flow is the IPC. However, other programs, which may not

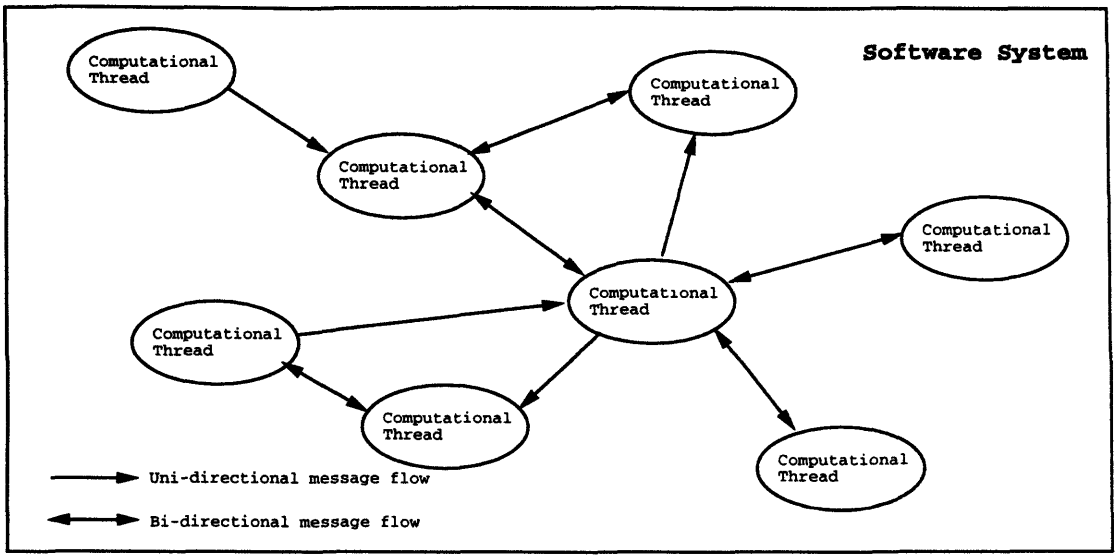


Figure 3-3: The EventFlow Model

be considered distributed at the process level, can still be modeled using EventFlow. Single process, multi-threaded systems are EventFlow based. Threads are objects and shared memory allows for inter-thread communications. Even sequential programs use the EventFlow construct. Object-based programs have data objects which are responsible for accomplishing a subset of the total work. Multiple data objects communicate with each other through exported methods. EventFlow could probably be used to model just about any software system. Arguably, even at the object-level, each instruction represents a separate object. Data is passed from instruction to instruction using registers and memory. However, the lower the granularity of the objects, the harder it will be to accurately model the system using EventFlow. EventFlow is intended to model systems in which the object granularity desired is significantly higher than source-level.

Event Driven Debugging and Terminology

As mentioned earlier, debuggers are event driven systems. Events occurring in the target environment signal actions to the debugger. Creatr is based on this design. Because Creatr models EventFlow, each target event must have some mapping into the EventFlow model.

Creatr internally models the EventFlow environment using two constructs: timelines and messages. Timelines represent computing objects while messages represent the event-flow items. Computational entities do not need to exist throughout the lifetime of a system. Often, one object will spawn another. When an assigned task is completed, objects may disappear. To model this behavior, timelines are dynamic. They have a starting time and an ending time. Messages are actions occurring at a specific time. They are commonly associated with timelines since the event-flow happens among objects.

Target events are used to signal timeline and message actions. There are three different classes of events, two for timelines and one for messages. Timeline-events include creation-events and destruction-events. A creation-event signals the creation of a new timeline. Destruction-events signal the destruction of existing timelines. Message-events are used to notify Creatr that a new message should be added to the current modeling environment.

Finally, EventFlow modeling is time-sequenced. When a model is first created, there are no timelines or messages and the model's logical clock is initialized. Every time a target event occurs, an appropriate action is taken by Creatr. The action is timestamped with the current time, and the time incremented. Thus, there is a causality relationship between all events in the EventFlow environment. Messages have an ordering in the time-space based on when a message-event occurred in relation to other message-events. Timelines have a limited lifespan. Their start and end times are determined by creation and destruction-events respectively.

3.2.2 Customization Facilities

Creatr's back-end and front-end are user configurable. This is accomplished by providing the user with linguistic support for customization. The user defines the customizations in various configuration files. Creatr then uses these definitions to interface to a target and to provide a graphical front-end to the user.

As mentioned earlier, Creatr is designed with ease of reuse in mind. Use of configuration files for customization lets users ignore implementation details internal to

the tool. Furthermore, once the configuration files are specified, building a completed Creatr is transparent. Creatr uses an interpreted environment for the linguistic support it provides. Instead of specifying a configuration, compiling some object based on the definition, and then linking that against Creatr's core to build a final executable, Creatr is already an executable with built-in interpreters.

Back-end

The back-end of Creatr is customizable. Users can provide event definitions. Using a language outlined in a later section, users specify the events that will be happening in the target environment and how they map to Creatr's notion of EventFlow. Based on the definitions, users need to add hooks to the target sources. These hooks are the run-time mechanism through which Creatr is notified of target events.

The main power of this approach is that Creatr becomes a generic debugger which can be used across multiple targets, each of which may have a separate notion of events. While users must add target-side hooks explicitly, they do not need to be aware of how Creatr deals with the hooks internally. Ideally, Creatr should provide an automated way of inserting hooks into a target (much like source-level debuggers embed symbolic information into code during compile time). Unfortunately, in the context of system-level debugging, this is a hard problem and thus has not been attempted. However, to reduce the interfacing costs, the back-end encapsulates the packet level protocol used to deliver events.

Front-end

Creatr's front-end lets users define exactly how they want to display things graphically and textually. Graphical presentation is used for event-flow visualization while textual presentation is used for event content visualization. Three customization facilities are provided: two for event-flow visualization and one for content visualization. Like event definitions in the back-end, these facilities are language driven and the configuration files interpreted at Creatr run-time.

In existing system-level debuggers, events are predefined. The tool developer

can hardcode a mapping from events to graphics. Before an event is visualized, the tool does a table lookup to see how the event should be displayed and then renders accordingly. Creatr's back-end allows users to define primitive events. A mapping from events to graphics cannot be easily embedded within Creatr when the events may be redefined across applications. In Creatr, users can define infinitely many different event types. Hardcoding rendering routines for some subset of these event types would probably not be useful. Furthermore, each event could be rendered in infinitely many ways. Some users may find one style useful while others may feel differently. Instead, Creatr lets the user define the graphical rendering of events. Rendering is accomplished by writing routines for each back-end event type in a visualization language.

Past experience suggests that filtering capabilities are useful because large systems have high event-flow traffic. If the user wants to debug a specific module within the target, he/she is probably not interested in seeing unrelated event-flow. Creatr exports a filtering language which can be used to filter events based on event type and content.

After events have been collected from a target, the user may be interested in viewing the content of specific events. However, event data coming from a target environment will be in binary form. From a debugging standpoint, binary data is not as useful as symbolic representations. Creatr has a data formatting facility through which users can specify symbolic mappings for data. During the textual display of event data, instead of displaying bit patterns, Creatr will match the bit pattern to a string based representation based on the configuration specified. This string is presented to the user in place of the original data.

Chapter 4

Design

The latter portion of the previous chapter gave a brief overview of Creatr from a conceptual viewpoint. This chapter details the complete design of a toolkit based on the ideas presented. It begins by presenting an overview of the toolkit itself and then details each specific module. Design issues concerning the user interface are discussed, as is the linguistic support the back-end and the customization facilities provide. Based on the design presented in this chapter, the reader should be able to implement a system-level debugger which incorporates the ideas Creatr presents.

4.1 High-Level View

Creatr is a system-level debugger which allows users to view events in a target program. To do so requires some form of run-time interaction between the target and Creatr. Creatr must be notified of events that are happening within the target so that it can display the information to the user. To accomplish this, Creatr is designed as a set of tools. The first pair of tools, CTRHDR and CTRLIB, are coupled to the target program. They act as the hooking mechanism within the target software, which delivers events to the other tool, CTRVIEW. CTRVIEW is the visualization tool. It receives events from the target, maps them to a EventFlow model, and based on user customizations, displays them. Figure 4-1 presents a system-level view of the toolkit set and shows how each of the parts interacts with a target.

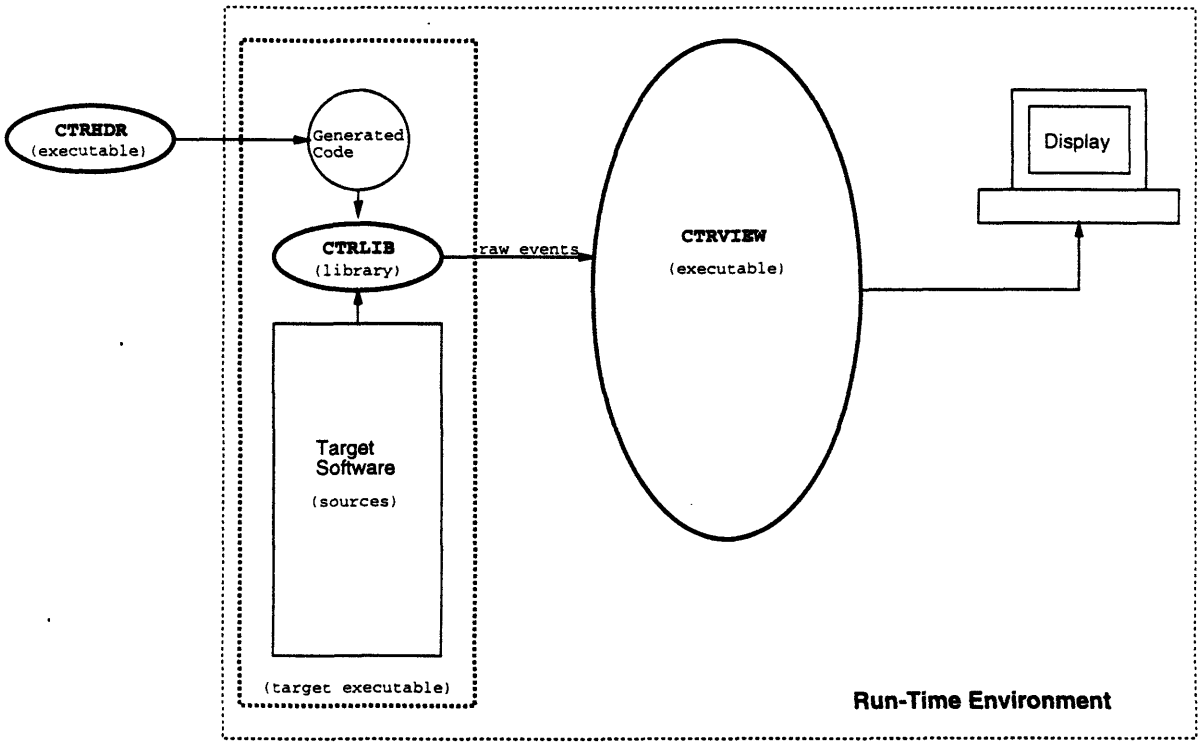


Figure 4-1: Creatr Toolkit

As mentioned in the last chapter, Creatr is composed of three modules: the back-end, the engine, and the front-end. While the engine and front-end are embodied only in the visualization tool, the back-end's functionality is divided among all three tools. The remainder of this chapter describes in detail the functionality of each of the three modules.

4.2 Back-End

The back-end of Creatr is responsible for maintaining the run-time interaction with the target application. It receives raw, user-defined events from an application, maps them into timelines and messages accordingly, and then delivers the synthesized timelines and messages to the engine. The back-end is customizable. Users can specify the mapping of raw events to timelines and messages.

4.2.1 Event Mappings

To describe a mapping of target events to Creatr timelines and messages, two pieces of information are needed. First, the user needs to decide what each event signals. Does the event signal a message or the creation/destruction of a timeline? The second piece of information is the content of the raw event. Useful events contain a timestamp (when did the event happen), and some information (where did the event happen and what was the surrounding context). The former is handled internally by Creatr. Creatr time sequences all generated events. The latter is accomplished by users defining the makeup of an event.

An Event Definition Language (EDL) represents the means through which users can define event mappings. Users write event definitions in EDL, which are then used by Creatr to negotiate the run-time communication between Creatr and the target. The definitions are also to build an EventFlow model from the generated events.

The EventFlow model is composed of objects and the interactions among them. As mentioned in the previous chapter, Creatr models the objects and flow using timelines and messages respectively. In the target environment, raw events are generated.

These events, when received by Creatr, will need to be mapped to either timelines or messages. In EDL, users define the complete set of raw events which could be generated during target execution. Users also specify what each event signals: message, timeline creation, or timeline destruction. Finally, they specify the information that accompanies an event. While event to timeline/message mappings are important because they provide the mechanism through which target systems can be modeled using EventFlow, the actual composition of an event is vital because it dictates much of the visualization scheme.

One important aspect of message-events is that they are most likely linked to various timelines in some manner. For example, in a distributed environment, IPC messages are associated with two timelines: the sending process and the receiving process. When viewing a message-event, it is common to need information about the timeline-event. EDL provides linguistic support for naming timeline-events and referring to such events from message-events. This way, message-events only need to have pointers to timeline events. As discussed later, this also proves useful for the visualization semantics.

Finally, while EDL is mainly for defining event structures, it also allows users to define static events. This is useful for persistent timelines. For example, in a distributed environment, multiple processes may be communicating with each other. A subset of these processes may always exist, while others are spawned and killed. For processes that are static, run-time communication overheads can be reduced by defining static timeline-events for each such process. Furthermore, this reduces costs associated with interfacing Creatr to new targets. Users do not need to add hooks for persistent events in the target sources. The next section describes in more detail how Creatr and a target interact and should help make clear why static events are useful.

4.2.2 Event Delivery

At run-time, Creatr needs to be informed of the events which a target application will be generating events. A communication link between Creatr and the target needs

to exist. The sending end of this link must be embedded within the target, and the receiving end within Creatr's visualization tool. To accomplish this, the back-end is distributed across all three tools. CTRHDR and CTRLIB embody the sending end and the receiving end is contained in CTRVIEW.

CTRLIB is an injection library. User can use this library to add event creation hooks to target software. Within the target sources, the user needs to decide when it is appropriate to create an event. Based on the event definitions the user provides, routines provided by the library allows users to build and transparently deliver raw events to the receiving end. This is useful because it provides a single entry point from the target to Creatr and also encapsulates the packet level protocol used to deliver events. As a result, the user need not know the protocol. It also reduces the total amount of extra code a user needs to add to the target sources when interfacing it to Creatr.

CTRHDR is a secondary tool which provides information concerning event definitions to CTRLIB. The sending end of Creatr needs to know the composition of raw events, but not how they map to timelines and messages. When the user makes calls to the library routines, the routines can internally parse the event definition file to get this information. Based on the composition, the library then needs to define a packet protocol which can be used to send the event from the target to the visualization tool. To reduce performance overheads, this functionality is pushed out of the run-time environment. Instead, CTRHDR parses the event definitions and generates a header file which defines the packet protocol for each event type. This header is then used by CTRLIB to build packets from events and deliver the packets.

To realize the packet-level protocol, the back-end of CTRVIEW also parses the event definition. When packets are received from the target, CTRVIEW can appropriately rebuild the raw event, map them to timelines or messages, and deliver them to the engine. The reason CTRVIEW directly parses the event definitions instead of using another tool like CTRHDR is because such a scenario would require recompiling CTRVIEW each time the definitions changed. As mentioned earlier, to reduce migration costs, the user should never have to rebuild the Creatr system.

4.2.3 Time Sequencing of Events

As mentioned in Chapter 3, all collected events are time-sequenced and ordered. However, the issue of correct ordering was not addressed. In distributed environments, multiple processes can be generating events. If process A generates a message-event before process B, will Creatr receive and order the two events correctly? Do the two events even need to be ordered correctly? Creatr is designed to display events using a total ordering scheme. Each event is assigned its own timeslot in an internal, global timespace. The ordering is determined by the order in which events are received. Thus, Creatr needs a communications substrate which guarantees that events are received by Creatr in the same order that they are generated in the target.

Currently, Creatr ignores this issue. The assumption is made that a reliable, totally ordered multicast mechanism exists. This communication system should enforces an ordering based on a global target clock[10]. The total ordering is time based. If event A occurred at a logical time prior to event B, then A "happened-before" event B. In other words, Creatr does not support event concurrency within the target. There is a one-to-one mapping between events and timeslots.

In reality, ordered multicast systems may not exist for various platforms. The current implementation of Creatr uses a FIFO piping mechanism. Pipes are one-to-one communication protocols. Thus, while Creatr can guarantee correct ordering for sequential targets, it does not currently support ordering across a distributed target. Future versions of Creatr should incorporate the communications layer into its design. Totally ordered multicast protocols have already been developed and do not need to be redesigned. Examples include the ISIS ABCAST primitive[4] and the xAMp system[13].

Creatr is designed to display a total ordering among events in the target environment. However, it may not be necessary to define a total, time-based ordering on all the generated events in a distributed environment. The user may feel that a partial ordering is sufficient. Since each event is given its own timeslot, Creatr is then free to derive a total ordering based on the partial ordering. User-defined partial orderings are needed in this case. One possibility could be to somehow extend

EDL to allow users to not only define event types, but also define causal relations between the events. A mechanism would also be needed to define a target "process". Then, Creatr can keep a separate clock for each "process" and use the causal relations to determine ordering among events generated during the same logical time. Partial ordering schemes which could help implement such a feature already exist. The Psync protocol implements a partial ordering scheme by assigning precedence to operations[12]. This protocol could be adapted to work within a modified Creatr. Precedence would be established across events instead of operations.

4.3 Engine

The engine represents the section of Creatr that is not involved in target interfacing. Engine functionality cannot be customized by the user. The back-end receives user-defined events from the target environment and maps them to timelines and messages. The front-end is used to customize the graphical and textual display of the different timeline and message types. The engine acts as the glue for the interfacing layers. It stores synthesized timelines and messages and also manages the user interface through which the front-end customizations are rendered. Both the back-end and front-end are language driven. Their main tasks are to parse configuration files and interpret them when appropriate. The engine drives the entire process by acting as the I/O mechanism for the end-user. Users tell the engine what history they want to view. The engine then displays all events in the requested history for the user. If customizations for certain events are specified in the front-end, the engine will make calls to the front-end. The front-end interprets the specified display routines, making appropriate calls back into the engine when output to the user is desired.

Creatr's engine is contained within the visualization tool, CTRVIEW, and is composed of two parts: the object database and the graphical user interface (GUI). The object database handles receiving timelines and messages from the back-end and storing them internally in a manner consistent with the EventFlow model. It also services requests to access the timelines/messages and their associated primitive events. The

GUI is the physical user interface to Creatr. It handles requests from the user dealing with the collection and viewing of events from a target. Often, GUIs are considered front-end modules. In Creatr, the GUI is part of the engine. This decision allows the core functionality to remain separate from the user-defined functionality. However, the GUI and the front-end are closely related. The GUI presents a standard system-level view of the target to the user. It is also used to render any customizations the user has specified in the front-end.

4.3.1 Object Database

The object database enforces the EventFlow model by collecting timelines and messages and storing them internally as histories. It also services requests by the GUI and front-end customization facilities to retrieve information about histories and the timelines/messages in them. Finally, the object database also acts as the manager for the GUI. It keeps track of open windows, the histories being displayed in various windows, etc.

Creatr supports real-time event collection and viewing through the object database. Histories do not need to be completed to have requests for them serviced. If an open history is requested for viewing, the object database will deliver all existing timeline and messages. The database keeps tabs on which window is requesting which history. If more events arrive for a history which is opened for viewing, the database will deliver them to the window as they are received.

To achieve the functionality described above, the object database is both passive and active. It passively receives target events from the back-end as the events are triggered. The interaction with the front-end and GUI is passive and active. A passive request to open a history causes the database to switch into an active role. It delivers all events currently in the opened history to the client. As more events are received from the back-end, they are automatically delivered to the requesting client. This keeps clients from having to busy-poll for new events.

4.3.2 Graphical User Interface

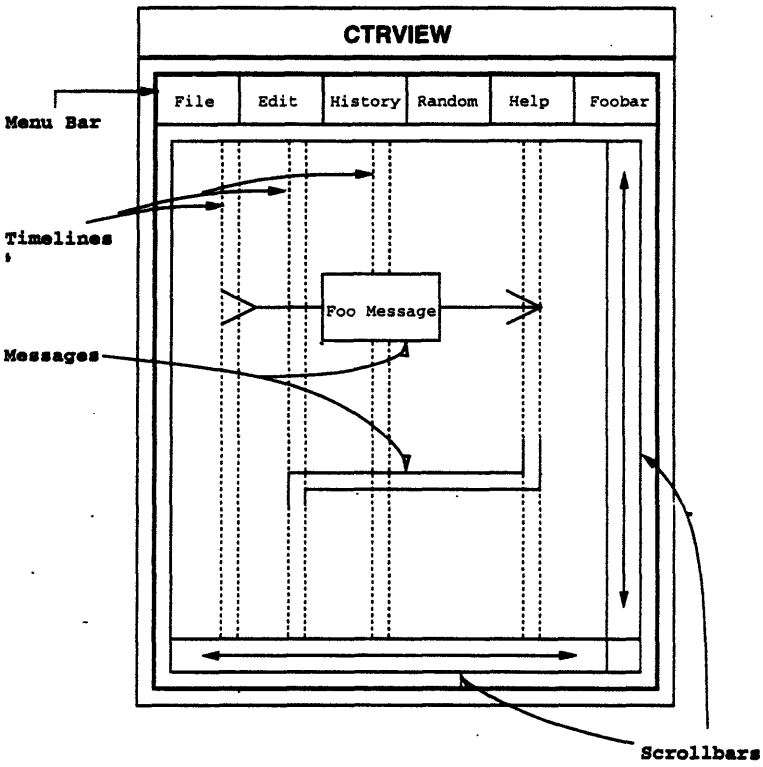
The GUI is the physical user interface. It is a window-based, menu-driven front-end to Creatr. It handles requests from the user concerning the collection and viewing of target events. The GUI is the portion of Creatr that users interact with during debug time.

The GUI is designed with user needs in mind. Based on experience, system-level debugging involves collecting many histories from various runs of a target and then comparing the histories for discrepancies, differences, etc. The GUI allows users to see many histories concurrently. It also lets users control the collection of timelines and messages into different histories. Because debugging is a time-intensive process, the GUI also lets users save and retrieve histories to persistent memory. The main job of the GUI is to allow users to view various collections of timelines and messages. To do so, it depends on user-defined customizations. Every time a timeline or message is to be rendered, the GUI calls the appropriate front-end routines. The GUI also provides services to the front-end routines. To actually render anything to the screen, during interpretation time, the front-end will make calls to user interface routines provided by the GUI. While event-level customizations are provided by the front-end, the GUI provides direct support for history-level customizations. Figure 4-2 presents a rough sketch of Creatr's GUI and also outlines some of the support it provides for manipulating histories.

4.4 Front-End

Creatr's front-end lets users customize the display of timelines and messages and is part of CTRVIEW only. As mentioned Chapter 3, it is useful to have both graphical and textual display. The graphical display should allow users to get a system-level view of the target flow while the textual display should provide for event-level visualization. Creatr graphically displays histories, a coherent set of timelines and messages. On the other hand, event data is textually presented to the user.

Creatr is designed with three levels of customization: two for the graphical display



Support for History Manipulation

A) Ability to chose which history to Display

B) Ability to Load/Save histories

C) Ability to move timelines around relative to one another

D) Ability to space timelines as desired (i.e. set the horizontal margins of a timeline)

E) Ability to select messages and move amongst selected messages (i.e.scroll history such that selected message is in view)

Figure 4-2: Creatr Graphical User Interface

and one for the textual display. Graphical visualization is used to define how timelines and messages get graphically drawn while filtering allows users to specify which subset of the timelines and messages are of interest. Message formatting is used to define how raw data within message-events gets mapped to symbolic strings.

4.4.1 Visualization

The visualization facility controls the graphical presentation of timelines and messages. When the user requests to view a history, the GUI will retrieve the appropriate history from the object database. Graphical display of the timelines and messages within the history are then based on the customizations provided by the user.

Graphical visualization in Creatr is based on the event definitions from the back-end. In a language called VL (visualization language), users define how each of the EDL based events will be drawn. During graphical display, Creatr will iterate through all the events, and in the process, call each event's drawing routine.

Events in Creatr can map to either timelines or messages. The general presentation of timelines and messages is outlined in Figure 4-3. Timelines are vertically running pairs of lines which have a defined start time and a defined end time. Sequencing of start and end times are determined by the events signalling creation and destruction respectively. Messages are vertically stacked rectangular regions. Events in Creatr are time sequenced based on when they are received. If message A is located above message B, then the event which triggered message A was received by Creatr prior to the event which triggered message B.

VL allows users to customize the drawing of the EDL-based events. Because events can map to either timelines or messages, VL provides different customizations for each event type. Timeline visualizations are limited while message visualizations are open-ended.

While timeline visualization is largely predefined, VL does allow the user some flexibility. Users can chose the draw pattern used for the outline and the interior of a timeline. The outline and interior can also be dynamically changed based on the messages associated with an event. This is useful for showing internal object state

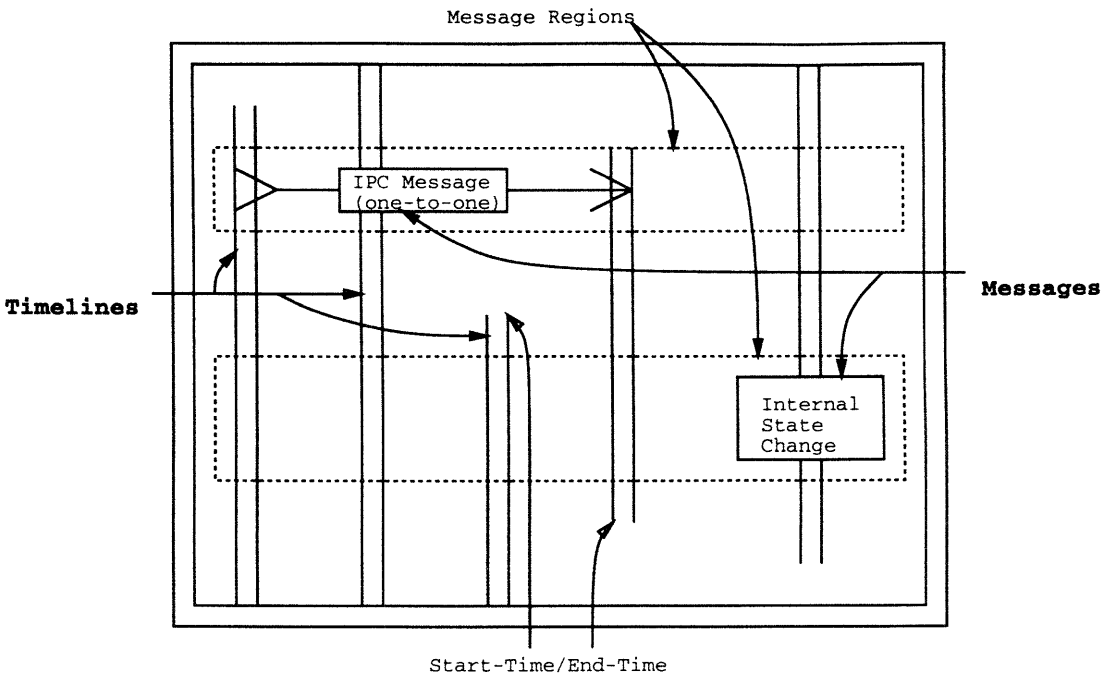


Figure 4-3: Basic Timeline and Message Visualization

changes of within the target. For example, in a distributed environment, processes can be active or asleep. Message events have been defined to signal wake-up and sleep. Furthermore, the user would like to view an active timeline as two solid lines and a sleeping timeline as two dotted lines. At run-time, the drawing state of the process timelines can be toggled based on the message-events received. Sleep events cause the timeline to start being drawn dotted and wake-up events cause it to be drawn as solid lines. This functionality allows users to graphically view the execution state of objects without having to manually view each event which signals a state change.

Message visualization is very flexible and draws much of its power from the fact that message events are likely to be bound to timelines. During viewing, each message is assigned a timeslot based on the order in which Creatr received the message event. To draw messages, Creatr will sequence through all the messages. Beginning with the message assigned the lowest timeslot, Creatr will call each events drawing routine. When a routine is called, it gains control of the entire viewing screen. The routine can then draw the message how and where it chooses to. To support message drawing,

VL provides primitive drawing capabilities. Primitives for line, square, circle, and text drawing are all supported. Furthermore, VL gives the user the ability to change the state of the drawing brush. This include the brush style, brush width, fill pattern, and foreground/background colors.

While users can draw each message anywhere on the viewing screen, in general, there should be some ordering. Since time-space progresses downward, messages which occur after other messages should be drawn below the previous messages. Display ordering is accomplished through vertical offsets. Each routine specifies how much vertical space it uses to render a given message and all drawing primitives accept relative y coordinates. The absolute y coordinate is derived by adding in the vertical space reserved for all prior messages. If users draw only within the reserved y space and never specify negative y coordinates, then each message will be rendered in the correct order with no overlapping.

In addition to the drawing support outlined above, VL provides extensions to EDL-based events. These extensions can be useful in rendering messages. First, the horizontal positioning of timelines is exported through timeline-event extensions. Messages can reference associated timelines. A timeline can reference its horizontal position. Thus, a message can horizontally position itself through its timelines. Second, the filter state of each event can be referenced. Messages containing filtered timelines are not automatically filtered. By referencing a timeline's filter state, a message can dynamically change the way it should be rendered when certain associated timelines are hidden. Finally, message-events have a selection field. While Creatr's GUI allows users to select/deselect messages, no general display capability is provided through which users can distinguish selected and unselected messages. User's can improvise by making message visualization a function of the selection state.

4.4.2 Filtering

Creatr provides a filter through which users can hide uninteresting events. Large target systems may have a variety of event definitions and multiple injection points. During run-time, Creatr could be collecting a huge number or events from the target.

However, the user may not want to view all the events. More often, the user will be debugging a subset of the entire target and wants to see events related to that section of the target only. By setting up a filter, users can hide unrelated timelines and messages. As mentioned above, after a GUI retrieves a history, it iterates through all the events, calling their drawing routines. Before a drawing routine is actually called, the event is processed by the filter. If the event passes through the filter, then the drawing is done. Otherwise, the event is simply skipped and not displayed to the user.

Filters are written in FL (filtering language). FL provides support for filtering based on event type and content. For each event type, users write a filter routine. Linguistic support for regular expression matching is used to filter based on event data. Users can specify a regular expression which some portion of the event data is matched against. Based on the success of the match, the event is passed through the filter and drawn, or caught and skipped. Language support for timeline-events and message-events is identical. While message-events directly map to messages, timelines have one or two events associated with them: creation and destruction events. Timeline filtering is based on the creation event. Events signaling timeline creation are filtered to determine if a timeline should be drawn. Destruction events are simply ignored during filtering.

One interesting aspect of the filter is that messages which contain hidden timelines are not automatically filtered. Situations may arise in which users want to render messages even when related timelines are hidden. To avoid compromising such functionality, no implicit filtering is performed. Rather, FL provides an event extension through which the user can access the filter state of an event (similar to the filter state extension provided by VL). Users can then explicitly filter a message based on the filter state of associated timelines.

The other important feature deals with timeslot allocation and ordering. All events have a distinct ordering. However, filtered messages lose their reserved space allocation in the view screen. This collapses the user-viewable time-scale when there are hidden messages. For example, suppose there are three messages, A, B, and C,

each having a vertical space reservation of 25 units. If nothing is filtered, the three messages will consume 75 units of vertical space in the view screen. However, if message B is filtered, only 50 units will be consumed by messages A and C. Message C will be rendered in the space directly below message A. No space is reserved to show that there is really a message between the two which happens to be filtered. The same is true for timelines. A timeline has a creation-event and possibly a destruction-event. If between the two events, there are five message-events and nothing is filtered, then the user will perceive the timeline to have existed through five timeslots. However, if any of the intermediate messages are filtered, then the user's perception of a timeline's lifespan is skewed as well.

4.4.3 Message Formatting

In the same manner that the visualizer and filter customize the graphical presentation of timelines and messages, the message formatter lets users customize the textual presentation of message-based event data. To view the content of message events, users select the graphical message. This causes a separate window to appear which contains the event data. However, event data is in binary form. Users like symbolic representations. Furthermore, it may be useful to process the event data and display a report accordingly. To support this functionality, each message-event is sent through the message formatter before having its data displayed. Instead of displaying the event data, the report generated by the message formatter is displayed.

The language used to write message formatters is MFL (message formatting language). MFL is similar in functionality to the FL. Separate routines are written for each event type (message-events only), and regular expression matching controls report generation. Primitives for report generation also exist and are like the "printf" routine in C.

MFL does not support state. In other words, the action performed on each message-event is only dependent on that event. This is a clear deficiency in the language and future versions of MFL should probably include such functionality. Static global variable support is an example of one simple approach.

One important aspect of the linguistic design of MFL is that it is backwards compatible with an existing formatting language. This is needed because an existing specialized system-level debugger exists which uses message formatting files. Creatr is intended to replace that debugger as a more general solution. However, to reduce the migration costs, MFL keeps much of the same syntax as the other debugger's formatting language. Porting existing format definitions to MFL-based definitions should requires minor changes only.

Chapter 5

Languages

A high degree of user customization makes Creatr very flexible. The previous chapter detailed the high-level design of Creatr and explained the features of the language-driven customizations. This chapter describes in some detail the syntax and capabilities of each of the languages. Examples are used to provide a clearer understanding of the functionalities.

5.1 Linguistic Style

Before detailing each of the languages, an explanation of the design decision which guided the linguistic development of the languages is given.

The foremost goal of the linguistic design was that the languages should be relatively easy to learn. If users are to embrace Creatr, it should have lower start-up costs than other available solutions. Experience suggests that the cost of learning new languages is directly dependent on the leveraging users get from languages they already know. If a new language is similar in syntax and functionality to a language the user already knows, it will be easier to learn. C is recognized as a leading industry standard for programming languages. To appeal to the widest audience, Creatr's languages try to be C-like in their syntax and semantics.

The second design decision deals with the breakup of capabilities into four different languages. Creatr requires users to write four different configuration files, each in a

different language. Why not a single configuration file in a single language? The linguistic design was secondary to the overall design of Creatr. Since each of the customizations was already modularized, it was easy to design separate languages for each module. Developing a generalized language would have added complexity to the high-level design of Creatr and would also have made the implementation of Creatr more difficult. However, as discussed in Chapter 7, a generalized language is probably the more elegant approach.

The final issue is backwards compatibility. An existing system-level debugger was being widely used and had features closely related to message formatting. A large numbers of message formatting definitions already existed. To reduce migration and maintenance costs, the message formatting language needed to closely resemble the existing formatting language. This constrained the linguistic design of the message formatting language. The capabilities of the filtering language closely matched those of the message formatting language. To reduce development time and complexity, the constraint was imposed on the filtering language as well. Unfortunately, it was recognized early in the development of the languages that the existing language was lacking in certain regards. Event definitions and visualization required significantly different linguistic features. Thus, the constraint was not imposed on the event definition language or the visualization language.

5.2 Event Definition Language(EDL)

EDL is used to define events. Based on the events, Creatr develops a communication protocol between the target and itself. Creatr also uses the definitions as the template for EventFlow modeling. Conceptually, EDL is a type definition language. Users define aggregate event-types using primitive types. To facilitate EventFlow modeling, events must belong to one of two classes: timeline or message. At run-time, packetized data is received from the target and parsed into one of the defined event-types.

5.2.1 Timeline-Events

Timeline-events are composed of two sections: identifiers and data. Object naming is accomplished through the identifiers and is used by message-events to reference timelines. Information about objects is passed to Creatr through the data portion of timeline-events.

Imagine that system developers are building a distributed database, and need a tool to model the communications among the nodes. The system has two types of nodes: master and slave. All nodes are identified using a unique address. The following timeline-events could be defined for such an environment.

```
type timeline (nodeid:int) {
     Master(SlavesIControl:buf, WhatIDo:string);
     Slave(MasterWhoControlsMe:int, WhatIDo:string);
     }
```

In this example, two timeline-events have been defined: "Master" and "Slave". The variable definitions following the keyword "timeline" are the identifiers. Identifiers are global across all timeline types. This allows message-events to reference timelines without constraining the specific timeline type a message-event can reference (timeline referencing explained below). Each instance of a timeline-event must contain a unique "nodeid" integer value. Data for each of the event-types is different and specified following the name of the event. "Master" events have two pieces of data: a buffer and a string. "Slaves" have an integer followed by a string.

As mentioned in Chapter 3, Creatr models timelines dynamically. Separate events signal the creation and destruction of timelines. In EDL, users only specify one event-type for each of the different timeline-events. The timeline-events the user defines are actually used for creation-events. Destruction-event-types are implied by the definitions and do not need to be specified. While creation-events need to include both identifiers and data, destruction-events only need to include identifiers. Because identifiers are globally unique across all timeline-event-types, only one destruction-event-type is needed. An event-type called "TIMELINE_DESTROY" is implicit in all definitions and is composed of the identifiers specified. If multiple identifiers are used for timeline definitions, "TIMELINE_DESTROY" events will contain all the

identifiers.

5.2.2 Message-Events

Message-events are composed purely of data. However, as mentioned in Chapter 4, messages are often linked to timelines. Timeline referencing is supported in EDL. Message-event data can be either a primitive type (integer, string, or buffer), a timeline identifier, or a timeline type. When a timeline identifier is specified, the raw data delivered from the target will be of the primitive type specified for the identifier. When received, the data is matched against the identifiers for all existing timeline. If a match is found, the message-event variable becomes a pointer to the matching timeline. The timeline type is used to reference an entire set of existing timelines. Variables of this type are arrays. Each element in the array references a different timeline of the type specified. For primitive and identifier types, data is expected from the target. Timeline variables do not expect such data. Rather, when a message-event instance is created, all such variables are mapped to timelines using the existing set of timelines which have not been destroyed (the set of instantiated timelines without destruction-events).

Using the distributed database example, suppose that communications among the nodes are to be either point-to-point or broadcast to all masters/slaves. To model this environment with Creatr, the following message-event definitions could be used.

```
type message {
     PointToPoint(FromWho:timeline->nodeid,
                  ToWho:timeline->nodeid, TheMessage:buf);
     MasterBroadcast(FromWho:timeline->nodeid,
                     ToWhoAll:timeline(Master),
                     TheMessage:buf);
     SlaveBroadcast(FromWho:timeline->nodeid,
                    ToWhoAll:timeline(Slave), TheMessage:buf);
     }
```

When "PointToPoint" events are injected into Creatr, the first piece of the packet will be an integer which is used to identify the timeline (node) from which the message is originating. The second integer identifies the destination timeline. A buffer of

the actual, target-dependent, message rounds out "PointToPoint" events. "MasterBroadcast" and "SlaveBroadcast" events are similar to "PointToPoint" events. The first data slot contains the sending timeline's identifier. The next variable is used to reference an entire class of timeline types. When instances of either event-type are created, the "ToWhoAll" field is mapped to all instantiated timelines of the appropriate type: "Master" timelines for the "MasterBroadcast" messages and "Slave" timelines for the "SlaveBroadcast" messages. For these two types of events, the second field in the event packet corresponds to "TheMessage" buffer.

5.2.3 Static Events

Events are triggered within the target environment and then delivered to Creatr. Certain events may be constant, in that all runs of a target cause the events to happen. In particular, timelines are often persistent. Certain creation-events may always happen at the beginning of target execution. To reduce event injection overhead and the interfacing costs of adding target hooks, EDL provides a way to specify static events. In the configuration file, users can create instances of event-types. These instances are available when a new history is opened. They are time sequenced according to the ordering of their definitions.

Suppose that a distributed database has two persistent Master nodes and four persistent Slave nodes. Instead of adding hooks to the target code to stimulate timeline creation, static events can be defined.

```
statics {
   Master(0x01, ''6 9'', ''Control Queries'');
   Master(2, ''7 8'', ''Database Changes'');
   Slave(0x6, 1, ''Parse Query'');
   Slave(7, 02, ''Add Records'');
   Slave(8, 2, ''Delete Records'');
   Slave(011, 1, ''Lookup Query'');
   }
```

When defining static events, the event definitions can be modeled as function prototypes with implicit function definitions. Static events are then defined as calls to the different functions with the appropriate data. Integers can be specified in hex-

adecimal, decimal, or octal form. Strings and Buffers are specified as string constants.

5.3 Visualization Language(VL)

VL is used to customize the graphical drawing of the events defined in EDL. For each of the EDL-based event types, users write a VL-based visualization routine. Chapter 4 outlined the basic templates available for timeline and message rendering. Timelines are constrained to vertical line-pairs while messages are rectangular regions. The drawing support for each of the types reflects these constraints and is discussed below.

5.3.1 Control Flow

VL has three different control flow primitives: "if", "for", and "while". The syntax and semantics of each statement is identical to that of its C counterpart.

5.3.2 Drawing Primitives

VL is designed to allow users to easily specify drawing routines. Functionally, VL is similar to other rendering languages like Postscript or Printer Command Language (PCL).

Brush State

Users of window-driven drawing programs are probably familiar with the notion of a brush state. The brush state represents the properties used when drawing basic figures. Consider drawing a polygonal shape. What color should the outline of the figure be? If the interior is filled, the fill color and fill pattern need to be specified as well. Drawing programs allow users to change such properties through menu options. Typical options include line width, foreground/background color, fill and line patterns, and text font. VL provides this functionality through a special variable called "drawstate". The variable is an aggregate type which can be conceptually

pictured as the following C structure.

```
struct DrawstateType
   {
     int brushwidth; # between 1 and 10
     BrushType brushpattern;
     FillType fillpattern;
     Color foreground;
     Color background;
     TipInfo *brushstart;
     TipInfo *brushend;
   } *drawstate;
enum BrushType
   {
     SOLID, DASHED, DOUBLEDASHED, NONE
   };
enum FillType
   {
     SOLID, DOTTED, SLASHED, CROSSED, NONE
   };
enum Color
   {
     BLACK, WHITE
   };
struct TipInfo
   {
     int baseangle;
     int tipangle;
     int length;
   }
```

The global variable, "drawstate", is implicitly declared and can be used to set the brush properties. When calls are made to drawing routines, rendering is based on the properties set in the drawstate at the time a specific drawing primitive is called.

Most of the properties specified in the drawstate are self-explanatory. The subtle ones are the "brushstart" and "brushend", which are used for line drawing. They are used to determine the exact manner in which the arrowhead and tail are rendered. Most drawing programs allow users to append simple arrows or tails to lines. However, a common property of Creatr messages is that they are directed. This theme suggests that having a finer granularity for line rendering would be useful. If users want to add an arrow or tail to a line, instead of drawing the arrow/tail explicitly, they can specify

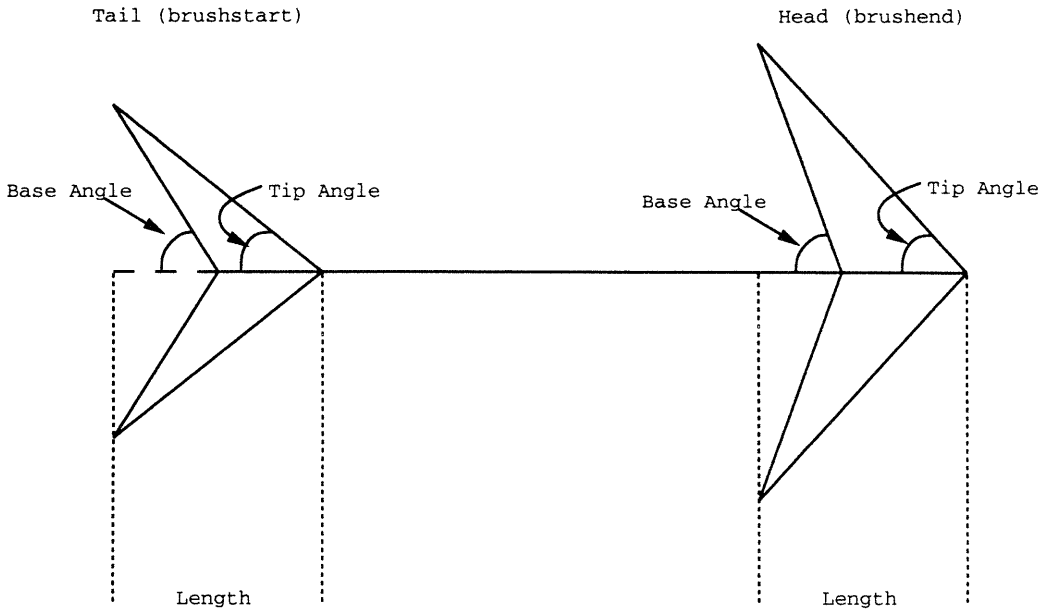


Figure 5-1: Properties of Line End-Points

brushstart and brushend properties. When a line is rendered, the starting point will have a tail appended to it based on the "brushstart" property. Similarly, the end point can have an arrow defined by "brushend". Each of the two properties take three arguments: baseangle, tipangle, and length. Figure 5-1 graphically describes how each of the three arguments affects an arrowhead or tail.

Drawing Routines

VL provides four general drawing routines which can be used to render figures. These drawing routines are implemented as built-in function calls. Each of the routines is outlined below.

```
drawline (int startx, int starty, int endx, int endy)
  # startx is the x location of the starting point
  #       (absolute terms)
  # starty is the y location of the starting point
  #       (relative terms)
  # endx is the x location of the ending point
  #      (absolute terms)
  # endy is the y location of the ending point
  #      (relative terms)
drawbox (int topleftx, int toplefty, int length, int width)
```

```
  # topleftx is the x location of the top left corner of the
  #         box (absolute terms)
  # toplefty is the y location of the top left corner of the
  #         box (relative terms)
  # length is the horizontal length of the box
  # with is the vertical width of the box
drawcircle (int centerx, int centery, int radius)
  # centerx is the x location of the center of the circle
  #        (absolute terms)
  # centery is the y location of the center of the circle
  #        (absolute terms)
  # radius is the radius of the circle
drawtext (int bottomleftx, int bottomlefty, string text)
  # bottomleftx is the x location of the bottom left corner
  #           of the text's bounding box (absolute terms)
  # bottomlefty is the y location of the bottom left corner
  #           of the text's bounding box (relative terms)
  # text is the actual string to be rendered
```

The routines described above can be used to render messages in a user-defined manner. However, they are not useful for timeline visualization. Three other, simple routines are provided through which the graphical state of timelines can be customized.

```
set_outline (BrushType brushpattern)
  # brushpattern specifies which pattern the timeline's outline
  #           should be drawn in
set_fill (FillType fillpattern)
  # fillpattern specifies which pattern the timeline's interior
  #           should be drawn in
set_label(string text)
  # text is the string which will appear at the top of the
  #      viewing window, directly above the given timeline
```

5.3.3 Event Referencing and Extensions

While control flow and drawing primitives provide the basis from which figures are drawn, the actual drawing decisions are based on the event data. Events are defined in EDL and instances created via the injection mechanism. Each drawing routine, when invoked, has an assumed parameter, the invoking event. This event is referenced through the event name which distinguishes the different routines. For example,

consider the definition below.

```
message PointToPoint (50)
  {
    # Code to realize PointToPoint message-events goes here
  }
```

This routine is used to specify the drawing routine which renders "PointToPoint" events. The variable "PointToPoint" will contain the message-event which invoked the routine.

Conceptually, event variables are record types. The types are as defined in their respective EDL definitions. For example, consider the EDL definition of the "Point-ToPoint" message-event.

```
# EDL definition of "PointToPoint"
type message {
     PointToPoint(FromWho:timeline->nodeid,
                  ToWho:timeline->nodeid, TheMessage:buf);
     }
# Implicit VL type and variable which corresponds to
# message-event above
struct
   {
     Timeline FromWho;
     Timeline ToWho:
     # buffer types are not defined in EDL
   } *PointToPoint;
```

Event types are defined to be pointer types. To reference items within the record, the C "pointer reference" notation is used: '->'.

In addition to the event items defined by EDL, implicit items are also available to users. Chapter 4 discussed the event extensions which are provided by VL. The extensions are useful for locating a timeline in the history space, figuring out if an event has been filtered or not, and for retrieving the selection state of messages. Timeline-events are extended with an integer "x" field which monitors the horizontal location of the associated timeline. Message-events have an extra "selected" field, which is 1 when the message has been selected and 0 otherwise. Every time the user selects/deselects a message, its visualization routine is re-run. Finally, all events have a "hidden" field. Like the "selected" field, the "hidden" variable will be 1 when the

event has been filtered and 0 otherwise. As discussed in Chapter 4, message-events are not automatically filtered when associated timelines are filtered. The "hidden" field is useful in such situation.

5.3.4 Example Routines

A complete example of VL usage is provided in Appendix A. The routines provided in the appendix complement the usage model described in the next chapter. The example covers most of the functionality provided by Creatr and should be illustrative in understanding how VL is actually used to render timelines and messages.

5.4 Message Formatting Language(MFL)

MFL allows users to define symbolic mappings to message-based event data. This is done by writing a formatting routine for each of the message-types defined in EDL. Like VL, MFL has control flow support. Support for regular expression matching and report generation represent the main power of MFL. MFL variable declaration and typing is semantically differs from that in VL. In VL, variable declaration is C-like. All variables are declared at the beginning of a routine and given a specific type. In MFL, variables are dynamically declared and typed. If a variable is used which has not been previously used it is instantiated. Subsequent usage is based on the existing instantiation. Furthermore, the type of a variable is determined by usage. MFL supports three basic types: integers, strings, and buffers. Implicit typecast conversions cause variables to switch type. Finally, MFL has a single level of scoping. All variable, when instantiated, have a global scope.

5.4.1 Control Flow

MFL control flow statements are composed of "if", "for", and "switch". The syntax of "if" and "for" statements is identical to that of the C analogs. The "switch" statement is similar to its C counterpart. In MFL's "switch", regular expressions can

be used for caseline matching.

5.4.2 Regular Expression Matching

Regular expressions are the key to message formatting. They are used both by "switch" statements and a special "match" expression. Examples should help to clarify the syntax of regular expressions and their usage.

Consider the following regular expression.

```
first=. second=. 123 . next_two=(..) rest=.*
```

This regular expression will match any data that is at least six bytes long, with the third byte equal to 123. If the data matches, then five new variables are introduced. The first character of the data, which may be any character, is stored in the variable "first". The second character is stored in the variable "second". The third must be 123, but is not saved to any variable. The fourth may be anything and is discarded as well. The fifth and sixth characters are stored in the variable "next_two". Any remaining characters in the data stream will be stored in "rest". The '*' operator means to repeat the previous subexpression zero or more times. Thus, ".*" matches zero or more characters.

Now consider this example.

```
'H' <012 0x13 14 '5'> (('a' 98 'c') | {''def''})* $
```

This regular expression would never be used in a serious formatter, but is illustrative. It matches a byte stream that begins with the letter 'H' and whose second byte is either octal 12, hexadecimal 13, decimal 14, or the character constant '5'. The stream may end with two bytes. If the stream is longer, then there may be any number of byte triplets, either 'a' followed by decimal 98 followed by 'c', or the string "def". The '$' symbol matches the end of the data. There may be nothing else following the matching triplet sequence. This regular expression would match the following data.

```
'H' 0x13 'd' 'e' 'f' 'd' 'e' 'f' 'a' 'b' 'c'
or
'H' 012 'a' 98 'c'
or simply
'H' '5'
```

Other regular expression operators include '+' and '?'. The '+' operator is like the '*' operator, but matches one or more of the previous item. '?' matches exactly zero or one of the previous item. In other words, the prior subexpression becomes an optional match.

```
The regular expression: 'a' = 'b' ? $ matches:
    'a' 'a' 'a' 'a' or
    'a' 'b' but not
    'b nor
    'a' 'a' 'b' 'c'
```

5.4.3 Report Generation

Report generation is done through the "print" statement. This statement takes a series of print items. A print item is either a constant or an identifier. String constants are printed verbatim, while all other items are printed as a series of hexadecimal bytes. Items can be followed by a ':' followed by either '1', '2', or '4'. Such syntax is used to print hex bytes, words, or double-words respectively. In addition, formatting functions to print in ascii, decimal, or octal, rather than hexadecimal are available.

Consider the following formatter:

```
message PointToPoint
  {
   if (match(PointToPoint->TheMessage, text=(....) rest=.*))
     print(``I got one:  '' ascii(text) octal(text:2)
           decimal(text:2) hex(text:2) ``\n'')
  }
```

If "TheMessage" variable is bound to "Nifty-message" data when the formatter is invoked, the following report would be generated for display.

```
I got one: Nift 047151 063164 28265 26228 4e69 6674
```

A secondary report generation primitive called "label" is used to build a report which can be used by VL. The syntax of "label" is identical to that of "print". When the VL statement, "mf_getlabel", is called, the appropriate message formatter is invoked and the report generated by "label" calls is returned. The "label" statement was deemed useful because graphical visualization will often display a subset of the more detailed information provided by text display. To reduce code duplication in the

different configuration files, VL routines make calls to MFL routines when formatted text, based on internal event-data, is desired for graphical display.

5.5 Filtering Language(FL)

Filtering is a preprocessing stage to visualization. The filter removes events which the user is not interested in viewing. Filtering is accomplished by writing a filter in FL. The syntax and semantics of FL is identical to that of MFL. However, instead of report generation support, VL provides filter primitives. Also, while MFL is used to write formatters for message-events, FL is used to filter both message-events and timeline-events. As specified in the last chapter, only creation-events are filtered for timelines. If the creation-event is filtered, the timeline is hidden. If a timeline has a destruction-event which sets the end time, visualization of the destruction-event will only happen when the timeline has not been filtered.

5.5.1 Filter Primitives

The filter primitives are used to hide and unhide the display of timelines and messages. When a "hide" or "unhide" statement is reached, the state of the specific event instance is set accordingly. When the filter routine completes, visualization is called. If the final state reflects a hidden state, then the visualization is bypassed. Otherwise the event is rendered.

The following example gives a basic filter.

```
timeline Master
  {
   # hide all Masters who control queries except the
   # the one whose nodeid is 1
   if (match(Master->WhatIDo, {''Control Queries''})) hide
   if (match(Master->nodeid, {1})) unhide
   # since nothing else happens, all Masters who change
   # the database are by default shown
  }
message MasterBroadcast
  {
```

```
# first hide everything
  hide
# If the second byte of TheMessage buffer is
# 0x77 then don't filter the message
if (match(MasterBroadcast->TheMessage, . 0x77 .*))
  unhide
# finally, if any of my timelines are hidden, then
# I should be as well
if (MasterBroadcast->FromWho->hidden)
  {
   hide
   return # no need to go on
  }
for (i=array_size(MasterBroadcast->ToWhoAll),
     i >= 1,
     i = i-1)
   {
    if (MasterBroadcast->ToWhoAll[i]->hidden)
      {
       hide
       return # no need to go on
      }
   }
}
```

For convenience, both "hide" and "unhide" statements are provided. While any filter can be written using only one of the constructs, having both reduces complexity.

Chapter 6

Usage

An implemented copy of Creatr was evaluated by interfacing it to an existing target environment. This chapter details the process of actually using Creatr in the context of a target application. The target environment and the debugging model are described in some detail. An explanation of how Creatr was interfaced to the target is also provided. Interfacing Creatr to a target provided a foundation from which an evaluation was performed. The chapter concludes with brief comments pertaining to the usage model. A detailed evaluation is deferred to the next chapter.

6.1 Target Model

Writing software for embedded systems is a common task during development of electronic products. However, development in a target environment is difficult. Gaining direct control of embedded software is hard, but can be achieved with in-circuit emulators which link to workstations running symbolic debuggers. This is an expensive and inflexible solution because new emulators must be acquired every time the hardware platform is changed.

To reduce the costs of embedded system development, a common solution is to simulate the software in a workstation-based cross environment. This environment remains stable across multiple hardware platforms and provides a single platform for development tools.

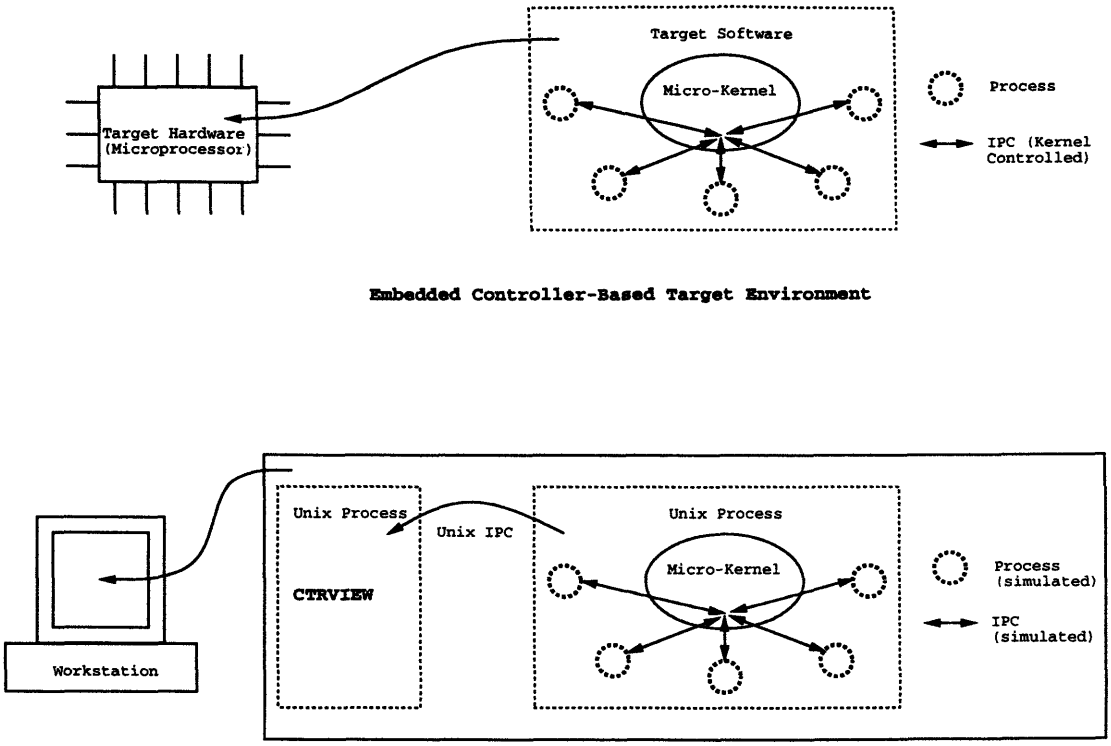


Figure 6-1: Target vs. Simulated Environment

Creatr was used as one of the development tools for such an environment. An embedded microkernel was already being simulated under Unix on the Sparc platform. The microkernel was multi-tasking with preemptive scheduling. The kernel also provided support for IPCs among the active processes. System development entailed dividing the functionality among many processes and using the IPC mechanism for communication and synchronization. Creatr was used to visualize the message flow among the processes. Figure 6-1 outlines the difference between the target environment and the simulated environment, and shows how Creatr was used in the latter.

6.2 Event Definitions

To interface Creatr to the simulated microkernel required developing an EventFlow mapping. Each of the processes was modeled as a timeline and IPCs as messages. Scheduling information was also desired. Task switches were modeled as a special type of message. Finally, operating system calls into the microkernel were mapped

through a common entry point. Engineers wanted to know what other operating system calls the processes were making aside from IPC calls. Thus, operating system calls were also modeled as messages.

Once the model was defined, event definitions were developed. The microkernel implemented IPCs using a mailbox model. Each process was assigned an identifier. Any process which needed to accept messages from other processes requested a mailbox from the kernel. This method was designed to allow for flexibility. A mailbox could be used as a common hub for multi-producer/multi-consumer relations. In reality, it was used as a hub for multi-producer/single-consumer situations. To send a message, a process would make a system call, giving the mailbox identifier and the message as the parameters. Receiving processes could retrieve messages from their mailbox through system calls.

After reading the kernel specifications for the various system calls and the scheduler, the following event definitions were developed.

```
type timeline (pid:int, mboxid:int) {
     process(name:string);
     }
type message {
     send_msg(send_flag:int, fromtm:timeline->pid,
              totm:timeline->mboxid, msg:buf);
     task_switch(fromtm:timeline->pid, totm:timeline->pid);
     os_call(tm:timeline->pid, msg:string);
     os_exit(tm:timeline->pid, msg:string, returncode:int);
     }
```

The event definitions above contain five different event types. One event signals timeline creation while the others signal messages. Each of the message-events is used to signal different target-level actions. The "send_msg" event is used to model IPC flow. The "send_flag" field tags the sub-action - sending to or receiving from a mailbox. Context switching causes a "task_switch" event to be generated. Finally, "os_call" and "os_exit" events capture system calls.

Based on the event definitions above, target hooks had to be added for each of the event types. At this point, a key observation was made. The application being developed on top of the kernel used a predefined, persistent set of processes. Instead

of embedding hooks for timeline events, static events were defined.

```
statics {
     (process, 0, 0, ''Task1'');
     (process, 1, 1, ''Task2'');
     (process, 2, 2, ''Task3'');
     .
     .
     .
     }
```

To embed hooks for the message events, the CTRLIB routine, ctr_inject_event, was used. The signature for the routine is straightforward.

```
void ctr_inject_event (char *event_id, ...)
     # event_id is the string encoding of a packet for a given event
     #        (as generated by CTRHDR).
     # ... is the variable length argument list of data comprising the
     #       packet defined by event_id.
```

An example of how this routine was used within the target microkernel is given below.

```
int system_ipc_send (int from_process, int to_mailbox,
                     void *msg, int msglength)
  {
   .
   .
                                  # send_flag = 1 implies sending message
    ctr_inject_event(send_msg, 1, from_process, to_mailbox,
                     msglength, msg);
                     # buf types require an associated length
                     # while string types are assumed null terminated
   .
   .
  }
```

6.3 Customizations

6.3.1 Visualization

Visualization required writing VL routines for each of the defined event types. However, the layout of the event types needed to be defined first. Figure 6-2 gives a

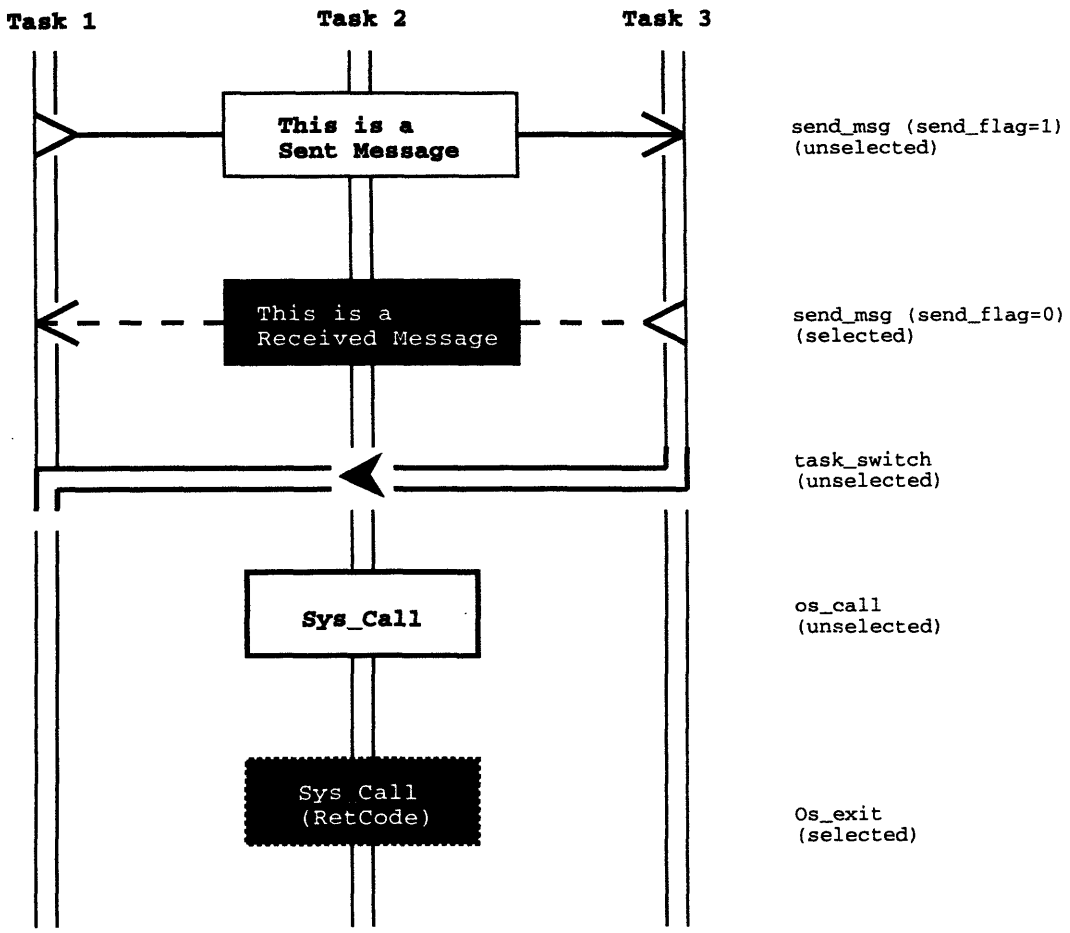


Figure 6-2: Graphical Realization of Events

picture of the template on which the visualization routines were based. Appendix A contains the complete listing of the visualization routines used to realize the template.

6.3.2 Message Formatting

Message formatting for this usage example was only needed for the "send_msg" event type. The "msg" field of this event type contained the actual message data that was being transmitted by the microkernel. The format of the data however, was defined by the application that was being developed on the microkernel. In general, each application could define its own protocol concerning the contents of a message. From a debugging standpoint, users were interested in seeing the content breakdown of the application level message - the data within the "msg" field. Thus, message formatting

for this usage model was dependent on the actual application that was being written. Consider the message breakdown as shown in Figure 6-3. For the example message shown, users would be interested in seeing the following textual presentation.

```
Protocol: Data Send
Message Number: 56
Control Data: 0x27
Channel Number: 166
Command Sequence: 0x826c
Data String: Testing 1 2
Checksum: 0x75
```

The following simple formatter could realize the formatting for the protocol described in Figure 6-3.

```
table protocol_table =
 {
  # default mapping starts at 0x00
        ''Null Message'',
        ''Clear Line'',
        ''Reset'',
  0xf0: ''Ready to Receive'',
        ''Data Acknowledgment'',
        ''Data Start'',
        ''Data Send'',
        ''Data Resend'',
        ''Data End''
 }
message send_msg
 {
  # First match the protocol byte.
  if (match(send_msg->msg, protocol=. rest=.*))
   {
    # See if the protocol really exists.
    if (??protocol_table[protocol])
     {
      print(''Protocol: '' protocol_table[protocol] ''\n'')
      # If its a Data Send type, then match the rest of
      # the message.
      if (protocol == 0xf3)
       {
        if (match(rest, msgnum=.... control_data=.
                        channelnum=. command_seq=........
                        data_string=............
                        checksum=.))
```

```
            {
             print(''Message Number: '' decimal(msgnum) ''\n'')
             print(''Control Data : '' hex(control_data) ''\n'')
             print(''Channel Number: '' decimal(channelnum)
                   ''\n'')
             print(''Command Sequence: '' hex(command_seq) ''\n'')
             print(''Data String: '' ascii(data_string) ''\n'')
             print(''Checksum: '' hex(checksum) ''\n'')
            }
           # If the match failed, let user know.
           else
            {
             print(''Data Send message not of Correct Length\n'')
             print(''Data is: '' hex(rest) ''\n'')
            }
          }
         # for non-Data Send messages, just print out the data.
         else
          {
           print(''Undefined format for given protocol\n'')
           print(''Data is: '' hex(rest) ''\n'')
          }
        }
       # For undefined protocol types, let user know and
       # print out the data.
       else
        {
         print(''Unknown Protocol Type: '' hex(protocol) ''\n'')
         print(''Data is: '' hex(rest) ''\n'')
        }
      }
     # If we can't even match the first byte, then message
     # is screwy.  Inform user and print out what there is.
     else
      {
       print(''Something went horribly wrong with the message\n'')
       print(''Data is: '' hex(send_msg->msg) ''\n'')
      }
    }
```

The example message formatter shown above captures the essence of the real formatter without introducing the complexities of the actual protocol. The embedded system was intended for used as a translation element for PBX switches. The card sat between two different buses, and translated packets from one format to the other.

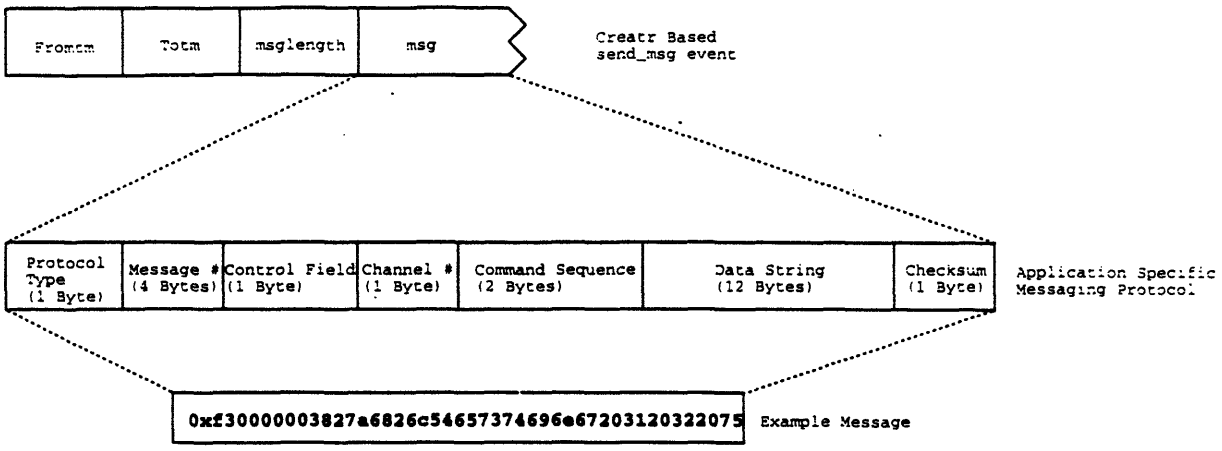


Figure 6-3: Example Internal Message Protocol

The message protocol of the internal data reflected the bus protocols.

6.3.3 Filtering

The event definitions and each of the customizations described above tend to be static throughout the lifetime of target development. If changes are required, they are usually applicable globally. On the other hand, filtering is dependent on the end-user, who actually uses Creatr to help debug the target application. Different users may be working on different sections of the application. The filter will change according to their requirements. As a result, the filter was not defined while interfacing Creatr to the target. Rather, users were allowed to define their own filter and then load it in during Creatr run-time.

The development of the target application was divided among a number of engineers, each responsible for certain subtasks. Each of the subtasks was isolated to a specific process or subset of processes. When engineers used Creatr, they complained of being flooded with messages unrelated to the task at hand. Since they were working with only a certain subset of the processes, they wanted to filter out all timelines mapped to non-relevant processes as well as messages among those timelines. To help do this, the following template filter was set up. Engineers could customize the filter

as needed.

```
timeline process
  {
   # For each process you are uninterested in, repeat
   # the following line, filling in the ... with the
   # name of the uninteresting process.
   if (process->name == ...) hide
  }
# all messages are hidden if any of their associated
# timelines are hidden
message send_msg
  {
   if (send_msg->fromtm->hidden ||
       send_msg->totm->hidden)
     hide
  }
message task_switch
  {
   if (task_switch->fromtm->hidden ||
       task_switch->totm->hidden)
     hide
  }
message os_call
  {
   if (os_call->tm->hidden) hide
  }
message os_exit
  {
   if (os_exit->tm->hidden) hide
  }
```

While this template was sufficient to please most developers, some were interested in creating more powerful filters. Content-based message filtering was desired. In particular, engineers wanted "send_msg" events to be filtered based on the content of the "msg" field. FL was identical to MFL, in that users could use regular expression matching to reduce scope. Enterprising engineers were able to build sophisticated filters which allowed them to view an extremely narrow portion of the message bandwidth.

6.4 Comments

Creatr was a success in the context of the usage model described above. Currently, over 20 engineers are actively using Creatr to help debug and maintain the software for an embedded system. It has proven to be instrumental in locating protocol and synchronization problems. Defining event mappings, adding hooks, and building customizations proved relatively easy.

Extended usage of Creatr reveals an interesting usage hierarchy. Creatr is used by two different groups - those who interface it to work with a given target and those who use the interfaced version to help with target development. The first group requires specialized knowledge about the infrastructure of the target and how Creatr works with it. The second need know only how to use Creatr's graphical user interface. As a result, a single person was assigned the task of becoming the administrator of Creatr for a given application. He/She would interface Creatr to the target and service requests from end-users for added functionality, modifications, etc.

The current usage hierarchy can be attributed to the high costs of customizing Creatr. For small projects, this represents a significant time commitment. In Chapter 7, Creatr design extensions are discussed. These extensions should help reduce customization costs and thus eliminate the usage dichotomy.

Chapter 7

Evaluation

While usage has been limited to a single application, a great deal has been learned from implementing and using Creatr. Discussed below are some of the interesting issues which were discovered during the development of Creatr.

7.1 Linguistic Support

7.1.1 Performance Constraints on VL

During implementation of VL and its interpreter, it became obvious that much of the customization flexibility desired would be at a high performance cost. Experience using a previous system-level debugger showed that typical target executions could result in over 10,000 events being generated, the large majority being of the message type. With Creatr's visual flexibility, a serious performance bottleneck during graphical realization of events would ensue.

First, Creatr was designed to allow messages to dynamically change the presentation of associated timelines. This means that if the user wants to view the state of the target near the time that message 9,000 happened, all 8,999 messages prior to that message would need to be processed by the interpreter. This happens because each of the message's drawing routines have the possible side-effect of changing the display of timelines. In other words, the current drawing state of timelines on the

screen is a function of all associated messages appearing before the current timeslot. This creates an unacceptable performance level for typical target runs. As a result, the functionality was abandoned during implementation. Instead, the graphical state of a timeline depends only on the event triggering its creation. A timeline's graphical state is composed of an outline, interior, and label.

Second, when a message is to be visualized, it gains control of the entire viewing screen. A mechanism is needed to give each message information concerning its time sequencing and how that relates to where it should begin drawing on the screen. Without such information, message 5 would have no way of knowing where to draw itself in relation to message 2. This was accomplished by internally keeping track of the y offset for each message. All primitive drawing routines supported by VL expected x coordinates in absolute terms and y coordinates in relative terms. This leads to the question of how the y offset for each message is determined. The original design called for each drawing routine to return the y height it used during realization. This height would be added to the existing offset and used for the next message. However, this meant that to get the y offset for message 9,000, all 8,999 prior messages needed to be rendered. Again, performance demands made this option infeasible. Instead, VL allows the user to statically denote the y height to be used for each message type. The assumption is made that the drawing routine will not draw outside this scope. However, the assumption is not enforced (i.e. a clipping rectangle is not implemented).

Finally, occurs the issue of how a user would be able to select/deselect a message using the mouse. One option states that if the user clicks on any location whose y value is within the offset denoted for a given message, then it should be assumed that the user has selected the message. A more likely scenario is that a user will select a message by clicking in an area where there are actual figures drawn for the message. Clicking in an area which does not have any drawings should not cause selection of a message. To accomplish this, an algorithm would be needed which can dynamically scope the bounding box of a message based on what has been drawn. This algorithm was not implemented due to time-constraints. Instead, a linguistic

feature was added through which the user can explicitly set the bounding box of a message. Since the y height of the box is statically set, only the x endpoints needs to be specified. If the user clicks in an area covered by the bounding box, then the message is selected/deselected, otherwise the mouse event is simply ignored.

7.1.2 Overall Linguistic Style

Creatr provided four different languages through which the user writes four different configuration files. In retrospect, having a single language provide all the capabilities may have been a better idea. Users would only need to write one configuration file. The question still remains of how to avoid the performance hit or confusion that results from having all the capabilities provided by a single language. The answer is that one of the features of the language should be that it breaks down the functionality it provides into separate non-interlaceable sections of the language. In other words, the new language should just be a conglomeration of the existing languages with consistent syntax and semantics. One possibility is to have keywords tag what customizations are being done.

```
types {
        # event type definitions here
      }
filter {
         # filter definitions here
       }
visualize {
            # visualization routines here
          }
formats {
           # message formatting here
         }
```

7.1.3 Expressiveness vs. Ease of Use

Creatr is designed to be customized linguistically. This requires users to learn four different languages. As mentioned at the close of Chapter 6, this has created a usage dichotomy. Specialists interface Creatr to targets and others use the interfaced version

for application development. Such a usage hierarchy may be acceptable for large projects requiring many developers. However, for small projects, the time invested in learning Creatr's languages represents a significant fraction of the development cycle. Small projects often only need simple customizations. In such situations, the expressiveness of the linguistic approach does not justify the reduction in ease of use. Future versions of Creatr should include a secondary, GUI-based approach to customization. The GUI builders should let users graphically specify some high-level subset of the customizations and then automatically generate appropriate linguistic definitions. If finer customizations are needed, users can learn the more expressive languages and make the necessary changes to the generated code.

7.2 Portability

Creatr has been implemented under Unix using X and Motif for graphical functionality. The CTRLIB library is C-based. Migrating Creatr to other platforms and interfacing non-C-based applications present two portability issues.

Creatr uses operating system support for IPC to implement event injection. Currently, Creatr uses Unix's piping mechanism. However, as mentioned in Chapter 4, a more general, distributed IPC subsystem should be added to Creatr. This layer may make use of OS-based IPC. If Creatr is rebuilt for a different OS, the IPC subsystem may need to be reimplemented. The current implementation of Creatr is highly modularized. Rewriting the IPC subsystem should entail localized changes. Migrating to different graphics platforms is more difficult. Different graphics packages make different assumptions about the programming model. Replacing the graphics package will require reimplementing Creatr's graphics module and possibly modifying the module's integration layer.

Currently, Creatr exports injection hooks for C-based target only. New CTRLIBs and CTRHDRs will need to be written to support targets written in other languages. Because CTRLIB and CTRHDR are decoupled from the rest of Creatr, multiple versions can exist. The IPC subsystem bridges the communication link between the

target hooks and Creatr.

The two portability constraints described above are minor. At a lower level, Creatr will always need to be rebuilt when migrating to new machine architectures. This may require code changes to reimplement features not supported by the new machine. Creatr has currently been built for three systems: Sun Sparc, HP Apollo, and Digital Decstation. Creatr was developed on the Sparc platform under BSD Unix. Migration to the other platforms required approximately thirty minutes each with minor code changes to deal with missing headers or libraries.

7.3 Application Domain

Creatr has only been built and interfaced against one application. This leads to the issue of whether it is actually useful against other applications. I believe that it is. The important point is that Creatr is only useful if the user can successfully map a target into the EventFlow model. This is most likely if the user already thinks of the application along such lines. However, there are various sets of applications which lie within the application domain of Creatr. A few of these sets are listed below.

Distributed Applications This includes the entire spectrum of distributed applications, from the low-end, multi-threaded software through the middle ground, multi-process programs to the high-end, host distributed systems. Distributed systems display the common feature of loosely coupled execution modules independently accomplishing subtask. Communication mechanisms are used to integrate the modules into a cohesive program. The mapping to EventFlow is apparent - execution modules are the objects and the communication medium the event-flow. However, for each of the subsets, a different mapping is needed. For multi-threaded programs, threads would be the objects. Communication in such an environment is commonly accomplished through shared memory using a semaphore model. The semaphore release by one thread and subsequent lock by another could be used to model event-flow. For multi-process systems, the mappings are a bit cleaner - processes are the objects and inter-process com-

munications are the event-flow. Finally, host distributed software would need to model each host on the network as an object and the network traffic as the event-flow.

Network Traffic Analysis Creatr could make a useful tool to analyze network protocols. The user would need to write a separate program which patches into a network and records all messages that travel the media (i.e. a snooper). The snooper would then pass on scanned messages to Creatr as message-events. The snooper could also dynamically create timeline-events by keeping a mapping of all new host addresses picked up from recorded messages. Similarly, different host addresses could be statically determined and static timeline-events created with the host addresses acting as the identifiers. Under this scheme, users could monitor and analyze network traffic graphically.

Simulated Operating Systems Creatr has already proven useful in an environment for cross-development of embedded, multi-tasking operating systems. Creatr could also be used as a standard testbed tool for the evaluation and development of new operating systems. Current approaches to operating system development involve simulating the operating system on top of an existing system and developing test application within the exploratory testbed. If the operating system is designed to provide system support for distributed communications, hooks should be added into this functionality. Users who want to test the experimental system can use built-in tools to help develop programs.

Object-Oriented Programming Visualization of the run-time control flow of object-oriented programs would be an interesting use for Creatr. Each object is modeled as a timeline and method calls are directed messages from the calling object to the called object. While this approach would make a novel debugging tool for object-based systems, it also presents difficult mapping issues. How is each object identified? How does a method call obtain a handle on the calling object? At what specific places are injection hooks embedded?

While each of the sets described above has a natural mapping into the EventFlow model, actually interfacing an application will require some thought. Users must pinpoint entry points in the target code where events are created and injected into Creatr. For some applications this is an easy task, as object creation or event-flow occurs at isolated instances within the code. Others, however, may spread the functionality over a large portion of the code. What the user would like to think of as an event is implemented in the code as a conglomeration of smaller functionalities. In this situation, the user will need to collect information from various pieces of the program until an event is completed and then inject the event. As a result, extra code on top of the hooks themselves may need to be added. This code would need to keep track of the execution state until an event has completely developed. If users are comfortable thinking about a system using an EventFlow model, modifying the system design to facilitate a clean map may also have the side-effect of reducing the complexity of the code itself.

7.4 Other Issues

While Creatr represents a less costly solution to generic system-level debugging, certain costs have not been completely eliminated. In particular, interfacing the back-end of Creatr to a target is not transparent or automated. Users must manually add event injection hooks to the target. This requires users to be familiar with the target coding scheme and the hook library provided by Creatr. A better solution would be to automate the hooking. Once users have defined event mappings, they simply run the target sources through an intermediary tool, which automatically adds the hooks at the appropriate places. At this time, it is unclear how exactly to do the automated hooking or even what extra information is needed to correctly place the hooks.

Another concern surrounding Creatr is that it is unidirectional. Users can use Creatr to view event-flow, but they cannot inject synthetic events into the target. A brief discussion on this issue suggests that in a system like Creatr, such functionality would be extremely difficult and would require further research. While building syn-

thetic events is not difficult, correctly injecting them into a target is not trivial. The target may have multiple hook points at which events are being injected into Creatr. How is the execution of a target suspended and the system state modified to reflect the injection of a synthetic event? At which hook point in the target is the event injected?

The prolonged usage of Creatr has made clear the benefits of an interpreted environment. First, the customizations are not platform dependent. If Creatr can be built for a platform, then the interpreter simply becomes an internal part of the executable. New compilers do not need to be written for each platform to which Creatr is to be migrated. Second, from an implementation standpoint, interpreters are less costly to implement. The implementor only needs to know about the functionality of the language. Knowledge about the underlying machine architecture is unnecessary. Interfacing Creatr to a target is an interactive process. When incremental changes are made to the customizations, the user does not need to recompile anything. Creatr simply parses the new configuration files and stores the tree at run-time.

While interpreted environments have their benefits, they are not without problems. In particular, interpreted code does not execute as quickly as compiled code. Performance problems constrained the functionality of VL. However, VL was modified due to throughput issues, not latency problems. A larger bandwidth of events can be rendered in a compiled environment. If a compiled environment is used, for runs of 10,000 messages, it may be feasible to re-introduce the removed functionality. But the problem has not been eliminated. Rather, the feasible upper bound on messages has been raised. If future targets begin producing 100,000 events, the performance problems discussed earlier will resurface. However, if flexible visualization if required and event traffic is reasonable, an incremental compilation approach may be a feasible alternative to an interpreted environment.

Chapter 8

Conclusion

Creatr attempts to show that generic debugging paradigms can be used to create specialized debugging environments for a variety of software systems. Specifically, Creatr lets users model system-level events using an EventFlow model. By providing a generic toolkit core, high levels of configureability, and low migration costs, Creatr presents an attractive solution to system level debugging.

So far, Creatr has been used in a limited context. It is being used to help develop software for embedded systems. Creatr was interfaced to the target environment with minimal effort and modest knowledge. The interpreted environment Creatr exports proved valuable in reducing latency costs associated with the iterative interfacing process. The interfaced Creatr is successfully being used by a number of developers. As new applications are developed on the embedded microkernel, Creatr should continue to be a valuable debugging tools.

8.1 Future Work

Creatr represents an initial foray into extensible system-level debugging. However, many of the problems Creatr addresses are approached from an academic standpoint. While the design has a solid framework, many improvements to Creatr are still possible and should be explored for a commercial-level debugger.

The most important area of improvement deals with the linguistic design of the

languages used by Creatr. As mentioned earlier, the four languages should probably be consolidated into one. Furthermore, added functionality should be explored. It may also prove useful to reduce the syntactical/semantical functionality of the language to that of C. The extra, Creatr specific, functionality can then be provided through simulated library routines. This will require users already familiar with C to only need to learn the extra functionality provided by the library routines.

Interactive debugging is the other key area for improvement. Creatr is designed for generic event-flow visualization. Viewing the state of programs is one powerful debugging paradigm. The other is being able to control that state. Future generations of Creatr should try to incorporate target control. Ideally, control should be generalized in the same way that Creatr has generalized viewing. Users should be able to define the control granularity. One possibility is to allow users to inject synthetic events into the target. This option was discussed in Chapter 7 with the conclusion that it remains a hard problem. Research into generic system-level control structures should be done before attempting to incorporate such functionality into Creatr.

Event ordering should also be addressed. While the usage model Creatr is being used for currently, does not suffer from ordering problems, other usage scenarios will almost most certainly need causality support. This is especially true for distributed systems, which are foreseen as the major users of Creatr.

Finally, Creatr's graphical user interface should be expanded to provide more functionality and gracefulness to the user. A professional product will have a redesigned menu layout, provide help facilities, and also expand on the available history management/viewing options.

Appendix A

Visualization Code

Listed below is the code used to realize the visualization of events for the usage example described in Chapter 6.

```
                    # need to allocate some y spacing for
                    # the timeline creation
timeline process (40)
  {
   set_outline(SOLID);
   set_fill(NONE);
   set_label(process->name);
  }
message send_msg (40)
  {
                    # Message formatter better suited for string
                    # manipulations based on message content.
   string label = mf_getlabel(send_msg);
   int boxwidth = 40;
                     # string types have two subfields: length
                     # and width of the bounding region.
   int boxlength = label->length + 6;
   timeline from = send_msg->fromtm;
   timeline to = send_msg->totm;
   int buflength;
   int headstartx, tailendx;
   int boxx;
   int boundx1, boundx2;
   # only realize message if both associated timelines are visible
   if (!from->hidden && !to->hidden)
     {
```

```
# first figure out which direction to draw in, and setup
# the appropriate boundaries
if (from->x < to->x)
  {
   buflength = (to->x - from->x - boxlength)/2;
   # when the length of the label is greater than the
   # distance from one timeline to the other, some special
   # stuff needs to be done
   if (buflength<=0)
     {
      headstartx = to->x - (to->x - from->x)/2;
      tailendx = from->x + (to->x - from->x)/2;
      boxx = from->x - boxlength;
      boundx1 = boxx;
      boundx2 = to->x;
     }
   else
     {
      headstartx = to->x - buflength;
      tailendx = from->x + buflength;
      boxx = tailendx;
      boundx1 = from->x;
      boundx2 = to->x;
     }
  }
else
  {
   buflength = (from->x - to->x  - boxlength)/2;
   if (buflength<=0)
     {
      headstartx = to->x + (from->x - to->x)/2;
      tailendx = from->x - (from->x - to->x)/2;
      boxx = from->x;
      boundx1 = to->x;
      boundx2 = boxx + boxlength;
     }
   else
     {
      headstartx = to->x + buflength;
      tailendx = from->x - buflength;
      boxx = headstartx;
      boundx1 = to->x;
      boundx2 = from->x;
     }
  }
```

```
        # before drawing, clear the area
        clear_drawstate;
        drawstate->fillpattern = SOLID;
        drawstate->foreground = WHITE;
        drawbox(boundx1, 0, boundx2 - boundx1, 40);
        # now draw
        clear_drawstate;
        drawstate->brushwidth = 5;
        # setup the arrow which is drawn at the beginning of
        # a brush stroke
        drawstate->brushstart = (70, 40, 20);
        # if this is a receive type message, set the brushpattern
        if (!send_msg->send_flag)
          {
           drawstate->brushpattern = DASHED;
          }
        drawline(from->x, boxwidth/2, tailendx, boxwidth/2);
        # breakdown starting arrow, setup ending tail
        drawstate->brushstart = NONE;
        drawstate->brushend = (75, 35, 15);
        drawline(headstartx, boxwidth/2, to->x, boxwidth/2);
        drawstate->brushwidth = 2;
        drawstate->brushpattern = SOLID;
        # Users should have some way of knowing if a message is
        # selected.  This is done by inverting the box containing
        # the label.
        if (send_msg->selected) drawstate->fillpattern = SOLID;
        drawbox(boxx, 0, boxlength, boxwidth);
        if (send_msg->selected)
          {
           drawstate->fillpattern = SOLID;
           drawstate->foreground = WHITE;
           drawstate->background = BLACK;
          }
        drawtext(boxx + 3, boxwidth/2 + label->width/2, label);
        if (send_msg->selected)
          {
           drawstate->foreground = BLACK;
           drawstate->background = WHITE;
          }
        # finally, setup the x scoping (y scoping statically set)
        bound_scope(boundx1, boundx2);
      }
  }
message task_switch (20)
```

```
{
 timeline from = task_switch->fromtm;
 timeline to = task_switch->totm;
 int boxx, boxlength;
 int boundx1, boundx2;
 int midpt;
 # only realize message if both associated timelines are visible
 if (!from->hidden && !to->hidden)
   {
    # drawing is based on relative locations of the two timelines
    if (from->x < to->x)
      {
       boxlength = to->x - from->x;
       midpt = from->x + boxlength/2;
       # clear the area
       clear_drawstate;
       drawstate->fillpattern = SOLID;
       drawstate->foreground = WHITE;
       drawbox(from->x - 5, 0,
               (to->x + 5) - (from->x - 5), 20);
       # now draw
       clear_drawstate;
       drawline(from->x + 5, 0, from->x + 5, 5);
       drawline(from->x + 5, 5, midpt - 10, 5);
       drawline(midpt + 10, 5, to->x + 5, 5);
       drawline(to->x + 5, 5, to->x + 5, 20);
       drawline(from->x - 5, 0, from->x - 5, 15);
       drawline(from->x - 5, 15, midpt - 10, 15);
       drawline(midpt + 10, 15, to->x - 5, 15);
       drawline(to->x - 5, 15, to->x - 5, 20);
       drawstate->brushwidth = 4;
       drawstate->brushend = (75, 35, 15);
       drawline(midpt - 10, 10, midpt + 10, 10);
       boxx = from->x - 5;
       boundx1 = from->x - 5;
       boundx2 = to->x + 6;
      }
    else
      {
       boxlength = from->x - to->x;
       midpt = to->x + boxlength/2;
       # clear the area
       clear_drawstate;
       drawstate->fillpattern = SOLID;
       drawstate->foreground = WHITE;
```

```
        drawbox(to->x - 5, 0,
                (from->x + 5) - (to->x - 5), 20);
        # now draw
        clear_drawstate;
        drawline(from->x - 5, 0, from->x - 5, 5);
        drawline(from->x - 5, 5, midpt + 10, 5);
        drawline(midpt - 10, 5, to->x - 5, 5);
        drawline(to->x - 5, 5, to->x - 5, 20);
        drawline(from->x + 5, 0, from->x + 5, 15);
        drawline(from->x + 5, 15, midpt + 10, 15);
        drawline(midpt - 10, 15, to->x + 5, 15);
        drawline(to->x + 5, 15, to->x + 5, 20);
        drawstate->brushwidth = 4;
        drawstate->brushend = (75, 35, 15);
        drawline(midpt + 10, 10, midpt - 10, 10);
        boxx = to->x - 5;
        boundx1 = to->x - 5;
        boundx2 = from->x + 6;
       }
      # selected task switches are filled in
      if (task_switch->selected)
        {
         clear_drawstate;
         drawstate->fillpattern = SOLID;
         drawbox(boundx1 + 5, 7, (midpt - boundx1 - 15), 6);
         drawbox((midpt + 15), 7, (boundx2 - midpt - 20), 6);
        }
      bound_scope(boundx1, boundx2);
     }
  }
message os_call (40)
  {
   string label = os_call->msg;
   int boxwidth = 40;
   int boxlength = label->length + 6;
   timeline tm = os_call->tm;
   int boxx = tm->x - boxlength/2;
   # only realize message if associated timeline is visible
   if (!tm->hidden)
     {
      # clear the area
      clear_drawstate;
      drawstate->fillpattern = SOLID;
      drawstate->foreground = WHITE;
      drawbox(boxx, 0, boxlength, 40);
```

```
      # now draw
      clear_drawstate;
      drawstate->brushwidth = 2;
      drawstate->brushpattern = DASHED;
      # selected messages have the label box inverted
      if (os_call->selected) drawstate->fillpattern = SOLID;
      drawbox(boxx, 0, boxlength, boxwidth);
      if (os_call->selected)
        {
         drawstate->fillpattern = SOLID;
         drawstate->foreground = WHITE;
         drawstate->background = BLACK;
        }
      drawstate->brushpattern = SOLID;
      drawtext(boxx + 2, boxwidth/2 + label->width/2, label);
      if (os_call->selected)
        {
         drawstate->foreground = BLACK;
         drawstate->background = WHITE;
        }
      bound_scope(boxx, boxx+boxlength);
     }
  }
message os_exit (40)
  {
   string label = os_exit->msg | " (" | os_exit->retcod | ")";
   int boxwidth = 40;
   int boxlength = label->length + 6;
   timeline tm = os_exit->tm;
   int boxx = tm->x - boxlength/2;
   # only realize message if associated timeline is visible
   if (!tm->hidden)
     {
      # clear the area
      clear_drawstate;
      drawstate->fillpattern = SOLID;
      drawstate->foreground = WHITE;
      drawbox(boxx, 0, boxlength, 40);
      # now draw
      clear_drawstate;
      drawstate->brushwidth = 2;
      drawstate->brushpattern = DOUBLEDASHED;
      # selected messages have the label box inverted
      if (os_exit->selected) drawstate->fillpattern = SOLID;
      drawbox(boxx, 0, boxlength, boxwidth);
```

```
    if (os_exit->selected)
      {
       drawstate->fillpattern = SOLID;
       drawstate->foreground = WHITE;
       drawstate->background = BLACK;
      }
    drawstate->brushpattern = SOLID;
    drawtext(boxx + 2, boxwidth/2 + label->width/2, label);
    if (os_exit->selected)
      {
       drawstate->foreground = BLACK;
       drawstate->background = WHITE;
      }
    bound_scope(boxx, boxx+boxlength);
   }
}
```

Bibliography

[1] Deborah A. Agarwal and Louise E. Moser. A Graphical Interface for Analysis of Communication Protocols. *Proceedings: The 20th Annual ACM Computer Science Conference*, pages 149–156, 1992.

[2] Peter C. Bates and Jack C. Wileden. High Level Debugging of Distributed Systems: The Behavioral Abstraction Approach. *The Journal of Systems and Software*, pages 225–267, December 1983.

[3] Thomas Bemmerl. The TOPSYS Architecture. *Proceeding: Joint International Conference on Vector and Parallel Processing*, pages 732–743, September 1990.

[4] Ken P. Birman and Thomas A. Joseph. Exploiting Virtual Synchrony in Distributed Systems. *Proceedings: 11th ACM Symposium on Operating Systems Principles*, pages 123–138, November 1987.

[5] S. Chaumette and M. C. Counilh. A Development Environment for Distributed Systems. *Proceedings: 2nd European Conference on Distributed Memory Computing*, pages 110–119, April 1991.

[6] Pierpaolo Degano, Roberto Gorrieri, Luca Zamboni, and Pierluigi Zanotti. The Kernel of a Graphic Environment for Analyzing Distributed Systems. *Proceedings: The International Conference on Parallel Computing Technologies*, pages 266–279, September 1991.

[7] J. Etkin and J. A. Zinky. Distributed Debugging: Network Analysis Tools. *Microprocessing and Microprogramming*, pages 307–312, January 1989.

[8] Mariano G. Fernandez and Sumit Ghosh. Ddx-LPP: A Dynamic Software Tool for Debugging Asynchronous Distributed Algorithms on Loosely-Coupled Parallel Processors. *Proceedings: IEEE International Conference on Systems, Man, and Cybernetics*, pages 639–644, 1991.

[9] Paul K. Harter, Dennis M. Heimbigner, and Roger King. IDD: An Interactive Distributed Debugger. *Proceedings: 5th International Conference on Distributed Computing Systems*, pages 498–506, May 1985.

[10] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, pages 558–565, July 1978.

[11] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, pages 471–482, April 1987.

[12] Larry L. Peterson Mishra, Shivakant and Richard D. Schlichting. Implementing Fault-Tolerant Replicated Objects Using Psync. *Proceedings: 8th IEEE Symposium on Reliable Distributed Systems*, pages 42–52, October 1989.

[13] Luis Rodrigues and Paulo Verissimo. xAMp: A Multi-primitive Group Communications Service. *Proceedings: 11th IEEE Symposium on Reliable Distributed Systems*, pages 112–121, October 1992.

[14] J. Scholten and P. G. Jansen. Distributed Debugging and Tumult. *Proceedings: Second IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 172–175, October 1990.

[15] R. S. Side and Shoja G. C. DPD: A Distributed Program Debugger for the REM Environment. *Proceedings: IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, pages 265–268, June 1989.

[16] Edward T. Smith. Debugging Tools for Message-Based Communicating Processes. *Proceedings: 4th International Conference on Distributed Computing Systems*, pages 303–310, May 1984.

[17] Edward T. Smith. A Debugger for Message-Based Processes. *Software - Practice and Experience*, pages 1073–1086, November 1985.

[18] Janice M. Stone. A Graphical Representation of Concurrent Processes. *SIGPLAN Notices: Workshop on Parallel and Distributed Debugging*, pages 226–235, January 1989.

[19] P.J. Venables and H. Zedan. Debugging and Monitoring Highly Parallel Systems with GRIP. *Microprocessing and Microprogramming*, pages 79–84, March 1989.