Debugging Programs in a Distributed System Environment

Peter C. Bates

COINS Technical Report 86–05
January 1986

Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 1986

Computer and Information Science

This research supported in part by the National Science Foundation under grants MCS-8306327, DCR-8318776, and DCR-8500332. And by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract NR049-041.

ABSTRACT

DEBUGGING PROGRAMS IN A DISTRIBUTED SYSTEM ENVIRONMENT

Peter Charles Bates

B.A., State University of New York at Buffalo

M.S., Ph.D., University of Massachusetts

Directed by: Associate Professor Jack C. Wileden

Debugging is an activity that attempts to locate the sources of errors in the specification and coding of a software system and to suggest possible repairs that might be made to correct the errors. Debugging complex distributed programs is a frustrating and difficult task. This is due primarily to the predominance of a low-level, computation-unit view of systems. This extant perspective is necessarily detail intensive and offers little aid in dealing with the higher level operational characteristics of a system or the complexities inherent in distributed systems.

In this dissertation we develop a *high-level* debugging approach in which debugging is viewed as a process of creating *models* of actual behavior from the activity of the system and comparing these to models of expected system behavior. The differences between the actual and expected models can be used to characterize errorful behavior. The basis for the approach is viewing the activity of a system as consisting of a stream of significant, distinguishable *events* that may be abstracted into high-level models of system behavior. An example is presented to demonstrate the use of event based model building to investigate an error in a distributed program.

Behavior abstraction and system understanding are characterized as problems in pattern recognition that must operate in a noisy, uncertain environment. Pattern recognition in support of behavioral abstraction is thus shown to be more than a

simple parsing exercise. A formal model is developed for event based behavioral abstraction which provides a basis for rigorous discussions of debugging as behavior modelling and forms a guide for implementing tools to support debugging in terms of events and higher level abstractions of system behavior.

A prototype distributed behavior recognition system which has been constructed to demonstrate and evaluate the feasiblity of the *EBBA* approach is described. The prototype toolset identifies a range of debugging tools useful for distributed systems. *Remote* debugging, *filtered remote* debugging with *preset actions*, *simple cooperative* debugging, and *distributed debugging* progressively increase the power of debugging agents at individual nodes by reducing communication requirements, increasing overall transparency of the debugging tools, and distributing debugging tool functionality throughout the system.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Software debugging is an activity that attempts to *locate the sources* of errors in the specification and coding of a software system and to *suggest possible repairs* that might be made to correct the errors. Debugging distributed software systems is a complex and difficult process due to the lack of adequate methods for debugging complex computer software in general, as well as special problems specific to distributed systems. While distributed computer systems enhance the capabilities of computing, they do so by introducing greater complexities due to their structural organization, their operational characteristics, and their potentially enormous size [Enslow78]. Traditional software debugging methods force users to deal with the complexities of systems from a low-level perspective through examination of uninterpreted computation-unit information [Elliot82], [Lausen79], [Model79], [SWAT82], [VAXDEBUG82]. The extant perspective on debugging was developed very early in the history of computing and does not heed the knowledge gained by software engineers in recent years which indicates that the key to dealing with complexity is through abstraction and problem decomposition [Dahl72].

Event Based Behavioral Abstraction (*EBBA*) is a framework for a high-level approach to debugging complex distributed software systems. Within the *EBBA* framework, errors in software are investigated by observing the identifiable interactions of the components of the software. The framework employs patterns of these observations to aid users to gain an understanding of important aspects of system behavior. *EBBA* characterizes system behavior as consisting of a stream of characteristic, atomic behaviors, termed *events*. Debugging methods based on *EBBA* use

this characteristic event stream to investigate erroneous behaviors exhibited by the system.

The *EBBA* framework is particularly useful for distributed software systems because of its reliance on events as a fundamental information unit and its use of abstraction to deal with complexity. Events possess uniform information carrying structure and their use as the medium of information exchange results in a high degree of independence from processor and software system architectures.

EBBA employs abstraction in a systematic way so that tool users may examine errorful systems in a top-down, detail-as-needed fashion. Tool users can focus their attention on suspected problem areas without being distracted by less relevant details and may vary their abstraction levels to match their understanding of system activity.

§1. View of Debugging

Our perspective on debugging is that it consists largely of building *models* of actual program behavior and comparing these to models of intended behavior. During this process, debugging tool users abstract portions of program text and artifacts of program execution into models which they deem accurately reflect the actual activity of the program. The constructed model of actual behavior is then compared to models of expected behavior held by implementors or users of the software. Errors in software are indicated by the behaviors responsible for differences between the model of actual software behavior and the model of expected behavior. Software errors have a manifestation or observable effect which is usually only a symptom of faults in the activities that were designed to produce useful outputs. The activities which led to the errorful outputs must be examined to determine why the actual behavior of the software components differs from the intended behavior.

During the process of creating and comparing models of the system behavior, only elements of the software deemed to influence the errorful behavior need be

incorporated into the model. The models of actual system behavior are obtained by *monitoring* the system activity through output producing probes placed into the software. Probes so inserted make the constituent behavior of the software visible and are intended to characterize some aspect of the behavior. The probe output provides the user with a set of behaviors to be incorporated into the actual behavior model and permits comparison to a model of how the software is intended to work. Models of intended behavior are derived from the user's understanding of what the desired functionality of the software is to be and knowledge of how that functionality is provided.

Monitoring is a fundamental component of debugging because it makes the activity of the system constituents visible and provides an alternative to the standard system outputs. Without some capability for monitoring constituent behavior of a software component, only error sources which are trivially obvious can be understood and corrected. As a result, all software tools intended to aid debugging include some technique for inserting probes into the software for the purposes of exposing underlying behavior.

Intervention in system activity is another important component of debugging. Intervention involves altering the system's behavior as a means of testing hypotheses about sources of errorful behavior. Intervention in system operation is useful for two purposes. The first is to slow the system, to allow a tool user to assess the current state of the system and consider information which has been gathered about its operation. The net effect is to bring the system activity and the tool user's comprehension of that activity into synchrony.

The other important use for intervention is to provide an ability to *experiment* with the system under scrutiny. Experiments are often performed by closely controlling execution of system constituents. Most commonly, a software component will be brought to an important point in its execution and then have some elements of its environment changed before proceeding further. A debugging tool that provides capabilities for intervention to experiment with the system must address three

issues: determining when the time is opportune to intervene, what elements are to be involved, and how the intervention is to be carried out (how to gain control and access the elements). Which subset of the available elements are to be affected is left to the user. The chosen subset will reflect the user's perception of the error under investigation.

Experiments serve several purposes. First, when the model of actual behavior is constructed, perturbing the behavior attempts to verify that model. If the perturbations produce results predicted by the model, then the user has found and understood a (perhaps) significant component of system behavior. Otherwise, more information gathering is necessary (possibly guided by the previous results). Another purpose for experimentation is to force the system from an incorrect into an assumed correct state, thus allowing execution to continue. This sort of technique is valuable when the tool user feels that identification of the possible error source has been made and the number of elements to be changed is manageable. In a variation on this, a user inserts new behaviors (or arranges to have their effects continually added) into system operation. This is accomplished either through the facilities available in the tools or by editing and rebuilding the system itself. Obviously, when changes affecting system operation result in more correct system behavior, the user has demonstrated understanding and possible repairs to the system.

It is important to contrast two notions which are related to debugging: those of testing and performance analysis. Software testing is intended to exercise software in such a way that if the software contains errors, they will be exposed. In contrast, debugging depends on knowing that errors exist. The primary direction of testing is to choose an input data set or sets which thoroughly exercises the system. The testing view of the system is through its normal functional outputs, whereas debugging requires the behavior responsible for these outputs to be illuminated. Testing *reveals* errors, debugging entails a search for their causes.

Performance analysis attempts to compare a working system with some standard to determine *how well* it works. Monitoring is an important component of perfor-

mance analysis. Performance analysis can point out bottlenecks, suggest areas for improvement and possibly predict when a software component might fail. However, performance analysis does not directly support a search for underlying causes of errorful behavior. Nor does it directly suggest corrections to be made to software so that it will work properly.

§2. *EBBA* as a High-Level Debugging Paradigm

Philosophy

EBBA characterizes a system in terms of the observable effects and interactions of system components as represented by events. *Primitive* events represent the finest observable granularity of a system's activity, generally that provided by the functional level the system defines for its users. In practice, different users of a system combine sets of system components in varying ways in order to provide new or virtual levels of system definition. These higher levels of definition are more problem domain oriented and/or represent decomposition of applications into manageable and coherent implementation units. This suggests that behaviors exhibited by a system can be partitioned into similar subsets and that these behavior subsets combine to produce required application functionality. Each subset represents a different *viewpoint* on system functionality. Each viewpoint is in turn describable by a specific set of primitive events.

The traditional debugging view of a system is through a succession of largely uninterpreted state vectors that hold the current progress of a computation. Within this traditional framework, the essential model building activity is accomplished by synthesizing elements of program state into behavioral models. In traditional debugging tools, users collect traces of statement execution and examine variables at important points of the execution to infer behavior. They must guess what the incorrect behaviors are, determine which pieces of state information will best illustrate these incorrect behaviors, then devise a plan for obtaining this information.

This model-gather-synthesize cycle is repeated until an understanding of what is in error is gained.

The traditional view of debugging is known as *break-examine* debugging; so named because of its principle monitoring and intervention technique. Break-examine debugging relies on a user's ability to set some kind of trap instruction in executing code with a view to gaining control of the computation. Whenever a breakpoint is encountered, the software component that executes the breakpoint is synchronously interrupted (the break) and control is transferred to a debugging routine that is aware of attributes of the breakpoint. The user is then free to examine virtually any component of the state vector associated with the suspended computation (the examine part). The efficacy of the break-examine techniques rely on the time-invariance property of sequential (non-real time) computations.

The break-examine snapshot view of behavior is replaced in *EBBA* by a view emphasizing observation of significant system transitions manifested as program behavior. This viewpoint is more complete, and transcends the step-by-step changes made to a system. *EBBA* encourages debugging tools that minimize the need to deal in computation unit details and favors system understanding through manipulation of behavior models. *EBBA* uses collections of events representing behaviors as a basis for high-level debugging. Models of system behavior are created in terms of the events forming a particular viewpoint on a system. Within the *EBBA* framework, the modelling nature of debugging is directly supported.

Recent programming technology has promoted abstraction [Guarino78] as a way to overcome complexity and aid in producing programs which work properly [Dahl72]. *EBBA* employs abstraction both to isolate a user from specific implementation details of a program and to make the modelling task more manageable. As considered in chapter II, *EBBA* may be used in a manner analogous to "stepwise refinement" [Wirth73] by using abstraction in a systematic way. Initially, a debugging tool user's attention is focused on undetailed but relevant high-level behaviors. Behaviors are repeatedly expressed in terms of their constituent behaviors until a

level of detail is achieved which permits comparison to the actual, errorful behavior. Through this systematic decomposition of behavior, debugging tool users can concentrate on isolating errorful behavior before examining program-level details that may be responsible.

Tools

The prototype *EBBA* toolset, described in chapters III, VI, and VII, is a collection of programs constructed to verify many of the ideas of this work, and to provide a basis for future work. The toolset is a highly modular set of components with well defined functions within the *EBBA* framework. The components themselves form a distributed program that provides various levels of debugging service by placing an appropriate complement of components at the nodes of a distributed system.

EBBA easily supports a distributed debugging tool consisting of a heterogeneous collection of cooperating debugging agents. From the *EBBA* perspective it is not necessary that all nodes that participate in debugging activity provide the same observation, abstraction, and intervention service levels. The service level provided by a node is based on the appropriate balance of local processing constraints at each node, overall needs of the debugging toolset in relation to the problems being investigated, and the degree of transparency required for effective investigation of the errorful system. This flexibility is achieved through exchange of high-level event information by cooperating nodes and through exploitation of load sharing characteristics possible in distributed systems. Where the information needs of debugging tools clash with processing autonomy or privacy of individual nodes, abstraction of activity into events serves to respect the boundaries required by distributed system components.

The toolset provides three functions necessary to implement the basic *EBBA* philosophy: tools to specify primitive behaviors and high-level event models, tools to maintain user defined viewpoints, and tools to recognize behaviors as they occur in the system. The Event Definition Language (*EDL*) and its compiler are the

important components for expressing behavior models. An Event Librarian helps a user maintain viewpoints on a system as well as any models constructed through that viewpoint. Construction of behavioral models based on a user's understanding of desired system functionality is only a first step to debugging a system. The models must be compared to actual system activity and their differences noted. The event stream characterization of system activity suggests the use of a syntactic pattern recognition [Fu82] approach to recognizing behaviors. The pattern recognition component of the toolset, found in the Event Recognizer, is based on a formal model for recognizing behavior in a distributed system.

Properties of EBBA

Recognized events may be placed into the event stream and subsequently incorporated into other event recognitions representing higher level behavioral models. This complicates event recognition slightly but it results in several advantages. Among them is the ability for a distributed debugger to exchange these high-level events among its components. This results in lowered communication bandwidth requirements and an increase in debugger transparency. Another is the ability to partially recognize behavior models and use this information to direct further debugging activity.

The intervention techniques which facilitate experimentation in break-examine debugging tools are not easily implemented within *EBBA*. In break-examine debugging tools, the issues involved in experimentation are readily addressed. The time to intervene follows the break that interrupts the program flow. Following the breakpoint interrupt, the debugging monitor generally has access to the entire procedure and data components of the computation. All that is necessary for experimentation is a capability to edit the storage locations holding the program and its active data space. The intervention techniques outlined in chapter VII explain the mechanism for intervention through *EBBA* but contains the caveat that *EBBA* is unable to detail *what* is to be affected by an experiment. Exactly what elements can be affected,

like primitive events, appears to be highly system dependent or event specific.

An important point concerning *EBBA* and the prototype toolset is that they are intended to work in real-time. Use of the tools should proceed roughly in step with the operation of the system rather than in a post-mortem fashion that analyzes previously collected system dumps or traces. The interactive nature permits tool users to shift viewpoints and constrast different views in response to changing system conditions.

EBBA is not intended to be used to create complete models of a system's behavior. Instead, the intention is to create small, focused models that aid in understanding the causes of errors. To effect this, users can create models of errorful *or* correct behavior and use appropriately constructed tools to compare these to actual behavior.

EBBA-based debugging tools will not provide an oracle with the ability to identify a section of program as the error containing piece. A user still needs to create the models and coordinate their use. Event based behavior modelling tools support model manipulation to encourage users to reason about behavior. The emphasis of behavioral abstraction based tools is on creating and exploring the models rather than on obtaining the information necessary for these purposes. A user employing behavioral abstraction based tools can describe models with low overhead and should be assisted in evaluating the goodness of these models by the provided tools. Also, like the structured programming methods, the systematic expression of behavioral models forces users to think in a "correct" fashion, thereby reducing the possibility they will expend effort in fruitless searches.

§3. Current Systems and Research

There has been little systematic research into software debugging. What are generally available are discussions of the merits of various implementations of debugging tools. Most debugging tools have been created to support the break-examine de-

bugging paradigm by providing facilities for examining and altering program state vectors and closely controlling program execution. Debugging distributed systems has been approached largely by extending traditional break-examine methods into a distributed system with a view to imitating the types of monitoring and experimenting that have been successful in the past.

The early days of computing were characterized by a hands-on approach to programming. Programmers generally had exclusive use of the machine and entered programs through console terminals or punched cards. Debugging at this time was done with the aid of the console lights and switches which could be connected to internal processor registers and memories. Programmers could examine the contents of memory by using the switches to set memory addresses and reading the values displayed in the console lights. In a similar fashion, parts of memory could be altered to modify data or to change the program itself. "Halt" instructions could be inserted into the code at important places to allow a programmer to gain control over the machine to apply these early monitoring and experimenting techniques.

These are the origins of the break-examine debugging technology that is dominant today. What has greatly improved is the ease of performing these operations and the presentation of information that is obtained. These debugging aids are generally integrated into the various compilers, linkers and other utilities that make up the programming environment. A tool user is presented with a consistent interface and predictable information presentation techniques. There are two main criticisms of using the traditional break-examine method as the vehicle for behavior modelling. The first is the low-level nature of its interaction with the system. Models are derived from details rather than having details bound to models. Selecting the wrong details could lead to a synthesized model which does not illuminate the system behavior of interest or one which is plausible, but only eventually proves to be incorrect. Also, the set of probes available and the presentation methods for gathered information are frozen into the tools and are, by definition, restricted to the computation-unit level. *EBBA* encourages a top-down style of modelling that

allows the modeller to notice differences between model and system at the highest possible level.

The second criticism, following from the first, is that break-examine encourages only ad-hoc modelling. Break-examine tools provide mechanisms for obtaining program state information rather than means for implementing a coherent modelling strategy. There is no provision for model manipulation or investigation as there is for state manipulation and investigation. Also, since models derived with break-examine tools lack consistent structure, their comparison is more difficult. In contrast, the event view has a uniform structure for its information and modelling strategy, and encourages the reuse of models.

§3.1 *Interactive Programming Systems*

Perhaps the most advanced break-examine technology is realized in Interactive Programming Systems (IPS) such as CoPilot [Swinehart74], Interlisp [Teitelman78], and the Cornell Program Synthesizer [Tietlebaum81]. In these systems, all monitoring and experimenting are performed at the level of the programming language that forms the basis for the IPS. Few specific debugging aids are provided since IPS users can quickly add debugging code or gain access to information using the facilities provided by the programming environment. An IPS allows very elaborate information displays and sophisticated filtering of information simply because it utilizes all of the power of the programming language it is embedded in.

CoPilot creates a programming environment in which a user is encouraged to deal with large, multiprocess programs using a total system view. CoPilot forms a very information intensive environment in which the relationships among various views of a system through its programs and its state information can be explored. Debugging is effected through the uniform screen editing interface to the system by which all interaction with CoPilot is accomplished.

The Interlisp environment is less information intensive but has more tools incorporated into the system to aid program development and debugging. The DWIM

(Do What I Mean) facility, an extensive set of breakpointing routines and a number of static analysis tools let the system deal with many smaller problems, leaving programmers free to reason about more difficult, higher level errors in their programs.

The Cornell Program Synthesizer incorporates PL/CS, which is an example of a structured programming language designed to encourage programmers to think in a "correct" manner, into an environment which enforces much of the programming philosophy of the language. The Cornell Program Synthesizer eliminates the possibility of creating many of the annoying simple errors that plague early program development but, not unexpectedly, has no facilities for dealing with higher level errors.

The IPS are perhaps the preeminent implementations of the break-examine technology. However, they still suffer from the basic flaws of this technology in that the user is required to synthesize behavior models from details obtained from low-level probes inserted into the system.

§3.2 *Postmortem, Trace-Based Debugging*

Early attempts at providing better behavioral information can be seen in the work of Satterwaithe [Satterwaithe75] in a batch oriented system and Balzer on EXDAMS [Balzer69]. Satterwaithe recognized the information overload created by higher speed output devices and incorporated techniques for information filtering through selective dumping of program state information. These dumps were heavily annotated with source-level program text to aid the user to easily assimilate the state and trace information. By juxtaposing state information with the statements that caused it, a reasonable behavioral model of program activity could be obtained.

The EXDAMS technique provided users with a history trace of system activity in which was recorded variable changes, choices made by control statements, subroutine calls, etc. This trace was to be used after program execution to simulate the execution with a view towards obtaining a more careful look at the details of the execution. EXDAMS contained aids for displaying static and dynamic information

and could be used in combination with user written routines to search for patterns of activity and graphically display the information obtained. The replay could be done at a pace determined by the user and could proceed in either a forward or backward direction. When an interesting point was discovered, the state information in the trace could be used to begin execution again from that point.

Similar to EXDAMS was IL [Cohen77]. The IL system accumulated trace information about a running program and included a regular expression based language to aid in searching for patterns of statement labels and values of variables. However, unlike EXDAMS, the uses for the information did not include a replay capability.

§3.3 *Primacy of Monitoring*

In a more recent work, done within the context of debugging complex artificial intelligence systems, Model [Model79] argues for higher level monitoring of system activity. Through "meta-monitoring," as introduced by Model, is recognized the need to observe behavior at more abstract levels than that provided by the implementation. His work largely attempts to provide monitoring tools which give users information about system activity instead of simply displaying program implementation level information. Normal application-level interactions with the system being debugged and a stream of information about its progress on its processing goals form the basis for all debugging activity.

§3.4 *Towards more Machine-oriented Tools*

The theme of having a large number of machine based tools to aid in debugging is important in the KRAUT debugger [Bruegge83] for Path Pascal [Campbell79] programs. Surrounding a user in the KRAUT system is an array of tools similar to those provided in programming environments. As a basic debugging environment, KRAUT contains a sophisticated set of debugging commands in the break-examine tradition. However, in a significant departure from the break-examine tradition, KRAUT allows users to describe certain behavioral patterns that might be of interest

to a Path Pascal user. Using an extended Path Expression formalism [Campbell74], [Andler79], a user can describe the expected behavior of a section of program in terms of its access to objects defined in the program. Facilities are provided to automatically carry out a series of debugging commands whenever the expected behavior matches or is violated by actual system activity.

The SPIDER debugger implemented in [Smith81] is intended to aid the debugging of programs which are written as a collection of loosely coupled cooperating processes. To this end, users of the SPIDER debugger specify "demons" containing debugging commands which alter interprocess events and message traffic. The demons are set into execution when an interprocess event having the appropriate characteristics occurs.

Although SPIDER does not allow a user to examine or affect intraprocess behavior directly, the power to affect interprocess behavior through the event triggered demons is considerable. The interprocess events used to initiate demon activity are a small, predefined set with three primary classes: debugger events, system kernel calls and message transmission events. All debugging activity in SPIDER is performed through the activation of the demons and their power to affect the interprocess events. No specific monitoring and presentation facilities are provided, but complex arrangements of demons and presentation processes to carry out these functions are conceivable.

§3.5 *True Distributed Program Debuggers*

Interprocess communication as the prime focus of attention for multiple process programs is used by Schiffenbauer [Schiffenbauer81] in a debugger for interactively debugging distributed programs. Tools developed by Schiffenbauer provide a user with considerable capabilities to alter and create interprocess communications and simulate transmission errors. The processes that make up the program cannot be directly manipulated, but the pattern of the interprocess communication can be controlled as a user desires. This effectively gives implicit control over the processes

making up the application. The use of this facility for debugging creates a simulation of the program being debugged because the amount of control exerted radically changes real time relations and patterns of process interaction. This simulation is demonstrated to be correct so long as certain simulated clock conditions are maintained.

BugNet [Curtis82] is another debugging tool developed specifically for a distributed system. BugNet, which operates within the MICROS [Wittie80] system, consists of a user interface, a database system and local event monitors at each node. Users reside at a debugging node and issue commands through the user interface that affect system processes, nodes or groups of processes. There is a predefined set of process and node events that are used to trigger actions containing control or display commands. It is the task of the local event monitor to detect these events and notify the system of their occurrence.

The database is created through the maintenance of a per process "silo" which stores message, event and state information for the process. All message traffic and event occurrences for a process are written into a data silo when this information is delivered to the process. In addition, each node periodically dumps all of the variables of all processes that reside at the node into the individual process' silo. When difficulties are encountered, the silo may be used to replay affected processes for purposes of observing where the errors occurred. Since a large amount of information may potentially be dumped into a process' silo, BugNet provides a mechanism to allow a user to select what information will be written into the silo during execution.

Another trace-based debugger for distributed programs is described by Garcia-Molina [Garcia81] and was implemented in the context of a distributed transaction processing system. In the approach developed, each node participating in debugging keeps a history (trace) of important events produced in the course of process execution. The collection of these traces forms a distributed database. Using a suitable relational database query language, the user can interrogate the database to learn about behavior and data states that may have had bearing on detected errors.

§3.6 *Event Based Performance Analysis*

Although performance analysis is not here considered to be debugging, some recent performance analysis work with distributed systems should be mentioned because of its emphasis on event-based information gathering.

METRIC [McDaniel77] was developed in a very complex minicomputer network in which different machines contain tailored versions of different operating systems. A METRIC implementation contains *probes* in the target system that continuously report events to *accountants*. An accountant performs filtering of events by only recording those events which have been "enabled". The recorded events are analyzed and tabulated by an *analyst* program that is constructed, on an as-needed basis, to illuminate particular behavior trends. Two results related to METRIC lend support to the debugging paradigm developed from *EBBA*. The first is the standard event message employed by METRIC to provide a high-degree of system independence, thus allowing it to work in a heterogeneous environment. Second, event filtering reduces the amount of information that must be dealt with to obtain meaningful interpretation of system activity.

In [Snodgrass82], significant issues related to collecting performance analysis data on a multiprocessor are explored. Sensor (probe) interference in system operation and correlation of sensed events to their delivery at the analysis site are important themes considered. Sensor performance is optimized by an attempt to minimize sensor activation overhead and reduce interference due to collection of the created event records. Event records are prepared and left in a *receptacle* – a container holding a small number of event records. Periodically, the receptacles are emptied and their contents reported to the performance analysis monitor. The caching of events in a receptacle, while greatly affecting the timeliness of event information, helps reduce the effect of the event gathering tools on system operation.

Consistency between an event record at the sensor and the performance monitor's view of the event, is handled mechanically through a sensor description file (SDF). SDF's are processed in the compile-link-execute cycle of system building and result

in addition of needed event generation code to target programs.

§4. Contributions

The important contributions of this work result from direct support of the modelling nature of software debugging. Most prominent is development of Behavioral Abstraction as a systematic approach to debugging complex systems. This step toward placing debugging on a level with other software development practices, employs a formal application of top-down, hierarchical construction techniques which have been shown by software engineers to be valuable for program development.

Behavioral abstraction completely updates the nature of the debugging process. Traditional debugging techniques only allow a tool user to observe a system from a detail-laden perspective with little aid for organizing a search for erroneous behaviors. Behavioral abstraction directs tool users to approach their search for errors in a step-wise refinement manner. Instead of viewing a system's behavior in terms of the details of its implementation, behavioral abstraction tools allow users to examine system behavior through abstraction levels that match the functionality provided by the layers of system implementation.

The behavioral abstraction approach introduces techniques to master the complexity of errors in large systems. Traditional debugging tools provide no means for dealing with complexity in a system. Behavioral abstraction aids in dealing with complexity by partitioning system behavior into viewpoints related to functional divisions with a system and allows users to describe the interrelationships of behaviors that produce effects in the system. Behavioral abstraction encourages users to ignore computation details that are unrelated to the errors they are investigating and to focus their attention from high-level perspectives down to lower levels as the relation of the complex interactions of system components to errorful behavior is understood.

The behavioral abstraction approach is particularly suited to distributed sys-

tems. The abstraction capabilities of behavioral abstraction are important to understanding the behavior of systems involving the cooperation of tens, or hundreds of processing elements. The uniform use of an event based view of system behavior results in a highly implementation independent way to view system activity. The simple behavior modelling paradigm supported by *EBBA* permits varying levels of debugging service from different nodes of a distributed system. The debugging service provided by a node can be tailored to match capabilities and needs of a heterogeneous distributed computational environment, without altering the fundamental behavioral view of the system or providing special-case functions.

Pattern recognition and analysis techniques are employed in a new role to illuminate and interrogate behavior patterns in system activity. The concurrency and timing issues that are important to distributed system behavior are considered by the shuffle automata model which is developed to deal with the unique problems of pattern recognition when applied to behavior recognition. The Shuffle Automata formalism is quite valuable for describing concurrent as well as sequential behaviors in terms of the event based view promoted by behavioral abstraction. The hierarchical model descriptions possible with shuffle automata easily account for behavior ascribed to heterogeneous distributed systems.

All of this groundwork is applied to a prototype implementation of a set of tools for debugging distributed systems. The basic toolset implements the event based view of system activity in a distributed environment. Toolset development has indicated what components are essential to this high-level debugging paradigm and how much tool is required at each system node to support mixed levels of debugging activity. A result of the tool development that should not be overlooked is its reinforcement of the notion that *EBBA* is independent of the implementation of systems it is being used to debug. The tools have been used to observe behaviors in two very different systems: the Vehicle Monitoring Testbed (VMT) [Lesser83] and the VAX/VMS operating system [Kenah84].

§5. Outline for the Sequel

Chapter II explains how debugging is accomplished through models of behavior. *EBBA* is more fully explained as a combination of behavior specification and behavior matching techniques. The view of systems through events that represent system functionality is explained and a simple notation is developed to describe events and higher level behaviors. A distributed program containing an error is described and using the notation developed previously, a set of models is constructed that are useful in finding the error. By way of this example, the important capabilities of tools needed to support *EBBA* debugging are outlined to be examined in later chapters.

Chapter III describes the Event Definition Language (*EDL*) which has been developed to support the event-based view of systems. *EDL* provides the means to construct behavioral models with minimal overhead and encompasses the descriptive machinery necessary for *EBBA*. *EDL* is a descriptive tool; it facilitates description of primitive events representing fundamental system behavior as well as high-level events representing models of higher level system functions. Note however, that use of *EDL* to describe activity is separate from creation of instances of that activity.

Chapter IV is a discussion of the problems with recognizing higher level behaviors in the system being debugged. A modification on the basic syntactic pattern recognition technique is introduced as the method for abstracting system behavior and sets a base for tools needed to use event based behavioral abstraction for high-level debugging. A discussion of primitive events and techniques for their creation in several systems is presented. An important activity connected with syntactic pattern recognition is the selection of the pattern primitive elements that compose more complex patterns. Likewise, behavior recognition depends on the creation of the primitive event representatives for system behavior. The final sections of chapter IV outline possible ways to obtain the primitive events representative of system behavior, and the costs in terms of debugging tool transparency.

Recognition of behaviors in a distributed system proves to be more than using simple finite state automaton to recognize strings of events. Chapter V presents

a formalism, the *Shuffle Automata,* useful for characterizing behavior recognition through an event based view. The formalism addresses the problems posed for syntactic pattern recognition in support of behavior recognition and provides guidelines for solutions to those problems. The shuffle automata description includes a procedure for translating an *EDL* event description into a shuffle automata. This procedure demonstrates the correspondence of *EDL*-defined behavior models to shuffle automata capable of recognizing the occurrence of the events representing the model.

The previous two chapters (IV and V) are the foundation for a collection of tools to be used to implement debugging from the behavioral abstraction viewpoint. Chapter VI presents the design of such a tool set. This tool set was implemented as a highly modular distributed program capable of providing the varying levels of debugging service necessary in a heterogeneous distributed system. The tools include a compiler for *EDL* descriptions, librarians that maintain viewpoints on a system, and a high-level event recognizer. This tool set attains a high degree of system independence and relies only on an ability to insert event gathering probes into the system it is to monitor.

Chapter VII discusses distribution of the debugging task and presents techniques that assist intervention actions. Debugging distributed programs is characterized as remote debugging or distributed debugging. The *EBBA*-based toolset from chapter VI is first extended into a distributed system as a remote debugging tool. Remote debugging is shown to be deficient in a number of ways. The deficiencies of remote debugging are overcome in stages that increase the monitoring capablities of each remote debugging node. Distributed debugging places more debugging burden on nodes that are participants in the debugging task to increase the overall power, transparency, and accuracy of debugging activity.

Finally, a summary of the dissertation, conclusions that may be drawn from it, and suggestions for further research are presented in Chapter VIII.

CHAPTER II

DEBUGGING THROUGH EVENT-BASED MODELS OF BEHAVIOR

An important goal for Event Based Behavioral Abstraction is to provide a system independent, high-level paradigm for debugging distributed programs. The approach implements this by providing direct support for the modelling activity that is involved in debugging. Modelling in the debugging context means that debugging tool users do not inspect every system state, or every variable, or every line of code contributing to a program and its execution. Rather, they select a relevant subset of program elements and use the selected items to understand overall behavior. In searching for complex or subtle errors, this subset of items will change when some items are judged not relevant and others take on significance.

In [Weiser82] are the results of some studies indicating that programmers employ mental "slices" of programs [Weiser79] when searching for errors. A program slice is a subset of the statements of a program that have bearing on a particular statement in the program. A program slice begins at the statement where an error has made itself known, e.g. a point where the program aborts. By working back from this point and including only statements which influence the errorful behavior, a program slice is made. A program slice made this way may be executed for purposes of only viewing behavior related to the statement of interest. Slicing is easily a form of modelling.

The *EBBA* approach is different from slicing in that models are expressed in terms of the primitive and high-level behaviors that occur in the system. Slices are not abstractions of system activity and users must still examine computation details to synthesize behavior models. Slices are valuable because they can be derived

mechanically, they reduce the amount of information to be examined, and, with proper tool application, users need only examine sub-slices that are most likely to influence errorful behaviors.

While slicing appears to be an attractive alternative to traditional tools it still provides only narrowly defined computation-level viewpoints. *EBBA* is better suited to dealing with increases in complexity and the interactions of autonomous, asynchronous components as are found in distributed systems. A program slice would detail what parts of a large program to subset, but their distribution to multiple processing sites and the asynchronous nature of their interactions would create difficulties for executing a slice.

EBBA provides a framework that directly supports modelling through an amalgam of components which supply two fundamental services:

- *Model specification* provides rules for combining behaviors into models and bounds the explanatory power of *EBBA* based tools,
- *Model matching* fits the activity of a system under investigation to the behavior models which are developed to explain system behavior.

Debugging tools that implement *EBBA* minimally need to supply components which support these activities as well as provide ways to obtain the events which are the basis for the technique. This basis toolset might be augmented with tools having the capability to exploit information collected during specification and matching.

This chapter will largely concentrate on model specification and will use examples to indicate what capabilities are useful in model matching. The next section will introduce the event based view of system activity. The information carrying content of events is detailed and given context within the event streams which reflect the dynamic behavior of a system. Following this is an explanation of how models of behavior are created and used to view system activity. The last sections present an example using event based behavior models to track an error in a distributed program and point the way to the capabilities needed in model recognition tools.

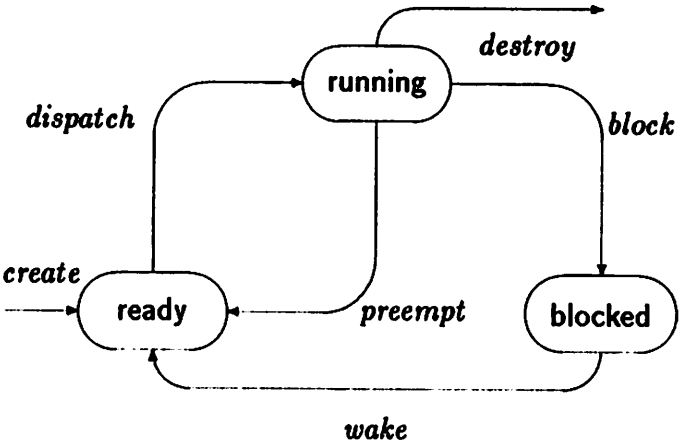


Figure 1: Process management state diagram

§1. Events, Event Streams, and Behavior

EBBA characterizes overall system behavior as a *stream* of observable, more simple, behaviors. The finest granularity of system behavior that is observable is represented by the *primitive events* which are characteristic of a viewpoint on a system. The selection of behaviors to express as primitive events is determined by the functionality provided by the system. For example, the classic process management view of operating system functionality provides six major functions: create process, dispatch process, preempt process, block process, wake process, and destroy process [Graham75], [Deitel84]. These functions cause a process to enter and leave a system and make transitions through the process state diagram of figure 1. For a user of these system functions, this is all there is to the system. The process management view of system behavior can be represented by six primitive events, corresponding to the occurrence of behaviors resulting from the six process management system calls which are made by or on behalf of a process as it changes states. A user of process management services could observe the lifetime of a process as it traverses the process state diagram by observing the sequence of events produced.

For example, the proper behavior of a simple process that is created to read and

transfer a single file block might be described by the following sequence of event instances:

create,	– –process given life
dispatch,	– –now running
block,	– –I/O issued to read file
wake,	– –read completes
dispatch,	– –now to write file
block,	– –waiting for write I/O completion
wake,	– –completed transfering block
dispatch,	– –to destroy itself
destroy.	– –process disappears

The user of process management services only needs to see this level of function. While there is indeed a great deal of work being performed below the process management level to effect the functionality, the user of the process management functional level must assume it works properly – this is the observable level of behavior.

Each event generated in a system belongs to a particular *event class* which contains like-named members. An event class represents some functions provided by a system or the transition to some state by a component of the system. As a system executes, *instances* of the event classes that characterize its behavior are created and form the observable event stream, as in the above process management example. Observing an event instance indicates that an example of the behavior represented by the class has occurred. The event classes in the above example are *create, dispatch, block,* etc., which correspond to the system management functions. In the example there are three *dispatch* event instances, two instances each of *block* and *wake* events, and one instance from each of the *create* and *destroy* event classes.

Individual event instances from an event class may be distinguished from each other by a set of *attributes* possessed by each event. All events within a class will have the same number and type of attributes. However, different instances of events from the same class may bind different values to corresponding attributes. Two attributes, time and place of occurrence, are possessed by all events, and provide temporal and physical locations for the event instance. Using this class and attribute characterization, all events can be represented as a tuple having the form

(EventClass attribute$_1$ attribute$_2$ ... attribute$_n$).

The actual, realized event instance might be represented simply by a string of machine readable bits, but it is an easily solved translation problem to extract the tuple form from an instance. (The topic of event form and generation is covered more in chapter IV.) Each event class is described by a *template* which includes the class name and the list of fields for its attributes. The templates for the process management event set might be

(create, time, node, creator, processid, processname)
(destroy, time, node, processid, destroyer)
(dispatch, time, node, processid)
(preempt, time, node, processid)
(block, time, node, processid, reason)
(wake, time, node, processid, wakeupstatus, waker)

The event stream to be used for *EBBA* may be thought of as a collection of distinguishable event instances represented as tuples. Each tuple has values appropriate for its class bound to its fields.

The event instances composing the stream are only partially ordered and may arrive at an observer simultaneously or in an order which does not reflect their actual occurrence. The *time* attribute of each event may only be relied upon to order events occurring at a single node of a multiple node system. This is the most natural view of a distributed system, so the specification and matching tools must account for this indeterminant behavior.

One final note concerning the nature of event tuples. An event class can represent simultaneous holding of a set of conditions or a transition from one holding to another [Holt70]. This categorization of events suggests different roles for the time attribute of an event. Events resulting from a set of condition holdings might have a time attribute which consists of two parts, a start time and a finish time. Transitional events would have a more instantaneous time associated with them – the time the key transition was made. Of course, the former can be represented by two transitional events, one to signal the start and one at the finish time. Events

employed by *EBBA* can be of arbitrary duration and complexity and thus may represent either category of event. In the use of time by *EBBA*, the time attribute designates a single, appropriate point in time, determined by the event generator.

§2. Clustering, Filtering and High-Level Events

Primitive events form a first level of abstraction of system behavior. However, simply observing a stream of low-level events is not very different from watching a trace of program execution. There will be much uninterpreted, seemingly unconnected, system activity; some relevant to a problem under investigation; some not relevant.

EBBA is designed to provide tools which aid a user in selecting only relevant activity to be modelled, at abstraction levels which match the user's understanding of system behavior. To effect this, *EBBA* defines two techniques, *clustering* and *filtering*. Clustering is used to combine primitive and previously defined events into aggregates representing higher level models of behavior. Filtering serves to eliminate from consideration events that are not relevant to a behavioral model being investigated. These two techniques work together to enable users to create and manipulate abstract models of system behavior. Users employ tools based on *EBBA* to investigate errorful systems by tinkering with models representing system activity rather than system implementation details.

Clustering is the means by which high-level behavior models are created from less complex behavior abstractions. The concept is simple: specify a collection of event classes and describe how they relate to each other to model a higher level behavior. High-level behavior models specified this way may be represented as event classes in their own right and subsequently may be incorporated into other high-level behavior models.

Filtering is realized in two ways: coarse filtering, which is closely related to behavior model structure; and fine filtering, through which users focus the intent of

a model. Selection of an appropriate set of member events for a cluster constitutes a coarse filtering of system activity. Coarse filtering specifies that instances from only certain event classes, those selected as cluster members, are eligible to be constituents of a particular high-level behavior model.

Fine filtering is effected by specifying relationships among cluster members. The relationships of event classes forming a behavior model are expressible in terms of temporal constraints designating acceptable orderings, and relational constraints among attributes defined by member events. The temporal and attribute relations help detail the structure of a behavior model and are used in a model matching role to guide recognition of the high-level events representing the model (chapters IV and V).

Temporal relations for a cluster are specified by an *event expression* defined over the event classes which have been selected as cluster members. Event expressions are similar to regular expressions [Hopcroft69], event expressions as in [Riddle76], and constrained expressions [Wileden78]. The temporal relations are indicated by a set of regular expression-like operators which specify acceptable orderings for instances of the member events.

Here we introduce some notation that will be useful for describing event expressions and relations among events in a cluster. The notation is described without rigor and will be used informally to describe event based models in this and later chapters. The notation is nearly identical to the event expression notation of the Event Definition Language described in chapter III but is less syntax laden.

An event expression is constructed from event symbol operands and explicit operators which indicate acceptable orderings that instances of the symbols might have. The event symbols correspond to the event class name field of an event template. Event operators include binary operators describing sequencing, choice, and concurrency and postfix operators for repetition. An event symbol, e, is an event expression. Sequences of events are denoted by the catenation operator, designated by " o ". An event expression written e_1 o e_2 indicates that e_2 occurs after e_1.

Choice among two events is indicated by the alternation operator, denoted by " | ". An event expression written $e_1 \mid e_2$ indicates that occurrence of **either** e_1 **or** e_2 will match the event expression. Similarly, concurrency is designated by the binary shuffle operator, " $\triangle$ ", which denotes that an acceptable event string is formed by any interleaving of its operand event classes. Repetition is designated by a unary postfix operator "+" which indicates that a series of one or more of its operand event classes is needed to match the model, or when designated by "*", zero or more instances of its operand event are required. In addition to these operators, parentheses may be used to group subexpressions in the normal fashion. Event expression operands may have simple exponents to indicate a bounded repetition.

For example, the event expression describing the file block transfer process from the previous example might be written

$$\textit{blockmove} = \textit{create} \circ \textit{dispatch} \circ ((\textit{preempt} \circ \textit{dispatch})^* \circ \textit{block} \circ \textit{wake} \circ \textit{dispatch})^2 \circ \textit{destroy}.$$

This expression indicates that the *blockmove* event begins with *create* process and *dispatch* process events that bring the process into existence and schedule it to run; followed by two strings that may include several *preempt* and *dispatch* events (which would normally result from quantum rundown) and a sequence that includes *block*, *wake*, and *dispatch* events as it performs I/O; followed finally by the *destroy* event that indicates the process has completed and been removed from the system.

Fine filtering in terms of event attributes more narrowly focuses a behavioral model by declaring that an event must have certain properties if it is to be included in the model. In this way, filtering based on event attributes serves to reduce the volume of information that needs to be considered for a particular behavior model. For example, when considering the process management view, proper operation requires that for every *preempt* there is a *dispatch* of *some* process. To reduce the number of *dispatch* events considered for inclusion, it is necessary to restrict the *dispatch* events to those whose *processid* field is the same as the *processid* of the *create* event which begins the higher level *blockmove* event. A further narrowing of

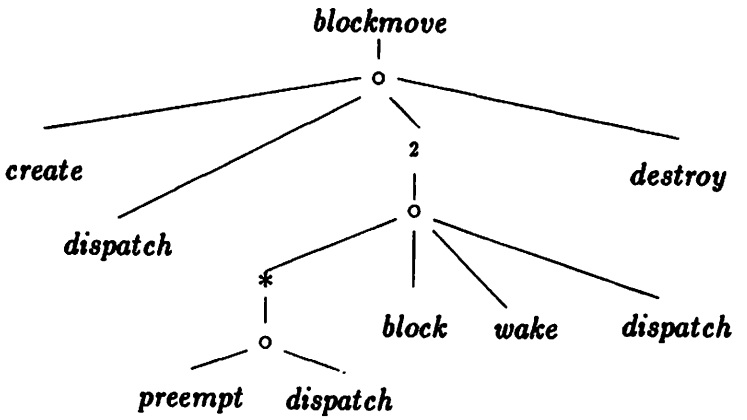


Figure 2: Simple high-level event structure

the intent of the model might be accomplished by constraining the *creator* or *system* fields of the *create* event to be one of a specific set of values.

References to event attributes are denoted by the dot notation used to reference fields of structures in many programming languages. An attribute slot reference is written using the event class to qualify the slot and the name of the attribute to access the value represented by the slot. For example, the attributes of the *destroy* primitive event from the process management view would be referenced

destroy.time,
destroy.system,
destroy.processid,
destroy.destroyer.

The *blockmove* example is a high-level event. Its structure is reflected in the structure diagram of figure 2. While this model might be adequate for the simple example, it is not very modular and does not return as much information as might be possible. It is also prone to error as it is being specified. It makes better sense to develop the model in a more structured way as a cluster of high-level and primitive events. The *blockmove* event might be expressed as follows

$$blockmove = startup \circ file\_access^2 \circ shutdown.$$

This is a more manageable and intuitive description of the *blockmove* event. This definition of *blockmove* would be adequate only if instances of its member events, *startup*, *file_access*, and *shutdown*, were available to be bound as constituents into the model. Since they are not available, they will need to be expressed in terms of their constituents – until a level is reached at which the actual system behavior and the model may be compared.

An important contribution of *EBBA* is being illustrated here. A user can express a model in a top-down fashion. In this way it is possible to expend a minimal effort to understand system behavior. Errors or deficiencies in the user's model of expected behavior may be detected at the earlier, less detailed stages of model construction. Also, owing to intuition or other knowledge, a user employing proper *EBBA* tools need only further investigate that part of the model thought to explain erroneous behavior.

Pressing on with the example, the *startup* event might be expressed as

$$startup = create \circ dispatch.$$

This event expression could be applied to many models which include program activations, and is thus reusable. For a complex or subtle error, there might be many such reusable clusters created to explain different parts of overall behavior. In the case where a complex system is riddled with errors, these reusable parts are easily combined into high-level models as the tool user's attention is directed to different areas of the system. The *EBBA* tool set described in chapter V facilitates the creation of *event libraries* based on a viewpoint.

In order to complete the *blockmove* high-level event, the *file_access* and *shutdown* events must be specified as, for example,

$$file\_access = (preempt \circ dispatch)^* \circ block \circ wake \circ dispatch,$$
$$shutdown = destroy.$$

The cluster of primitive and high-level events representing the *blockmove* will yield the structural model of figure 3.

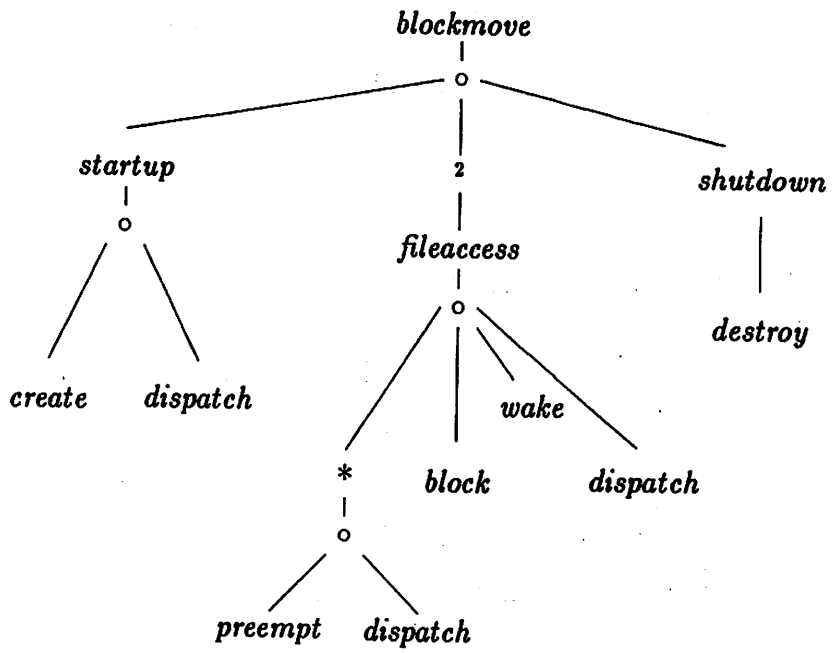


Figure 3: Hierarchical view of *blockmove*

In a distributed system, use of high-level events can reduce the communication bandwidth required to observe a behavioral model and distribute the effort necessary to recognize high-level event occurrences. For example, the *blockmove* event might have its *startup* and *shutdown* events occur on a node different from that of the *file_access* event. Further, all of these events might occur on a node other than the one where the *EBBA* tool user resides. Using the high-level model, the user would send several messages to remote agents to look for the *blockmove* constituents and wait for reports of their instantiation. The earlier, one-abstraction level version would require all of the low-level events to be reported to the user's node where filtering would be performed. This case has high communication overhead and low load distribution.

§3. A Distributed Programming Example

This section presents an example of a simple distributed program that contains an error and demonstrates how it might be uncovered through *EBBA*. The example closely resembles an error discovered while implementing the prototype debugging monitor and will be described in the context of the prototype. Through the example, we explore the limitations and capabilities of the *EBBA* approach and motivate development of tools to support *EBBA*.

The physical environment is a system containing a number of processor nodes connected by a communication subnet. Each node is a general purpose processor, time-shared among several users. These users enter the system through both hard-wired login terminals and processes created by actions of the software controlling the communication subnet.

The program being considered is the Network Interface (NI) component of the distributed debugging toolset that is described in chapter VI. The NI is responsible for controlling debugging activity at remote nodes of the distributed system and exchanging event tuples with other cooperating nodes. As such, the NI forms

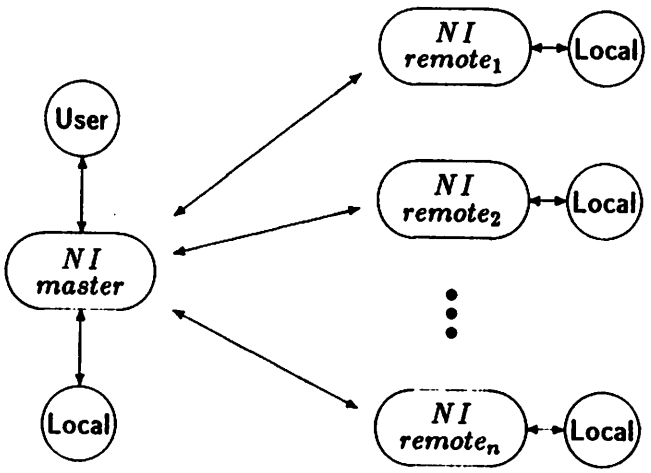


Figure 4: Communication structure for example

a distributed program consisting of cooperating processes, in which one process is initially designated as the master process with responsiblity for establishing the others. Following the initial setup period, each remote NI process started by the master NI engages in a two way message exchange with the master. The communication pattern is asymmetric in that the remote components do not communicate with each other, only the master process. This connection pattern and the idealized local components are illustrated in figure 4.

The Network Interface recognizes two types of message: control messages and event messages. Control messages are exchanged by the NI components at cooperating nodes to alter the communication pattern and affect local debugging tool activity. Event messages are text records of arbitrary length (up to a large maximum) constructed as a parenthesized list of symbols, numbers, and strings. Event messages are logically exchanged by the local component at each node by delivery to a local network interface which will send a copy to each cooperating network interface it has knowledge of.

The software environment is defined by a layered structure providing process

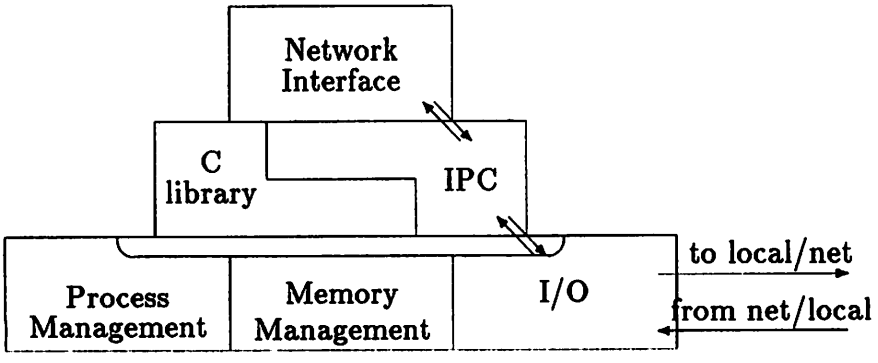


Figure 5: Layered architecture of network interface example

management and interprocess communication. (See figure 5.) The NI is layered on an interprocess communication (IPC) scheme that is capable of establishing connections and exchanging messages with both local and remote processes. The IPC uses runtime library functions common to the NI (or any other layers written in the implementation language) for storage management and miscellaneous buffer formatting, but the primary layering is on the host operating system services level [Kenah84].

The error to be tracked manifests itself as corrupted event messages being delivered from the master NI to its local components. An important characteristic of the observed error is that correct and incorrect messages both occupy the event stream and there is no discernible pattern established by this interleaving. As for the messages which are in error, they possess the following characteristics:

- The "beginning" of each message appears to be correct,
- A message is wrong because either it ends early, having no closing parenthesis or because it has extra characters that form a valid, second ending. In many cases the characters forming the extended message appear to be the missing characters from an earlier shortened message,
- The messages represent events with variable length attribute fields encoded as text strings, and all of the attribute values are syntactically valid values,
- Control messages appear to be intact and correct in all cases.

As in all properly layered communication software, actual message exchange only occurs between the lowest levels of software. All other levels implement virtual message exchange by appropriate use of functions provided by the layer beneath them. In this distributed program, messages flow from the low-level host I/O system responsible for actual transport of messages through IPC routines to the network interface. The interfaces between layers mainly copy message buffers and affix layer specific protocol notations to the buffers. This results in creating a number of possible points where the message stream could be corrupted but none are readily suspect.

The NI could mishandle a message as it turns it around for its journey to its receiver, the IPC might damage the message as it passes it up or down, or the host I/O level might impose some unknown limitation which is not observed by the IPC.

The corrupted message stream does not appear under certain conditions. If the master starts up any number of remote nodes and does not send out event messages of its own, then it receives all messages sent from all clients without error. However, when the master attempts to send messages, the message stream it receives tends to be damaged even though all sent messages appear to be intact at the receiving sites.

The error is somewhat subtle, and because of reasons given above, where to begin investigation of the error is not completely clear. It appears that there is some interaction of the send and receive sides of the master NI which is causing the observed errorful behavior. This easy explanation is complicated by evidence that the remote NI's do not manifest the error. It also appears that the IPC works in a satisfactory manner – messages may be exchanged in both directions successfully. One difficulty in searching for the error is that it is tedious and error-prone to determine what is wrong with each message since some of the messages appear to be correct even when they are incorrect. Further, since the message exchange is asynchronous it is impossible to predict when a bad message will occur. Finally, since messages have random arrival times, it is impossible to determine a total order

for the messages to determine if or when certain messages might be affecting others.

Debugging this problem was originally attempted with conventional break-examine debugging tools. The error was eventually found in the IPC layer and was related to misuse of a shared structure that included length information about messages which have been sent and received. The error was not obvious because, by coincidence, the event messages did not appear to be complete nonsense and the interactions of the components sharing the structure was quite complex.

Misuse of the shared structure occurred at the junction between IPC and host I/O layers where software interrupts and procedure call driven components handed control of message buffers to each other. The break-examine tools masked the error by altering the timing involved in the interaction of the components. The break-examine tools were connected to several of the nodes and the movement of messages through the software was charted. The messages appeared to be intact and correct at each stage. As soon as the tools were removed, destroying the enhanced visibility they provided, the corrupted message stream once again surfaced.

Tracking this error with *EBBA* tools requires that there be available a set of primitive events which describe the appropriate level of behavior. The primitive events will be supplied by three sources: the host system services level providing process management, synchronization, and I/O services; the functional level of the IPC; and the implementation level of the IPC.

This example is more concerned with specification of models to explore the erroneous behavior rather than the specifics of matching the actual system activity to the models. Assume there is a mechanism capable of detecting a high-level event model by selecting from the stream an appropriate set of events matching its cluster members. The desired properties of such an event recognizer will be accumulated here and considered in later chapters.

§4. Tracking the IPC Error With EBBA

The initial assumptions for finding the error are that the IPC is correct, there are no inherent limitations imposed by the IPC or the host system services, and it is the NI process which is incorrect – all reasonable assumptions. The top level model of master NI execution is described by the following event expression.

$$master\_NI = startup \circ create\_remote^{+} \circ msg\_exchange \circ delete\_remote^{+}.$$

The NI selected as the master is given life, creates the remote NI components, and proceeds to engage in event tuple exchange in the manner prescribed earlier. Following this activity, requests are made by the master NI that its remote components be deleted. This cluster ignores many aspects of actual behavior involved in NI operation – details that are not relevant to the errors in the tuple exchange. Among the ignored details is that of how the NI is selected as master. The activity of each remote NI is defined as

$$remote\_NI = startup \circ msg\_exchange \circ shutdown.$$

The master NI as well as the remote NI components become active through a flurry of activity modelled as

$$startup = create \circ bind.$$

Process creation is followed by a declaration of the process as a server. The *create* event is a primitive obtained from the host system and *bind* is an IPC functional level primitive.

A difficulty with using this model to recognize the beginning of a process is there is no guarantee that a *create* and *bind* selected from the event stream as model constituents are related. The importance of attributes in narrowing the focus of a model should now be apparent. The templates for *create* and *bind* supply attributes which might be used to focus the *startup* model and relate its cluster member *create* and *bind* events.

(create, time, node, creator, processid, processname)

(bind, time, node, name, processid).

In order that the *create* and *bind* are guaranteed to occur at the same site, the *startup* model should constrain the *node* and and *processid* attributes of its cluster members, i.e.

create.node == *bind.node,*

create.processid == *bind.processid.*

The model for *startup*, forming a high-level event in its own right, defines the following template:

(startup, time, node, processid, server_name)

The *server_name* field is necessary to distinguish startups of the NI process from other servers in the system and is derived from the *bind.name* attribute. Alternatively, the *startup* event could specify a constraint to apply to the *bind* event in its event expression such as

bind.name == "NI".

This constraint will be necessary at some level of the model definition in order to filter events that are not relevant to the model.

Event attributes may be used as constraints to be propagated down to lower levels of abstraction. This results in filtering of event instances and reduction of the volume of information that needs to be considered by higher level models. Specifying constraints at a low level provides the highest degree of filtering. Although the volume of information is reduced by specifying constraints at lower levels, the user's ability to observe peripheral behavior when not sure of the characteristics of a behavior is correspondingly reduced. In their other role of helping to match models, the attributes of those event instances actually selected as constituents to fit a model may be passed up levels of abstraction to supply information regarding the model constituents.

A tool set can aid a user by allowing constraint satisfaction to be relaxed at user specified levels or for specified parts of a model. The effect obtained would be similar to a user requesting a model match which "...looks something like this...". Users can explore models and obtain information concerning system behavior without complete knowledge of its structure or the interactions of its components. They can refine their models or change viewpoints with minimal investment in unproductive models. The toolset of chapter VI allows such constraint relaxation on an individual event recognition basis.

Returning to the model being developed for NI operation, the master NI, following its startup, will create a number of remote NI components and proceed to exchange event messages with them. When the message exchange is completed, the partners are removed from the system, but the model need not specify how this is done. The view of master NI operation through this model is a rigid one, but is sufficient for the problem at hand. Actual master NI operation is more accurately modelled as

$$master\_NI = startup \circ (create\_remote^{+} \triangle msg\_exchange \triangle delete\_remote^{+})$$

This is a more comprehensive and complete model because it allows for dynamic addition and removal of partners. However, for the conditions defined by initial NI development, the simple model is sufficient to explain its behavior. The important point illustrated here is that *EBBA* permits flexibility in model definition that is driven by current needs or viewpoints.

Both the simple and the alternative model of master NI operation might be considered weak because they do not closely relate the creation and deletion of remote components. Ignoring the fact that this relation is not interesting to the problem at hand we digress a moment to explain. Matching repetitive strings of events cannot be specified solely through the event expression mechanism, which has a regular expression genealogy. Instead, event attributes may be used to express relationships among cluster members. A first solution might be to tie pairs of

create_remote/delete_remote events by their *processid* attributes, i.e. constrain these attributes

$$\forall\ i\text{: } create\_remote_i.processid == delete\_remote_i.processid,$$

but this appears to force the order of the *create_remote/delete_remote* pairs to be strictly adhered to. Another suitable solution for some cases might involve a function over the strings of events accepted as model constituents,

$$number\_of(\ create\_remote\ ) == number\_of(\ delete\_remote\ )$$

Another possible remedy is to more fully specify this aspect of master NI behavior and use attribute information only to clean up loose ends. For example, the central part of the *master_NI* behavior might be specified as

$$master\_NI = \ldots (partner\_mgt^{+} \triangle\ msg\_exchange^{+}) \ldots$$

with this "partner management" modelled as

$$partner\_mgt = startup \circ shutdown$$

with a constraint to relate the startup and shutdown phases of the partner

$$startup.processid == shutdown.processid.$$

Event attributes may be used extensively to filter unneeded event instances, to pass characteristics of model constituents up to higher levels, and to overcome limitations of the event expression specification method. As shown above, the model matching machinery should have mechanisms available to supply characteristics of the collection of event instances bound to the model. These characteristics, as well as event attributes may then be used to constrain models and pass attributes to higher levels.

The message exchange model, *msg_exchange*, only needs to reflect the event message exchange behavior associated with master/remote NI interaction. Control messages have been absorbed into the *create_remote/shutdown_remote* behavior models. The first pass at a message exchange model might be

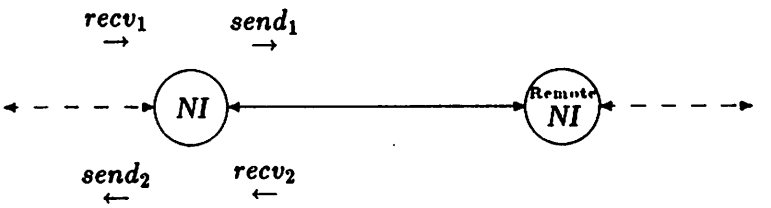


Figure 6: Event sense for *msg_exchange*

$$msg\_exchange = (send \mid recv)^{+}.$$

This model indicates that at a node, a *msg_exchange* is a mixed sequence of *send* and *recv* events. Incorporating this message exchange model into the *master_NI* will tell little about the error being investigated. This model is too simple. If a tool user were to employ this simple model to observe behavior, the model would match and simply indicate that the system does exchange messages. However, it is the messages which are in error.

To investigate further, the *msg_exchange* model will be altered to explain the behavior better.

$$msg\_exchange = (recv_1 \circ send_1)^{*} \triangle (recv_2 \circ send_2)^{*}$$

The intent is that the $recv_1$ and $send_1$ events will describe the master NI obtaining a local event message and distributing it to the remote components. The $recv_2$ and $send_2$ indicate an event message being delivered from a remote NI to the local component of the master NI. Figure 6 illustrates the message exchange events. Some constraints are necessary to express this properly:

$$recv_1.node == recv_1.from == send_1.node,$$
$$recv_2.node == send_2.node == send_1.node,$$

will describe the node relations adequately. Correct behavior of this *msg_exchange* model is reflected in the lengths of the messages, i.e.

$$recv_1.length == send_1.length,$$
$$recv_2.length == send_2.length.$$

Together the node and length constraints specify that an event message is received from the net by the master NI and handed to its local component properly (or vice versa).

Now we have a reasonable model of master NI behavior that applies when viewing the behavior solely from the node at which the master NI executes. By requesting that the system be viewed in terms of this model the user will observe that sometimes the model will match, other times it does not match. If the length constraint is not observed, the model will always match since the relaxed model only is concerned with message exchange protocol. The user can easily model the incorrect behavior by changing the length constraint to read

$$recv_1.length \neg = send_1.length,$$
$$recv_2.length \neg = send_2.length$$

The user should observe that correct messages arrive when the model of proper behavior matches and incorrect messages arrive when the model of improper behavior matches.

The model developed thus far ascertains that the error is related to the length of a message. None of the three components IPC, NI, or host I/O software has been eliminated from consideration as the site of an error. To eliminate the host I/O software, an internode *msg_exchange* model could be written

$$inter\_msg\_exchange = recv_1 \circ send_1 \circ recv_2 \circ send_2$$

which describes a transfer message from arrival at an NI to exit at another NI (figure 7). With addition of constraints on the internode portion of the message movement,

$$send_1.length == recv_2.length,$$
$$send_1.to == recv_2.node$$

the *inter_msg_exchange* model will match all event message exchanges at the same time that the *master_NI* viewpoint model of correct or incorrect behavior only matches occasionally. By adding constraints on the message length handled by the NI component to the *inter_msg_exchange* model such as

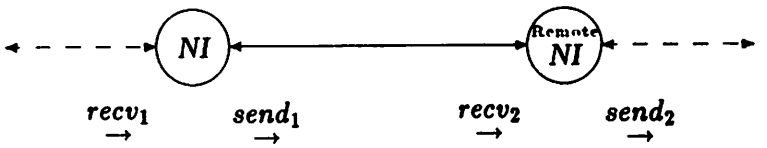


Figure 7: Event sense for *inter_msg_exchange* event

$$recv_1.length == send_1.length,$$
$$recv_2.length == send_2.length$$

it can be determined that the NI properly transfers messages from its local side to the network.

Evidence suggests that there is some problem with the IPC layer or its interaction with the NI and I/O layers. Examination of these interactions will require a closer look a IPC functionality, requiring a viewpoint defined by another set of primitive events that represent IPC functionality.

§5. Chapter Summary

This chapter has examined an approach to monitoring the activity of an errorful system. The approach supports top-down, hierarchical description of system activity with a view to modelling the errors which have made themselves known. An example was presented to demonstrate the use of the approach. The example is based on an error that arose during the implementation of the prototype toolset. The error was caused by the misuse of a data structure that reported the length of an I/O operation between the layers of a hierarchical inter process communication support library. The error could not be found by using conventional break-examine tools because the tool use completely disrupted the timing relations and caused the error to appear transient.

CHAPTER III

EVENT DEFINITION LANGUAGE

The Event Definition Language (*EDL*) is a mechanism that enables its users to express models of system behavior as abstractions of the detailed computation comprising that behavior. *EDL* users express their behavioral abstractions as high-level event clusters in terms of the primitive events that characterize the system, and high-level events that were previously defined through that viewpoint. Behavior models expressed as *EDL* definitions themselves define event tuples which may be incorporated into other high-level event descriptions. Thus, each *EDL* event definition provides a template for an event tuple that represents an event class which might occur in the system.

Both primitive and high-level events may be described in *EDL*. Primitive events result from system operation so users of *EBBA* descriptive tools have little control over creation of instances of primitive event classes. Hence, primitive events require little descriptive power. High-level events on the other hand, represent user models of behavior and in a sense are a creation of the *EBBA* model matching tools. They require more powerful means to describe their resultant event tuples as well as means for describing the intended behavioral model.

Primitive events described in *EDL* simply indicate their unique system-wide event names and supply a list of names for the attributes possessed by the event. Descriptions of higher level events are more elaborate since they involve use of *EBBA* clustering and filtering techniques. Clustering to describe a high-level model is effected by specifying a set of previously defined events in an event expression and imposing constraints on the event expression members' attributes in a set of

relational expressions. The event and relational expressions serve in dual roles of describing a behavioral model and providing a guide for recognizing the occurrence of an event representing the model. The attributes of a high-level event are expressed in terms of the attributes defined by the cluster member events and other information obtainable from the *EBBA* model matching environment.

This chapter describes the Event Definition Language as implemented in the prototype toolset (chapter VI). The syntax is quite simple and will not require explanation beyond its lexical elements. However, *EDL* syntax is detailed in a BNF description contained in appendix A for those who are interested. Here we will informally detail the meaning of an event description and relate descriptions to the subset of events they would match from an event stream. The form of a model recognizer defined by an event definition is more formally described in chapter V. The translation of *EDL* descriptions into a guide for recognizing the model they represent and details of their use in this capacity are likewise left until later.

The next section is a short description of the lexical elements from which *EDL* definitions are composed. The following section explains in detail how high-level events are written in *EDL*. Section three describes how the primitive events which characterize a system viewpoint are described in *EDL* and relates these event definitions to the event templates defined in the previous chapter. The final section provides a complete example of the use of *EDL* to describe a couple of behavior models.

§1. Lexical Elements of *EDL* Descriptions

EDL event descriptions are composed from a limited set of lexical elements including keywords, identifiers, values, expression operators, and punctuation marks. Keywords introduce clauses and special values and will be written here in bold face letters, e.g. **event, end, primitive, with, cond, true.** Usage of keywords will be described in appropriate sections.

§1.1 *Identifiers*

Identifiers are constructed from contiguous sequences of letters, digits, and underscores with the restriction that the first be a letter. Identifiers are used to represent event attributes, event class names, and parameters, constants, and external functions. Identifiers will be written in italic throughout this and following chapters. Some valid identifiers would be

Send	*CreateHyp*	*Create_Special_Node*
Event3	*TIME_OF_DAY*	*legitimate_10_folds*

Identifier strings are not case sensitive. Upper and lower case alphabetic characters are normalized into an upper case form, i.e. *MiXeD* is the same as *mIxEd*.

§1.2 *Values*

Values may be either numbers or character strings that are used to represent absolute quantities. Users must exercise care in using values and value bearing identifiers. *EDL* does not include means for explicitly data typing the expressions involved in assigning attributes or evaluating filtering constraints. The assumption is made that this weak typing can be resolved in a satisfactory manner by whatever system uses the *EDL* definitions to guide behavioral abstraction.

Numbers

EDL allows numbers to be specified as signed decimal and floating point numbers or radix-specific bit strings. The range of values, the container size, alignment, and representation are, as always, system dependent. Some examples

```
37.661          -392          0xFF01
0b101101        12.2e-2       0o77376
```

The keywords **true** and **false**, intended for use in relational expressions, represent the values 1 and 0, respectively.

Strings

A character string is written as a sequence of characters enclosed in quotation marks, ' " '. Non-printing characters can be inserted into string constants by writing the integer values preceded by a "\" escape character. As with numbers, all specifics of the representation are system dependent. Some string examples

```
"vax9"
"An escape sequence \27[H"
```

§2. Describing High-Level Event Models

A single high-level event definition describes a model of some aspect of a system's overall behavior. Each event definition is composed from a heading which supplies the name for the event class being defined and up to three kinds of defining clause which specify the structure and intent of the event. The **is** clause defines the event expression that specifies what event classes constitute the cluster and indicates acceptable orderings for their instances. The **cond** clause details a set of relational expressions defined over the attributes of the event expression event classes. These relational expressions constrain the attributes of events that appear in the event expression to specific values or value ranges. The event heading together with a set of attribute binding expressions defined in the **with** clause of a definition describe the event tuple for the event class represented by the event definition. The attribute binding expressions are also expressed in terms of the event expression members' attributes.

§2.1 *Event Heading*

The event heading of an event definition associates a name by which the event class is known and referred to by users of the modelling tools employing *EDL*. The event name that appears in the heading is not necessarily the identifying name used

internally by the *EBBA* model matching components. However, it is important that there is available a unique mapping of system defined event names onto the external names with which an event view is constructed by a user. This is necessary so that users have some basis to believe that what they see represents their view of a system [Snodgrass85]. Within a given view, all event names are required to be unique.

An event heading may have an optional parameter list which provides a means of parameterizing events. The parameter list is specified as a list of names acting as place holders for values that are supplied when a request is made to match the event description to actual system activity. This permits a limited degree of tailoring an event recognition request to dynamic system conditions. The event heading of figure 8 introduces an event class named *FrontEndCompletion* which is

event *FrontEndCompletion(node)* **is**

Figure 8: Event heading with one parameter

invoked with a single parameter, *node*. The *node* parameter can appear later as an operand in relational or arithmetic expressions in the **cond** or **with** clauses of the event description. A value is bound to this parameter when a request is made for recognition of an instance of the *FrontEndCompletion* event.

The example heading might be useful where a user is interested in instances of the *FrontEndCompletion* event only if they occur on a specific node of the distributed system. For example, in the prototype debugging monitor, a request for recognition of an instance of the *FrontEndCompletion* would have the form[1]:

```
user-> (recognize (FrontEndCompletion "vax9"))
```

The event recognizer would be requested to watch the event stream for an instance of the high-level *FrontEndCompletion* event with the string value "vax9" bound to the *node* parameter.

[1]The "user->" is a prompt from the toolset indicating that it is looking for something to do.

§2.2 *Event Expression*

The event expression is a regular expression-like string of event symbols and event operators. The event symbols represent constituent behaviors of the high-level event model which is described by the enclosing event description. The operators indicate alternative orderings that sets of event instances may have in order to match the event expression members. In general, the event expression will only describe an outline for a behavior model. The event operators define acceptable orderings, thus implicitly imposing time relations on event expression constituent events. However, events carry a richer set of attributes that may be used to relate collections of events. These relations are handled by the **cond** clause, described later.

Figure 9 expands the earlier *FrontEndCompletion* example to include an event expression. The *FrontEndCompletion* event is defined in terms of three other events: *InstantiateKsi, CreateHyp,* and *InvokeKs.*

event *FrontEndCompletion(node)* **is**
    InstantiateKsi[1] ∘ *CreateHyp** ∘ *InstantiateKsi[2]* ∘ *InvokeKs*
end

Figure 9: Event heading with event expression

The event expression provides a means to perform coarse filtering of event stream instances. When an attempt is made to recognize an instance of the event, only event stream instances in the same class as event expression symbols need be examined for possible inclusion as constituent events.

The subscript notation appended to each of the *InstantiateKsi* event symbols serves to distinguish the two occurrences when they are used as operand qualifiers in **cond** and **with** clauses. Without the subscripts all expression references to attributes of an *InstantiateKsi* event would access the values contained in the instance bound to the first *InstantiateKsi* of the event expression.

Event expressions need to describe a wide range of behavioral patterns. In distributed systems, behaviors will consist of sets of events which are only partially ordered. Sometimes, the set consists of events which need to be ordered in strict sequences. Other times the events occur concurrently, and will be observed in arbitrary order. Conventional regular expression formalisms are capable of describing sequences in various ways but do not provide means for expressing concurrency. *EDL* event operators include operators that specify sequencing, choice and repetition as well as an operator describing concurrently occurring sets of events.

Sequencing

The binary sequence operator, indicated with " o ", specifies that its operand events are to occur in the left to right order in which they are written. For example the partial expression

$$\ldots \textit{InstantiateKsi} \circ \textit{InvokeKs} \circ \textit{SendHyp} \ldots$$

indicates that the described behavior consists of an instance of the *InstantiateKsi* event followed at a later time by an *InvokeKs* event, which is subsequently followed by a *SendHyp* event. Implicitly, the three events are related by a relation of their time attributes where

$$\textit{InstantiateKsi.time} < \textit{InvokeKs.time} < \textit{SendHyp.time} .$$

Repetition

Indefinitely long sequences of events are specified by two postfix unary operators, plus and star, indicated by $^+$ and *, respectively. Plus indicates that the described event sequence consists of one or more repetitions of its event operand. Star indicates a sequence of zero or more instances of its operand event. Thus, the partial event expression

$$\ldots \textit{InstantiateKsi} \circ \textit{CreateHyp}^* \circ \textit{InstantiateKsi} \ldots$$

describes the sequence

```
(InstantiateKsi . . .)
(InstantiateKsi . . .)
```

or

```
(InstantiateKsi . . .)
(CreateHyp . . . )
(InstantiateKsi . . .)
```

or

```
(InstantiateKsi . . .)
(CreateHyp . . . )
(CreateHyp . . . )
(CreateHyp . . . )
(CreateHyp . . . )
(InstantiateKsi . . .)
```

and so on. Whereas

$$\ldots \mathit{InstantiateKsi} \circ \mathit{CreateHyp}^{+} \circ \mathit{InstantiateKsi} \ldots$$

will describe the all but the first of the above event sequences.

Choice

Specifying a choice between different events is accomplished by the alternation operator, " | ". Alternation indicates that an instance of at least one of its operand events is necessary to match the event expression. For example, the partial event expression

$$\ldots \mathit{CreateDDGoal} \circ (\mathit{ReceiveHyp} \mid \mathit{CreateHyp}) \circ \mathit{InvokeKs} \ldots$$

matches the event subsequence

```
(CreateDDGoal . . .)
(CreateHyp . . .)
(InvokeKs . . .)
```

or the subsequence

```
(CreateDDGoal . . .)
(ReceiveHyp . . .)
(InvokeKs . . .)
```

Random choice among a group of operand events is possible by specifying a form:

$$CreateHyp \mid SendHyp \mid CreateGoal \mid \ldots \mid SendGoal$$

The above expression indicates that the specified behavior consists of any *one* of the *CreateHyp... SendGoal* events. The alternation operator also has the property that the fully parenthesized expression

$$(\ldots((CreateHyp \mid SendHyp) \mid CreateGoal) \mid \ldots \mid SendGoal)$$

is the same as the above, unparenthesized expression.

Note that the definition of alternation does not state "... but not both ..." – exclusion is not its purpose. The nature of the event stream is such that all alternates could occur in the proper context. However, to successfully match the event expression to the stream only requires one of the alternates. Which event is chosen will depend on the tools employed to match the models to event streams.

Concurrency

Concurrency is described using the shuffle operator, written " $\triangle$ ". The shuffle operator designates a set of events, *all* of whose members must occur, but there is no preferred ordering imposed on the set members. For example,

$$\ldots InvokeKs \circ (CreateHyp \triangle SendHyp) \circ InvokeKs \ldots$$

describes the sequence

```
(InvokeKs . . .)
(SendHyp . . .)
(CreateHyp . . .)
(InvokeKs . . .)
```

or the sequence

```
(InvokeKs . . .)
(CreateHyp . . .)
(SendHyp . . .)
(InvokeKs . . .)
```

Similar to the alternation operator, a series of event names joined with shuffle operators in an event expression string such as

$$\ldots \mathit{CreateHyp} \triangle \mathit{ReceiveHyp} \triangle \mathit{CreateDDGoal} \ldots$$

is the same as the parenthesized version

$$\ldots ((\mathit{CreateHyp} \triangle \mathit{ReceiveHyp}) \triangle \mathit{CreateDDGoal}) \ldots .$$

This use of the shuffle operator designates a cluster of events in which *all* of the events participating in the shuffle must occur.

The use of the shuffle operator to denote concurrent events does not imply that (in the above example)

$$\mathit{CreateHyp.time} = \mathit{ReceiveHyp.time} = \mathit{CreateDDGoal.time}$$

It is also not true that if one operand of a shuffle occurs "first", the rest will occur "later". That is, if the following are true about the constituents bound to the event instance represented by the above shuffle

$$\begin{aligned} \mathit{CreateHyp.time} &= t_1 \\ \mathit{ReceveHyp.time} &= t_2 \\ \mathit{CreateDDGoal.time} &= t_3 \end{aligned}$$

there is no guarantee that an event sequence matching the shuffle event string implies any relations between the attribute times t_1, t_2, or t_3.

§2.3 *Cond Clauses*

The **cond** clause contains a set of relational expressions which are used to constrain the attributes of event expression members to certain values or value ranges. The set of **cond** clause relationals specifies that only instances of event expression members possessing certain characteristics are to be considered eligible to match the described high-level behavior.

The relations among events expressed by **cond** clause constraints provide finer grained filtering than that provided by the event expression since they are defined in terms of the attributes of event expression members. For recognition of a particular event, the high-level event recognition machinery only selects, from the event stream, event instances that match event expression members. Following this coarse filtering, any **cond** clause relations defined in the high-level event involving attributes of the instance matching an event expression member must be evaluated to determine if the selected instance possesses appropriate attributes.

These constraints on event attributes thus serve both to narrow the scope of a model and as filters which eliminate unnecessary instances of events from inclusion into a behavior model. A **cond** clause is an optional part of an event description. In the absence of a **cond** clause, any set of events from the appropriate classes which match a pattern prescribed by the event expression will constitute an instance of the defined event. Figure 10 is the earlier *FrontEndCompletion* example extended to include **cond** clause constraints.

The set of operators available for **cond** clause relational expressions is a rich set of arithmetic, logical, and relational operators. Arithmetic operators include addition (+), subtraction (−), multiplication (*), division (/), and modulus (%). Bitwise logical operations are shift left (<<), shift right (>>), and (&&), and or (||). The arithmetic and bitwise logicals are all binary numeric operations and return numeric values. Available relational operators are the usual less than (<), less than or equal (<=), equality (==), inequality (¬ =), greater than or equal (>=), and greater than (>). These are defined over all values and return boolean

```
event FrontEndCompletion( atnode ) is
    InstantiateKsi[1] ∘ CreateHyp* ∘ InstantiateKsi[2] ∘ InvokeKs
cond
    InvokeKs.node ==atnode
        ==InstantiateKsi[1].node ==InstantiateKsi[2].node
end
```

Figure 10: Event definition with constraining clauses

false (value 0) and true (value 1) results. Logical connectives and ($\&$) and or (|) supply conjunction and disjunction of boolean values. Unary complement operators are negate ($-$) for arithmetic values, bitwise logical complement ($\sim$), and not (!) for boolean expressions. Of course, all of these may be augmented with external functions defined in the *EBBA* tool environment.

Expression operands have five sources: event heading parameters, event instance attributes, so-called "lexical" functions, system defined functions and simple constant values. In the prototype system of Chapters VI and VII, no explicit means is provided to associate data type information with operands. Thus, no type compatibility checking is performed for individual expressions to ensure they can actually be evaluated meaningfully. An assumption is made that only simple numbers and character strings may be bound as operand values. The following subsections detail each source of expression operand by describing the access mechanism and when an operand takes on a value. This discussion of operands and value-producing expressions applies equally well to attribute binding expressions in **with** clauses.

Event Heading Parameters

Event heading parameters are identifiers listed in the parameter section of an event heading. Their importance is that they can be used to parameterize the value producing expressions in an event definition. Event heading parameters are

accessed in expressions as unadorned identifiers. Values are bound to event heading parameters when a recognition request is made for an instance of the event class. In **cond** clause relational expressions, use of event heading parameters serves to tailor the filtering effect of the **cond** clause to dynamic system conditions. Refer to figure 10 for an example of the use of an event parameter as an expression operand.

Event Attributes

Event attributes are the most important kind of valued operand because event attributes are the primary means for focusing particular behavior abstractions. Values represented by these operands are obtained from event instance tuples bound to event expression members. Event attribute operands are specified by using the attribute name prefixed by the name of the event with which the attribute is associated. The qualifying event symbol must appear as a member of the event expression of the enclosing event description.

If the *FrontEndCompletion* event expression members have the event templates

(InstantiateKsi time node ...)
(CreateHyp time node ...)
(InvokeKs time node ...)

the focus of the *FrontEndCompletion* event can be narrowed by specifying constraints in terms of the event expression member attributes. The attribute access is written using the common field access dot notation as in numerous programming languages [Horowitz84]. For example, the event definition of figure 10 constrains all of the event stream events which are bound to the event expression members to have occurred on the same node of the system.

Recall that the subscript notation is important when an event name is used more than one time in an event expression. Value expressions use the subscripts to specify which qualifying event supplies the attribute value when the expression is evaluated. The subscript notation only applies to the explicit mentions of an event in an event

event *FrontEndCompletion(count)* **is**
InstantiateKsi[1] ∘ *CreateHyp** ∘ *InstantiateKsi[2]* ∘ *InvokeKs*
cond
numberof(CreateHyp) $<$*count*
end

Figure 11: Event expression "Lexical" functions

expression. The correspondence between event expression events and qualified event attributes is made when the event description is created. The implication of this is that the attributes of a repetitive event instance, e.g. the *CreateHyp* events bound to the event expression

$$\ldots \circ \mathit{CreateHyp}^{+} \circ \ldots$$

are not accessed as *CreateHyp[1].node*, *CreateHyp[2].node*, etc. The only valid subscripted event attribute qualifier for this string would be *CreateHyp[1].node*, since only a single mention of the *CreateHyp* event is made in the event expression. Accesses to specific instances of the string of *CreateHyp* events bound to the *CreateHyp*$^+$ event expression member are performed through lexical functions.

Event Expression "lexical" Functions

Certain event expression constructs have attributes which result from the binding of event stream instances to the event expression member events. Attributes include the total number of events bound to a repetitive event expression or the absence of an event instance from the bound event string. The first is given by the *numberof()* lexical function, the later by the *undefined()* lexical function. For example, as shown in figure 11 the *FrontEndCompletion* event might only be "interesting" if the number of *CreateHyp* events is smaller than some number, *count*, supplied as a parameter. Other types of lexical functions help to unravel things which are not easily expressed

in the event expression format, such as accessing the first or last instance of a repetitive sequence. For example the function

last(CreateHyp).node

selects the *node* attribute of the last *CreateHyp* instance which has been bound to the $CreateHyp^{+}$ part of the event expression string.

Environment Specific Functions

These functions are supplied to return values obtained from the environment supporting the abstraction of event instances. What functions are supplied is highly implementation dependent, but they at least include a *time-of-day()* and a *localnode()* function. Environment specific functions are provided largely as a catch-all for unanticipated, but necessary, functionality in the evaluation environment.

§2.4 *With Clauses*

The list of names for attributes possessed by an event and a description of how to bind values to each attribute when an instance of the event is recognized is found in the **with** clause expression. Each **with** clause expression names and describes a single event attribute. The attribute is defined as an identifer which is the target of an assignment operator. The expressions which provide values for event attributes are composed from the same set of operators and operands as **cond** clause expressions.

When a set of events matching one of the possible event expression sequences has been accumulated and all **cond** clause constraints evaluate satisfactorily, the event is instantiated in its own right. Using the context supplied by its parameters and attributes of the constituent events, the attribute binding expressions are evaluated and the value for each slot in the event tuple template is filled in.

To illustrate, in figure 12 the familiar *FrontEndCompletion* event can be expanded to possess attributes (in addition to the predeclared attributes for time and place) for *elapsedtime* and *hypcount*. The *elapsedtime* attribute is bound to the

```
event FrontEndCompletion( count ) is
    ...
with
    hypcount := numberof(CreateHyp);
    elapsedtime := InvokeKs.time -InstantiateKsi[1].time
end
```

Figure 12: Example attribute binding expressions

difference in the time attributes of the events which are first and last in the event string bound to an instance of the *FrontEndCompletion* event. The *hypcount* attribute reflects the length of the string of *CreateHyp* instances bound to this event expression.

Like the **cond** clause, the **with** clause is an optional part of the event definition. However, every event instance will carry with it certain predefined attributes, such as time of occurrence, that might serve to distinguish various instances of the class. Other predefined attributes might be dependent on specific characteristics of the system, such as the name of the processor node on which the event occurred.

§2.5 *Summary of High-Level Event Descriptions*

EDL high-level event descriptions subsume the informal notation introduced in chapter II for describing models of system behavior. *EDL* carries the notation forward by providing means to parameterize events, providing links to an actual environment that supports modelling as an activity, and explicitly describes the structure of high-level event instances.

An event cluster is described by a single *EDL* description through its event expression and **cond** clause parts. The event cluster from section four of chapter II:

$$msg\_exchange = (recv_1 \circ send_1)^* \triangle (recv_2 \circ send_2)$$

and its constraint

$recv_1.node == recv_2.node$

is expressed in *EDL* as

event *msg_exchange* **is**
(recv[1] ∘ send[1]) △ (recv[2] ∘ send[2])*
cond
recv[1].node ==recv[2].node
end

These components provide the guide to recognizing an instance of the event and detail the behavior model intended by this description. To permit incorporation of events into other high-level event descriptions the **with** clause details the structure of an event template described by this event.

§3. Describing Primitive Events

Primitive events, those events that represent the lowest level of observable behavior, are also defined by *EDL* descriptions. The description for a primitive event is an abbreviated one which contains no event expression, **cond** clause, or parameter list. Each of these parts of an event description is related to some active property of event recognition. Primitive events are created by the system which is under investigation and are characteristic of that system. They are atomic entities that a user of model specification and matching tools has no control over. If a user desires to see only primitive event instances with certain attributes, it is necessary to construct a high-level event that expresses the appropriately constrained cases.

Primitive event descriptions consist of a special event heading and a reduced form of **with** clause. The event heading names an event class and declares the event class as **primitive**. Following the **primitive** keyword is a value, the system specific identifier, that the event generating source puts into the event class slot of an instance of this event class. The form and value of the system specific identifier

event *CreateHyp* is primitive "CREATE_HYP"
with
    time; node; hypname
end

$$(CreateHyp, time, node, hypname)$$

```
(CREATE_HYP 57391 "VAX9" "h:09:0034")
(CREATE_HYP 159318 "VAX2" "h:02:0173")
```

Figure 13: Primitive description, event template, and instances

following the **primitive** keyword is system and implementation dependent. It could be a number, a string, or some arbitrary bit pattern.

The unique system identifier has an effect of permitting flexible binding of names to actual events. As noted in the example of chapter II it is sometimes necessary to shift the viewpoint on a system to a "lower" level of primitive event. The capability in *EDL* to easily rebind primitive events facilitates this type of rebinding of event to event generator.

In a primitive event definition, the set of attributes defined by the primitive event must be listed as for high-level events. However, there are no code strings associated with the attributes. Event attributes result from action by the event generator that is, in the primitive event case, an unaccessible system activity. The *EDL* primitive event definition thus defines the tuple for a characteristic primitive event and provides a handle to match it with system generated events. Figure 13 shows the relationship between an *EDL* primitive description, the template for the event tuple, and a few instances of the primitive event. All that is necessary to use this primitive event is that the system generate some event tuple which matches the event definition.

event *FrontEndCompletion(atnode)* **is**
InstantiateKsi[1] ∘ *CreateHyp** ∘ *InstantiateKsi[2]* ∘ *InvokeKs*
cond
InvokeKs.node ==*atnode*
==*InstantiateKsi[1].node* ==*InstantiateKsi[2].node*
with
hypcount := *numberof(CreateHyp);*
elapsedtime := *InvokeKs.time* −*InstantiateKsi[1].time*
end

Figure 14: Complete *FrontEndCompletion* event definition

§4. A Complete Example

Now that all of the parts of a high-level event description have been explained it might be useful to illustrate many of the views of an event definition. A complete *FrontEndCompletion* event might be described as in figure 14. This description represents the event tuple

(FrontEndCompletion time node hypcount elapsedtime) .

The implicit attributes *time* and *node* are always present and are part of every event tuple. Some example instances of this event could be

```
(FrontEndCompletion 233 "vax9" 25 371)
```

and

```
(FrontEndCompletion 184 "vax6" 4 54)
```

If the observed event stream contained a sequence of events

```
(InstantiateKsi 10 "vax7" . . .)
(InstantiateKsi 9 "vax6" . . .)†
(InvokeKs 12 "vax7" . . .)
(CreateHyp 13 "vax7" . . .)
```

```
(InvokeKs 121 "vax6" .  .  .)
(CreateHyp 146 "vax6" .  .  .)†
(CreateHyp 172 "vax7".  .  .)
(InstantiateKsi 223 "vax6" .  .  .)†
(InvokeKs 250 "vax6".  .  .)†
```

then an instance of the *FrontEndCompletion* event derived from this stream would have the events marked with † as constituents. The instantiated *FrontEndCompletion* event could be

```
(FrontEndCompletion 250 "vax6" 1 241)
```

The *FrontEndCompletion* event can also be viewed in terms of its structure, which is derived from its constituent events, their constituents, etc. Since self-referent event descriptors are not allowed, the structure defines a tree with the given high-level event as the root. If the *FrontEndCompletion* event is incorporated into a higher level event, *FirstSigActivity* with the definition of figure 15 the structural

event *FirstSigActivity(atnode)* **is**
    FrontEndCompletion ∘ *(ReceiveHyp | CreateHyp)*

cond
    CreateHyp.node ==atnode;
    ReceiveHyp.node ==atnode;
    FrontEndCompletion.node ==atnode

end

Figure 15: A higher level event

representation of *FirstSigActivity* would be as shown in figure 16.

This structural view is useful when trying to understand the overall structure of an event based behavior description. With proper annotation this representation might make it easier to judge how well this abstraction matches the actual system activity. Also, the structural view aids in analyzing certain difficulties with the abstraction process itself. More of this will be described in later chapters.

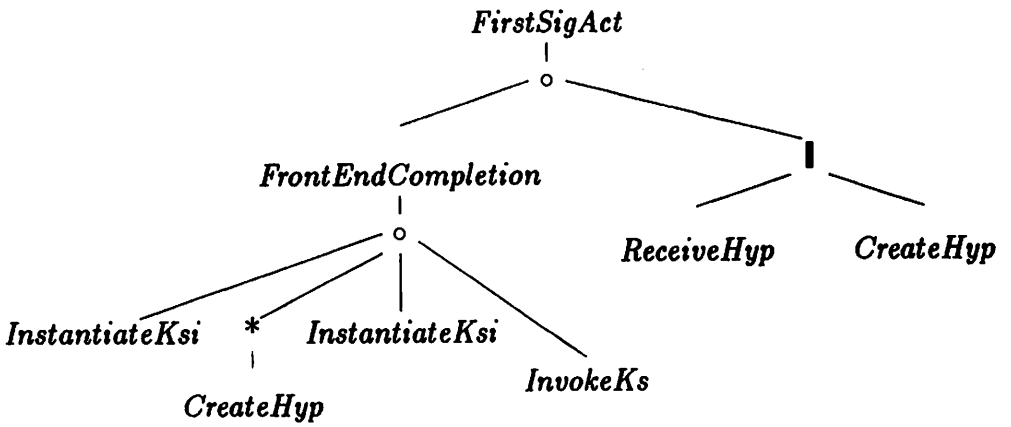


Figure 16: Structural representation of a high-level event

CHAPTER IV

RECOGNIZING BEHAVIOR IN COMPLEX SYSTEMS

Event Based Behavioral Abstraction characterizes the activity of a system as a stream of distinct, observable behaviors. Using this characterization, the behavior of a system may be abstracted into models that represent important system functions. The Event Definition Language presented in the previous chapter may be used to define models of "interesting" system functions in terms of patterns of events defined for a viewpoint on the system. In order for the stream characterization and the abstract modelling to be brought together into a useful tool to aid debugging complex systems, it is necessary to capture the event stream provided by the system and attempt to fit it to the user's model of system activity. The token-like event stream and the regular expression based *EDL* model descriptions suggest a *syntactic pattern recognition* approach [Fu82] to support the necessary behavior modelling.

A syntactic pattern recognition (*SPR*) system consists of two major parts: analysis and recognition (figure 17). The recognition part consists of components that perform pattern preprocessing, decomposition, primitive recognition, and syntax analysis. The analysis part consists of components involved with primitive and relation selection, and structural inferencing from a set of sample patterns. The analysis part accepts as input sample patterns from the domain in which the pattern recognizer is to operate. From these sample patterns, analysis determines what the atomic elements of the patterns are and how they combine to form pattern primitives. The primitive information will be used by the recognition part to decompose candidate patterns. After the primitive selection component derives a description of pattern primitives, the structural inference component determines a grammar

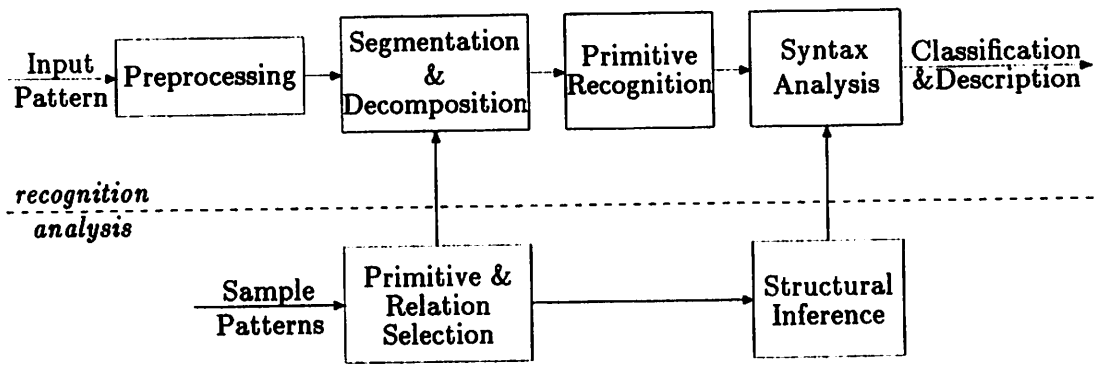


Figure 17: Syntactic Pattern Recognition System [Fu82]

describing acceptable patterns of the primitives.

The pattern recognition component is driven by the availability of candidate patterns formed from a set of primitive pattern elements. Preprocessing cleans the candidate pattern of any obvious defects before the segmentation and decomposition component isolates and marks possible atomic elements. The primitive recognition component groups atomic elements to identify the primitive tokens which will serve as input to the syntax analysis component. Syntax analysis proceeds to parse the now tokenized pattern to determine its validity and classify it within the acceptable set of patterns.

This general description spans a range of pattern recognition systems. At one end are the syntax checkers found in programming languages. Their atomic elements (letters, digits, punctuation, etc.), primitives (tokens such as keywords, identifiers, and numbers), and syntax are quite rigidly determined before any patterns need to be recognized. Other pattern recognition systems have great uncertainty and noise in their input candidate patterns. The result is that the preprocessing, segmentation, and primitive recognition phases require a large part of the effort to recognizing patterns.

The remainder of this chapter will describe pattern recognition as it is applied to event based behavioral abstraction. Pattern recognition for behavior monitoring purposes has many characteristics of syntactic pattern recognition systems. How-

ever, the characteristics of the distributed programming environment and the use of information in a debugging environment introduce significant particulars that are not adequately addressed by syntactic pattern recognition. The first section will describe the logical components of a behavior recognizer to outline its task and use of information. The second section describes the difficulties that need to be overcome by behavior recognition. The final section lists possible methods for creating the event stream that serves as input to the behavior monitor.

§1. Pattern Recognition Applied to Behavior Recognition

Like general syntactic pattern recognition systems, a pattern recognizer for event based behavior monitoring consists of analysis and recognition parts (figure 18). Analysis contains a component that translates *EDL*-described event classes into a

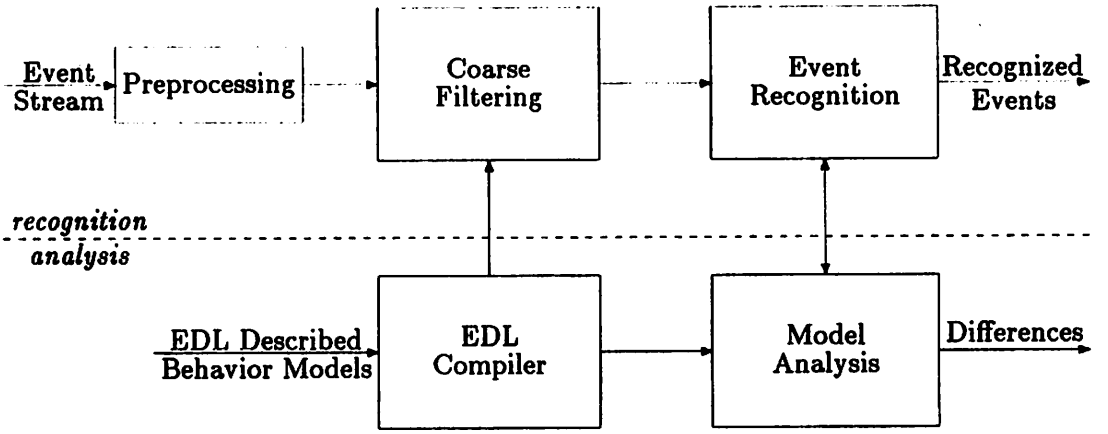


Figure 18: Syntactic PR for Behavior Monitoring

recognizer that guides detection of those event classes, and describes event templates for event instances expected in the event stream. For event based behavioral abstraction, there is no real structural inference component that is concerned with a grammar-creating role. Instead, to support behavioral abstraction in a difficult environment, a model analysis component is concerned with organizing the search

for constituents of a complex behavior model. This can be as simple as spawning recognizers for individual members of an event expression or as sophisticated as decomposing an event into sub-parts to be parcelled out to cooperating nodes of a distributed system.

Another task of the model analysis component is to help evaluate the accuracy of user defined event models. This evaluation is performed by comparing the description of an event to the progress made by the recognition part towards recognizing an instance of the event. This capability is useful since it helps a tool user discover the differences between the actual behavior and the models, thereby possibly pointing out sources of errors in the software. See the description of the Behavior Monitor component of the prototype toolset for more specifics of model analysis.

The recognition part of behavior monitoring is concerned with extracting behavior patterns from the input stream of events. In general, the stream input to the recognition part is a continuous one rather than a fixed length string representing a single pattern. Preprocessing of the event stream is simply responsible for translating an event instance tuple into a normal form to be used internally by the remaining recognition components. The coarse filtering component is likewise simple. Coarse filtering compares event templates defined for a viewpoint to event instances contained in the stream and can reject any event instances not relevant to the viewpoint in use.

The event recognition component is charged with the selection of subsets of the coarsely filtered events which match specified behavioral models. In syntactic pattern recognition systems, this component is usually served by a context-free parser or finite state machine interpreter. A number of factors complicate the pattern recognition component for event based behavioral abstraction. They include the fine filtering constraints that may be applied to a model's constituent events, resolution of the concurrency and time inaccuracies that are possible in event generation, and a need to at least partially recognize potentially inaccurate behavior models. Accounting for these needs requires the event recognizer to be more capable than

the simple pattern recognition state transition machines. These complications are examined in detail in the next section.

A significant difference of pattern recognition for *EBBA* from basic syntactic pattern recognition is the use of the patterns recognized by the pattern recognition component. One instance of this is the use by the model analysis component of partially recognized events to characterize differences between actual behavior and models of intended behavior. Another use of recognized patterns is the cycling of an event tuple that represents the pattern into the event stream. This enables high-level patterns to be recognized in terms of their event expression member events. This is especially beneficial in recognizing behaviors whose constituents might be distributed throughout a system. A distributed event recognizer does not need to know the details of an entire behavior model. Instead it only needs to be aware of those events that are members of its defining event cluster.

§2. Difficulties for Behavior Monitoring Through Events

Syntactic pattern recognition methods for behavior recognition must be approached with some caution. In their traditional role, syntactic pattern recognition systems are intended to be used over a domain which is essentially static in nature. When a candidate pattern is presented to the pattern recognizer, its decomposition into a set of input primitives will have a known size and will form a single representative of the permissible patterns. An important problem for pattern recognition systems is the initial identification of the primitives that compose the pattern. After the pattern primitives have been isolated, classification of the pattern is a straightforward parsing exercise. In general, the patterns are reasonably perfect and are easily recognized if the pattern primitives can be accurately located.

In contrast with normal syntactic pattern recognition, pattern recognition to support behavior recognition is assured of perfect, distinguishable pattern primitives forming the stream. However, the behavior models supplied by a user as event

based pattern descriptions only sometimes match the patterns in the event stream. Rather than working to resolve primitive identification problems, pattern recognition supporting behavior recognition must resolve differences between pattern descriptions and patterns. In order to be useful for debugging, the differences between pattern descriptions and event stream contents should be information available to users.

The problems encountered by the event recognizer as it attempts to fit the event stream to a high-level event may be classified into three catagories:

Assignment problems are concerned with determining what behavior model will be able to make use of an event instance. After a pattern recognizer determines that an event instance is relevant to a model, the instance might be judged not suitable for a specific model due to attribute constraints. The other facet of assignment is that since many models may be under scrutiny simultaneously, means for appropriate distribution of event instances among models is necessary.

Resolution problems are concerned with how well an event instance fits a model. Resolution difficulties occur because of the dynamic environment within which the recognizer operates. Uncertainties concerning time in distributed systems and some combinations of attribute constraints create difficulties determining how well an event instance fits a model. When attribute constraints relate many of the cluster member events it might be necessary to accumulate an entire set of potential constituent events before any individual instances can be rejected as not applicable.

User problems result because of inaccuracies in user models. Users can supply inaccurate models due to their lack of knowledge of actual system behavior or simply because of errors in specifying their behavior models. However, when faced with models that do not match, the pattern recognizer and associated tools must provide adequate feedback to the tool user to enable a recasting of the models.

The following sections detail the causes of recognition problems and discuss their implications for the event recognition and model analysis components. Some of these

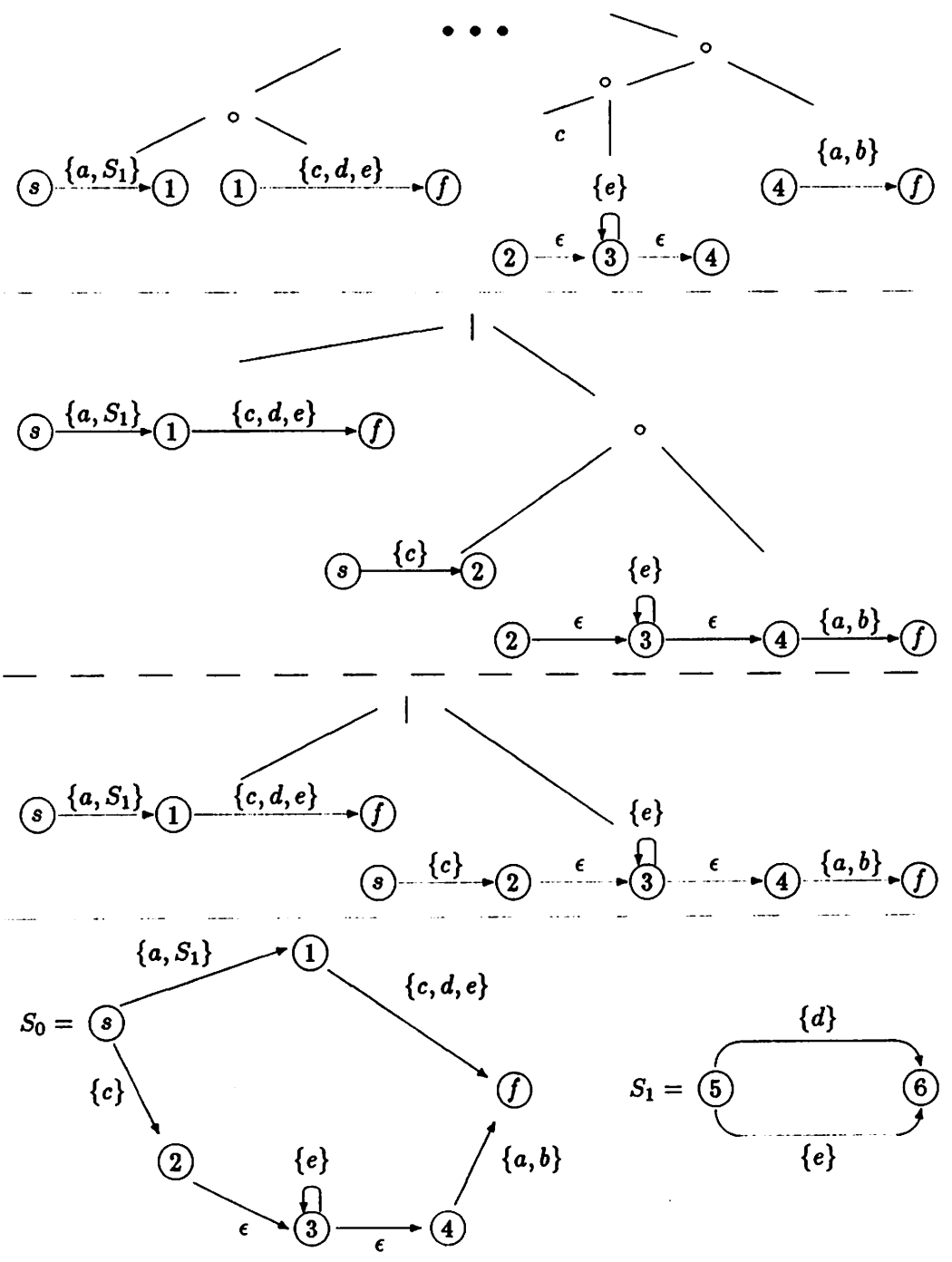


Figure 40: Final stages and complete NDSA

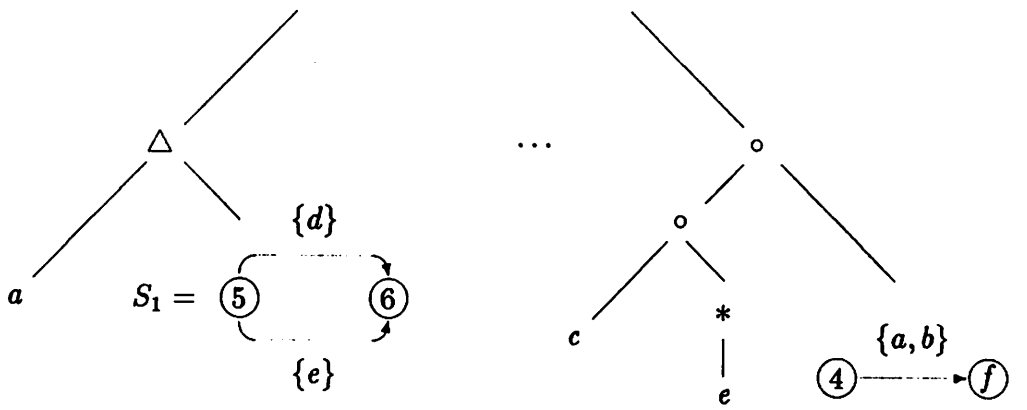


Figure 39: Early stages in synthesis of NDSA

All of the non-ϵ transition sets defined in the NDSA are defined in the DSA. That is

$$T_{NDSA} = T_{DSA} - \epsilon.$$

The state set and transition function of the DSA are formed by executing a variant on a standard algorithm [Aho77] using the states and transitions of the NDSA as input. Briefly, the algorithm accomplishes this by two "collapsing" actions:

- from a given state, s, all states reachable from s by ϵ-labeled transitions only, are collapsed into s, and
- two transitions $m(s,t)$ and $m'(s,t)$ from a state s on the same transition set, t, have their target states merged and a single transition $m''(s,t)$ is created in the DSA.

The collapsing actions result in each of the new DSA states being formed from a subset of NDSA states, elimination of ϵ-labeled transitions, and possibly a reduction in the number of transitions. The DSA to NDSA algorithm is detailed below. While the algorithm is being performed, assume there exists a set of "unmarked" states in the DSA. These are states which have been added to the DSA but have not had any outgoing transitions created yet. Recall that, in general, the *BSS* consists of a

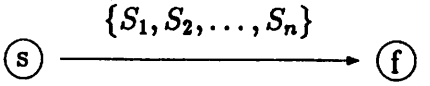
```
procedure SA(ee, s, f)
begin
   if ee = Φ then add_transition(s,ϵ,f)
   else
      case ee.operator of
         eventsymbol:
            add_transition(s, {ee.symbol}, f);
         alternation:
            SA(ee.left, s, f);
            SA(ee.right, s, f);
         catenation:
            i = new_state();
            SA(ee.left, s, i); SA(ee.right, i, f);
         star:
            i = new_state();
            add_transition(s, ϵ, i); add_transition(i, ϵ, f);
            SA(ee.operand, i,i);
         plus:
            i = new_state();
            SA(ee.operand, s, i); SA(ee.operand, i,i);
            add_transition(i, ϵ, f);
         shuffle:
            tset = {};
            while ee.operator = shuffle do
               if ee.rhs.operator = eventname then
                  tset = tset ∪ {ee.rhs.symbol}
               else
                  tset = tset ∪ {ShuffleRoutine(ee.rhs)};

               if ee.lhs.operator = eventname then
                  tset = tset ∪ {ee.lhs.symbol}
               else if ee.lhs.operator ≠ shuffle then
                  tset = tset ∪ {ShuffleRoutine(ee.lhs)};
               ee = ee.lhs
               end while
            add_transition(s, tset, f)
         end cases
end SA
```

Figure 38: Routine to synthesize an NDSA

2. The e_i represents a subexpression. For this case, *ShuffleRoutine(e_i)* is called to create the transitions for the *SA* described by e_i and return the symbol S_i for that *SA* subexpression.

The net result is the transition



where each S_i is either a simple event symbol or a symbol representing a shuffle automaton in the set S for the basic shuffle system.

The procedure is expressed in figure 38. The argument *ee* is the root of the basic part to be translated, *s* and *f* are the start and finish states to be connected. The routine *add_transition(s,t_set,f)* creates a transition from a start state *s* to a finish state *f* on transition set *t_set*, i.e. the transition

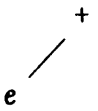
$$m(s,t\_set) = f$$

is added to the transition function m for a shuffle automaton. The *add_transition()* function is also responsible for creating the set of transition sets T for the basic shuffle system. Each time function *add_transition()* is called to add a transition, $m(s,t\_set) = f$, to the transition function, *add_transition()* checks to determine if *t_set* is already an element of T, if not, *t_set* is added.

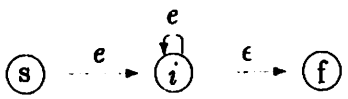
At this stage of the construction of the shuffle automaton from an event expression, a non-deterministic shuffle automaton (NDSA) exists which consists of a set of states connected by transitions that are labeled with transition sets. Some of the transition sets have sets of input symbols on their arcs, others are null sets. The last parts of figure 40 are the components of a non-deterministic *BSS* derived from the expression of figure 36.

To create the deterministic shuffle automaton (DSA) from the NDSA it is necessary to eliminate the non-deterministic and ϵ-labeled transitions from the NDSA.

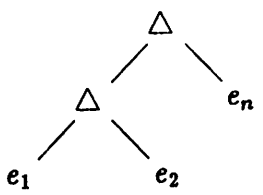
plus operator, representing the event subexpression e^+, partitioned into the basic part



Since e^+ is equivalent to $e \circ e^*$, create a new state i to be the iteration starting point for e, call $SA(e, s, i)$ to create the transition for the mandatory e, add a transition from i to f on the null symbol, and call $SA(e, i, i)$ to add the iterative transition represented by e. The resulting machine is

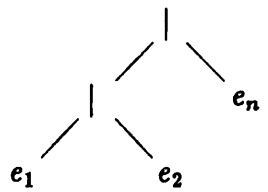


shuffle operator series, from the event expression string $e_1 \triangle\ e_2 \triangle\ \ldots \triangle\ e_n$ with the corresponding basic part

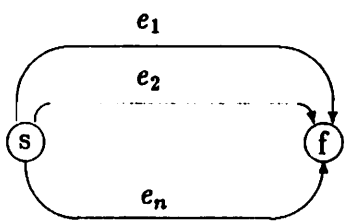


Construct a transition set from symbols $s_1, s_2, \ldots, s_n$ which will correspond to subexpressions $e_1, e_2, \ldots, e_n$. There are two cases for each of the e_i's

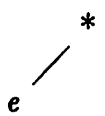
1. The e_i is a symbol name. For this case the symbol itself is added to the transition set as the s_i element (even though it forms a basic part in its own right).



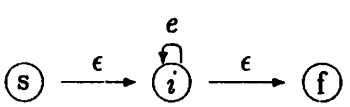
Alternation creates no new states, only alternate paths from the start to finish states, each based on one of $e_1, e_2, \ldots, e_n$. For each of $e_1, e_2, \ldots, e_n$, call $SA(e_i, s, f)$ to result in the machine



star operator for the event subexpression e^*, i.e. the basic part



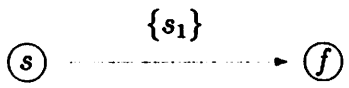
create a new state, i, to be the iteration starting point for e. Add transitions on the null symbol[3], ϵ, from s to i and i to f. Then create the iterative transitions represented by e by calling $SA(e, i, i)$. The result is a machine



[3] ϵ is the empty transition. Since no transition symbol is required, the empty transition is taken at the whim of the finite state control.

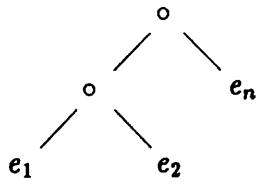
described by the event expression. Each call to *SA()* may add states to the shuffle automaton as well as transition sets to the *BSS*, depending on the argument event expression. If the event expression represents a basic part consisting of a(n):

event name symbol, s_1, create a transition set consisting of the event symbol, i.e. $\{s_1\}$, and add a transition

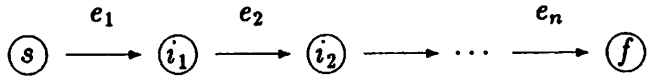


to the *SA*;

catentation operator series, representing the event expression string $e_1 \circ e_2 \circ \ldots \circ e_n$, the basic part will be in the form



Create new states $i_1, i_2, \ldots, i_{n-1}$ to connect transitions represented by each of $e_1, e_2, \ldots, e_n$ into a machine



To create the transitions, for each e_i perform $SA(e_i, i-1, i)$ (Starting with $SA(e_1, s, i)$ and finishing with $SA(e_n, i_{n-1}, f)$.)

alternation operator series, representing the event expression string $e_1 \mid e_2 \mid \ldots \mid e_n$, the basic part will be in the form

```
function ShuffleRoutine(ee) returns SAsymbol
begin

    -- create unique starting state s, finishing state f and
    -- a symbol for the resulting shuffle routine.
       s ← new_state(); f ← new_state();
       S ← create_name_S_i();

    -- create the transition from s to f on occurrence of the event
    -- subexpression given by the argument ee.
       SA(ee, s, f);

       ShuffleRoutine()←S
end;
```

Figure 37: Create a symbol representing an *SA*

set. A basic part consisting of a subexpression represents a start and finish state connected by a transition described by the part. If a subexpression basic part is an operand of a shuffle operator, it will contribute a shuffle automaton recognizing that subexpression to the *BSS* as well as a shuffle automaton symbol to the transition set for the transition. Other basic parts contribute transitions from the start and finish states and possibly contribute created states to the *SA*.

The algorithm consists of two mutually recursive procedures. One procedure, *ShuffleRoutine()*, is responsible for initial creation of an independent shuffle automaton within the *BSS*. The other, *SA()*, creates transitions and transition sets corresponding to a simple shuffle automata. Figure 37 is the controlling procedure *ShuffleRoutine()*. Its argument is an event expression parse tree partitioned into its basic parts. Each call to *ShuffleRoutine()* adds a shuffle automaton to the *BSS*. Consequently, *ShuffleRoutine()* returns a symbol representing the *SA* that has been added.

The arguments to *SA()* are an event expression that has been partitioned into its basic parts, and start and finish states that are connected by the automaton

$$(a \triangle d \mid e) \circ (c \triangle d \triangle e) \mid c \circ e^* \circ (a \triangle b)$$

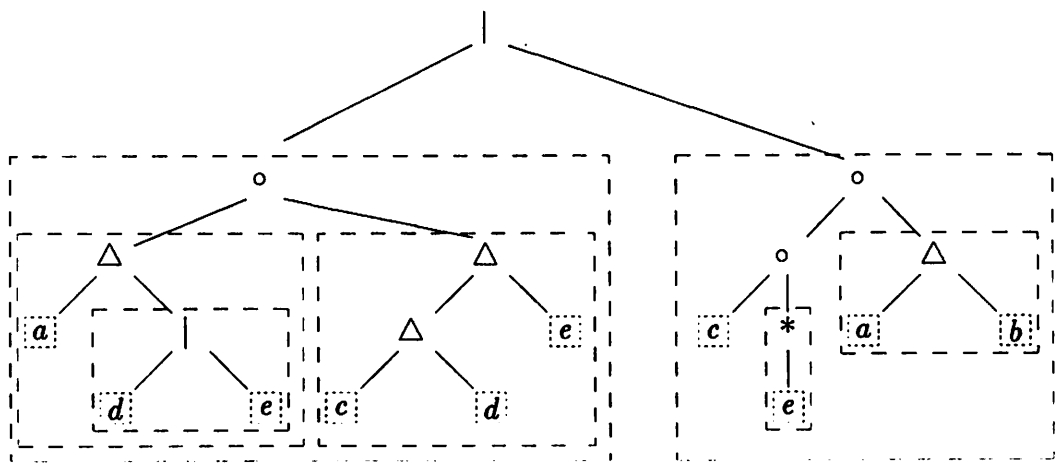


Figure 36: Event expression basic parts

plan is to synthesize a non-deterministic shuffle automaton (*NDSA*) from the event expression; then eliminate non-deterministic and ϵ-valued transitions.

To construct a *BSS* from an event expression the event expression is first logically partitioned into its basic parts. Each part is either a simple symbol, an expression operator and its operand(s), or a subexpression for a shuffle operator. The basic parts are easily derived from a parse tree for the expression (see figure 36). Each of these basic parts or subexpressions contributes a symbol to a transition set and/or a transition that will recognize an instance of the subexpression. The subexpressions (machines) are recursively synthesized into a basic shuffle system. Figures 39-40 shows some of the steps in the synthesis of a shuffle automaton from the expression tree of figure 36.

To synthesize a *BSS* from the basic part partitioning, perform a traversal of the partitioned expression tree to create transition sets and transitions from the basic parts. Each basic part that is an event symbol is simply included in a transition

continues to make transitions, with sub-machines called as needed, until the *fmove* shuffle automaton enters one of its final states and accepts.

The actual behavior model for the *fmove* behavior model is

$$E_{startup} \cup E_{S1_1} \cup E_{S1_2} \cup E_{fileaccess} \cup E_{shutdown} \cup E_{fmove}$$

The *BSS* model is also capable of characterizing sharing of events. Note in the preceding example that

$$E_{startup} \cap E_{S1_1} \cap E_{S1_2} \cap E_{fileaccess} \cap E_{shutdown} \cap E_{fmove} = \{\}.$$

In this example, no events are shared among the constituents of the actual behavior model. Sharing occurs when the event sets containing actual behavior model constituents overlap, i.e.

$$E_{startup} \cap E_{S1_1} \cap E_{S1_2} \cap E_{fileaccess} \cap E_{shutdown} \cap E_{fmove} \neq \{\}$$

In chapter IV, sharing is presented as a problem for behavior recognition. Given the current definition of shuffle automata operation sharing events cannot be directly controlled by the individual members of a *BSS*. In the prototype toolset, sharing can be controlled by not permitting the members of a *BSS* to share events, as in the example just presented. However, sharing across two distinct high-level event requests (represented by basic shuffle systems without a common root) is permitted, since its is assumed that a user requesting recognition of multiple high-level events described by different *BSS* is more interested in whether the two high-level events exist, and less in their being disjoint.

§4. Construction of Shuffle Automata from Event Expressions

Constructing a basic shuffle system from an event expression is straightforward. The algorithms presented in this section will effectively create a *BSS* consisting of a set of deterministic simple shuffle automata from an event expression. The basic

$create_1$
$dispatch_1$
$startup_1$
$preempt_1$
$dispatch_2$
$S1_1$
$preempt_2$
$dispatch_3$
$S1_2$
$block_1$
$wake_1$
$dispatch_4$
$fileaccess_1$
$destroy_1$
$shutdown_1$
$fmove_1$

$E_{startup} = \{create_1, dispatch_1\}$
$E_{S1_1} = \{preempt_1, dispatch_2\}$
$E_{S1_2} = \{preempt_2, dispatch_3\}$
$E_{fileaccess} = \{S1_1, S1_2, block_1, wake_1, dispatch_4\}$
$E_{shutdown} = \{destroy_1\}$
$E_{fmove} = \{startup_1, fileaccess_1, shutdown_1\}$

Figure 35: Event stream and event sets for *fmove*

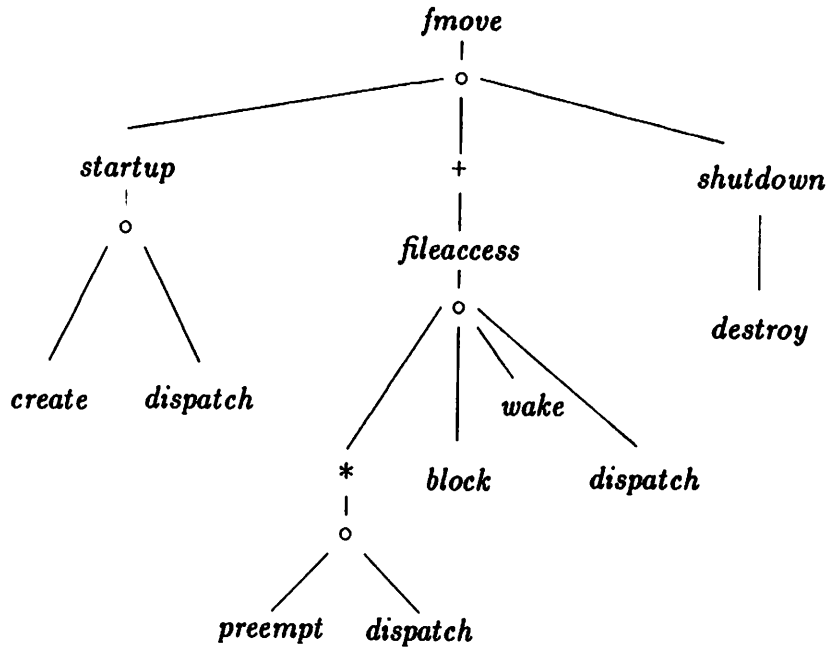


Figure 34: *fmove* model structure

As the event stream enters, *startup* makes transitions and accepts with event set

$$E_{startup} = \{create_1,\ dispatch_1\}.$$

The event symbol for *startup* is placed into the input register for *fmove* allowing *fmove* to take a transition to state 2. From state 2 the *BSS* requests a *fileaccess* event. *Fileaccess* is added to the active list, the finite state control for *fileaccess* is placed into its initial state (state 8), and a request for the subexpression *S1* is made. *S1* is initialized accordingly. The active set is now

$$\{fmove,\ fileaccess,\ S1\}.$$

S1 is recognized with event set

$$E_{S1} = \{preempt_1,\ dispatch_2\}$$

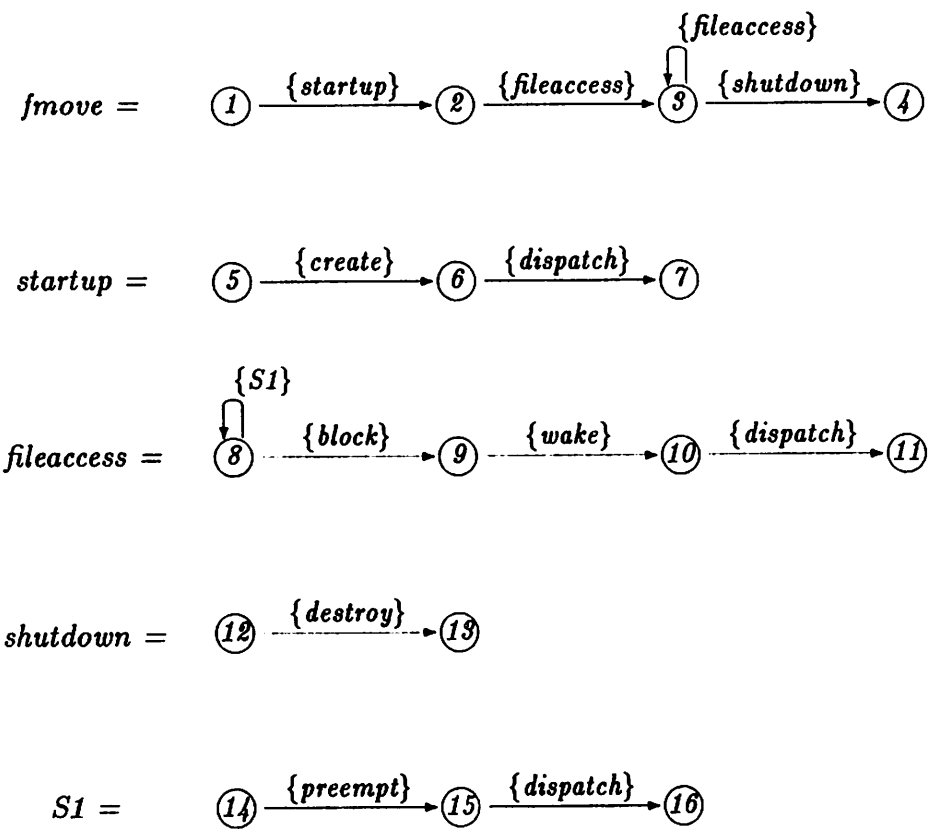
and returns an *S1* symbol to the calling state of *fileaccess*. *Fileaccess* takes the transition, *m(8,{S1})* and upon re-entering state 8, requests *S1* again. The *BSS*

$$fmove = startup \circ fileaccess^{+} \circ shutdown$$

where

$$startup = create \circ dispatch,$$
$$shutdown = destroy,$$
$$fileaccess = (preempt \circ dispatch)^{*} \circ block \circ wake \circ dispatch.$$

A basic shuffle system to recognize the behavior described by the event expression is:



and has the structure indicated by figure 34.

Using the event stream of figure 35, we can trace the activity for the *fmove BSS*. Initially, the *BSS* is placed in its start state (state 1) and a request for *startup* is made. The finite state control for *startup* is placed into its initial state (state 5). The active set for the *fmove BSS* is now

$$\{fmove, startup\}.$$

A higher level behavioral model is recognized when its defining shuffle automaton reads a set of events which cause it to enter a final state.

For a simple shuffle automaton

$$S = (Q, \Sigma, T, m, q_0, r_{map})$$

a recognition by S defines an event set

$$E = \{s_i \mid s_i \in \Sigma\}$$

such that there is a series of transitions from the initial state to one of the final states based on the sets of symbols presented to S

$$m(q_0, t_j), m(q^1, t_j^1), \ldots, m(q^m, t_j^m) \in F$$

where

$$\forall i, \exists j, m : s_i \in E \;\Rightarrow\; s_i \in t_j^m$$

Recognition of a high-level event by a simple shuffle automaton defines a model of actual system behavior whose constituent events are members of the event set E. For this to work for all event expressions, it is necessary to generalize to the basic shuffle system.

Each $S_i \in S$ in a *BSS* can be shown to be the same as a simple shuffle automata with an appropriately defined alphabet. Viewed in this way, a *BSS* represents a collection of simple shuffle automata, each having its own event set. Some event set elements are primitive events, some are representatives of complex shuffle expression operands, and some are high-level events. The actual behavior model for an event expression represented by a *BSS* is the union of all the event sets bound to the constituent $S_i \in S$ for the basic shuffle system. A recognition by a basic shuffle system

$$BSS = (\Sigma, S, T, A)$$

occurs if a stream of events presented to the shuffle automaton contains a set of symbols which cause the basic shuffle system root to perform transitions and eventually enter a final state. For example, for the event expression

the input register accordingly,

if $\exists i,j : m(q_{S_k^i}, t_j) \neq \perp$ **and** $t_j \subseteq R_{S_k^i}$

then

$q_{S_k^i} \leftarrow m(q_{S_k^i}, t_j)$,

$R_{S_k^i} \leftarrow r_{map}(R_{S_k^i})$

forall $j,l : m(q_{S_k^i}, t_j) \neq \perp$ **and** $S_l \in t_j$

$A \leftarrow A \cup \{S_l^{new(i,q)}\}$,

$q_{S_l^{new(i,q)}} \leftarrow q_{0S_i}$,

$R_{S_l^{new(i,q)}} \leftarrow \{\}$

else goto step 2

4. When a shuffle automata S_k^i in the active set enters a final state, the S_k^i returns to its calling shuffle automaton and is removed from the active set,

if $\exists i : S_k^i \in A$ **and** $q_{S_k^i} \in F$,

then

$A \leftarrow A - \{S_k^i\}$,

if $k = 0$ **then** accept,

$R_{S_k^{parent(i)}} \leftarrow R_{S_k^{parent(i)}} \cup S_k$

5. **goto step 3**

§3. Recognition, Models and Sharing

A shuffle automaton is capable of detecting a designated pattern of symbols generated by a set of sources for these symbols. These patterns can be made of symbol sets which are sequential or concurrent in nature. Behavioral models expressed as *EDL* event expressions can be represented as a *BSS* which is capable of recognizing the behavior. In the automaton model, the input symbols in the alphabet of the shuffle automaton correspond to events generated by the system being observed.

shuffle automata they should be added.

In the more rigorous explanation of *BSS* operation that follows, S_k^i indicates an instance of shuffle sub-machine S_k with a unique id i. The unique id maps the instance back to the state which made the call. A new identifier can be made from the parent identifier by catenating the current state number for S_k^i (q) to the parent identifier (i)

$$new(i, q) = i \bullet q.$$

The created identifier for a sub-machine can have the state part stripped away and the identifier of the parent sub-machine returned by a function, *parent(i)*. The more rigorous definition of *BSS* operation follows.

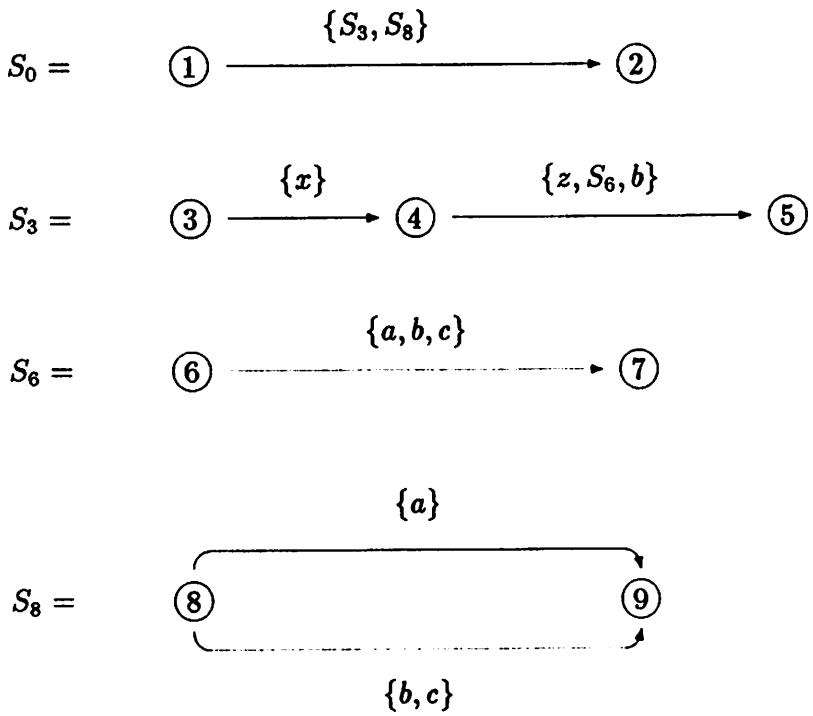
1. the root machine for the *BSS* is preset, and any sub-machines needed to exit the initial state are started,

   $q_{S_0^0} \leftarrow q_0$,
   $R_{S_0^0} \leftarrow \{\}$,
   $A \leftarrow \{S_0^0\}$
   forall $j, l : m(q_0, t_j) \neq \perp$ **and** $S_l \in t_j$
   $A \leftarrow A \cup \{S_l^{new(0,q)}\}$,
   $q_{S_l^{new(0,q)}} \leftarrow q_0 s_l$,
   $R_{S_l^{new(0,q)}} \leftarrow \{\}$

2. As the symbol generator presents symbol sets $\{s_1, s_2, \ldots, s_n\}$, the finite state controls for active shuffle automata evaluate a symbol distribution function to determine which symbols to add to their input registers

   input $\{s_1, s_2, \ldots, s_n\}$,
   forsome $i, j : S^i \in A, s_j \in \{s_1, s_2, \ldots, s_n\}$,
   $R_{S_k^i} \leftarrow R_{S_k^i} \cup \{s_j\}$

3. When the input register of a shuffle automaton in the active set contains one of the outgoing transition sets for the current state of the shuffle automaton, the finite state control for the shuffle automaton makes a transition and alters



$$\Sigma = \{a, b, c, x, z\}$$
$$S = \{S_0, S_3, S_6, S_8\}$$
$$T = \begin{array}{c}\{t_1 = \{S_1, S_8\}, t_2 = \{x\}, t_3 = \{z, S_6, b\},\\ t_4 = \{a, b, c\}, t_5 = \{a\}, t_6 = \{b, c\}\}\end{array}$$
$$A = \{S_0\}$$

$Q_{S_0} = \{1,2\}$, $q_{0S_0} = 1$, $F_{S_0} = \{2\}$, $M_{S_0} = \{m(1,t_1) = 2\}$.
$Q_{S_3} = \{3,4,5\}$, $q_{0S_3} = 3, F_{S_3} = \{5\}$, $M_{S_3} = \{m(3,t_2) = 4, m(4,t_3) = 5\}$.
$Q_{S_6} = \{6,7\}$, $q_{0S_6} = 6$, $F_{S_6} = \{7\}$, $M_{S_6} = \{m(6,t_4) = 7\}$.
$Q_{S_8} = \{8,9\}$, $q_{0S_8} = 8$, $F_{S_8} = \{9\}$, $M_{S_8} = \{m(8,t_5) = 9, m(8,t_6) = 9\}$.

Figure 33: Basic shuffle system

Each column is labelled with the name of a shuffle automaton, $S_i \in S$; likewise the rows. For each transition set t_j for which $m(q, t_j)$ is defined for S_i, if some shuffle automata symbol $S_k \in t_j$, then indicate in the matrix that S_i is connected to S_k. The reflexive transitive closure of the connectivity matrix indicates whether there are any cycles in the graph.

At any time, many *SA* in the *BSS* can be active. To monitor these active shuffle automata, the finite state control contains a set, A, of "active" shuffle automata. This set contains symbols representing those sub-machines which have been called from a state, but have not yet completed their recognition tasks. Figure 33 shows an example basic shuffle system.

A *BSS* begins operation by setting the active shuffle automata set to contain only the root machine S_0 and placing the root machine in its distinguished start state, q_{0S_0}. The root machine proceeds in a similar manner to the *SSA* by comparing the input register to transition sets and performing a transition when a transition set is contained in the input register. However, the finite state control must account for any S_k in the transition sets. When a state q of an active S_k in the *BSS* is entered, each transition $m(q, t_j)$ is examined. Any S_k which are elements of the t_j are called from the state q by adding a copy of the S_k to the active shuffle automata set, placing its finite state control in the starting state q_{0S_k}, and clearing the input register for S_k. Each S_k so started can operate in parallel with other active shuffle automata. Whenever an S_k enters a final state it is removed from the active set and its symbol is added to the input register associated with its caller. The *BSS* *recognizes* a pattern when the S_0 enters its final state. It is not necessary that the active set be empty for the *BSS* to accept an input.

An important issue here concerns the addition of input symbols to the input register of an active shuffle automaton. The extremes are: an input symbol is added to one input register of a single shuffle automaton; enough copies are made to give to all of the shuffle automata in the active set which can use one. To these possiblities add that of having the *BSS* decide how many copies should be made and to which

A *BSS* is defined by the 4-tuple,

$$BSS = (\Sigma, S, T, A)$$

where:

Σ – input alphabet for the *BSS*, $\{s_1, s_2, \ldots, s_n\}$,
S – set of Shuffle Automata, $\{S_i \mid S_i \text{ is a shuffle automaton}\}$,
T – transition sets, $\{t_i \mid t_i \subseteq \Sigma \cup S\}, \forall\, i, j\ t_i \neq t_j$
A – "Active" shuffle automata $A \subseteq S^*$.

The set of Shuffle Automata, S, contains at least the "root" shuffle automaton, S_0. The definition of an individual shuffle automaton, S_i, in the *BSS* is changed slightly from the *SSA* definition to accomodate the input alphabet that is shared by all of the $S_0, S_1, \ldots, S_n$. Each S_i is defined as a 5-tuple,

$$S_i = (Q_{S_i}, m_{S_i}, q_{0S_i}, F_{S_i}, r_{mapS_i})$$

with

Q_{S_i} – set of states for shuffle automaton S_i,
m_{S_i} – transition function, $Q_{S_i} \times T \rightarrow Q_{S_i}$,
q_{0S_i} – starting state for S_i, $q_{0S_i} \in Q_{S_i}$
F_{S_i} – final states for S_i, $F_{S_i} \subset Q_{S_i}$
r_{mapS_i} – input register mapping function for $S_i, (\Sigma \cup S)^* \rightarrow (\Sigma \cup S)^*$

Each component serves the same purpose and has the same meaning as the corresponding components of the *SSA*.

The transition sets, T, tie all of the shuffle automata in the *BSS* together. The definition of transition sets has been extended from the *SSA* definition to include names of other shuffle automata in the *BSS*. This permits a state to invoke other shuffle automata to supply symbols needed to satisfy a transition set and hence to move from the state. It is important that the transition sets are not defined in a circular or recursive fashion. It is easy to determine if there are cyclic definitions of events. Create an $n \times n$ connectivity matrix for the shuffle automata in the *BSS*.

[Bates78]. Among the significant differences is the parallel operation style of the shuffle automata versus the subroutine invocation style of ATN's.

The need for sub-machines also results from an implication of the *SSA* model that the shuffle operator can only connect simple event symbols. The use of a shuffle operator

$$e_1 \triangle e_2 \triangle \ldots \triangle e_n$$

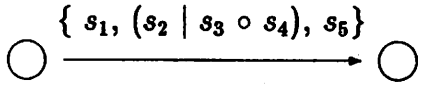
is only possible if all of the $e_1, e_2, \ldots, e_n$ are event symbols. The reason for this is the transition sets have no means to represent objects other than symbols as set members. For example, the event expression

$$s_1 \triangle (s_2 \mid s_3 \circ s_4) \triangle s_5$$

is not directly describable as a simple shuffle automaton because the subexpression

$$\ldots(s_2 \mid s_3 \circ s_4) \ldots$$

needs to be represented in a transition such as



The *BSS* overcomes this difficulty by replacing the $(s_2 \ldots)$ subexpression with a symbol to represent it in the transition set.

The shuffle automata forming the *BSS* are effectively merged into a single pattern recognition entity having a distinguished *root* machine. Once started in the root machine initial state, the *BSS* operates as a set of parallel *SA*. Each *SA* in the *BSS* may be started independently of the others and multiple copies of a *SA* may be active at any time. These possiblities arise because many transition sets marking transitions from a given state may have called for sub-pattern recognition when the state is achieved. From this perspective, the *BSS* is more of a higher organizational level than a machine radically different from the simple shuffle automata.

turned loose on the event stream. When a shuffle automaton enters a final state, it has recognized a behavior in the system. In the tool use of abstracted events, the recognized event is sometimes pushed back onto the event stream so it might be incorporated into higher level event models or distributed to remote nodes of a distributed system. The tools can supply information regarding the event set that was bound to the behavior model indicated by the simple shuffle automaton.

§2. Basic Shuffle System

The *SSA* is a good basis for pattern recognition in support of event based behavioral abstraction primarily because of its ability to easily model concurrency and filter unnecessary information. The formalism also provides improved time and space characteristics over the finite state automata model for complex behavior descriptions. What are lacking for behavior modelling purposes are capabilities for describing hierarchical models and employing event attributes as an aid to filtering. Filtering will be treated in a later section. To facilitate hierarchical description of complex models, the *SSA* model will be extended so that complex shuffle automata may be abstracted into single transition set elements.

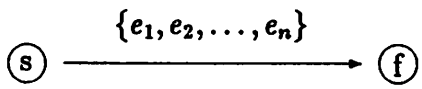
A *Basic Shuffle System (BSS)* is a collection of shuffle automata[2] (*SA*) with a common input alphabet and whose transition sets are allowed to include symbols representing other shuffle automata. A shuffle automaton in the *BSS* operates as does a simple shuffle automaton but is capable of *calling* other shuffle automata in the *BSS* to recognize subparts of a pattern of symbols. For each recognition by a shuffle automaton in the *BSS*, a copy of the symbol used to represent the shuffle automaton is created and may be placed into an input register. Thus, shuffle automata sub-machines may be used in the same way as ordinary alphabet symbols to effect transitions from state to state by a shuffle automaton. The call mechanism is similar to that found in the Augmented Transition Network formalism [Woods70],

[2]Shuffle automata (*SA*) includes both the *SSA* and *BSS*.

Finally, the shuffle operator which indicates concurrency for a model

$$E = e_1 \triangle e_2 \triangle \ldots \triangle e_n$$

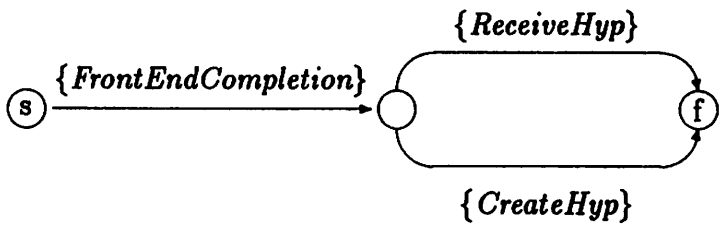
is recognized by the following single transition machine.



As an example, consider the set of behaviors specified by the example from the last section of chapter III, *FirstSigActivity.*

$$FirstSigActivity = FrontEndCompletion \circ (ReceiveHyp \mid CreateHyp)$$

The recognizer for this model is the simple shuffle automaton



Where

$$FrontEndCompletion = InstantiateKsi \circ CreateHyp^* \circ InstantiateKsi \circ InvokeKsi$$

has a corresponding recognizer

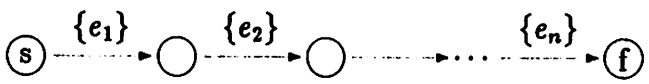
A suitable set of tools (described in later chapters) allows a user to describe behaviors as *EDL* descriptions which are then recast as recognizing automata and

automata. *SSA* are not capable of using constraints to filter events and cannot model complex shuffle expressions without eliminating their concurrency characteristics. The *BSS* described in the next section is capable of describing all event expressions and the *CSS* described in a following section can use constraining expressions to filter the input event stream.

A behavior model specified by an event expression consisting of a series of sequential behaviors such as

$$E = e_1 \circ e_2 \circ \ldots \circ e_n$$

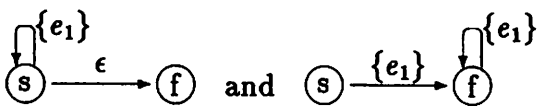
is recognized by a simple shuffle automaton



Likewise event expressions involving the iterative operators such as,

$$E = e_1{}^* \text{ and } E = e_1{}^+$$

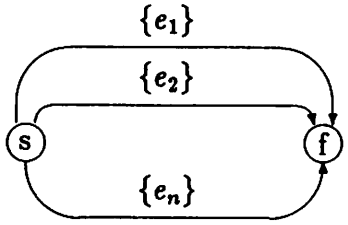
have the respective recognizers



Choice among behaviors, such as in the event expression

$$E = e_1 \mid e_2 \mid \ldots \mid e_n$$

is recognized by a simple shuffle automaton



```
procedure create_transition_net(t_set, start, finish);
- - create a transition network, from start to finish,
- - based on the elements of t_set. This effectively enumerates
- - all of the subsets of t_set.
begin
      node_set = {(ϵ, start), (t_set, finish)};
      create_net(ϵ, t_set, start)
end create_transition_net()
 procedure create_net(prefix, remaining, start);
- - create transitions for each element in remaining
- - from the start node (representing prefix)
begin
      if remaining = {} return;        - - this is the finish node
      forall σ ∈ remaining do
            s = statefor(prefix · σ);
            δ_fsa = δ_fsa ∪ δ(start,σ) = s;
            create_net(prefix · σ, remaining−{σ}, s)
end create_net()
 function statefor(prefix) returns state;
- - return the state that corresponds to prefix in node_set
begin
      forall i : (string_i, state_i) ∈ node_set
            if string_i = prefix then return state_i
      node_set = node_set ∪ {(prefix, new_state())}
end statefor()
```

Figure 32: Functions to create transition nets

it is necessary to construct a transition network like those of figure 30.

$s \leftarrow selectone(U)$,
forall $m(s, t_i) = s'$
  if $\mid t_i \mid = 1$ **then**
    $\delta_{fsa} = \delta_{fsa} \cup \delta(s, \sigma \in t_i) = s'$
  else
    create_transition_net(t_i, s, s')

4. If U is empty, the construction is complete, otherwise more transitions need to be created,

   if $U = \{\}$ **then** return **else** goto step 3

The function *create_transition_net(tset, start, finish)* (figure 32) constructs a digraph whose paths from the *start* to *finish* nodes enumerate all strings of length $\mid$ *tset* $\mid$. The algorithm creates nodes corresponding to each possible substring formed from the set *tset*. The edges are labeled with single symbols which carry a string of length l from a node labeled *prefix* to a string of length $l + 1$, labeled *prefix* · *symbol*. This is accomplished depth first from a start node to the finish node by the auxilary function *create_net(prefix,remaining,start)* that creates the individual transitions. The *create_net()* (figure 32) function assumes the existence of a global variable *node_list* that contains pairs representing the nodes already created. Each pair is a *(string, state)* tuple that represents the state that the *FSA* would be in after it has seen the substring of symbols from the transition set as indicated by *string*. The *statefor(prefix)* function searches the created node list for a node labelled by its argument and returns the corresponding state name. If the prefix does not exist yet, it is added and a new state name assigned.

§1.4 *SSA as Behavior Recognizers*

An *SSA* can be used to recognize behaviors that are specified by *EDL* event descriptions. The correspondence between *EDL* event expressions and the *SSA* will be sketched here. Not all *EDL* descriptions can be represented by simple shuffle

result in system activity, but the actual software behavior will depend on other elements of the system.

Taking the preceding operational differences into account, we can create an *FSA* that is equivalent to an *SSA* with $rmap : R \leftarrow \{\}$ and which reads its symbols from a sequential stream. For a *SSA*

$$SSA = (Q_{SSA}, \Sigma_{SSA}, T_{SSA}, m_{SSA}, q_{0SSA}, F_{SSA}, r_{mapSSA}).$$

with

$$T_{SSA} = \{t_i \mid \forall \sigma_i \in t_i, \sigma_i \in \Sigma\}$$
$$r_{map} : R \leftarrow \{\}$$

construct an *FSA*

$$fsa = (Q_{fsa},\ \Sigma_{fsa},\ \delta_{fsa},\ F_{fsa},\ q_{0fsa})$$

as follows.

1. The alphabets are the same and all of the states of the *SSA* are contained in the state set for the *FSA* so

   $\Sigma_{fsa} = \Sigma_{SSA}$,
   $Q_{fsa} = Q_{SSA}, q_{0fsa} = q_{0SSA}, F_{fsa} = F_{SSA}$

   What remains to be done is to construct δ_{fsa}, which might add states to Q_{fsa}.

2. Create an "unexplored state" set from the *FSA* states that correspond to the *SSA* states,

   $U \leftarrow Q_{fsa}$

3. Select and remove[1] a state s from U and construct *FSA* transitions from s which are equivalent to *SSA* transitions $m(s,t) \neq \perp$. For simple transitions (i.e. $\mid t_i \mid = 1$) simply add a transition to δ_{fsa}, on the symbol $\sigma \in t_i$. For $\mid t_i \mid > 1$

[1] *selectone()* is equivalent to a random choice of an element of a set, then removal of this element from the set.

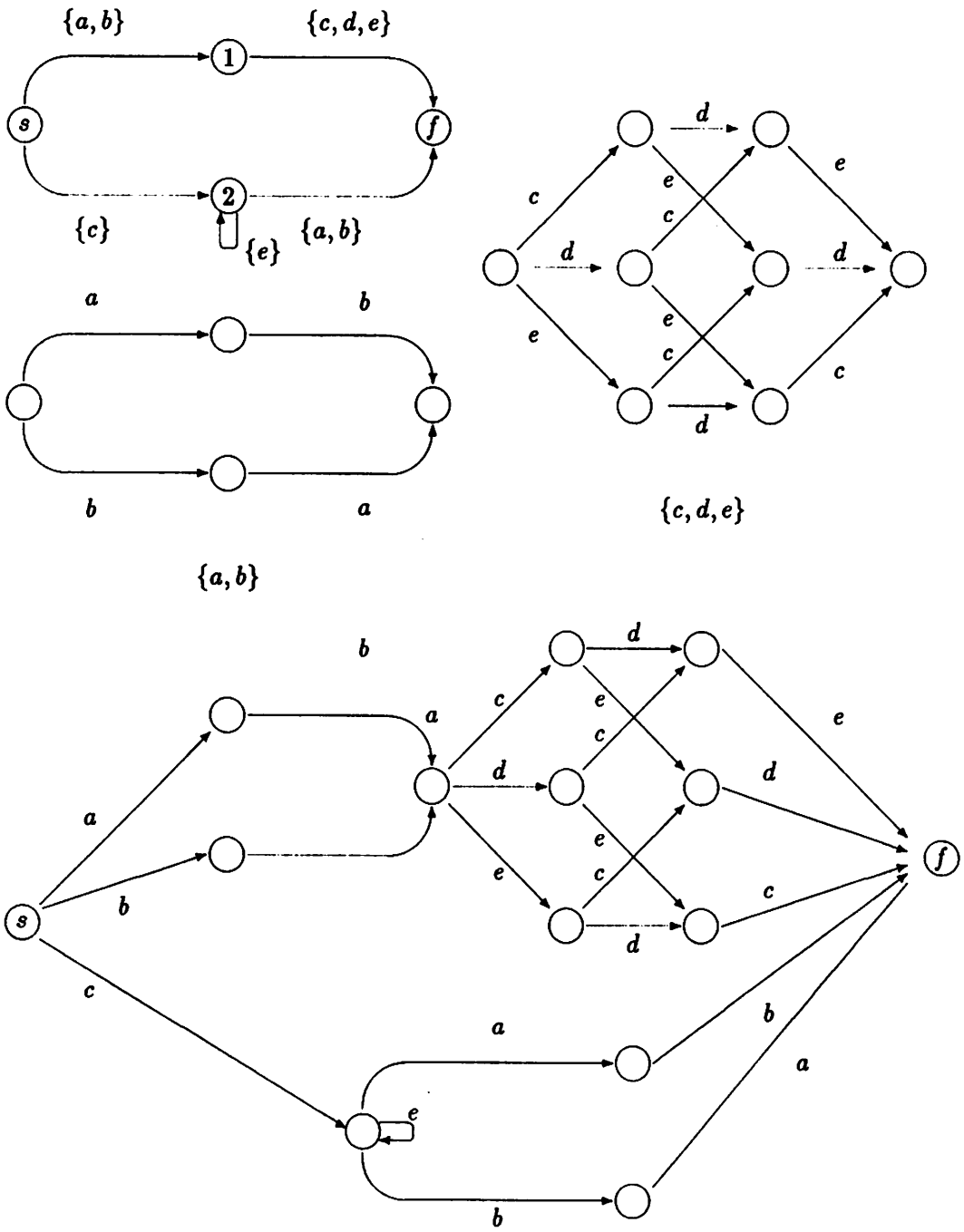


Figure 31: *SSA* **with expanded transition sets**

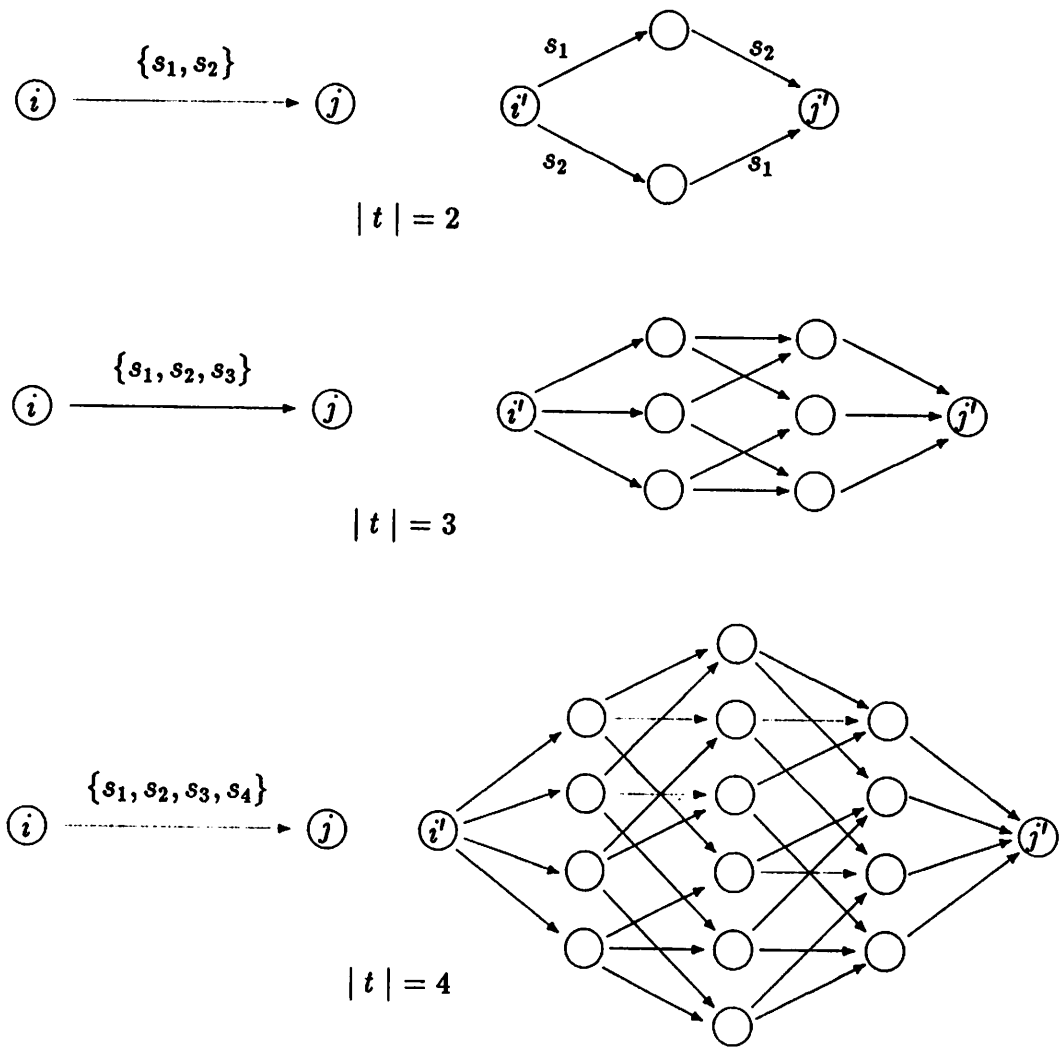


Figure 30: *SSA* **transitions with corresponding** *FSA*

Using sets of symbols for transitions introduces a simple and succinct way to model concurrency and is a primary difference between shuffle automata and finite state automata. An *SSA* transition $m(q,t)$, where $\mid t \mid = 1$, can be seen to be easily modelled as an *FSA* transition. The *SSA* of figure 29 clearly will accept the same set of strings as its corresponding *FSA*. However, the *SSA* is more powerful since if the input alphabet is extended by, say,

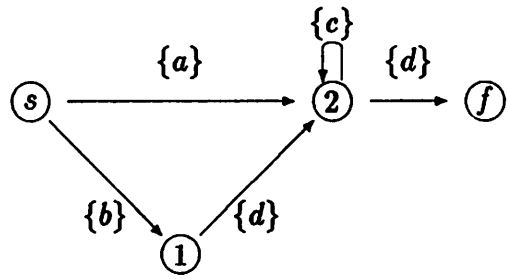
$$\Sigma' = \Sigma \cup \Sigma_0,$$

the shuffle automaton will still accept the same set of strings. The *SSA* effectively ignores the symbols which are elements of Σ_0 because the transition function is defined in terms of symbol subsets of the most recent input symbols, rather than exact matches. When applied to behavior recognition, this simple mechanism contributes to robustness and aids in filtering noise and some undesirable events from incorporation into models of actual behavior.

An *SSA* transition $m(q,t)$ for which $\mid t \mid > 1$ can be modelled with a corresponding finite state automaton if the constraint is imposed that input symbols are from a single serial symbol stream. The *FSA* and its corresponding *SSA* are equivalent in the sense that they *might* accept the same set of symbols by the time they enter a final state. An important difference is that the serial stream used by the *FSA* imposes an implicit order on the acceptance of individual symbols. Figure 30 shows the *SSA* transition and a corresponding *FSA* which might be used to model the transition, for transition sets of size two, three, and four. In each case, the *SSA* transition takes an *SSA* from a state i to a state j when $q = i$ and $R \supseteq \{s_1, s_2, \ldots, s_k\}$. The corresponding *FSA* model forms a network which effectively enumerates all possible strings of the transition set elements, $s_1, s_2, \ldots, s_k$, in taking the *FSA* from i' to j'. Figure 31 shows the shuffle automaton from figure 26 with each of its transition sets fully expanded into a finite state representation.

The other important difference from *FSA* models is the operational nondeterminism mentioned earlier. This is valuable for modelling complex distributed software since it is often possible to describe the structure and interactions that

$$
\begin{aligned}
Q &= \{s, 1, 2, f\} \\
\Sigma &= \{a, b, c, d\} \\
T &= \{\{a\}, \{b\}, \{c\}, \{d\}\} \\
m &= \begin{cases} m(s, \{a\}) = 2, \\ m(s, \{b\}) = 1, \\ m(1, \{d\}) = 2, \\ m(2, \{c\}) = 2, \\ m(2, \{d\}) = f \end{cases} \\
q_0 &= s \\
F &= \{f\} \\
r_{map} &= R \leftarrow \{\}
\end{aligned}
$$



$$
\begin{aligned}
Q &= \{s, 1, 2, f\} \\
\Sigma &= \{a, b, c, d\} \\
\delta &= \begin{cases} \delta(s, a) = 2, \\ \delta(s, b) = 1, \\ \delta(1, d) = 2, \\ \delta(2, c) = 2, \\ \delta(2, d) = f \end{cases} \\
F &= \{f\} \\
q_0 &= s
\end{aligned}
$$

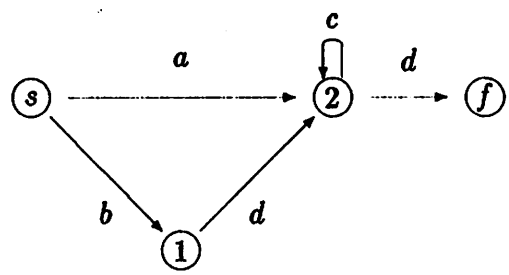


Figure 29: *FSA* with corresponding *SSA*

3. To construct the transition function for the *SSA*, for each transition from state q to q' defined in the *FSA* on the alphabet symbol σ, there is a corresponding transition in the *SSA* on a transition set $\{\sigma\}$.

$$\textbf{forall } \delta(q, \sigma) = q', m(q, \{\sigma\}) = q'$$

4. Finally, the r_{map} function must clear the input register R after each transition to ensure strict sequencing of input symbols:

$$r_{map} : R \leftarrow \{\}$$

Figure 29 is an example of a finite state automaton with its corresponding simple shuffle automaton.

Figure 27 illustrates the operation of a simple shuffle automaton (the one defined in figure 26) which watches a parallel event stream. The event stream first produces the set of symbols $\{a\}$, then the set $\{c, e\}$, and so on. The event set for the recognition by the shuffle automaton – the set of symbols finally accepted – is, $\{c,\ e,\ e,\ a,\ b\}$.

§1.3 *SSA Equivalence to FSA*

It is easy to see that by placing restrictions on transition sets and providing an appropriate r_{map} function, any finite state automaton has an equivalent shuffle automaton. In the simple case, a finite state automaton

$$fsa = (Q_{fsa},\ \Sigma_{fsa},\ \delta_{fsa},\ F_{fsa},\ q_{0fsa})$$

has a corresponding shuffle automaton

$$SSA = (Q_{SSA}, \Sigma_{SSA}, T_{SSA}, m_{SSA}, q_{0SSA}, F_{SSA}, r_{mapSSA}).$$

This is demonstrated by the following algorithm for construction of a shuffle automaton from a finite state automaton.

1. The alphabet and states of the *FSA* correspond exactly to those of the resultant *SSA*,

$$\begin{aligned} \Sigma_{ssa} &= \Sigma_{fsa} \\ Q_{ssa} &= Q_{fsa} \end{aligned}$$

   as well as the starting state and the set of finish states,

$$\begin{aligned} q_{0ssa} &= q_{0fsa} \\ F_{ssa} &= F_{fsa}. \end{aligned}$$

2. To construct the transition sets of the *SSA*, each symbol of the *FSA* alphabet is made into a transition set of the *SSA*,

$$\textbf{forall } \sigma \in \Sigma_{fsa},\ T_{ssa} = T_{ssa} \cup \{\sigma\}.$$

   Note that $\forall t_i \in T : \mid t_i \mid = 1$.

   This completes the static, structural parts of the *SSA*.

state control in state q_0 and the input register empty. As event symbols are presented to the control, they are added to the input multiset. When the finite state control is in a state q and the current contents of the multiset R contains one of the outgoing transition sets, t_i from state q, the transition determined by $m(q, t_i)$ is taken. The input register, R, is then altered according to the r_{map} function defined for the shuffle automaton. More precisely:

1. The finite state control is placed in its initial state and its input register is cleared:
   $q \leftarrow q_0$,
   $R \leftarrow \{\}$,

2. The control waits for the symbol generator(s) to present an input symbol set, $\{i\}$, to the shuffle automaton. The control then adds the input into the input multiset R,
   input $\{i\}$,
   $R \leftarrow R \cup \{i\}$,

3. Compare each transition set t_i leading from the current state q to the contents of the input register R. If one of the transition sets is contained in the input register, make a transition to the state given by $m(q, t_i)$. If no transition results from the current input register contents, the control loops back to the step which will change the input register,
   if $\exists i : m(q, t_i) \neq \perp$ **and** $t_i \subseteq R$,
   then
   $q \leftarrow m(q, t_i)$, $R \leftarrow r_{map}(R)$
   else
   goto step2.

4. If the shuffle automaton has entered a final state it *accepts,* or *recognizes,* one of the patterns it describes, otherwise return to step 3 to attempt more transitions.
   if $q \in F$ **then** accept **else** goto step3.

input symbols	R	*state*	*applicable transition*
–	$\{\}$	s	
$\{a\}$	$\{a\}$	s	
$\{c,e\}$	$\{a,c,e\}$	s	$m(s,\{c\}), R \leftarrow R - \{c\}$
$\{d\}$	$\{a,e,d\}$	2	$m(2,\{e\}), R \leftarrow R - \{e\}$
$\{c\}$	$\{a,d,c\}$	2	
$\{e\}$	$\{a,d,c,e\}$	2	$m(2,\{e\}), R \leftarrow R - \{e\}$
$\{b\}$	$\{a,d,c,b\}$	2	$m(2,\{a,b\}), R \leftarrow R - \{a,b\}$
–	$\{d,c\}$	f	

$$event\ set = \{c,e,e,a,b\}$$

Figure 28: *SSA* **operation with** $r_{map} : R \leftarrow R - t_i$

Another effect of holding unused symbols in the input register is an operational non-determinism. Non-determinism is introduced, not in the definition of a shuffle automaton model, but during its operation. It is always possible to construct a shuffle automaton that appears deterministic in the traditional sense – all transitions from a given state are unique, i.e.

$$\forall\, q,i,j : m(q,t_i) \neq m(q,t_j) \Rightarrow\ t_i \neq t_j.$$

This is the usual restriction for deterministic finite state automata. However, shuffle automata are capable of non-deterministic behavior. It is certainly possible that during operation, a shuffle automaton could be waiting in state q with the above restriction on the transition sets from that state, but both $t_i \subset R$ and $t_j \subset R$. Which transition, m or m', will be taken, depends on other factors.

§1.2 *SSA Operation*

The shuffle automaton operates by making a sequence of moves which are determined by the current state of the finite state control and the contents of the input register multiset. The operation of a simple shuffle automaton begins with the finite

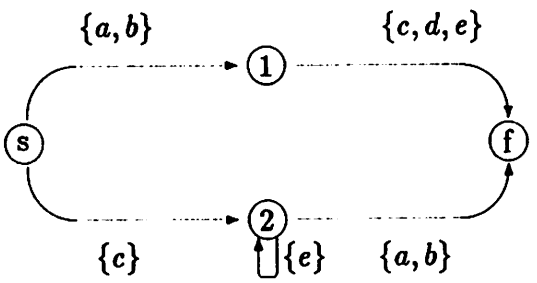
input symbols	R	*state*	*applicable transition*
–	$\{\}$	s	
$\{a\}$	$\{a\}$	s	
$\{c,e\}$	$\{a,c,e\}$	s	$m(s,\{c\}), R \leftarrow \{\}$
$\{d\}$	$\{d\}$	2	
$\{c\}$	$\{d,c\}$	2	
$\{e\}$	$\{d,c,e\}$	2	$m(2,\{e\}), R \leftarrow \{\}$
$\{b\}$	$\{b\}$	2	
$\{e,e\}$	$\{b,e,e\}$	2	$m(2,\{e\}), R \leftarrow \{\}$
$\{a\}$	$\{a\}$	2	
$\{a,b\}$	$\{a,a,b\}$	2	$m(2,\{a,b\}), R \leftarrow \{\}$
–	$\{\}$	f	

$$event\ set = \{c,e,e,a,b\}$$

Figure 27: *SSA* **operation with** $r_{map} : R \leftarrow \{\}$

roughly as specified by the shuffle automaton, or one defective in some way that is yet to be analyzed, is sought. This r_{map} is illustrated in figure 28 using the same event stream as the previous example. While the series of transitions is the same for both examples (figures 27 and 28) the symbol instances that form the event set are different. In fact, the a symbol that is used in the transition from states 2 to f occurs earlier in the stream than all other symbols used in the previous three transitions.

Due to the use of the input register to accumulate transition sets, shuffle automata possess some interesting properties. In using shuffle automata to model behavior patterns, the modelling of concurrent behavior is facilitated by considering R to represent the parallel event stream generated by concurrent event sources (see figure 27 for an example recognition). As will be seen later, the input register contributes to information filtering and robustness of the shuffle automata model in its behavior recognition role.



$$
\begin{aligned}
Q &= \{s, 1, 2, f\} \\
\Sigma &= \{a, b, c, d, e\} \\
T &= \{\{a, b\}, \{e\}, \{c\}, \{c, d, e\}\} \\
m : &\begin{cases} m(s, \{a, b\}) & = 1, \\ m(s, \{c\}) & = 2, \\ m(1, \{c, d, e\}) & = f, \\ m(2, \{e\}) & = 2, \\ m(2, \{a, b\}) & = f \\ otherwise & = \bot \end{cases} \\
q_0 &= s \\
F &= \{f\} \\
r_{map} &: R \leftarrow \{\}
\end{aligned}
$$

Figure 26: A Simple Shuffle Automaton

called the *event set* for an instance of the pattern described by the *SSA*.

A directed graph may be associated with an *SSA* as follows. Each state of the *SSA* has a corresponding node in the graph. If the *SSA* defines a transition from a state q to state q' on a transition set t_i, then the graph contains a directed arc from node q to node q' labeled with t_i. A set of symbols presented in such a way as to cause the *SSA* to traverse the graph from its initial state to a final state results in recognition of one of the patterns described by the *SSA*. The recognized pattern is the one described by the path taken through the graph. A simple shuffle automaton and its graph are shown in figure 26.

The r_{map} function is used in conjunction with the input register to alter its contents following a transition. Changing the definition of r_{map} leads to different interpretations of the sets of symbols which cause a simple shuffle automaton to make transitions and eventually enter a final state. A simple r_{map} is one which clears the input set following a transition,

$$r_{map} : R \leftarrow \{\}.$$

The effect is to impose a temporal ordering on the symbol subsets which will cause the simple shuffle automaton to enter an accepting state. Symbols not consumed in a transition are discarded and not considered for further transitions. The r_{map} that clears the input register at each transition guarantees that a transition from state n to n' which is followed (in time) by a transition from n' to n'' occurs in response to symbol sets that are likewise ordered in time. No symbol that is consumed in the transition from n' to n'' was available for the transition from n to n'. Figure 27 illustrates this for the *SSA* of figure 26.

Another simple r_{map} is one which only removes symbols consumed by a transition $m(q, t_i)$

$$r_{map} : \ R \leftarrow R - t_i.$$

This r_{map} removes the temporal ordering imposed by the previous function. This might be useful when an exact match is not required, but only a pattern that is

where

Q – finite, nonempty set of states,
Σ – finite alphabet of input symbols,
T – transition sets, $\{t_i \mid t_i \subseteq \Sigma\}, \forall i,j : t_i \neq t_j$,
m – transition function $Q \times T \rightarrow Q$,
q_0 – initial state of the *SSA*, $q_0 \in Q$,
F – set of final states, $F \subset Q$,
r_{map} – input register map, $\Sigma^* \rightarrow \Sigma^*$.

The set of states, Q, the input alphabet, Σ, the transition function, m, and the set of final states, F, are all similar to the like-named elements in any of the various finite automata definitions. The *SSA* started in q_0 reads symbols and moves from state to state according to the transition function until it enters a final state $q \in F$. The transition sets, T, are a collection of multisets of input symbols. Each $t_i \in T$ is a multiset whose elements are members of Σ. It is possible that the transition sets overlap, that is, $\forall i,j : t_i \in T, t_j \in T, t_i \cap t_j \neq \{\}$. It is necessary that every transition set element is a member of the alphabet of the simple shuffle automaton, that is, $\bigcup_{i=1}^{|T|} t_i \subseteq \Sigma$. The transition function m is defined over the transition sets, $t_1, t_2, \ldots, t_m$.

The finite state control associated with a *SSA* is in some state, q, from Q and maintains an input register which contains symbols from Σ. The input register, R, is a multiset which holds the input symbols that have been presented to the shuffle automaton by the symbol generating sources, but not yet consumed by a transition. In a single *move,* the *SSA* in state q examines its input register for a set of symbols which will match any one of the transition sets t_i. If $m(q,t_i)$ is defined then the finite state control enters the state given by $m(q,t_i)$ and applies the r_{map} function to the input register. If the state given by $m(q,t_i)$ is one of the final states F, the *SSA* is deemed to have *recognized* one of the patterns it describes. The recognized pattern corresponds to the list of transition function applications

$$m(q_0,t),\ m(q',t'),\ \ldots, m(q'',t'') \in F.$$

The set of input symbols which were responsible for the sequence of transitions is

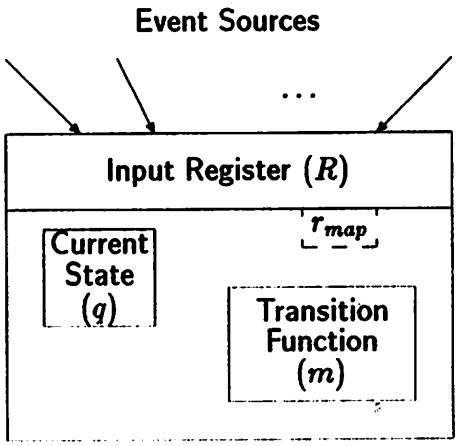


Figure 25: *SSA* control

meaning of behavior recognition.

§1. Simple Shuffle Automata

Simple shuffle automata (*SSA*) are similar in form and operation to familiar models for finite state automata [Hopcroft69]. The important difference is that transitions from state to state are based on the simultaneous availability of all elements of a subset of symbols from the machine alphabet. In order to use sets of symbols as transition symbols, the *SSA* finite state control (figure 25) includes an *input register* to hold symbols which have been generated by the symbol sources but not yet used in a transition by the finite state control. This mechanism is the basis for modelling concurrent behavior. The symbol generation sources are considered to operate concurrently and may fill the input register in parallel.

§1.1 *Simple Shuffle Automata Definition and Component Properties*

A simple shuffle automaton is described by the 7-tuple:

$$SSA = (Q, \Sigma, T, m, q_0, F, r_{map})$$

behaviors which are only partially ordered in time. In addition, shuffle automata are also capable of describing the usual sequential behaviors: sequence, iteration, and selection.

The simplest type of shuffle automata, the *Simple Shuffle Automata* (*SSA*), bears strong resemblence to the familiar *FSA* generally associated with type 3, or regular, languages. The basic characteristics of *EDL* event expressions and their recognition by shuffle automata are established by the *SSA*. However, the *SSA* has some inherent limitations that bound its use as a recognizer for behavior models. The *SSA* can become unwieldy for complex expressions, does not fully support hierarchical behavior descriptions, and does not account for the use of event attributes to perform fine filtering of event information.

The first two of these problems are overcome by the *Basic Shuffle System (BSS)*, a more general shuffle automaton, which consists of collections of simple shuffle automata. The *BSS* extension permits high-level events to be easily incorporated into behavior recognition models and simplifies their description. To employ event attributes for filtering, the *Constrained Shuffle System (CSS)*, extends the input alphabet symbol representation to be a multiple field tuple similar to the event tuples of chapter II. Each alphabet symbol is an event symbol together with a list of attributes. The algorithm that implements the finite state control for a *CSS* includes the use of constraining expressions to effect filtering based on the attributes possessed by an event instance.

This chapter first details the simple shuffle automata model and describes its resemblance to finite state automata. With this grounding, the more general basic shuffle system is introduced. The equivalence of event expressions and shuffle automata is established by providing an algorithm for translating an event expression into a non-deterministic basic shuffle system. Then it is demonstrated how to change any non-deterministic shuffle automaton into a deterministic shuffle automaton. In the final sections, the most general shuffle automata-based recognizer for behavioral patterns, the constrained shuffle system, is described and then used to formalize the

CHAPTER V

SHUFFLE AUTOMATA AND RELATIVES

The previous chapter discussed behavior recognition as a pattern recognition problem. For various reasons this approach to behavior recognition is more involved than simple application of parsing techniques to an input stream of symbols. This chapter will present a formalism, the *Shuffle Automata* model, that is useful for describing the meaning of event based behavior abstraction and model recognition.

Shuffle Automata are a Finite State Automata-like (*FSA*) formalism that comprise a family of machines with a common basic operation. Each higher family member enhances the operational characteristics of the basic model to take in more of the needs posed for behavior recognition.

A shuffle automaton consists of a set of states and a finite state control that effects transitions from an initial state to some final state. Transitions are made from state to state based on the availability of an appropriate sequence of input symbols. The connection of shuffle automata to behavior recognition is made by representing events by the symbols forming the input alphabet for the shuffle automata. User models of behavior described by *EDL* definitions are expressed as shuffle automata in which transition symbols represent events. These shuffle automata are then used as accepting automata in a recognition role to guide extraction of a set of behaviors from the system event stream.

The important difference between the shuffle automata and *FSA* models is that in order to make transitions in the shuffle automata, the finite state control examines *sets* of input symbols, rather than individual symbols from the input alphabet of the machine. This is a simple means for describing concurrency and collections of

A more efficient implementation of the combined occurrence primitive event generation requires the ability, available on many computer systems, to set hardware "watchpoints" on memory locations. Whenever a watched location is accessed, an interrupt is generated in the same manner as a breakpoint instruction. To be used for event generation, the system routine servicing the watchpoint interrupts will implement a combined occurrences test and generate events when they are appropriate.

§4. Chapter Summary

This chapter has expressed the monitoring aspects of *EBBA*-based debugging as a problem in pattern recognition. Modelling system behavior and investigating suspected erroneous behaviors can be achieved by examining information bound into observed patterns of system activity. Pattern recognition to support *EBBA* is complicated due to the noisy, uncertain environment in which debugging distributed systems must take place. A pattern recognizer for event based behavioral abstraction can address this by providing an information rich environment; not a detail intensive environment as in traditional break-examine debugging tools, but one that interprets and explains differences between user models and actual system activity.

§3.3 *Event Generation Through Breakpointing*

Instead of directly adding procedure invocation code to the system, hardware breakpoint instructions are inserted at the appropriate places as needed. Hardware breakpoints interrupt the normal flow of instruction execution and, following a minimal processor context save, begin execution at a specific system routine. The interrupted instruction stream is resumed upon completion of the system routine. Except for the passage of time, the interrupted routine is unaware of the event. For event generation, the system routine called from the breakpoint is the event generation routine.

This method does not require rebuilding of a system to insert event generation probes. It does require the appropriate hardware support and understanding of system structure. Implementation of this technique requires the knowledge of the location of these places (usually obtainable from a symbol map) and appropriate data structures to describe and use the placed breakpoint instructions. In an important departure from the procedure call method, the system pays no overhead when the event generation is not needed. Also, using the breakpoint method, only a relevant subset of available primitive events need be generated, and no testing needs to be performed to determine which events are members of this set.

§3.4 *Combined State Occurrence*

A generalization of the procedure-call and breakpoint event generation techniques allows a collection of state variables to be observed for a specific set of values. When this set of conditions holds, the event it represents is instantiated. The monitored state variables can be dispersed over a large area and thus the technique does not rely on a computation reaching a specific program counter value. The technique does rely on having another routine watch for the combined state to occur. [Zobrist77] contains an algorithm for detection of the combined occurrence of specific values by a set of variables. The algorithm is efficient, but requires considerable computational and storage overhead.

objects in the distributed system. Obviously, from an event generation point of view this is the most desirable kind of primitive event. The events are, in some sense, for free, and an observer only needs to observe and note their presence.

Using this set of messages as a basis for the recognition of behaviors will in many cases be too restrictive. It limits the view of the system to one defined by message orderings defined in the message protocols or to whatever can be gleened from message content (when a message may be examined at all). Investigation at this level can reveal protocol errors or timing problems related to the message exchange. However, if the constituent behaviors for the protocol level need to be investigated (e.g. process creation or resource acquisition), they will not be available.

§3.2 *Procedure Call*

Perhaps the simplest method of adding event generation to a system is to seed the system under investigation with procedure calls to an event generation routine. This strongly resembles the time-honored method of inserting output producing statements at strategic places in the code. The important difference is organizational. Event generation will be more thought out, more consistent, and will have support within the system software base. Each event generation call would represent a point where the execution constitutes primitive behavior. Procedure-call event generation will operate best when the system being debugged is a high-level application. The effect on system resources will be minimal when event generation resource use or frequency is not large compared to system operation use. Conversely, procedure-call event generation is most obtrusive where timing dependencies are closely related to the primitive events or where system resource availablity is tightly constrained. Also, it should be clear that if the source code is not available or the system can not be rebuilt with event generation code in place, event generation via procedure call cannot be used.

is intended to operate in real time, the timeliness of event reporting is important. Behavior investigations that answer user queries result in new queries for further information. For the new information to be used effectively requires that it be current. In order for *EBBA* to be useful, primitive event generation must occur with resource usage commensurate with the granularity of the system being observed.

Given the nature of computer systems it is generally not possible for an observer to sit passively to the side and note when events occur. Practical considerations require primitive event generation to consist of at least a procedure invocation when an event instance occurs. Primitive event generation requires some active agent to:

1. recognize a related set of simultaneous condition holdings or a transition from one holding to another [Holt70] (an event class),
2. gather the attributes together, and
3. report the event.

For current technology, the simultaneous holding exists when some subset of a process or processor state vector has attained a specific set or range of values. This definition does not rule out distributed process state – although observing this is subject to the communication delay and synchronization problems of distributed systems.

The following sections will discuss a number of ways to generate primitive events. The method chosen for a particular system reflects the "appropriate granularity" alluded to previously, the transparency required of the primitive event generation, the hardware architecture available for event generation, and the software structures involved in the computation.

§3.1 *Interprocessor Communication (IPC)*

In the IPC form of primitive event, there is no explicit generation of primitive events. All observable primitive events correspond to message traffic present in the network. These messages may encompass all or only part of protocols used by

§2.5 *Inaccurate Behavior Models*

A model of behavior which is in use to explain an erroneous system activity is itself potentially inaccurate, thus failing to match any actual system behavior. In general though, the most important task of debugging activity is to collect information that explains errors exhibited by a system. The failure of a model to match system behavior is not devoid of information. So, while a perfect match between a behavior model and system activity demonstrates that a user understands some aspect of a system, models that fail to match also contain information – some explanation of the mismatch between model and activity.

The event recognition component must be capable of matching as much of a user model to system activity as possible. This is aided in two ways by a pattern recognizer for event based behavioral abstraction. First, the model analysis component is capable of decomposing a user model into finer grained parts by requesting recognition of subexpressions or other valid partitions of a complete model. This permits the event recognizer to match much system activity independently and synthesize the simpler, but higher level, components into the model undergoing investigation. Any missing parts represent potential differences to be more closely investigated.

Second, the pattern recognizer can relax constraints among events constituting a model as it accumulates a set of events that roughly match the event expression. With a set of constituent possibilities available the relaxed constraints may be evaluated to indicate where and why the potential constituents fail to match the user model.

§3. Primitive Event View

Primitive events form the basis for the *EBBA* debugging techniques. In previous chapters, primitive events have been assumed to appear as needed, in a form that is suitable for use. Primitive event generation is an active process which consumes time in a system that has genuine time and other resource constraints. Since *EBBA*

§2.4 *Constraints on Events*

The constraining clauses expressed in the **cond** clause part of an event definition serve as filters for the events that are to become the constituents of a high-level event instance. Evaluation of constraining clauses determines the suitability of a particular event class instance to a model that includes that class. When an event instance is to be considered as a possible constituent of a behavior model, its attributes must satisfy any constraining clause relations in which they appear.

Constraints are either simple or multidimensional. Simple constraints are those which rely only on the relation of an attribute of a constituent event to a constant value. The acceptability of an event as a model constituent can be determined immediately if all constraints applied to an event's attributes are simple constraints. A constraining clause involving two or more attribute operands, each qualified by the same behavior model member event, is also considered simple since the suitablity of the instance is readily determined.

Multidimensional constraints are those that involve relations among attributes of a number of constituent events. While the constraints themselves are no more difficult to evaluate than simple constraints, the inter-event dependence requires that all constraint related events have an event class instance bound to the model before any filtering can take place. The difficulty with multidimensional constraints is now apparent. In general, an event which fails to fit as a model constituent due to a failure of constraint evaluation is not necessarily unsuitable for the behavior model. Rather, it may be the case that the unsuitable event is simply not compatible with those events previously bound to the model. The unsuitable event might be compatible with another set of event bindings. Multidimensional constraints better indicate what an acceptable set of events is, rather than what events are to be excluded. In the prototype toolset all events which are not removed by simple constraints are retained as possible model constituents until a set that results in model recognition is accumulated.

§2.3 *Concurrency and Time*

Much behavior in a distributed system is time dependent. When multiple processing sites exist, there is the potential for true concurrent processing and load sharing. For processors to cooperate effectively and use these advantages, they must exchange results with each other. Programs which are otherwise correct can exhibit errors because of synchronization problems due to network delays over which they have no control or due to timing relations which are too tightly constrained. The behavior modelling tool, *EDL*, provides ways to use time and time constraints in creating behavior models.

The inability to obtain accurate time information in a distributed system greatly complicates fitting event stream instances to behavior models. In a traditional single processor sequential system time is a known quantity. It is accurate and reliable. In distributed systems, there is no time authority to appeal to in order to accurately determine time relations among events occurring on different nodes. Events occurring at a single node still have accurate time relations (if they are simple sequential systems). However, the best that may be said for events observed at distributed nodes is that they are out of synchrony by no more than some bounded amount [Lamport78].

Time relations are implicitly stated in event expressions defined by catenation and repetition operators. Other time relations are explicitly stated in constraining clause relational expressions. The time attribute which is attached to each event instance can be used only as an estimate of the true event time. The time at which an event arrives at an observer is not to be relied on to totally order event occurrences. A method such as Lamport's time synchronization algorithm can be used to understand the time variances, thus minimizing difficulties due to time indeterminancy.

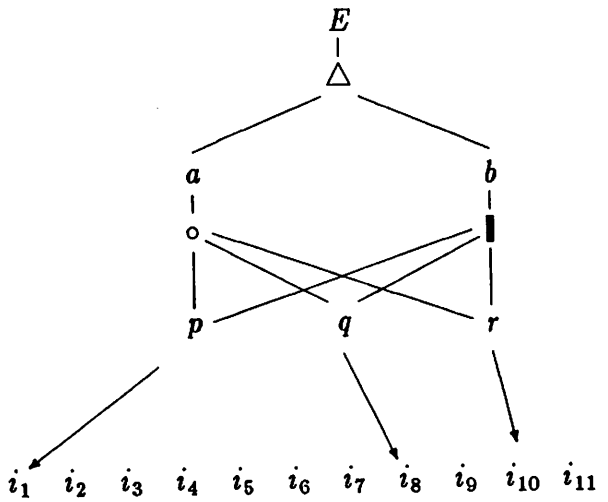


Figure 24: Unacceptable event sharing

of some prioritizing scheme or some evaluation of relative event instance suitabilty for a model.

An additional consideration takes into account exchange of high-level events by cooperating nodes. If sharing is not allowed, local recognizers must ensure that high-level events recognized at remote nodes are not sharing. Furthermore, assurance is needed that low-level constituents of remotely recognized high-level events are not mistakenly incorporated into local high-level events.

The prototype toolset permits users to control sharing by allowing recognition requests to specify that no constituents of the requested model may be shared. The effect is an all or none type of sharing that is applied to each abstraction level of an entire behavior model. Control of remote recognition-caused sharing problems is dealt with by controlling the communication patterns of cooperating nodes. The nodes of a distributed system may be partitioned into subsets. Each subset has a single abstraction node that communicates only with other high-level nodes. The result is that high-level nodes never see low-level information.

as in figure 23. The case where the instance bound to one of event b's member events

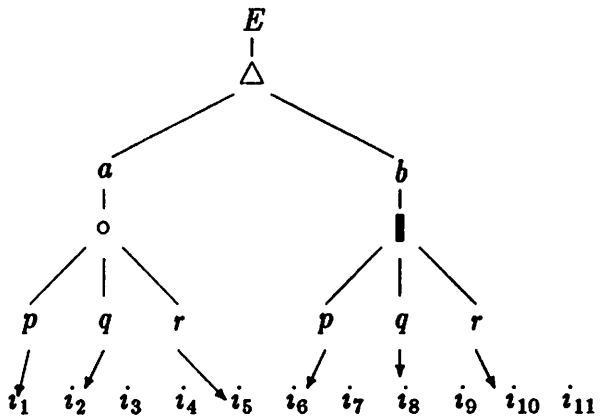


Figure 23: Acceptable event bindings

(thus causing its instantiation) is the same as one in event a would not be acceptable.

For example, using thee same representative event stream, $i_1, \ldots, i_{11}$, assume p is bound to i_1, q to i_8, and r to i_{10}. This sequence will instantiate event a. If event b picks one of p, q, or r as its model constituent, an unacceptable sharing results. A suitable event b requires a p, q, or r which has an alternative binding: i_6, i_2, or i_5 respectively. Figure 24 illustrates the possibilities for sharing.

On the other hand, for a user requesting recognition of both an a and b event this situation might be acceptable. An enumeration of cases where sharing is permitted will not help. The most general solution will require the tools to provide ways to describe what a user wants and to be informative when a user has not been specific.

The event recognizer has yet another perspecitve on this problem – how to control whatever level of sharing is permitted. If event sharing is allowed, some minor problems occur in the recognizer regarding the actual mechanics of sharing the event instance records. When sharing is not to be allowed, an important problem is again the assignment problem. The decision regarding which worthy high-level event is to be given an event instance as a constituent will need to be made in light

of event streams discussed in the previous section is the possibility that distinct high-level events may have non-distinct cluster member events. Thus the possiblity exists that the constituent event sets bound to two distinct high-level events might have common members. If the intersection of the constituent event sets is not empty the common members are said to be *shared* between S_1 and S_2. Figure 22 illustrates this. In this example, the behavior models represented by events E_1 and E_2 have a

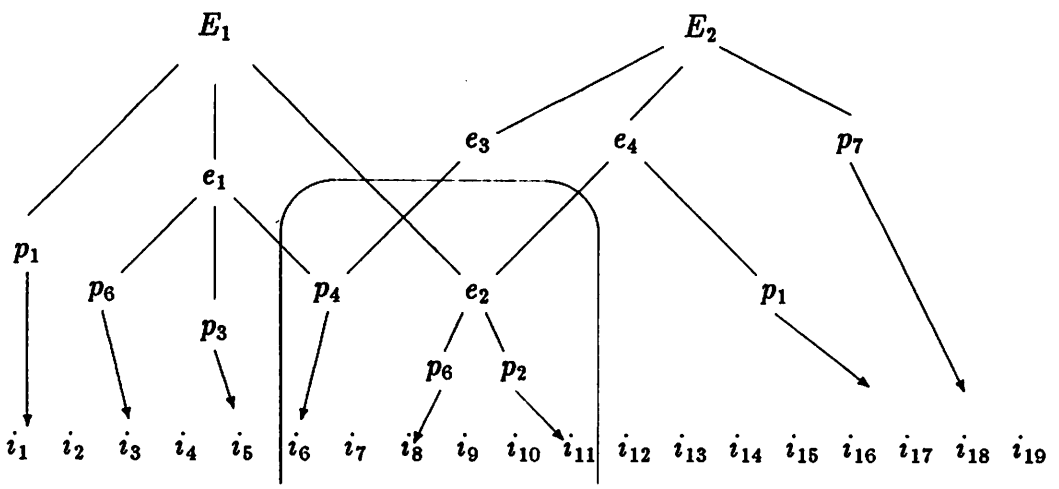


Figure 22: Distinct events with common constituents

number of primitive event classes (p_1, p_2, p_4, and p_6) and a high-level event class (e_2) as common model members. If event instances are bound to these common named members such as i_6 to p_4, i_8 and i_{11} to e_2, then the recognized models, E_1 and E_2, share some structure.

The question from the *EBBA* perspective is: When and to what degree is this sharing to be permitted? This question has no clear answer. Sometimes sharing is not permissible, for example, if the behavior model specifies

$$E = a \triangle b$$

with

$$a = p \circ q \circ r \text{ and } b = p \mid q \mid r.$$

the user probably intends there to be distinct a and b events as E model constituents

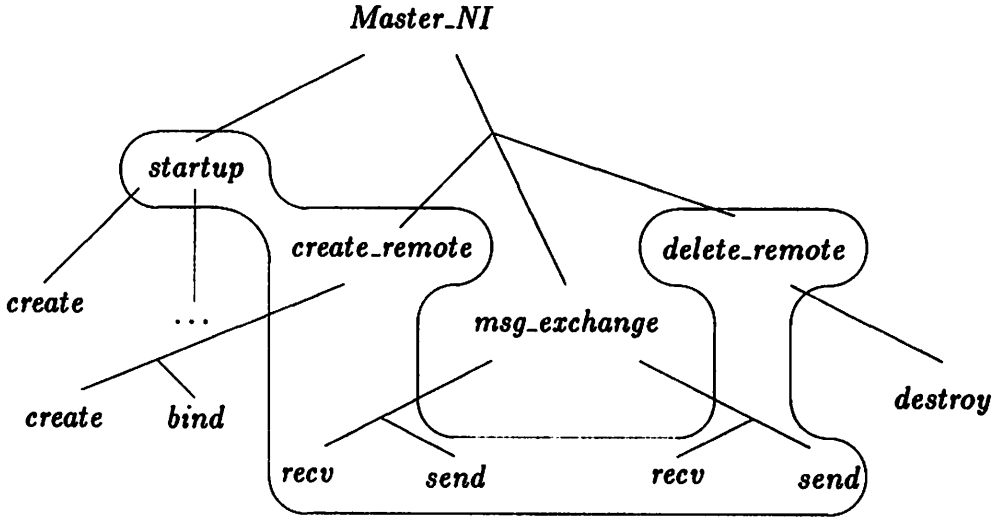


Figure 21: High-level event with mixed level constituents

of the model to be recognized. The ability to use this mixed event set for recognition is greatly facilitated by a recognizer which easily deals with decomposition of high-level events.

In a distributed system, the recognition task could be centralized at a single processing node or distributed among the nodes of the system. The mixed event stream could result from the cycling of high-level events back into the stream or from nodes reporting *only* locally recognized high-level events to remote cooperating nodes. In this case, a binding as illustrated in figure 21 might be the only one possible. For a distributed system, exchange of high-level events can lead to significant economies in use of the communication subnet by the debugging tools. In a distributed behavior recognizer, much of the event stream that is local to a node is filtered with a large part of what remains abstracted into high-level models. Only events representing the abstracted models are exchanged by communicating partners. Filtering and abstraction thus reduce the number of events to be exchanged by cooperating nodes.

An important complication to the pattern interleaving and noise characteristics

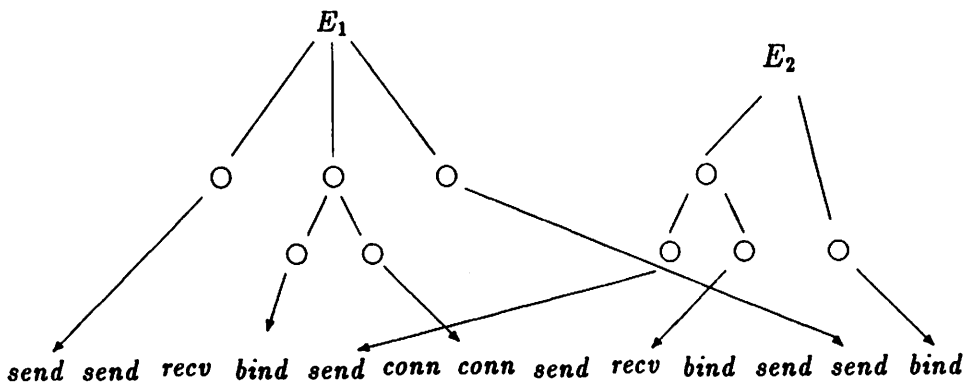


Figure 20: Interleaved patterns, different models

As is evident from figure 19, figure 20, and some examples of the previous chapter, not all events in a given subsequence of the event stream are constituents of each (or even any) interesting higher level behavior. Coarse filtering removes events not relevant to a model under investigation. However, many events not removed by coarse filtering are extra and should be treated as noise, to be ignored by behavior models that are uninterested in them.

Another form of the noise problem arises when event instances that match members of a high-level event model are not necessarily appropriate constituents of a *specific* high-level event. This effect occurs as a result of the constraints that may be placed on model constituents – expressed in *EDL* as **cond** clause relations. The recognizer for the model must examine each event instance in which it has interest, to determine if it fits with the other instances already bound to the model.

§2.2 *High/Low -Level Model Constituents and Sharing*

Infusing the event stream with high-level events raises the possiblity that recognition of a high-level behavior model may be possible by binding a mixed set of high-level and primitive events to the model members. Figure 21 shows a high-level model with a possible set of event bindings (in outline) which could cause an instance

problems do not neatly fit into a single category. Rather, when different aspects of the problems are emphasized, they span the previously discussed categories.

§2.1 *Interleaved Event Streams and Noise*

Behavior recognition for debugging purposes operates in a dynamic environment that is represented by an event stream containing multiple patterns. These patterns often exist without distinct boundaries. Additionally, at any time, debugging tool users will likely be monitoring the system for the occurrence of more than one pattern. An implication of these factors is that many patterns which are described by the same *EDL* pattern description may be interleaved in the event stream (figure 19). Likewise, different pattern descriptions will have instances of patterns that they

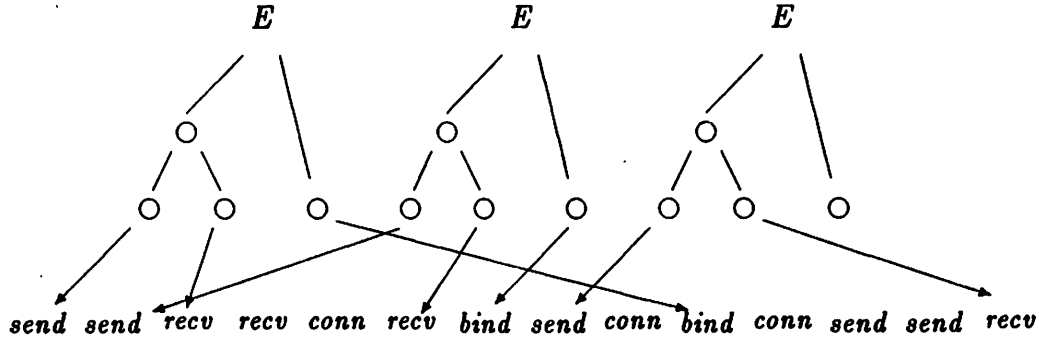


Figure 19: Many patterns for the same model

match, similarly interleaved (figure 20).

The appropriate solution to this interleaving is for the behavior monitor to operate as a collection of parallel event recognizers. Each recognizer will watch the event stream and extract from the stream a set of events to bind as constituents of the behavior model represented by the event recognizer. A recognizer that operates in this fashion can be easily adapted to recognizing complex patterns. The request for recognition of a high-level event is followed by a decomposition of that request into simpler sub-tasks. As each sub-task completes, the recognized event is pushed into the event stream for higher level recognition requests to use.

set of shuffle automata so the NDSA to DSA algorithm must be executed for each $S_0, S_1, \ldots, S_n$, in the *BSS*.

The NDSA to DSA algorithm is aided by a graph tracing function used to eliminate ϵ-labeled transitions and create new states for the DSA. The *ϵ-closure()* function accepts as input a set of states of the NDSA and determines the reflexive transitive closure of the member states which are connected by ϵ-labeled transitions in the NDSA. For each machine in the *BSS* the algorithm is as follows:

1. The alphabet, set of shuffle automata and all non-ϵ NDSA transition sets are the same in the DSA.

   $\Sigma_{DSA} = \Sigma_{NDSA}$
   $S_{DSA} = S_{NDSA}$
   $T_{DSA} \leftarrow T_{NDSA} - \epsilon$,

2. Set up the initial state of the deterministic shuffle automaton from the initial state of the non-deterministic shuffle automaton and the ϵ-closure (figure 41) of that NDSA state. Place this initial state into the set of "unmarked states" (U) of the DSA,

   $q_{0DSA} \leftarrow \epsilon\text{-}closure(\{q_{0NDSA}\})$,
   $U \leftarrow \{q_{0DSA}\}$,

3. If the set of "unmarked states" is empty, the DSA is complete, so quit,

   if $U = \{\}$ **then** return

4. Select and remove a DSA state $d = \{q_1, q_2, \ldots, q_n\}$ (where each of the elements $q_1, q_2, \ldots, q_n \in Q_{NDSA}$) from U. Create a temporary list containing copies of all the transition sets to be used to enumerate all of the transition sets,

   $d \leftarrow selectone(U)$,
   $T \leftarrow T_{DSA}$

5. Select a transition set, t, from the temporary list T, then create a new DSA state set N from all of the states $q_i \in d$ for which there is a transition in the NDSA on transition set t, (eliminate non-deterministic transitions),

$t \leftarrow selectone(T)$
$N \leftarrow \{\}$
forall $q_i \in d : m(q_i, t) \neq \perp$
$\quad N \leftarrow N \cup \{m(q, t)\}$
if $N = \{\}$ **then goto** step 9.

6. Form the ϵ-*closure* of this new state (collapse chains of ϵ-labeled transitions),

   $N \leftarrow \epsilon\text{-}closure(N)$

7. If this created state is unique, add it to the state set of the DSA, and to the "unmarked states" set,

   if $N \notin Q_{DSA}$
   then
   $\quad U \leftarrow U \cup N,$
   $\quad Q_{DSA} \leftarrow Q_{DSA} \cup N$

8. If there is not yet a transition from d to N on the transition set t in the transition function m_{DSA}, add one,

   if $m(d, t) = \perp$ **then** $m(d, t) = N$

9. If all transition sets of T have not been examined as possible transitions from all of the states in d go back to step 5 to do more transition sets, otherwise, go to step 3 to select another unmarked set from Q_{DSA},

   if $T \neq \{\}$ **then goto** 5 **else goto** 3.

§5. Constrained Shuffle System

The extension of the simple shuffle automata to the basic shuffle system model was important because of the need to easily express hierarchical behavior models and complex subexpressions. In the larger context of *EBBA*, the *BSS* model is also valuable because it is tolerant of event streams which consist of mixed high-level and primitive events.

```
function ε-closure( T: state-set ) returns state-set;
- - Algorithm for collapsing chain of connected ε transitions
- - into a single state. (After [Aho77])
begin
      forall i : s_i ∈ T do push( STACK, s_i);
      ε-closure() := T;
       while not empty( STACK ) do
            s := pop(STACK);
            forall t : (m(s,ε) = t) ∈ M do
                  if t ∉ ε-closure() then
                        ε-closure() ← ε-closure() ∪ {t};
                        push(STACK, t)
                        endif
                  endfor
            endwhile
end ε-closure()
```

Figure 41: Empty transition closure algorithm

The pattern recognition model of the *BSS* assumed that events are like symbols in a regular language, featureless and content free. All symbols with the same name are therefore indistinguishable (except for their position in a partially ordered set that defines the event stream). *EBBA* assigns more meaning to events by insisting that events in a class are distinguishable by the attributes they possess. The *Constrained Shuffle System* (*CSS*), introduced here, provides a capability to narrow the focus of a behavioral model by including or excluding events based on the relationships of their attributes.

The first step to providing this capability is to extend the notion of the input symbols to include attributes. Similar to events (of chapter II), a symbol in the alphabet of a CSS is a tuple:

$$(e, a_1, a_2, \ldots, a_n)$$

where e corresponds to the *class_name* field of an event and $a_1, a_2, \ldots, a_n$ correspond to the event's attributes. The event attributes are simple values (the natural numbers) associated with a specific instance of the symbol. All symbols with the

same class name field possess the same number of attributes. Different instances of a symbol may have the same values bound to their corresponding attribute slots. However, events in a distributed system are all made unique by their combined time and place attributes.

A *CSS* is similar to a *BSS* but contains a set of constraining functions which are defined in terms of the attributes of the input symbols. The finite state control of a *CSS* performs transitions in a similar manner as a *BSS*, but has the additional task of using the constraining expressions to determine if an input symbol should be included in an event set. A *CSS* then is a 5-tuple

$$CSS = (\Sigma, S, T, C, A)$$

where:

Σ – input alphabet, $\{s_i = (e_i, a_1, a_2, \ldots, a_n) \mid \forall j,\ a_j \in \aleph\}$
S – set of shuffle automata, $\{S_0, S_1, \ldots, S_m\}$,
T – transition sets, $\{t_i \mid t_i \subseteq \Sigma \cup S,\ \forall i, j : t_i \neq t_j\}$
C – set of constraining functions, $\{c_i \rightarrow \{0, 1\}\}$,
A – "Active" shuffle automata $A \subseteq S^*$.

The set of shuffle automata, S, the transition sets, T, and the active set, A are the same as the corresponding *BSS* elements. The input alphabet Σ is a set of $n+1$-tuples where e_i represents the class name field of the tuple and $a_1, \ldots, a_n$ are the attributes associated with the event.

The constraining functions, C, are defined over the same set of values as the attributes of the event tuples. Without loss of generality, each transition set t_i may have its own constraining function, c_i, associated with it. Each constraining function $c_1, c_2, \ldots, c_n$ in C for the *CSS* is defined in terms of the attributes associated with the event set for the *CSS*. Recall that the event set is a multiset of event instances bound to the event expression member events. In the *CSS*, the symbols $s_1, s_2, \ldots, s_m$ that are bound into the event set, each are formed from a tuple $(e, a_1, a_2, \ldots, a_n)$. For the event set there is a corresponding $m \times n$ array that represents the arguments to the constraining function c_i. The j^{th} row of the array contains the attributes for the j^{th} symbol from event set $E = \{s_1, s_2, \ldots, s_m\}$. The maximum number of columns is

determined by the event symbol with the largest number of attributes. It is useful to visualize the event set E the following way

$$E = \left\{ \begin{array}{l} s_1 = (e_1, a_1, \ldots, a_{n_1}) \\ s_2 = (e_2, a_1, \ldots, a_{n_2}) \\ \ldots \\ s_m = (e_m, a_1, \ldots, a_{n_m}) \end{array} \right\}$$

The constraining function for a transition set t_i is defined

$$c_i \left(\begin{array}{cccc} a_{1,1} & a_{1,2} & \ldots & a_{1,n} \\ \vdots & & a_{i,j} & \vdots \\ a_{m,1} & a_{m,2} & \ldots & a_{m,n} \end{array} \right) \rightarrow (0,1)$$

where each $a_{i,j}$ is the j^{th} attribute of event symbol s_i. For those event set entries with fewer than n attributes, fill their positions in the argument array with 0's. The constraining function c_i is the conjunction of all constraints involving attributes of event symbols defined for t_i.

The constraining functioin c_1 for a transtion set t_i is formed from a conjunction of all the constraint expressions associated with the symbols that are members of t_i. For example, if a set of primitive events have the following templates

(create, node, time, pid, creator)
(preempt, node, time, pid)
(dispatch, node, time, pid)
(destroy, node, time, pid, length, function, device)
(IO, node, time, pid, destroyer)

an event defined

$$fmove = create \circ \ (preempt \triangle dispatch \triangle IO)^{+} \circ destroy$$

with constraints

$$create.node == destroy.node,$$
$$IO.pid == create.pid$$

would define the transition sets

$$t_1 = \{create\},$$
$$t_2 = \{preempt,\ dispatch,\ IO\},$$
$$t_3 = \{destroy\}$$

with a corresponding set of constraining functions

$$c_i \begin{pmatrix} node_1 & time_1 & pid_1 & creator_1 & 0 & 0 \\ node_2 & time_2 & pid_2 & 0 & 0 & 0 \\ node_3 & time_3 & pid_3 & 0 & 0 & 0 \\ node_4 & time_4 & pid_4 & length_4 & function_4 & device_4 \\ node_5 & time_5 & pid_5 & destroyer_5 & 0 & 0 \end{pmatrix} \rightarrow \{0,1\}$$

$$\begin{aligned} c_1 &\leftarrow node_1 = node_5 \wedge pid_4 = pid_1, \\ c_2 &\leftarrow pid_4 = pid_1, \\ c_3 &\leftarrow node_1 = node_5. \end{aligned}$$

The constraining function associated with a transition set is useful because it can effectively eliminate event instances from consideration as event set constituents. This constitutes fine filtering for a behavior model.

Each shuffle automaton S_i of the *CSS* is similar to those of a *BSS* with the exception that the transition function m involves the constraining functions as well as the transition sets. Each $S_i \in S$ for a *CSS* is a 5-tuple

$$S_i = (Q_{S_i}, m_{S_i}, q_{0S_i}, F_{S_i}, r_{mapS_i})$$

where

Q_{S_i} – set of states
m_{S_i} – transition function, $Q_{S_i} \times (T, C) \rightarrow Q_{S_i}$
q_{0S_i} – starting state
F_{S_i} – set of final states
r_{mapS_i} – input register map

Operation of the *CSS* is identical to a *BSS* with added rules to evaluate constraints associated with a transition set. Refer to the operation defined for the *BSS* for a complete description but substitute the following step 3,

3. When the input register of a shuffle automaton in the active set contains one of the outgoing transition sets for the current state of the shuffle automaton, and the transition function for the transition set evaluates to 1, the finite state

control for the shuffle automata performs a transition,

if $\exists i,j : m(q_{S_k^i}, t_j) \neq \perp$ **and** $t_j \subseteq R_{S_k^i}$ **and** $c_j = 1$
then

$q_{S_k^i} \leftarrow m(q_{S_k^i}, t_j)$,
$R_{S_k^i} \leftarrow r_{map}(R_{S_k^i})$
forall $j,l : m(q_{S_k^i}, t_j) \neq \perp$ **and** $S_l \in t_j$

$A \leftarrow A \cup \{S_l^{new(i,q)}\}$,
$q_{S_l^{new(i,q)}} \leftarrow q_{0S_i}$,
$R_{S_l^{new(i,q)}} \leftarrow \{\}$

else goto step 4

§6. Chapter Summary

This chapter has developed the shuffle automata formalism which is useful for performing pattern recognition for the purposes of behavior modelling. The shuffle automata model describes a guideline for recognizing behaviors in complex systems rather than a completely rigorous formal system. The input alphabet of a shuffle automaton is representative of the events described for a view on a system.

The most basic shuffle automata, the simple shuffle automata, defines the basic operational characteristics of the model. *SSA* use an input register holding recently occurring events to absorb potential time variations possible in distributed systems and to model concurrency.

Simple shuffle automata are not capable of satisfying all the needs of a behavior recognizer for event based behavioral abstraction, so more capable members of the shuffle automata family are defined. A collection of simple shuffle automata formed as a basic shuffle system will represent any event expression which can be described by *EDL* event descriptions. Using the *BSS* model, the definition of an actual behavior model is given and event sharing among behaviors can be described. Finally, to tie in the fine filtering constraining expressions that are expressed as **cond** clause relations in an *EDL* description, a basic shuffle system augmented with the ability

to evaluate constraints based on the attributes of events is defined as a constrained shuffle system.

CHAPTER VI

A TOOL FOR EBBA DEBUGGING

EBBA provides a framework for debugging programs which focuses attention on evaluating models of system behavior rather than examining detailed artifacts of its computation. The *EDL* described in chapter III provides a simple mechanism for describing succinct behavioral models using this event view of a system. Syntactic pattern recognition together with the shuffle automata formalism provide a paradigm for abstraction of system behavior as an aid to isolating and identifying errorful behavior. This chapter describes a prototype toolset that draws these notions together into an integrated whole to demonstrate the feasibility and explore some constraints of a high-level debugging approach.

It is important to note that the toolset is not designed to enable its users to create complete, formal models of system behavior. For complex systems this would prove to be an arduous and potentially error prone process in itself. Instead, the intent of the toolset is to promote creation of small models that focus on suspected areas of errorful system activity. Nor is the toolset capable of automatic error identification — it will not isolate the error-causing section without help from someone to direct the search. Even then, the toolset cannot distinguish errorful from correct behavior. In this respect, the *EBBA* high-level debugging paradigm says nothing about errors, only patterns of behaviors and how they differ from each other.

§1. Basic Debugging Toolset

Figure 42 illustrates the main functional components of a prototype *EBBA* toolset. The toolset illustrated in figure 42 describes a logical partitioning of functionality. This logical partitioning is indicative of a true operational partitioning of toolset components that operate asynchronously with respect to each other and the system being observed. Since component operation is largely independent, they have been designed this way rather than as a single scheduled process with an appropriate mechanism for switching attention among the components.

The asynchrony and independence of toolset components is evident in the operation of all the components of figure 42. Event stream generation should be considered non-periodic and random so event reception and behavior abstraction by the event recognizer are independent of the tool user's consideration of the system behavior. Operation of the behavior monitor, event queuing, and event recognizer components is continuous and cooperative. The compiler executes as a response to user model creation efforts while the librarian is more of a service provider for all components.

Tool users describe their models of system behavior using *EDL* descriptions. The set of primitive and high-level events provided by a user form a view of the system which is kept in a *library*. The contents of a library are the translated events in the view along with some clerical information. Each event library contains the events defined for only one view of a system. A *Librarian* has the task of maintaining and distributing the contents of a library. Each event description stored in a library contains a shuffle automaton definition that will recognize an instance of the described behavior and clerical information that describes the event's relation to others in the view. The consistency of a library is maintained through cooperation between the *EDL* compiler and the librarian that maintains a library.

Files of event descriptions are checked for syntactic correctness and translated by an *EDL Compiler* into the form suitable for the remaining system components to use effectively. In order to monitor the activity of a system, a description of the primitive events defined for the system must be prepared as *EDL* primitive event

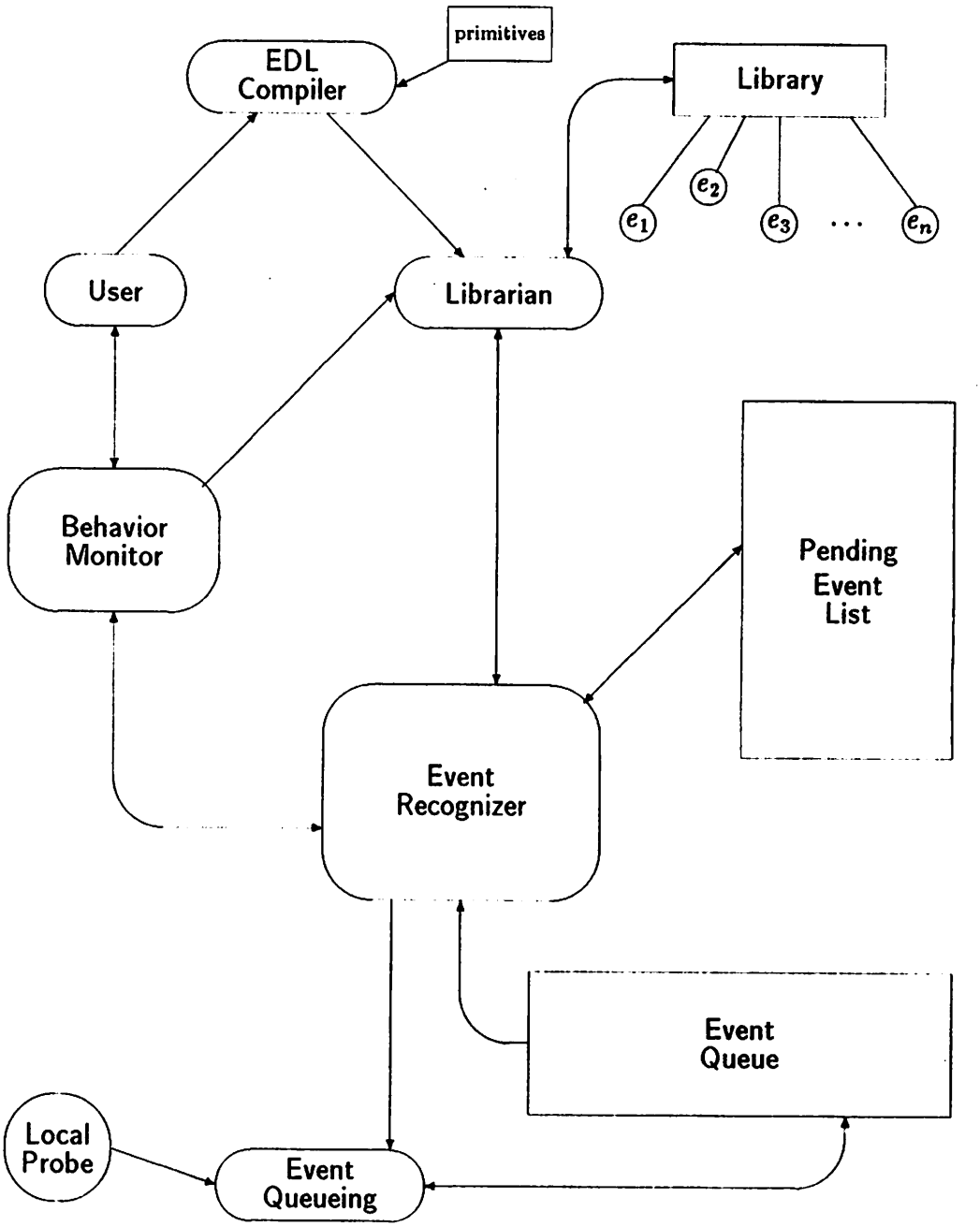


Figure 42: Basic debugging toolset

descriptions. These events are compiled by the *EDL* compiler and a new event library is created for the viewpoint represented by these primitive events. Each time the *EDL* compiler is invoked, the user supplies the name for a library that forms the translation context for the event descriptions to be compiled.

The event stream that forms the basis for the *EBBA* debugging framework is created by probes embedded in the system to be observed. The probes can be constructed using one (or a combination) of the techniques outlined in chapter IV for generating primitive events. Here and in chapter VII the event generators are idealized as *local probe* components. Their task is to take the record representing the instance of an event, create the instance tuple in the proper form, and send the instance to an agent of the toolset to be placed into the event stream. The event stream is presented to the event recognizer on the *event queue* structure. This structure, in which each event queue entry is an internal encoding of a system generated event tuple, is maintained by the *Event Queuing* component. Event queuing is the behavior interface between the external world of the system and the debugging monitor.

The *Event Recognizer* component implements the pattern recognition part of the *EBBA* pattern recognizer. The task of the event recognizer is to match user defined behavior models to the event stream generated by the system. A user, after having created a behavior model and compiled it with the *EDL* compiler, may request that the event recognizer monitor the event stream for an instance of the event describing the behavior model. The event recognizer requests the definition of the event, as well as any high-level events that are members of its event cluster, from the librarian and creates an internal structure to hold the state of the shuffle automaton in the definition. The *Pending Event List* describes recognition requests and the current state of the progress made by the event recognizer towards their recognition. The pending event list is made of two views of the set of pending events. One reflects the structure of a basic shuffle system (*BSS* Chapter V) for a high-level behavior model. The other is a scheduling queue that defines the order of

interpretation for the pending event list entries as they compete for event instances. As event instances are queued to the event queue, their applicability to pending events is determined and the instances are possibly added to the input registers of one or more shuffle automata. When a shuffle automaton enters a final state a set of actions is carried out that may include placing an instance tuple into the event stream, simply updating the input register of a calling shuffle automaton, or invoking some intervention activity.

Structures maintained on the pending event list provide information for pattern analysis by the *Behavior Monitor*. The behavior monitor serves as the user interface to the event monitor. Users make event recognition requests and debugging action requests through the behavior monitor. Another important function of the behavior monitor is to compare abstractions which the user has requested to the progress made recognizing them. In this capacity, the behavior monitor serves as the model analysis component of the *EBBA* pattern recognition system. Thus, the behavior monitor aids the user in directing the search for erroneous behaviors. The behavior monitor component in the current implementation is quite simple as it only passes user requests to the recognizer and is capable of simply examining recognizer structures in response to user requests. The behavior monitor component is envisioned as a very capable advisor to the user – although this capability is not currently provided. The following sections detail the function and major structures of each component in the toolset.

§1.1 *EDL Compiler*

The *EDL* compiler translates user defined behavior models into forms appropriate for the remaining components of the toolset. The compiler accepts both primitive and high-level event definitions for translation. Each *EDL*-described event is translated in the context of a specific event library. Input definitions are checked for syntactic correctness and consistent use of names. Local names, i.e. event heading parameters and attributes defined by the event, must all be unique within an

individual definition. Names of event expression members and the attributes they define are checked by obtaining structural information about each member event from the library associated with the compilation. After an event definition completes its syntax and consistency checks successfully, it is encoded into the shuffle automata library form and stored into its event library.

Translation of source *EDL* definitions into the library form proceeds in three phases. The first phase is syntax driven and results in translation of the definition into an internal abstract syntax tree. The abstract syntax tree that represents an event becomes the input for all subsequent compiler passes and will eventually hold all of the information necessary for the final outputs.

The second phase performs semantic analysis and preparation for output code generation. This phase has four parts, one for each major section of an event definition. The first part establishes a connection to the library to announce the new definition, verifies that event heading parameters and attribute names are unique, and assigns slots in the template event tuple for the attributes defined by the event. The net effect of this part is to establish the general shape of the event being defined.

Next, the event expression is analyzed. All event expression members have their definitions brought into the compiler's definition cache so that event heading argument lists can be compared and proper use of attribute names in **cond** and **with** clauses verified. This appears to force a user to describe events in a bottom-up fashion. This constraint exists because of an early implementation decision. The reasons are related in large part to the lack of a coherent user interface that would tie all of the tools together neatly. In a reimplementation, it will be possible to delay the binding of an event definition to an event name until the time that the unbound event is actually required for recognition of a cluster in which it is a member.

An important output of event expression analysis is an ordered list of the event expression member events. This list will be later used by the table generator to assign unique local symbols to these event symbols. The symbol mapping is useful for a model recognizer to simplify transition set acquisition and examination. The

event list is also used by the event recognizer to form the access context for expression attributes in **cond** and **with** clause expressions.

The final parts of this second phase examine any **cond** and **with** clause expressions to determine bindings for expression operands. Expression operands have three sources: event parameters, attributes of event expression member events, and constant values. Operands are not typed so no operand type compatiblity checking is needed and no type coercion operators are implicitly inserted into the expression trees. The result is that expression analysis is quite simple for the compiler to perform since it involves only operator selection and operand location binding.

Parameters and attributes are bound to event requests and event instances while a system is running and being monitored. These operands have known locations in the event recognizer structures representing event instances and event recognition requests. Locations for parameters are determined when the heading is analyzed. Since parameters are visible only to the event in which they are defined, parameter information does not need to be exported from a definition. For this reason, parameter name and location information is maintained only during the analysis of a single event description. Actual parameter values are determined when a request is made for an event recognition.

An event attribute location is determined by its position in the event tuple of the event that defines the attribute. After an event definition is analyzed, the shape of the tuple is known and descriptors for each attribute are bound into the structures describing the event. Values are bound to attributes of an instance when the event is generated by an event source. The recognizer for an event accesses the attributes of potential constituent events indirectly through a pointer list ordered by the list of member events compiled during event expression analysis. The actual event recognizer structures involved in supplying the operands will be described in a later section.

The final compiler phase outputs the event definition in the event library format. This form is the one which is passed to components requesting event definitions.

Each event in the library, whether primitive or high-level, contains a heading part that describes the general shape of the event. The heading contains all of the externally visible names defined and used by this event including the event's name, a list of the event expression members, and the list of the attribute names defined by this event. High-level events have an additional table part describing a recognizer for an instance of the event. In the prototype toolset, the recognizer is expressed as a coded shuffle automaton. The filtering constraints of the **cond** clause and the attribute binding expressions of the **with** clause are output as a set of code strings in S-expression form. The form of an *EDL* definition will be detailed in the next section.

§1.2 *Librarian*

The task of the librarian is to maintain libraries of event definitions. Each library represents a view of a system that is based on a set of primitive events. All event descriptions, primitive and high-level, are translated by the *EDL* compiler and coded into a form that is conveniently stored by the librarian. An event library is conceptually similar to a database with two types of objects:

- a directory that contains library status and indicates currently valid and usable event descriptions, and
- the text that fully describes an individual event definition.

Maintaining a set of event libraries that represent multiple system viewpoints adds another level of organization to the library structure in order to name individual libraries, but does not add any new objects.

The prototype toolset implements this organization within a hierarchical file system provided by the host operating system. Figure 43 illustrates this organization in the prototype toolset. The set of event libraries is collected into a directory that contains a list of subdirectory entries, one for each event library known to

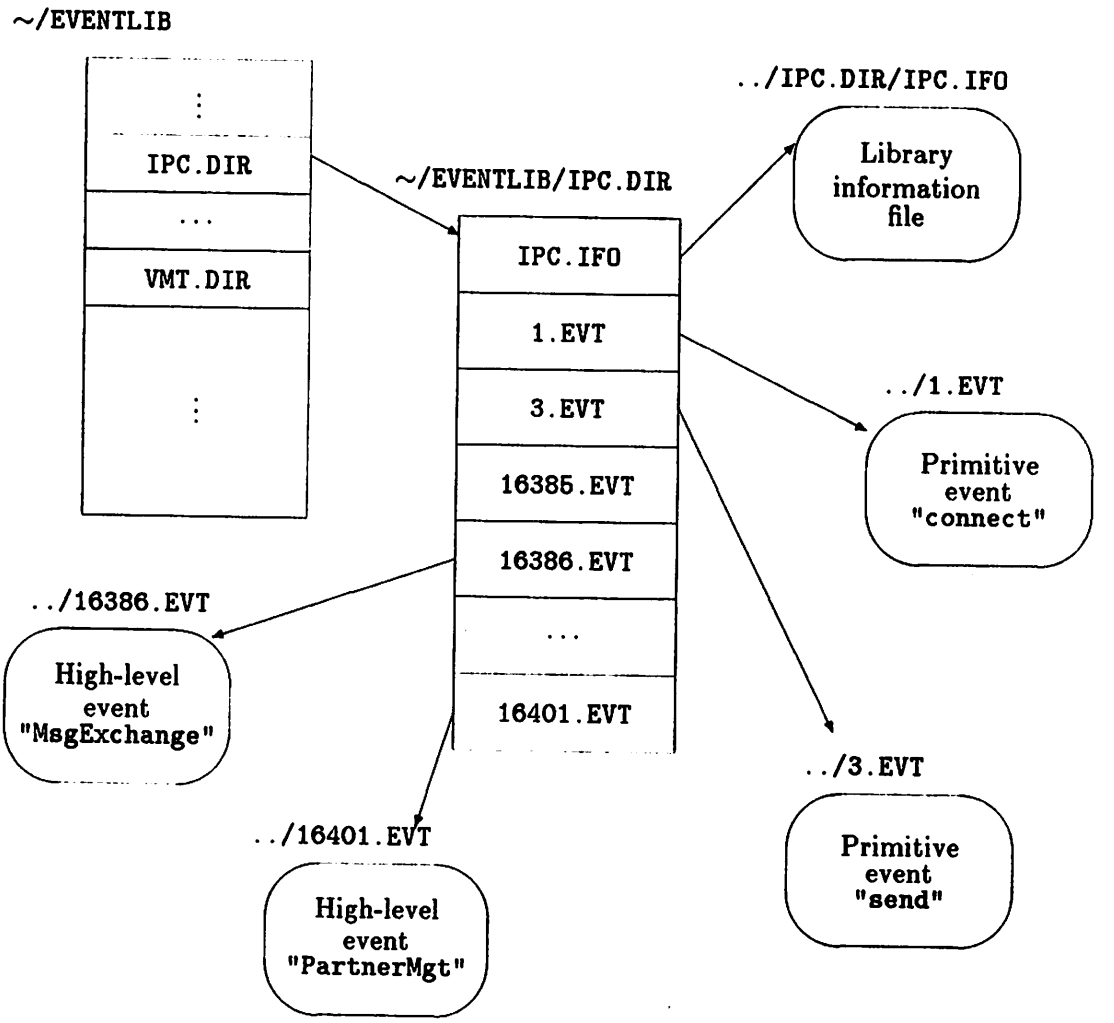


Figure 43: Event library external organization

the debugging monitor[1]. Within an individual library subdirectory is a single file entry that contains the library status and lists currently valid library events. Each event description is entirely contained in its own, separate file in the subdirectory that holds the library. The remainder of this section will describe the information carrying components: the library information file and the event files.

Library Information File

For each event library there is a distinguished file, the *Library Information File* (figure 44), which consists of two parts: one contains the event library status; the other, a list of index entries, one for each active event in the library. Each event in an event library is known to the toolset by three names: the user supplied name; the external name found in an event tuple; and a unique name assigned by the library. Each index entry collects all of the names by which an event is refered to by the toolset.

The user supplied name is the event class name as specified in the event heading of the *EDL* definition for an event. This is the name the user refers to when interacting with the debugging monitor components. All debugging activity originates at the behest of the user via some user-known event name. Since events are described in terms of user supplied names, toolset components also use the user supplied name in their interactions. For example, when the user requests a high-level event recognition from the event recognizer, the user supplied name is used. The event recognizer will request the definition of the event in terms of this name from the librarian. Any events which are members of the requested high-level event's cluster will also be referred to by their user supplied names.

The external name is the string that is found in the event class field of an event instance tuple which is created by an event generating source. The external name for a primitive event is specified as the system specific identifier in the event heading

[1]The root of the library structure, ~/eventlib, may be changed to any place in the file system. All that is required is for the library structure to be kept.

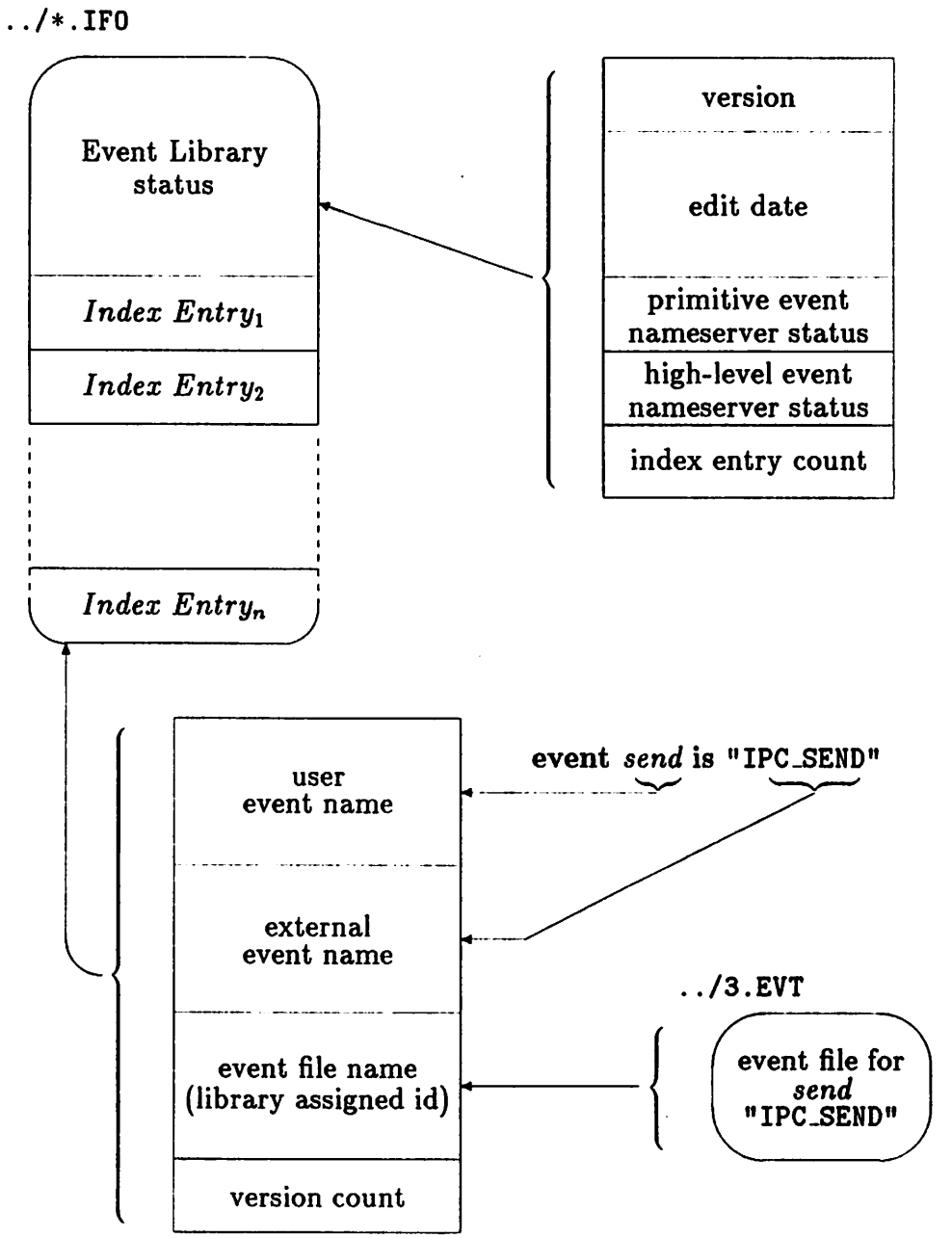


Figure 44: Event library info file structure

of its *EDL* description. High-level events are generated by event recognizers so, for high-level events, the external name is the same as the user supplied name.

The library assigned name provides a fixed length shorthand notation and unique identifier for an event definition. The librarian uses this name internally to refer to an event. Also, an encoded version of the library assigned name is used as the event file name by the library. Over the course of debugging a system, a user might change the meaning of an event in order to extend its scope or refine the model it represents. This activity will result in the situation where different event models are refered to by the same user defined name. The unique identifier assigned by the library ensures that there is an unambiguous current use for an event name, without destroying or deleting old definitions that still might be useful. Cleanup or renaming of old event descriptions is performed by utility functions that are explicitly invoked by a *tool* user.

Also attached to each index entry is a version counter for the user supplied name. The version count represents the total number of event definitions currently in the library that have the user supplied name. Each time the definition for the event is replaced by a more current one, the counter is incremented. Each time an old definition is removed by user action, it is decremented. This is a simple component that aids clerical tasks in the toolset.

The library status part of the library information file contains information pertaining to the library as a whole. The version field refers to the version of the librarian that created the event library. The edit date is the time of the last modification to the library. In order for the librarian to supply the unique library identifiers the status part keeps two numeric fields which act as name servers for primitive and high-level events. Primitive events are assigned a number in the range $1 \ldots 2^{15} - 1$ and high-level get $2^{15} \ldots 2^{16} - 1$. Each time the *EDL* compiler requests that an event be added to the library (or replaced), the librarian uses the appropriate name server to supply a unique identifier.

Event File Entries

Each event definition in a library is a self-contained unit. All information necessary to recognize an event and everything needed to describe its structure is found in its library-form event definition file. The library form of an event definition consists of two segments: the heading and table segments. Each segment has a fixed length prefix that contains sizes and counts for the text of the remainder of the segment.

The event heading segment (Figure 45) contains all of the information needed to describe the shape of an event. The event name and attribute descriptors completely define the tuple that represents the occurrence of an event instance. The event name part is a string of characters that contains the user defined name for the event. The attribute descriptors are a series of records, each describing a single attribute of the event. The descriptor has three fields: name, code index, and offset. The name field is the attribute name listed as a target of a **with** clause expression. This attribute name is used by high-level events as an operand in **cond** and **with** clause expressions. The code index field indicates which code string in the table segment can bind a value to this attribute when an event instance is created. The offset field is the attribute index at which the attribute is located in an event tuple for an instance of the event.

The constituent descriptors simply list the name of each event that appears in the event expression. Regardless of the number of times an event name is used in the event expression, it is only mentioned one time in the constituent list. The constituent name list is later used by the recognizer to find out what events must be available to recognize an instance of this event. For primitive events, the constituent name list is empty.

The table segment for an event definition describes the recognizer for an event – currently a description of a constrained shuffle system. The event table segment consists of five parts: the start state, the final states, the transition sets, transition tables, and code strings for **cond** and **with** clauses. The layout of the table segment for an event is illustrated by figure 46. The start and finish state list are simple coded

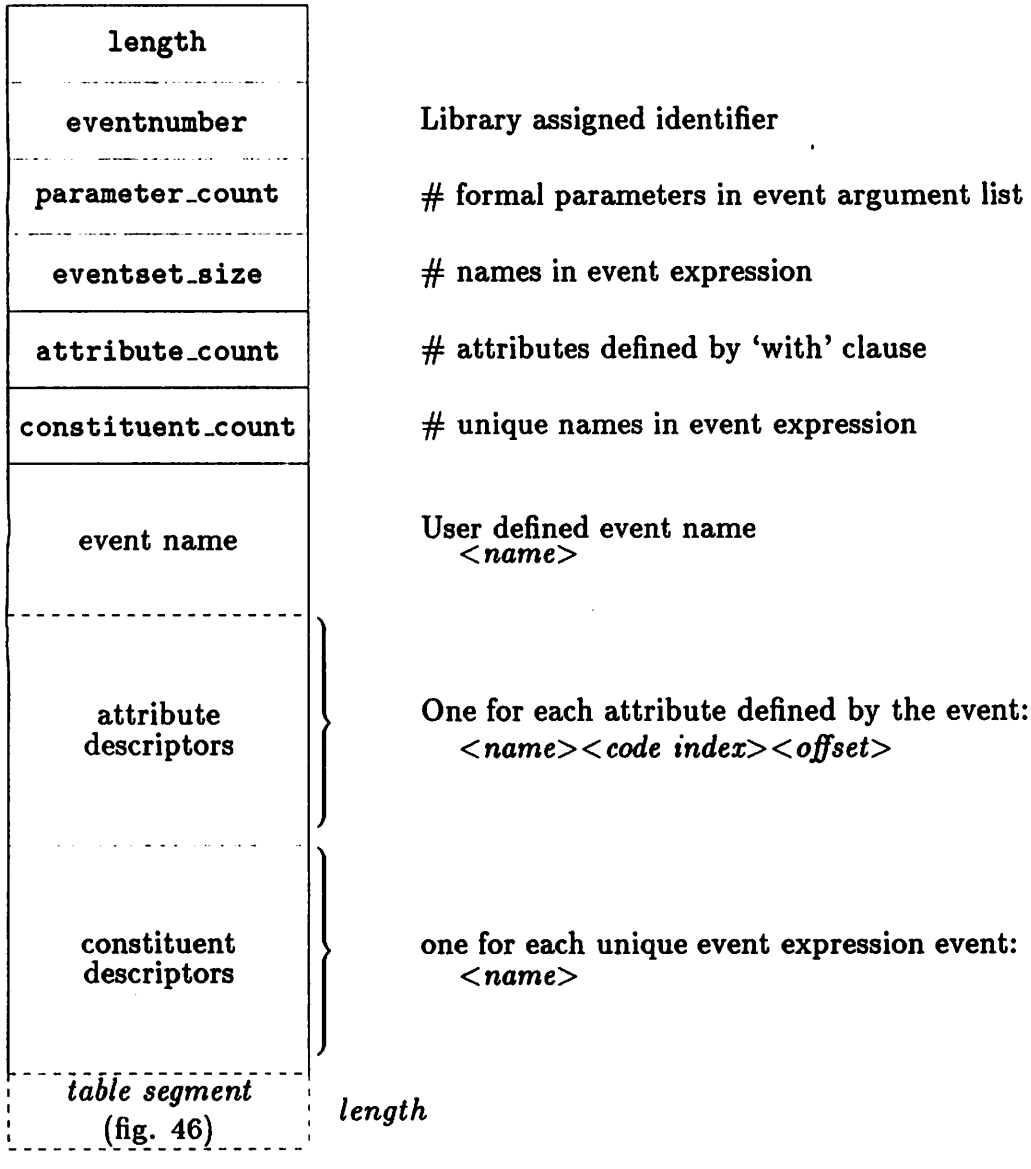


Figure 45: Event file - heading part

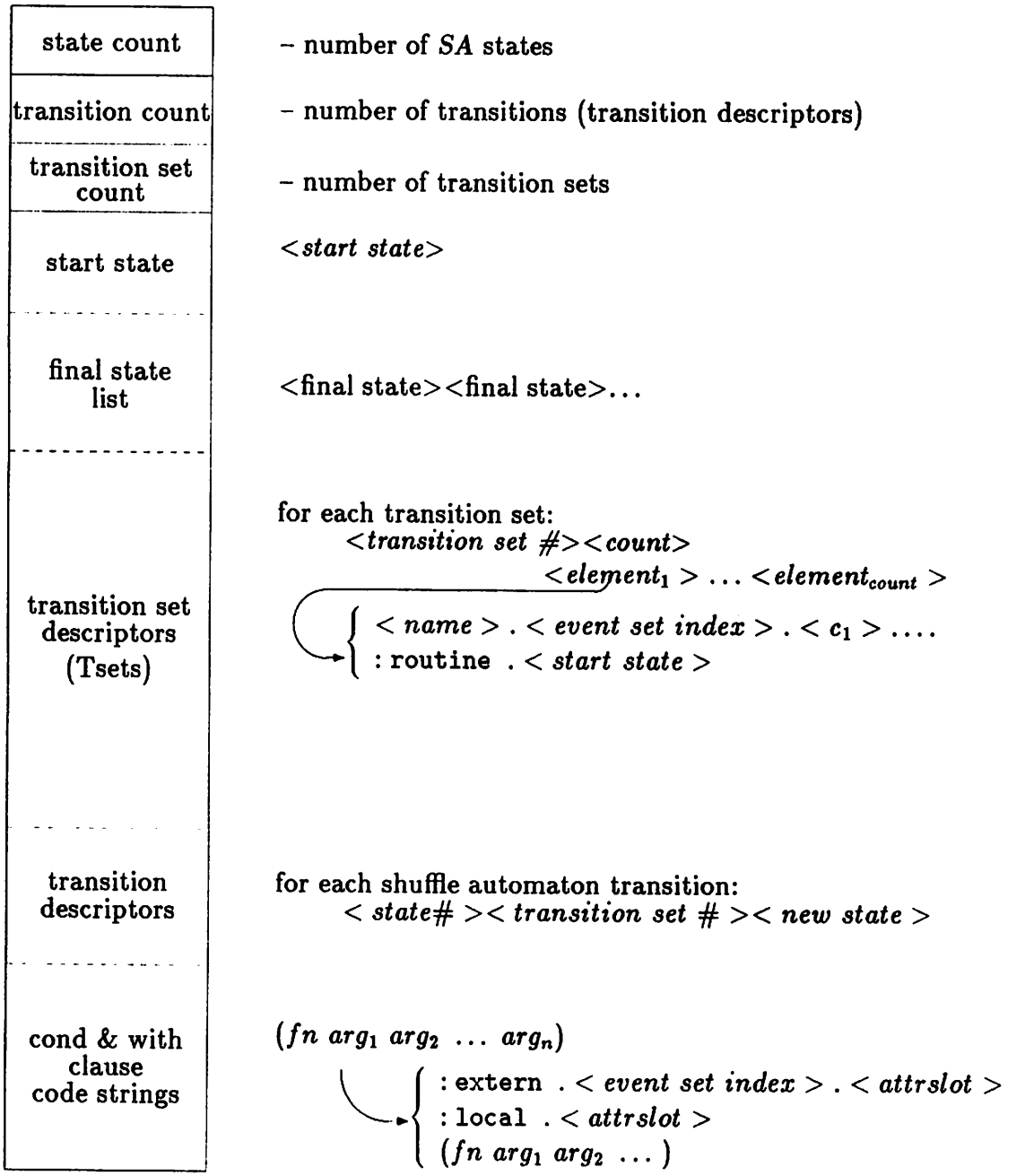


Figure 46: Event file - table segment

strings representing q_0 and F from the *CSS* root machine. Each *CSS* transition set is coded into a single transition set record that consists of a list of set element descriptors. The element descriptors have a form to represent a set element that is simply an event and a form for set elements that are shuffle routines. Descriptors for event name set elements include fields for the event name and the list of **cond** clause expressions in which the attributes of this transition set member must be evaluated. Shuffle routines do not have associated **cond** clause expressions. Instead, each shuffle routine of a transition set indicates the start state within the *CSS* for the sub-expression it represents. The use of the event set index field will be discussed in the sections describing the pending event list and event recognizer.

Shuffle automata transition tables are a list of tuples that define the state transition function of the shuffle automaton that will recognize an instance of the event. Each transition for which the function is defined is represented in a separate record as a 3-tuple consisting of a current state, a transition set number, and a new state. The transition set number indicates which of the above defined transition sets will move the shuffle automaton from the current state number to the new state number.

The last part of the table segment is a list of code strings used to evaluate **cond** and **with** clause expressions. The prototype event monitor contains a Lisp-like function evaluator used to obtain values from these expressions. Each code string, then, is represented as an S-expression in the usual prefix notation

$$(function\ operand_1\ operand_2\ \ldots operand_n).$$

Access to event attributes is provided by the `:extern` and `:local` forms of the S-expression arguments. The former refer to attributes of the constituent event set for an event. The latter form accesses the parameters bound to the event heading arguments.

The toolset structure diagram of figure 42 is somewhat misleading in that the librarian does not exist as a standalone, autonomous entity. Rather, it is a set of sharable routines capable of properly formatting and accessing the externally stored structures representing an event library. The routines form two layers. First is

a *cache* layer which is the interface to components such as the compiler or event recognizer. The cache layer maintains an internal representation of a library format event. The library interface layer is capable of translating between the internal representation and the secondary memory representation. The primary difference between the two representations is that memory addresses (internal representation) replace juxtapostion (external representation) as the structuring mechanism for the representation.

§1.3 *Event Queuing*

Event based behavioral abstraction describes its interface to a system under investigation as a probe capable of observing the event stream which is characteristic of the system. The event queuing module (from figure 42) is this interface between the event stream and the event recognizer. Note that the shuffle automaton formalism that is used as a pattern recognizer is not limited to observing serial event streams. Unfortunately, except in a setting where specialized hardware is available, receiving a parallel event stream at the recognizer is not possible. The prototype toolset uses an event stream that is serialized from whatever medium realizes it, onto the event queue.

The event queue (figure 47) is a shared structure that is maintained on behalf of the queuing module and the event recognizer. When the event queuing component is started, its event queue is empty. Following startup, the general scenario for event queueing is carried out each time an event instance is presented to the event queuing module. When an event message arrives, the event queuer wakes and accepts the message from the event stream. The event class is extracted from the event message and, if necessary, used as a key to request the corresponding event identifier and template from the librarian. An appropriately sized event descriptor is allocated; the library identifier is inserted into the descriptor; and each attribute is translated into a recognizer object and stored into the correct tuple field.

The completely described event is then added to the end of the event queue

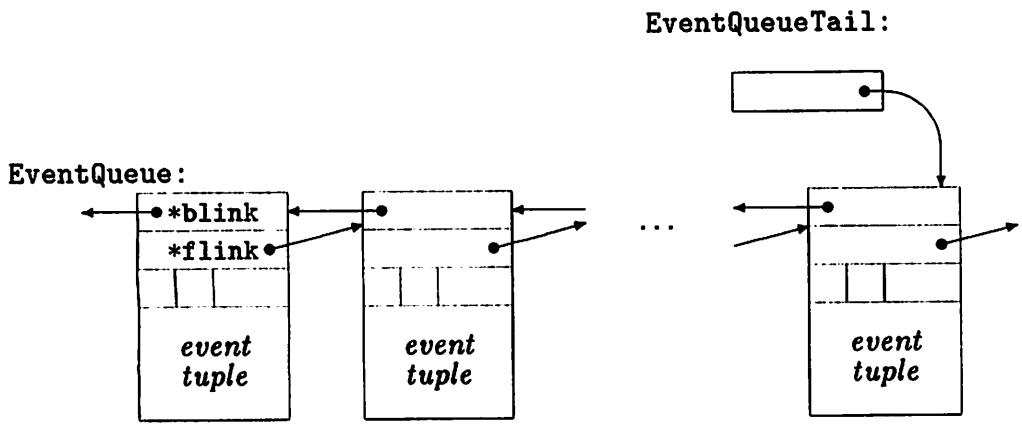
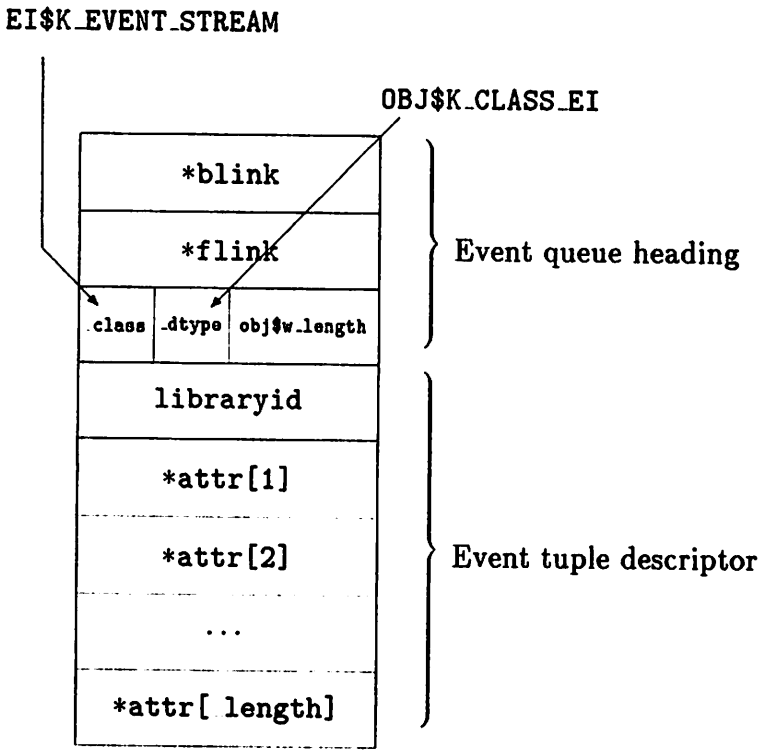


Figure 47: Event queue structure

(at `EventQueueTail`). The resulting event tuple queue describes a list that the event recognizer may examine as it needs to. The event queue can grow to an indefinite length, bounded only by the amount of space available for the structure. Alternatively, a tool user can issue a request to the queueing module to limit the length of the queue. This effectively creates a window that allows observation of only the most recent event stream instances.

An event queue entry fits the general structure of all data objects defined for use by the event recognizer. The internal event representation defined for the prototype is diagrammed in figure 48. The event tuple descriptor is catenated onto a heading that describes the object as an event instance, contains a count of the number of attributes the instance possesses, and contains forward (`*flink`) and backward (`*blink`) queue links. The forward queue link indicates the queue entries considered to be successors to the entry holding the link; the backward link indicates entries considered predecessors. The successor-predecessor relations are determined solely by arrival at the observer that maintains the queue.



blink - backward queue link,
flink - forward queue link,
attr[*i*] - event attribute descriptors.

Figure 48: Event instance internal representation

§1.4 *Event Recognizer*

The event recognizer fits the system behavior represented by the contents of the event stream into user defined behavioral models. In operation, the recognizer compares behavior models expressed as event patterns through *EDL* event descriptions to the actual event stream generated by the system under observation. Initially, a request for recognition of an event is made by a user via the behavior monitor component. The event recognizer requests the translated event description from the librarian and adds a description of the request to its list of pending requests, the pending event list (figure 49). Any high-level events which are members of the originally requested event are likewise requested, but do not result in pending event requests until they are needed to satisfy a transition from a state. The recognizer reads events from the event queue to try to find an appropriate set of constituent events which will result in instantiation of one of its pending event list entries.

Pending Event List

In the prototype toolset, event recognition is performed by matching the event stream (which has been serialized onto the event queue) with recognition requests described by pending event descriptors (structure *pe$pending_event*). Each descriptor forms an interpretation context for a shuffle automaton. The main loop of the event recognizer switches the interpreter's attention among the active pending event descriptors. The pending event list (figure 49) consists of two structures which tie all of the active pending event descriptors together. The two structures reflect the order of interpretation (through the `ScheduleQueue` list head) and the structural relations (reflected in the `PendingList` vector) of the set of currently active pending events. New pending events are added to the list via calls to an event recognizer routine, `InsertPendingEvent()`. The insertion point is designated to follow a specific list entry, thus permitting scheduling disciplines to be implemented externally. List deletions are handled by `RemovePendingEvent()` which removes a pending event from the list, thereby eliminating the chance for a pending event to compete for

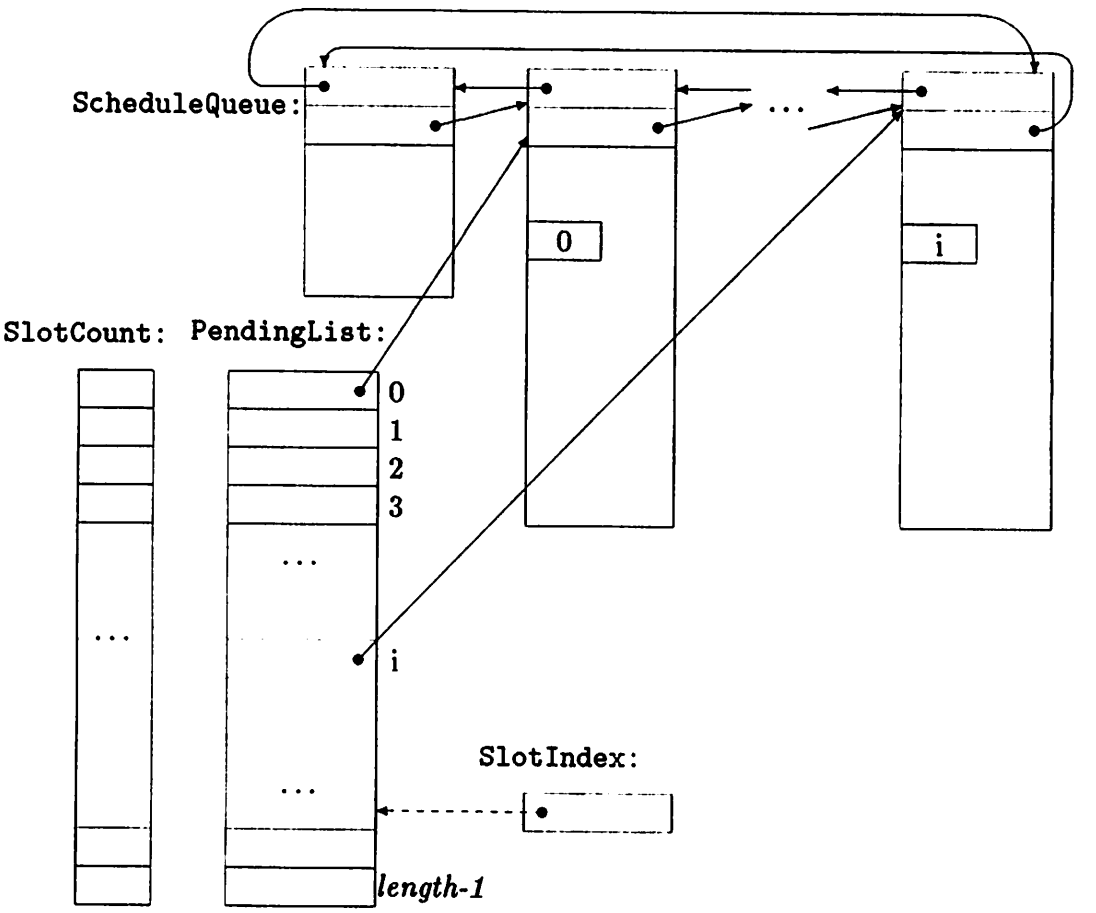


Figure 49: Pending event list

event queue entries. `RemovePendingEvent()` does not destroy the pending event list entry, thus allowing it to be inserted back onto the pending event list.

The structural relationships of the pending event list entrys are reflected in the `PendingList` (with its associated `SlotCount` array). The structural view defines a k-ary tree structure reflecting the dynamic state of the structure of a basic shuffle system. It is currently useful for determining sub-machine relations for a *BSS* and is a potential source of information to be used to (re)order the scheduling queue.

The `PendingList` is a vector of pointers to pending event descriptors. The entries in the `SlotCount` vector are usage counts for the associated `PendingList` slots. Each time a pending event is placed onto the pending event list it is threaded into the scheduling queue and given an available slot in the `PendingList` vector. The next available `PendingList` slot is indicated by the value of `SlotIndex` which is updated (modulo the vector size) to indicate the next empty vector entry. (This distributes the use of the slots and does not require compaction or linking together of the empty entries.) The `SlotCount` entry corresponding to the allocated `PendingList` slot is incremented each time the slot is allocated for a new pending event list entry. The slot number together with the usage count provide a unique name for each pending event throughout the life of the toolset (or some incarnation thereof).

Each pending event descriptor has fields that contain its own pending list slot number and the unique name of its parent pending event. If the pending event is a top-level pending event (created via a call to `Mon_RecognizeEvent()`), the parent name will be zero. When the pending event enters a final state, the unique name is used to determine disposition of the newly instantiated event. In particular, the unique name may be used to update input sets of a requestor. Another important function of the unique name is to allow the event monitor to forgo orphan elimination. Orphans are created when an event makes a transition not involving some of the sub-expressions and high-level events it has requested (as parts of transition sets). Rather than track down orphans, they are allowed to continue to operate until the parent spawning them has been removed.

Each pending event descriptor (figure 50) has fields which group into four distinct parts: those related to pending event list maintenance; those that describe the progress made in recognizing an event; a field detailing the explicit tasks to be performed when an instance is recognized; and a static part that points to the shuffle automaton and event definition structures for the event expression represented by the pending event list entry. Pending event list fields link the pending event descriptor to the scheduling queue (through *blink and *flink), and as a node in a high-level recognition hierarchy (_slot and _count fields).

Progress toward an event recognition is reflected in pending event descriptor fields that hold the current state of the shuffle automaton: its input register (*input_set); the current best fit of constituent events (*event_set); and the parameter and attribute values associated with this event instance. These will be more fully described in the later section that details recognizer operation.

Explicit tasks to be carried out upon recognition of an event are chained as a list of response control blocks (*rcb). This is an extremely powerful mechanism for invoking activity in response to successful event stream matching. Through the response control block mechanism, quite complex debugging tool responses may be easily associated with recognition.

When a pending event descriptor indicates that an event recognizer has recognized the event expression it represents, the monitor actions indicated by the pending event's response control block field are executed. A response control block (see figure 51) has three fields: a pointer to a monitor function (or procedure) that is to be invoked (*function()), a parameter to supply as an actual parameter to the called function (parameter), and a link to subsequent response blocks to execute. The indicated function is actually invoked with two arguments: the user supplied argument and a pointer to the pending event list entry that is responsible for this response control block. The response elicited by function invocation can be as simple as deletion of the pending event list entry, or as complex as executing a function at a remote node of the distributed system under observation.

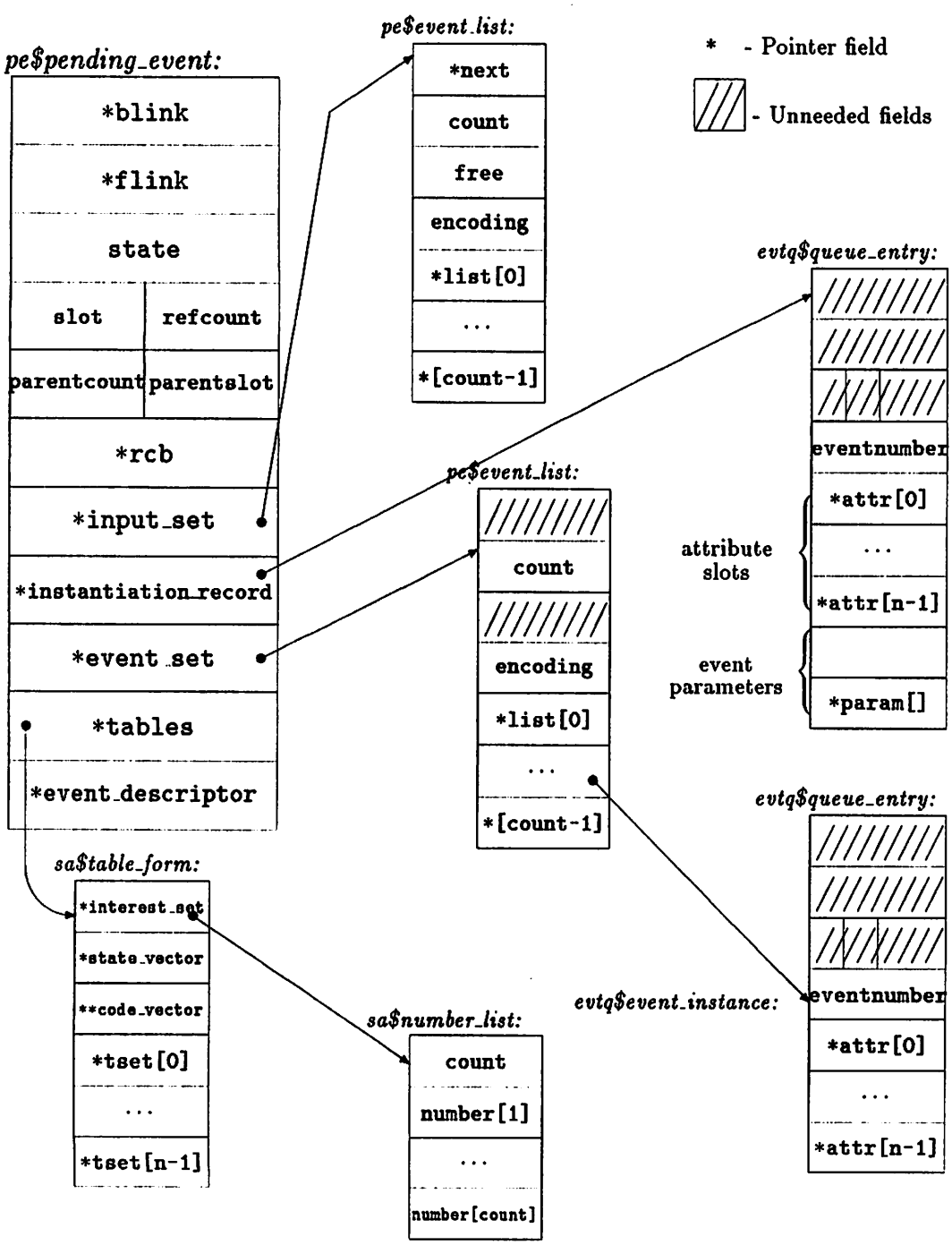


Figure 50: High-level pending list entry

Mere presence in the event stream does not ensure that an event will be reported to some tool set component. For any event to be recognized and create a tool response, it must have a pending event descriptor created and placed onto the pending event list. Three classes of pending event list entry are distinguishable; each results in different activity when instantiated

1. "Normal" recognition requests have been made explicitly by a call to the monitor routine `Mon_RecognizeEvent()`. These requests are always made at the behest of the user or through the behavior monitor. Each request may be for a high-level or a primitive event. High-level events are encoded and pushed back onto the stream when recognized, primitives are already in the stream.

2. "Internal" requests originate within the recognizer itself. These pending list entries represent high-level members of event expressions. Internal requests are scheduled as independent entities but are linked into a hierarchy through the `PendingList` structure. They are also pushed into the event stream.

3. "Shuffle Expression" requests correspond to subexpressions from a shuffle operator. A shuffle expression pending event shares all the progress marking data structures with its parent, sibling, and offspring recognition requests. Instances resulting from shuffle expression recognition are not placed back into the event stream. They only fill entries in the input register event sets.

Prototype Toolset Recognizer

The previous section described the organization of the pending event list which, when viewed through the hierarchical `PendingList`, reflects the status of a tool user's investigation of a system. The prototype recognizer implements an interpreter for constrained shuffle system-type automata. The interpreter uses structures bound to pending event list entries to guide and account for its pattern matching activities. In order to understand event recognizer operation it is reasonable to follow the

operations performed on a pending event descriptor by the recognizer in response to event stream inputs.

A pending event passes through three phases during its lifetime: creation, event accumulation, and final state activity. The creation phase begins with a request for recognition of an event subexpression. Then the recognizer will ensure that the description of the event to be recognized is available. A pending event descriptor is allocated and from information bound to the event description, the appropriate structures can be created that enable the recognizer to mark the progress made towards recognition of an event instance for the model. The kind of event subexpression represented by the recognition request will determine the amount of structure attached to the basic pending event descriptor.

Recognition of a normal high-level event requires allocation of a number of volatile structures and linkage to static tables that represent the shuffle automaton. The current state of the shuffle automaton is found in the pending event entry itself – thus requiring no additional storage. The shuffle automaton input register (figure 50, `*input_set`) is made from a chain of fixed length pieces, each holding a number of pointers to event queue entries. The input register structure can grow to arbitrary length during event recognition but initially only a single chunk is allocated and attached to the pending event entry. The set of events that have been matched to event expression members to move the shuffle automaton to its current state is kept in a structure similar to the input register. The `*event_set` represents the current best fit of event stream to behavior model. Unlike the input register, the event set has a known, bounded length – the number of event expression member events.

The final volatile structure needed for a high-level event recognition is one that will hold the parameters bound to event heading arguments so that they may be available for **cond** and **with** clause evaluations. To satisfy this need, an event instance structure is allocated that will be used to record the event instance resulting from recogniton by this pending event descriptor. The allocated instance record is

extended to accomodate any necessary event heading parameters and is linked to the pending event through the `*instantiation_record` field. When the instance represented by the pending event is recognized, the instantiation record will be sent to the queuing module for addition to the event queue. An appropriately coded version (without the parameter values) will be sent to any cooperating debugging nodes.

Requests for recognition of shuffle automata subexpressions are a bit simpler (figure 51). They share all of the volatile structures except their current state value with their parent pending event. Their creation is simple. Upon entry to a state that requires subexpression recognition, the list of pending event templates dangling from the state entry for the entered state is used to supply starting information for the subexpressions. Each entry is copied and and their hierarchy links (`_slot` and `_count` fields) directed at the parent pending event. The new pending event list entries are then scheduled to compete for entering events like any other.

Sharing within a *BSS* is controlled because all subexpression recognizers share their input set. The r_{map} function used by a subexpression recognizer only removes the events that the subexpression recognizer uses in its transition.

Requests for primitive events result in very simple pending event list structures (see figure 52). Primitive events do not need event lists for an input register or event set since they have no event expresssion members and will have no constituents when recognized. The instantiation record that will be returned by a recognition is simply the event queue instance that resulted in the recognition.

The shuffle automata tables (see below) for a primitive event contains only an interest set – indicating that the pending event list entry is interested in an instance of the primitive event being recognized. When an instance of the primitive event is available, the shuffle automata interpreter indicates that the primtive event entry has entered a final state and final state activity is carried out as for any pending event list entry.

In addition to the volatile structures that describe a pending list entry's recog-

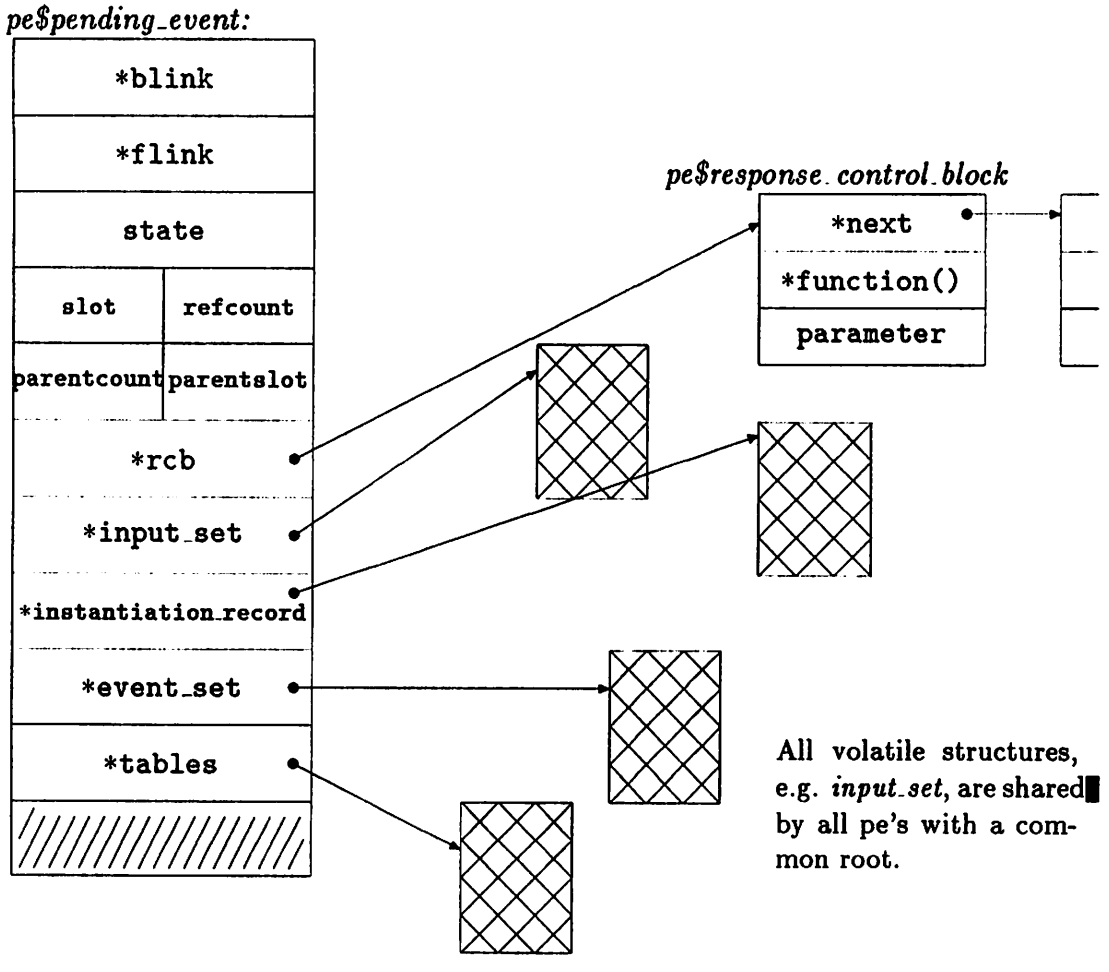
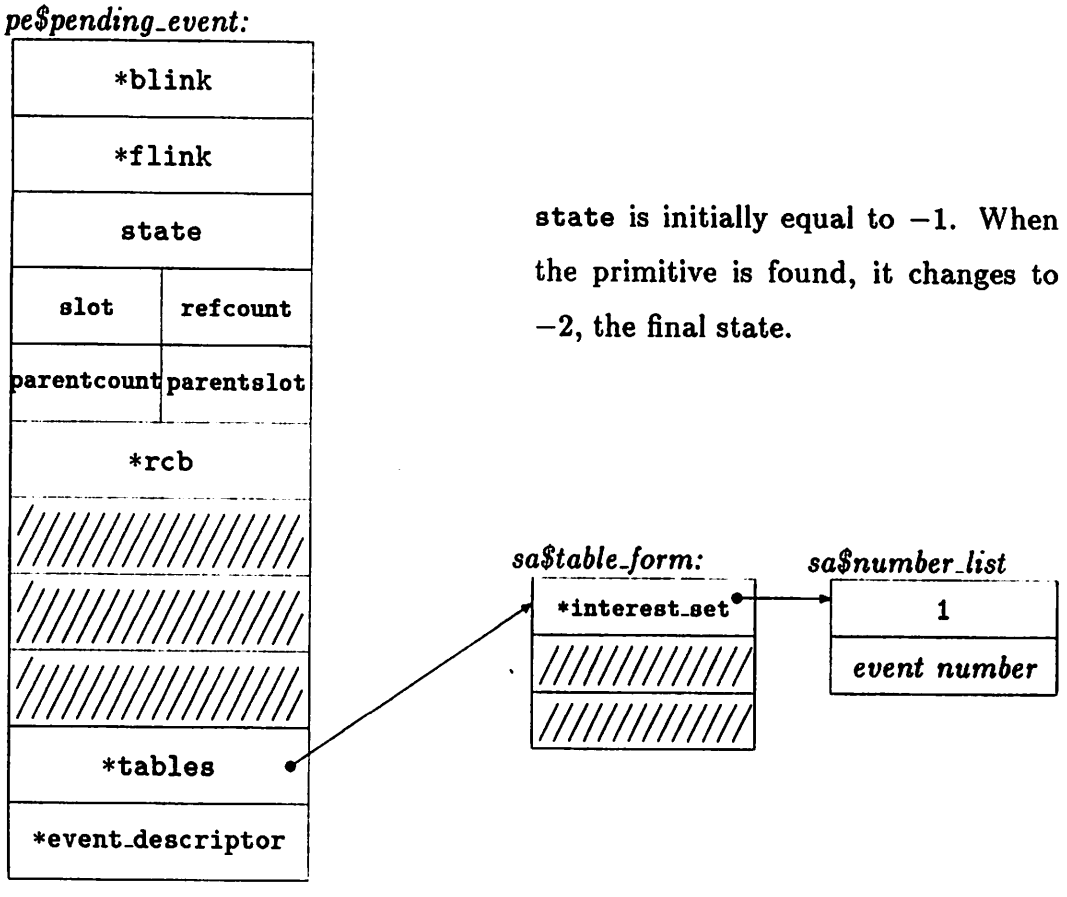


Figure 51: Subexpression pending event



state is initially equal to −1. When the primitive is found, it changes to −2, the final state.

Figure 52: Pending event list entry for primitive event

nition progress, tables that describe the event are necessary so that recognition can take place. Figure 53 outlines the form of the tables needed to interpret the shuffle automaton. The tables are read-only structures and so may be shared by all pending event descriptors whose events and event subexpressions they represent. The root of the tables (structure *sa$table_form*) holds pointers to a list of events this event can use (`*interest_set`), a vector (`*state_vector`) that locates the list of transitions from each state and a list of pending event descriptors for subexpressions relevant to the state, a list of pointers to descriptors for transition sets (`*tset[]`) defined by the shuffle automaton, and finally, a pointer to the list of code structures that are executed as **cond** and **with** clause expressions. For a normal high-level event (or subexpression) these tables are quite complicated.

The pending event descriptor is finally added to the `PendingList` and `ScheduleQueue` structure to complete the creation phase.

Following structure allocation and pending event list insertion, event accumulation for an event is performed according the coded shuffle automaton that is attached to the pending event descriptor. When the recognizer selects an event queue entry, it determines if the selected event is relevant to a pending event by examining the list indicated by the `*interest_set` field of the shuffle automaton description. The interest set is an ordered list of event numbers (the library assigned unique ids) that correspond to the event classes that are event expression members. If the class is located in the list, the event is potentially useful. In addition, the index of the event's location in the list is used as the event symbol in transition sets.

When an event instance is determined to be applicable to a pending event it is added to the input set pointed to by the `*input_set` field of the pending list entry. The rules for shuffle automaton operation state that if the input register contains the transition set needed for one of the transitions from the current state, a transition is made. In the event recognizer, this requires that the input set of the pending event be compared to some (or all) of the transition set descriptors attached to the shuffle automaton table vector through its `*tset[]` fields (see figure 53).

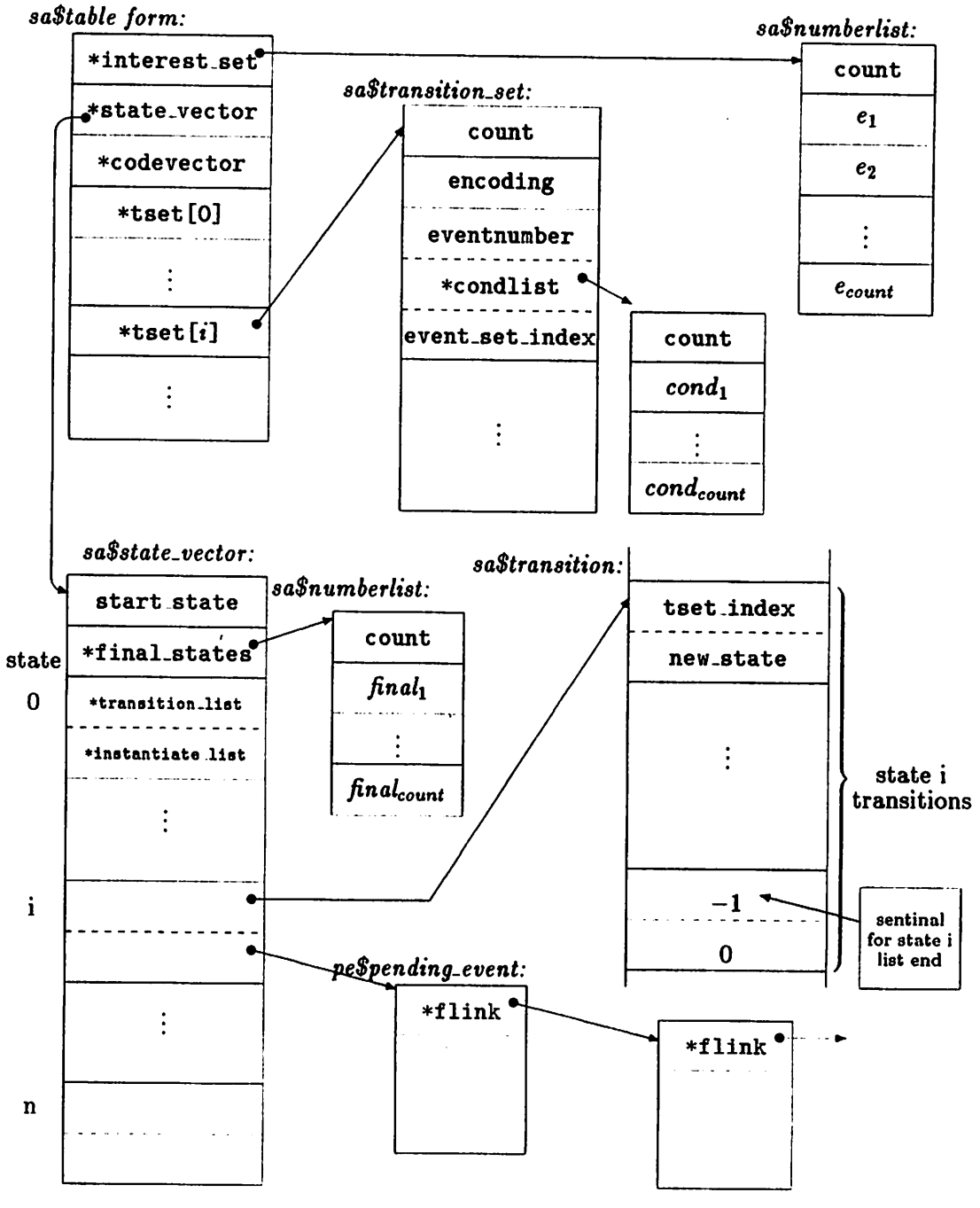


Figure 53: Encoded shuffle automaton description

Each transition set descriptor corresponds to a unique transition set of the shuffle automaton. Each event symbol that is a member of a transition set has three fields in the transition set descriptor. The `eventnumber` field is the index of the symbol mapped by the interest set. The `*condlist` field refers to a list of constraining clause expressions that are applicable to this event. The `event_set_index` field is the location in the event set that the match for this transition set element fills.

The process of determining that all members of a transition set have a corresponding instance in the input set would be quite cumbersome if each transition set member needed to be looked up in the current input set. A simple mechanism is used to greatly accelerate the comparison. Each index in the interest set (corresponding to an event symbol) has an associated prime number. The `encoding` field of all event lists is the value of the prime factorization of the list contents. Each time an event is added to the input set, the encoding is multiplied by the prime that corresponds to the added event. If the interest set for a shuffle automaton has events $e_1, e_2, \ldots, e_n$ with corresponding primes $p_1, p_2, \ldots, p_n$, then the encoded value for the input set is

$$p_1^{c_1} \times p_2^{c_2} \times \ldots \times p_n^{c_n}$$

where $c_1, c_2, \ldots, c_n$ count the number of instances of the corresponding events in the input set. This encoding is quite useful as it not only tells whether or not an event is in a set, but also *how many* instances of the event are in the set. Membership is easily determined by division of the encoding by the prime representing an event.

Determining that a transition set is contained in the input set is a simple matter of dividing the transition set encoding into the current input set encoding. If the encodings divide evenly (no remainder from integer division), the transition set is contained in the input set. So, the process of determining if a transition may be made from the current state is a matter of dividing the input set encoding by the encoding for each transition set that leads from this state. In the current implementation, if there are two transtion sets that can cause a transition, the first transtion set for which this is determined is used.

All that remains is locating the set of transitions for the current state. This is done indirectly through the `*state_vector` field of the shuffle automaton tables vector. For each state of the shuffle automaton, the state vector has an entry containing a pointer to the list of transition-set/next-state pairs defining transitions from that state (see figures 53 and 54).

Each entry in the transition list corresponds to a single transition defined by the transition function for the shuffle automaton. The entry `tset_index` field selects one of the `*tset[]` pointers from the shuffle automaton table vector. The `next_state` field is the state to change to if the transition set referenced by `*tset[tset_index]` matches the input register. The list for a particular state begins at the entry indicated by the state vector pointer and is bounded by a special entry (-1 in the `tset_index` field).

To briefly recap: event applicablity for a pending event is determined by examining the interest set, which also returns a symbol number and an encoding factor; the event is added to the input set whose encoding is updated with the encoding factor; the current state is used to locate the state descriptor that points to the list of transition descriptors for the state; each transition descriptor indicates a transtion set to compare to the input set, via their encodings; if one of the transition sets matches, any applicable **cond** clauses may be evaluated before the state is changed to the state indicated by `next_state`.

Following a transition to a new state, the final states vector is examined. If the new state of the shuffle automaton matches one of the final state entries, final state activity is performed. If not a final state, any subexpression or high-level events that might be required for the transitions from the new state must be prepared for recognition. The `*instantiate_list` field of the state vector entry for the new state will provide this information. Chained from this field is a list of pending event descriptors templates; each is copied and appropriate parent pending event information filled in before they are added to the pending list structures. Once added to the pending event list the subexpressions are able to stand alone as they

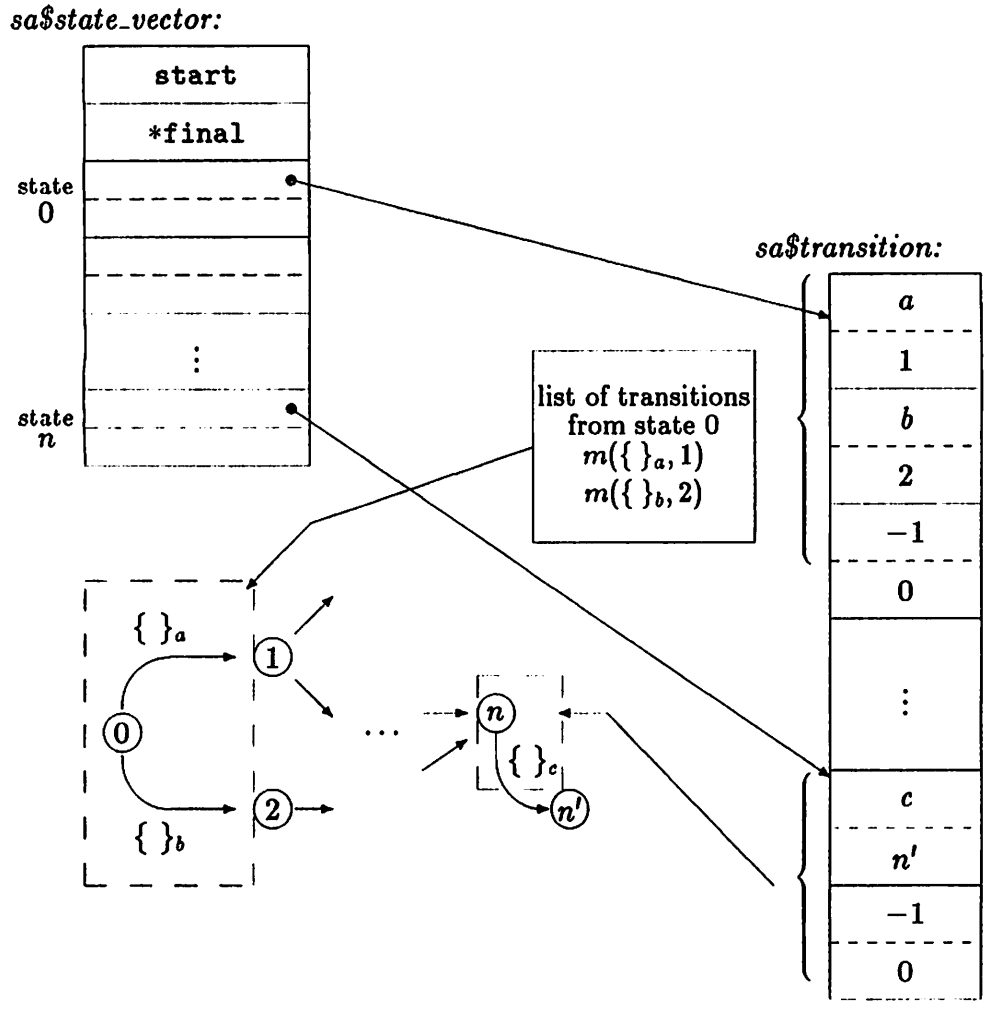


Figure 54: State transition vector

accumulate a set of event instances that will cause them to enter a final state.

The third phase for a pending event, final state activity, occurs when the pending event has attained a state that is listed in the final states list of the state vector. First, the list of response control blocks associated with the pending event list entry is executed. Next, the final disposition of the pending event list entry is determined, based on how the pending event is classified.

- Pending event list entries that represent shuffle subexpressions and high-level event members of transition sets return an entry to be added to the input set of their calling pending event. The subexpression is then recycled unless the parent pending event no longer exists (because it has been recognized) or is not in the state this subexpression was called from (because the parent made a transition). In either case, the child is deleted since it is no longer needed.

- If the pending event list entry is a high-level event request made by a user (or a transition in the continuous mode, see below), the event instantiation record is pushed into the event stream. The pending event entry is then deleted.

Recognizer Operation Modes

The event recognizer can have two basic modes of operation. In the *continuous* mode, the event stream is monitored for all instances of some subset of the high-level events contained in a view of the system. Each recognized high-level event instance is placed into the stream for possible integration into higher level events. The effect is a continuous bottom-up growth of high-level abstractions, resulting in a constant, high-level view of the system. Another effect is that sharing of events is less controlled. A high-level event instance recognized explicitly for a transition set could be incorporated into another transition set in the same basic shuffle system. The continuous mode is useful for gaining an overall picture of system activity through a viewpoint.

The *goal-driven* mode begins with a request for recognition of a high-level event. Members of the event cluster defining the high-level event, which are themselves

high-level events, similarly become the object of recognition requests and are requested. This way, only events needed for a specific behavioral model are abstracted from the event stream constituents. Goal driven recognition requires the minimal amount of processing for behavior recognition and most tightly controls sharing. Goal driven observation is useful when a user desires to focus attention on specific behaviors.

§1.5 *Behavior Monitor*

The Behavior Monitor is the user interface to the debugging toolset. Monitoring and intervention requests made by tool users are passed through the behavior monitor to relevant tool components. Analysis of recognized patterns – the pattern analysis component of *EBBA* pattern recognition – is performed in the behavior monitor component. Pattern analysis tools are defined in terms of structures contained on the pending event list maintained by the event recognizer. There is no explicit connection of the behavior monitor to the pending event list in the prototype. The reason for this is that the event recognizer is considered to be a data abstraction. Requests may be made for execution of functions it defines such as, "recognize an event" or "how is this recognition proceeding?". The data abstraction treatment is sufficient for the early version of the toolset, but is less satisfactory for more sophisticated behavior monitor components. A more efficient structure for the pending event list would be in a shared memory.

The current prototype toolset only cursorily treats the tools that compose the behavior monitor component. In all, three kinds of pattern analysis tools can be distinguished. The first simply provides statistics on the volume of information contained in a partially recognized pattern. These tools allow a debugging monitor user to view the gross progress of the model fitting activity performed by the event recognizer.

The second kind of behavior monitor tool is capable of determining differences between a user behavior model and the event stream representing system behavior.

This kind of tool is presently not provided by the behavior monitor. However, preliminary investigations indicate that the error correcting component of a minimum-distance error-correcting parser (MDECP) [Aho72] might be employed to provide some of this function. An MDECP is capable of determining a measure for the distance between two strings. The distance is defined in terms of the substitutions, deletions, and insertions of symbols that might be used to derive one string from another. When applied to behavior model differences, an error correcting component could inform a user about possible changes to make to a behavior model or the incoming event stream so that they would match more closely. These changes might be indicative of behaviors that need to be altered to produce a properly executing system (or an improved user model).

Another candidate for inclusion as a model differencing component is based on an error-correcting tree automaton (ECTA) [Lu78]. In this method, errors in tree structures are defined in terms of the substitution, stretch, split, branch, and deletion transforms that might be made to a tree to make it more correct. ECTA could be employed in the same advisory role as MDECP.

The third kind of tool that the behavior monitor should provide is one for graphical display of behavior model information. Users may find it difficult to expliot sophisticated high-level tools for examining behavior without similar sophistication in the information presentation techniques [Model79]. For the prototype, a preliminary pass has been made at providing interactive structure diagrams that represent a behavior model (similar to figures 2, 3, and 16).

§2. Chapter Summary

This chapter has presented the implementation of a prototype toolset designed to operate within the *EBBA* framework. The toolset consists of a collection of asynchronous, cooperating components whose goal is to recognize user models of behavior in the event stream created by a system under observation. The toolset

is described independently of any connection to a system other than through the event stream. The event recognizer that is the heart of the toolset implements an interpreter for shuffle automata models of behavior recognition. Several deficiencies in the prototype toolset have been pointed out, together with changes that might be made in a re-implementation to alleviate these deficiencies.

CHAPTER VII

DISTRIBUTION OF THE DEBUGGING TASK

The previous chapter detailed the structure and operation of a toolset that implements monitoring in terms of *EBBA*. The main task performed by the toolset is fitting actual system activity, represented by the event stream, to user models of that activity. Missing from the tool set description is an explicit connection to the distributed programs the toolset is intended to investigate. This chapter will continue development of the toolset and the *EBBA* paradigm by describing the distribution of the toolset in a distributed system.

The underlying structure of *EBBA* provides the means to extend the approach into a more powerful distributed program for debugging distributed programs. Events as a medium of tool information exchange isolates both users and distributed tool components from the vagaries of heterogeneous systems. This together with the ability of *EBBA* tools to operate with varying levels of abstraction granularity permits an *EBBA* distributed debugger to balance node performance, administrative and logical system partitioning, communication performance requirements, and logical program residency to achieve a level of tool/system interaction necessary to explain errorful behavior. Because distribution evolves naturally from a basic implementation of *EBBA* rather than as an afterthought, providing a distributed program debugger is straightforward.

Another tool feature missing from the prototype toolset description is any means for experimentation with a system undergoing investigation. Intervention in system activity at a local node is provided in a manner similar to non-*EBBA* debugging tools. It is only in a distributed system that advantages of *EBBA* for intervention

are apparent. Thus, along with the description of the possible distribution levels, strategies for intervention in system behavior through the *EBBA* framework will be examined.

The model of programming that is assumed by this dissertation is one in which the procedural and data components of the computation are physically dispersed in a computer network consisting of heterogeneous computational nodes. The collections of components that form a computation must cooperate with one another to produce an overall computational effect. Communication among components is via message passing as well as transfer of control. The components can operate asynchronously with respect to one another, and may be created, destroyed, and moved in response to local or non-local conditions. The communication medium is simply a transport mechanism for interprocess communication and is not assumed to be totally reliable.

Distributed debugging can mean that there is a centralized tool for debugging distributed programs, or that a tool may be located at one node and be used to debug a program at another node, or that the debugging tool itself may be distributed along with the distributed program. Most extant interactive debugging tools for distributed systems are of the first two types. They are constructed from existing break-examine debugging tools and either replace the interactive user with a remote terminal-type connection across a network or dedicate an interactive terminal to the debugging at each node (a physical combination of the first two kinds).

EBBA-based debugging treats debugging tool distribution as a metaphor for system organization that is accomodated naturally and without need for special cases. Extension of *EBBA* as a distributed program can enhance tool transparency, reduce latency and uncertainty for monitoring and intervention, and take better advantage of computational power available in a network computer system. An important result of extending *EBBA* as a distributed program is an ability to provide various levels of debugging service tailored to the capabilities and constraints of specific nodes in the distributed system. Intervention and experimentation strategies likewise have various levels of service. As noted in the introduction (Chapter I),

EBBA provides no guidelines for which individual system components are to be manipulated by experimentation activities. However, increased distribution and cooperation of the event monitor components provides a means to improve the accuracy and usefulness of interventions that are to be performed.

Initially, distribution of the prototype implementation requires addition of an additional component to the basic toolset, the *Network Interface (NI)* (see figure 55). Each node of the distributed system that is to participate in debugging activity contains some form of an *NI* component to act as the agent for the central toolset. The functionality of the network interface components will vary according to the level of service provided by an individual node, but each *NI* will have certain basic capabilities. As more capabilities are needed at a remote node in support of more sophisticated node participation in overall debugging activity, more functions are added to the *NI* or additional components from the basic toolset are moved to the node. Each increase in node sophistication comes with a corresponding increase in the amount of local resources needed to support distributed debugging.

§1. Remote Debugging

Remote debugging is implemented by placing a user and the set of debugging tools employed by the user at a single node of the distributed system. The node that contains the debugging tools may or may not be a participant in the distributed computation under investigation. The central node provides a way to direct attention to a specific node among the various participants in the session. All monitoring and intervention requests to be effected at the remote nodes of the system originate at the central node. Each remote node that is participating in debugging tasks necessarily contains an agent to aid the central debugging tool. The agent has tacit knowledge of the local environment and will respond to requests made by the central site. As the computation progresses, the tool user interacts with the programs in the computation through the central toolset and its remote agents.

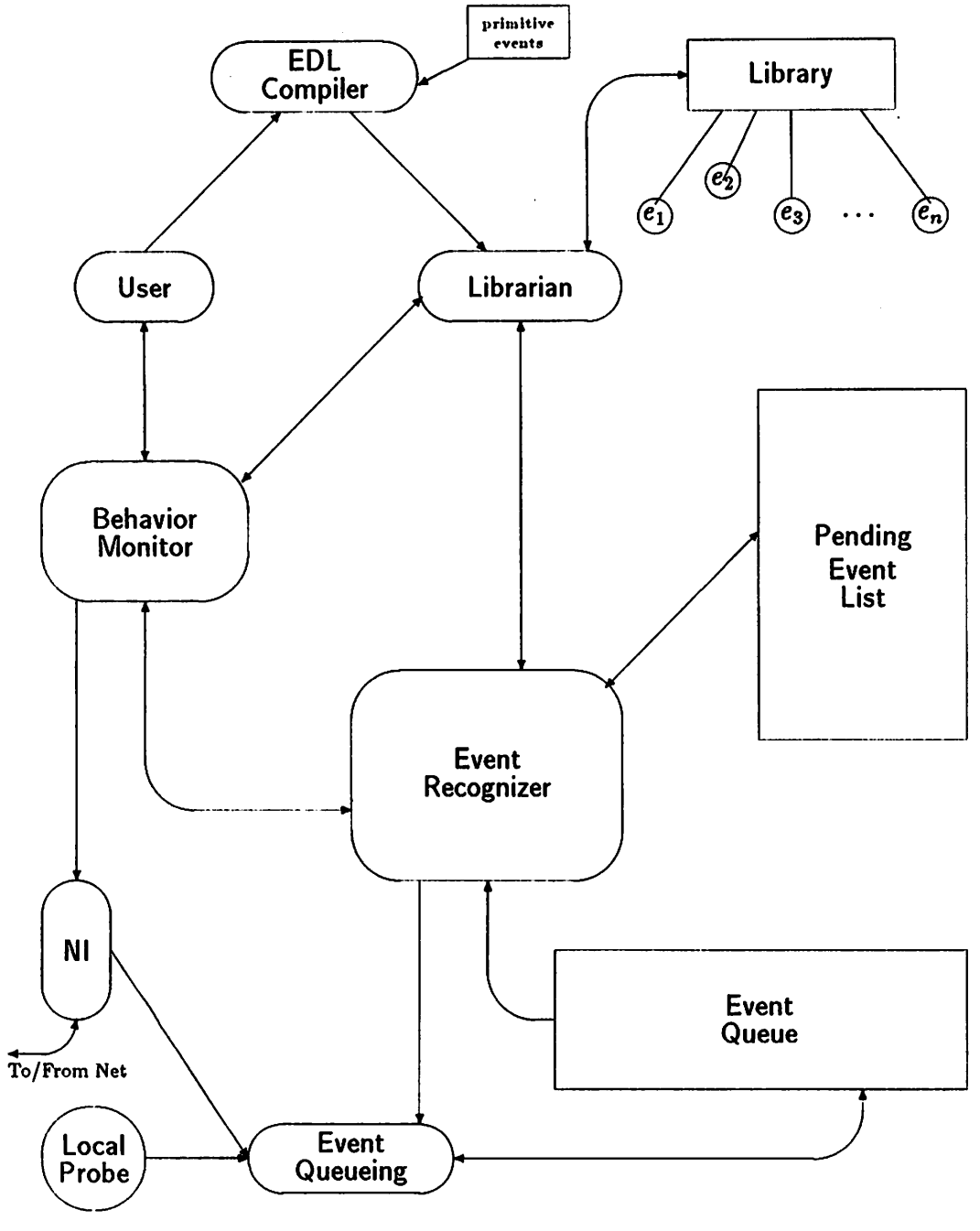


Figure 55: Central node of distributed toolset

Remote debugging facilities are easy to provide. Indeed, debugging tools that implement some form of remote debugging are the predominant sort currently provided for distributed programs. The primary drawbacks to the use of remote debugging are:

- latency associated with reading information and effecting intervention activities often renders the information out of date and the intervention lacking the desired effect, and
- because of the heterogeneous nature of the processing elements in a distributed system the computation details change from node to node of the system. In traditional debugging tools this creates difficulties for obtaining a coherent view of the computation.

Remote debugging within the *EBBA* framework is simple to provide and is the method that is least invasive to a remote node. The very nature of *EBBA* that provides a uniform view of a system overcomes the coherent view difficulty. Engineering tradeoffs that move the use of information closer to the place where it originates help overcome the latency problem.

To extend *EBBA* into a distributed system for remote debugging requires simple remote agents. The remote agents are network interface components that simply implement the communication protocol required for successful cooperation with other nodes and are able to listen for locally generated events. This configuration, diagrammed in figure 56, represents the minimal contents of the *EBBA* toolset for distributed systems. Through simple additions to the network interface component, the latency associated with intervention activity can be improved upon and communication bandwidth requirements for event reporting lowered.

§1.1 *Simple Remote Debugging*

The minimal arrangement of remote agents and central event monitor provides *simple* remote debugging. The toolset remote agents send all locally observed event traffic, unfiltered, to the central site. The central site consists of a complete toolset

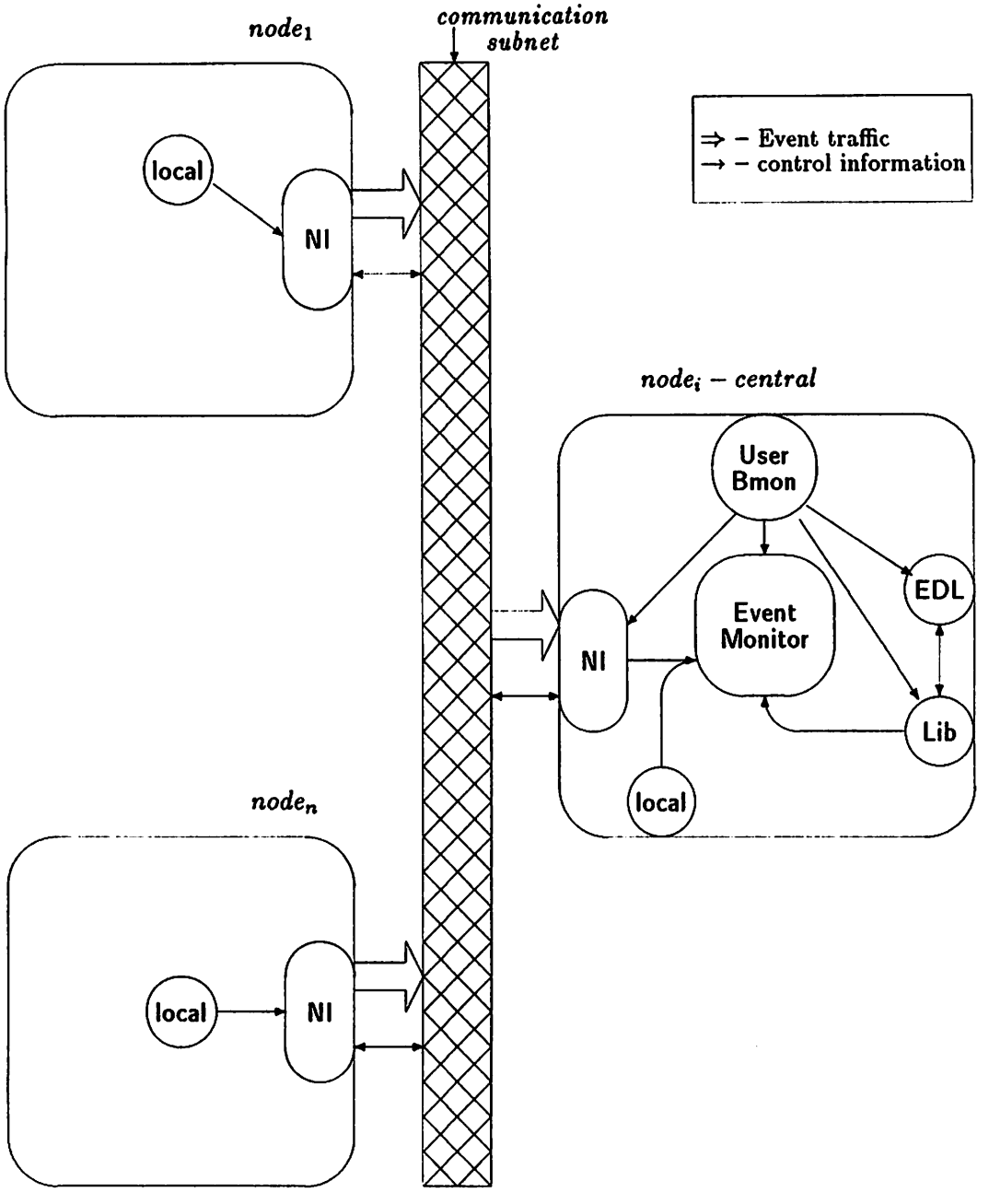


Figure 56: Remote debugging toolset

(figure 55) which is responsible for recognition of higher level behaviors based on the primitive events received from the remote agents. When the central monitor detects a higher level event of interest to the debugging tool user, it issues requests to relevant remote agents to intervene in the activity of the participating computing elements.

This is remote debugging in its classic form. This blind form of event reporting requires communication costs to be directly proportional to the volume of local primitive events. The latency for intervention is the time required for the message exchange plus whatever is necessary for the central abstraction node to perform its task. Simple remote debugging is useful because it is quite easy to provide and, if monitoring is central to uncovering errorful behavior, intervention latency is less an issue.

Comparison of the distributed toolset (figure 55) to the basic toolset (figure 42) shows the only addition is the *NI* component. The *NI* connection to the behavior monitor permits the user to direct attention to remote node *NIs*. The central *NI* is directly connected to the event queuing component for the purpose of moving the event instances sent from remote nodes to the event recognizer.

The use of the remote debugging configuration begins when the tool user starts up the central tool. Requests are made through the behavior monitor to start up the *NI* component at desired nodes. The local *NI* establishes a communication path to the indicated nodes and requests each remote *NI* to start and connect to its local event probe. Each event observed by the probe at a remote node is passed to the *NI* for transmission to the central tool. Whenever the central tool detects a high-level event that requires activity (intervention or otherwise) at a remote node, the behavior monitor is informed and a message to that effect is sent by the central node *NI*. The requests for remote debugging activity can be stored in a response control block for a pending event. Direct user requests for remote node activity are likewise directed through the behavior monitor to central *NI* to remote *NI*.

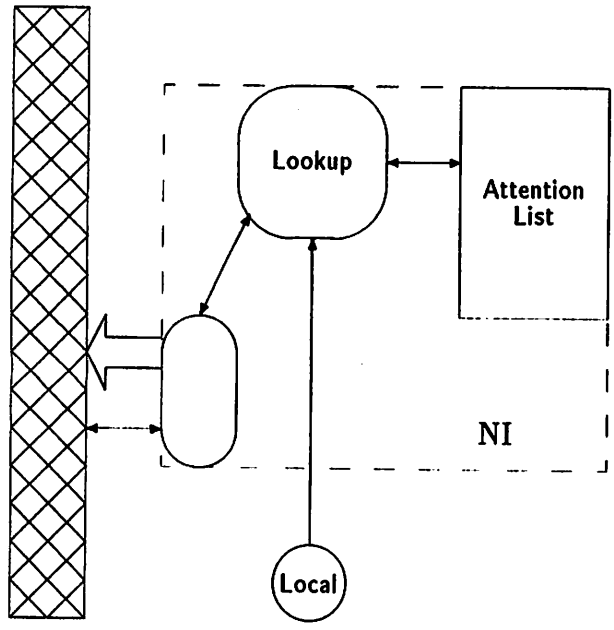


Figure 57: Remote agent with filtering capability

§1.2 *Filtered Remote Debugging and Preset Actions*

An improvement in the communication performance can result if the remote agent is instructed to report only certain primitive events. This comes at a slight cost. The remote agent must be altered to hold a table of event class names and implement the appropriate table manipulation routines (figure 57). Each primitive event generated at the remote agent's node is checked against the table and only those currently requested are sent out. This *filtered remote* debugging reduces the communication bandwidth requirements by removing event instances that would be filtered as unnecessary by the central tool. Perhaps more important is the inspection capability added to a remote agent to effect the filtering process.

With the addition of inspection to the remote agent's capabilities another level of intervention service is provided. In simple remote debugging the physical separation of the originator and effector of the intervention request can introduce delays which render the response too late for the desired effect. In order to improve on the latency,

requests for intervention at a remote node are made in advance of the time they are to be carried out. Execution of these *preset actions* are stimulated by detection of a local event. The event observed at the remote node that results in the intervention action is a primitive event since these are the only events available to the node.

The preset actions technique is useful because all high-level behavior models ultimately are composed from primitive events. Thus a user requiring intervention activity based on recognition of a high-level model implicitly specifies the action in terms primitive events. The difference, of course, is that a primitive event incorporated as a high-level model constituent is a product of filtering and constraint satisfaction. With preset actions the latency to intervention has improved, but since the intervention occurs in response to a locally observed primitive event which has not undergone filtering and constraint satisfaction, many unnecessary (and, depending on their nature, error causing[1]) interventions will occur.

§1.3 *Task Distribution Through Simple Cooperative Debugging*

A simple next step to further distribute the debugging task is to give each remote agent the ability to examine the network event stream. This extension allows remote nodes to initiate local debugging activity based on events occurring at *other* remote nodes. This limited listening capability implements *simple cooperative* debugging in which many nodes may be active participants in debugging activity. Now it is possible to effectively cause network-wide patterns of debugging activity to occur. Instead of being able to act only in reponse to locally observed events, groups of nodes may react to conditions that affect each other. The simple cooperative capability is useful where an intervention is required to affect program components at multiple nodes of a system such as an experiment where a tool user would like to synchronize distributed components upon the occurrence of an event at another node.

It is quite straightforward to extend the debugging tool set to offer simple coop-

[1] Repeated, unnecessary interventions might disrupt timing relations radically.

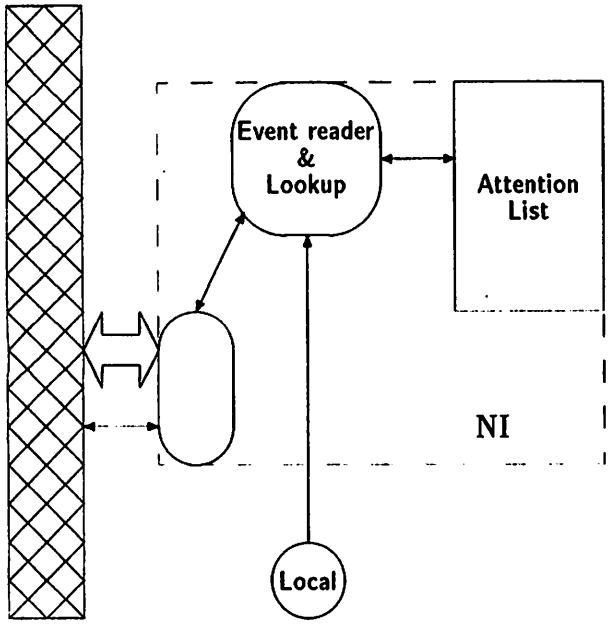


Figure 58: Simple cooperative debugging local agent

erative debugging services. The event reader of the central toolset's event queuing module is added to the toolset at the remote node. No pattern matching or constraint evaluation is required, only the simple table lookup required by preset actions (Figure 58). The remote agent now must listen to the communications medium for event traffic and pass this to the reader. The reader's task is to extract the event class part of the event tuple which is then used as a key to search the attention table. When a match occurs, any activity associated with the matched attention table entry is carried out. A simple optimization of simple cooperative debugging, intended to reduce the number of unnecessary interventions, is obtained by adding a simple constraint evaluation mechanism. Constraint evaluation is only possible in terms of simple attribute constraints since only single events are involved in the reduced recognition task performed by the attention table lookup.

This exhausts the possiblities of remote debugging. Remote debugging is simple and does not consume a large quantity of resources at remote nodes. However, the

communication medium is heavily used for low-level event traffic. This potentially poses a problem where contention for the communication medium is affected by this traffic. The latency to intervention problem is improved by adding simple event identification capabilities to each remote agent. In order to further reduce the communication requirements and provide more meaningful remote node intervention, it is necessary to perform local model abstraction and communicate higher level information among participants. The next section explores this approach.

§2. Distributed Debugging with Distributed Event Recognition

Distributed debugging from the *EBBA* perspective is more than remote debugging of distributed programs. *EBBA*-based distributed debugging emphasizes model abstraction at remote nodes and exchange of resulting high-level events by participating nodes. While simple cooperative debugging, described in the previous section, forms a distributed program for debugging distributed programs which is quite powerful, it can still impose unacceptable levels of event message traffic and result in unwanted interventions in system activity. Benefits accrued from a more fully distributed event recognizer include:

- lowered communication bandwidth due to exchange of only necessary or important events,
- improved intervention accuracy when intervention is based on local abstractions,
- load distribution of the processing required to effect debugging, and
- an ability to handle more general distributed system architectures that include gateways and subnets that are not fully connected.

Distributed *EBBA* high-level debugging nodes are capable of much autonomous activity and, once set in motion, may carry out a large portion of the monitoring and intervention activity necessary to understand a system error without requiring interaction by a tool user.

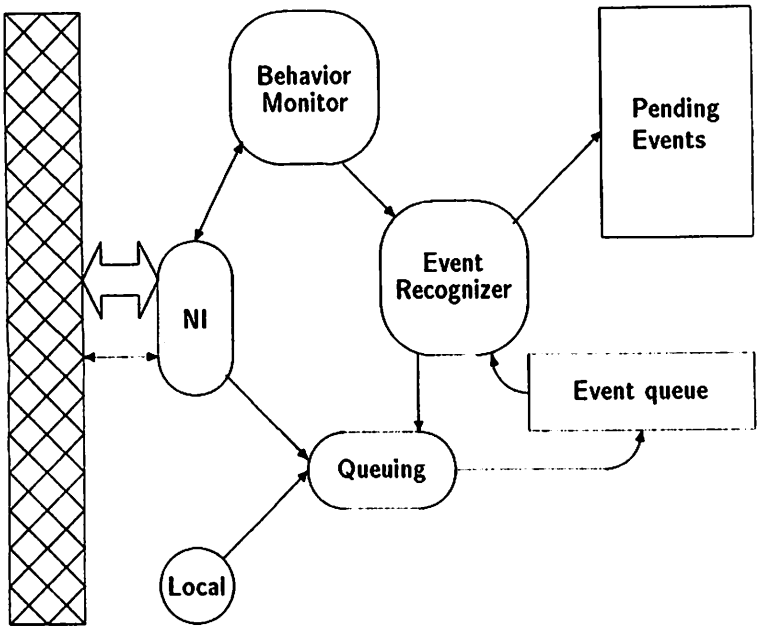


Figure 59: Distributed debugging agent

§2.1 Centrally directed

The components of the remote agents at *EBBA* high-level nodes are indicated in figure 59. Each component performs the same task as its central toolset counterpart. Missing are the components dedicated to viewpoint creation and maintenance: the *EDL* compiler and event librarian. The *EDL* compiler only responds to user created behavior models so it only needs to reside at a node where a tool user might need access. This is fully in keeping with the *EBBA* caveat that debugging requires some user to direct the search for errors.

However, the services of the Event Librarian (from figure 55) are required by all nodes that perform behavior abstraction. Each node that is involved in high-level event recognition should have the same view of the system. Therefore, the librarian should be the same one accessed by all nodes cooperating on recognition tasks. The librarian, which may be located at any single node or be a distributed component itself, acquires an additional connection to the network so that all high-level nodes

may access its contents. An example of a central librarian is seen in the central node of figure 60.

In the centrally directed use of the distributed toolset, coordination of debugging activity at a remote node is fully under control of the user acting through the central toolset. The tool user is responsible for partitioning the modelling tasks that are designed to uncover errorful system behavior and then directing the appropriate remote nodes to work on their portion of the overall modelling activity. The remote nodes that have been assigned activities make arrangements with each other to obtain high-level events and exchange locally recognized event tuples required for effective cooperation.

For example, the model describing internode communication from chapter II, *inter_msg_exchange*, can be partitioned into two node events: one for the master node, the other for the slave node. The user would direct the master node to look only for its portion of the abstraction and likewise the slave to look for its own portion. Each node working on abstracting its local behavior reports its findings to the central node whose inter-node event constraints are evaluated to complete model evaluation. The cooperating nodes contact the librarian to obtain the definitions for the events they have been assigned to observe. The centrally directed, distributed use of the *EBBA* toolset is the limit of its capability in the prototype implementation. Further enhancements that more fully automate searches for erroneous behaviors are best covered by artificial intelligence techniques beyond the scope of this research. However, in order to provide some completion and point to possible future work, the next section will speculate on enhancements to the toolset which are compatible with the basic toolset design.

§2.2 *Cooperative Debugging*

The central theme to extending distributed debugging with artificial intelligence techniques is to apply more of the information that is available to the modelling process. The goal is to improve the accuracy and relevance of that process. We are

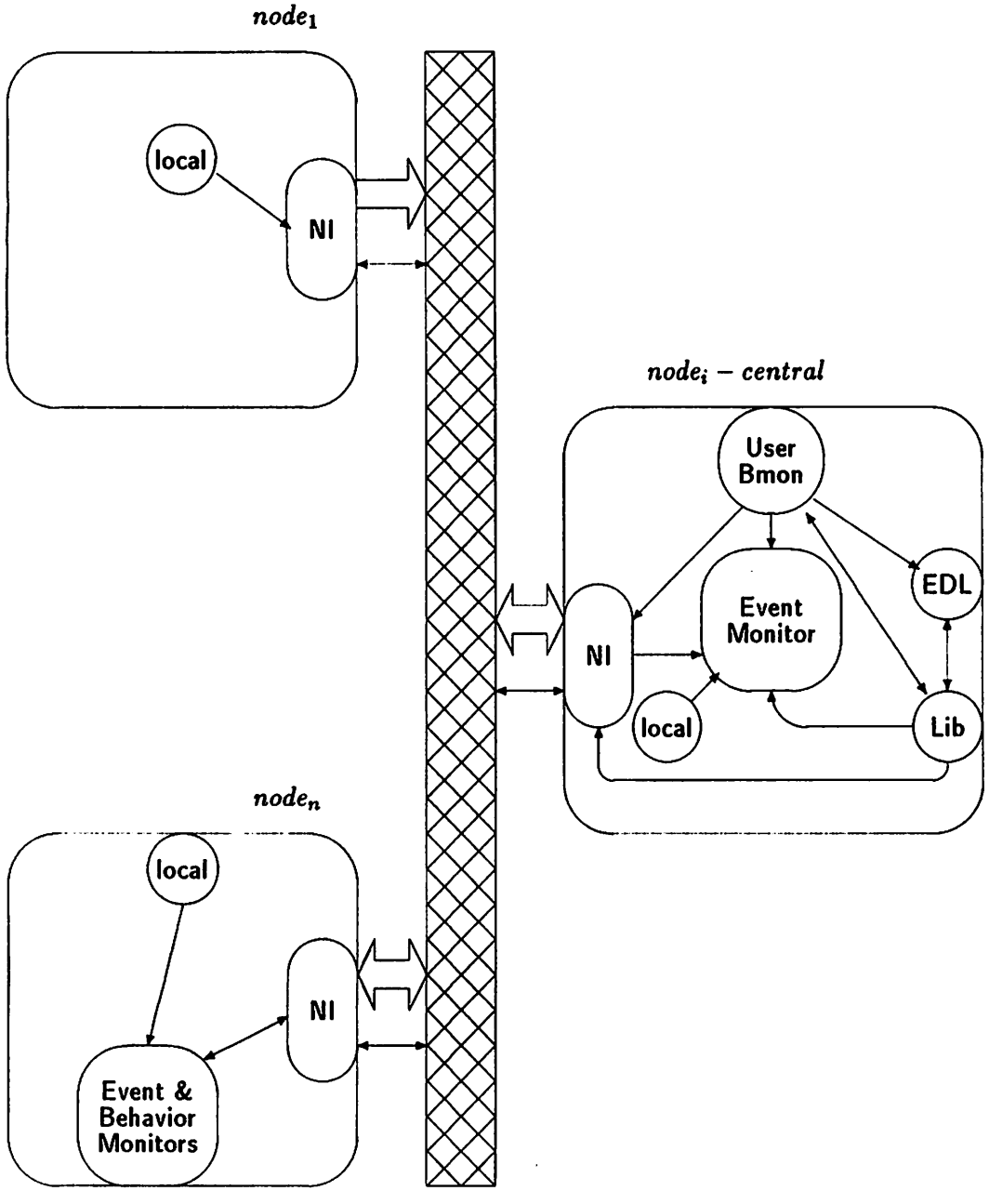


Figure 60: Mixed service level distributed tool

not looking for an automatic debugger which, given an errorful program, indicates where its erroneous behaviors originate and what needs to be done to correct them. Instead, the assistance envisioned would be of two kinds:

1. Explanatory aids, which could give extensive analysis of the difference between user behavior models and actual system activity, and
2. Speculation aids, which would take notice of user goals and information accessing trends to try out models derived from this information. The objective here is to provide a suggestive role for the tools by filling in areas of a model that the user may have overlooked; or the tools might work on variations on the models the user has specified.

The partitioned design of the toolset components allows easy integration of these kinds of aids into the behavior monitor component.

Various techniques suggest themselves to assist in providing these tools. One purpose of event libraries is to encourage reuse of abstractions. Sophisticated user aids might extend reuse by structuring strategies and plans for debugging complex problems around libraries and users' prior experience with similar situations. More detailed task decomposition, driven by the goal directed nature of plans for debugging, could result in partial result exchange among cooperating distributed debugging nodes.

Supplying this more cooperative debugging environment will require much more intensive use of system computational resources. The need for and use of these higher order tools will naturally need to be balanced against their impact on the system being debugged and the subtleness or complexity of the errors undergoing investigation. It is seldom advisable to crack an egg with a pile driver.

§3. Summary

This chapter completes the description of the *EBBA* toolset as a distributed program. It was shown that by working tradeoffs between remote information processing and communication, an *EBBA*-based distributed debugging toolset easily

and naturally provides a range of solutions to monitoring and intervention in a distributed system. Complex, heterogeneous, or arbitrarily structured network architectures are accomodated easily because of the uniform view of system activity provided by events and the ability of the distributed *EBBA* tools to operate on high-level abstractions of behavior. The increased distribution of abstraction capabilities helps *EBBA* to overcome inaccuracies in debugging activity that result from physical distribution of the computation undergoing investigation.

CHAPTER VIII

SUMMARY, CONCLUSIONS, AND FUTURE WORK

§1. Summary

In this dissertation we have described event based behavioral abstraction and shown how it might be used as a paradigm for debugging distributed systems. We have described how abstraction can be applied in a systematic way to model the behavior of an errorful system, and illustrated how *EBBA* can be used to explain an error in a distributed program. We have characterized the process of matching actual system activity to user defined models of behavior as a special pattern recognition problem that must derive useful information from a noisy, uncertain environment. A model is developed that formalizes description of abstract behavior models and provides a means to recognize occurrence of actual behavior patterns that match behavior models. In addition to these uses, the model is capable of describing a number of the problems posed for behavior recognition to support *EBBA*. The model indicates what capabilities are needed in a toolset designed to assist its users in debugging systems from the event based perspective. The implementation of a basic software toolset that permits its users to monitor system activity through abstract behavior models is then detailed. Finally, the extension of the toolset as a distributed program capable of exploiting desirable characteristics of distributed systems to perform debugging is presented.

§2. Conclusions

The goal of the research for this dissertation was to investigate event based behavior abstraction as a paradigm for high-level debugging of distributed software systems. The intention was to develop techniques and tools that would facilitate debugging in the face of the complexity and uncertainty attributed to programming in a distributed system environment. The research was required to address distributed debugging from a conceptual standpoint since no previous work could supply any theoretical or rigorous treatment of debugging, and as a practical problem of explaining erroneous behaviors in ways that do not require examination of low-level program details.

Our results demonstrate that software debugging can be approached in a systematic manner. Techniques for top-down program construction have shown themselves to be valuable for engineering more reliable software by reducing the complexity of individual tasks and using abstraction to manage the complexity of the overall system. Use of similar concepts in *EBBA* could lead to "better debugging."

EBBA employs abstraction to both manage system complexity and to improve the operational quality of debugging tools in distributed systems. Using a set of events as a basis for debugging appears at first to limit the monitoring power of a debugging tool by forcing a fixed view in terms of only those events. However, by varying the levels of granularity of events and describing different system components through varying viewpoints all of the activity of a system can be described and monitored.

The use of abstraction for debugging isolates a user from the computation details of individual systems and, in a distributed system environment, allows a user to concentrate on system behavior rather than implementation details of heterogeneous system components. Use of abstracted events by distributed *EBBA* debugging tools allows the tools to minimize their effect on system components and improve the accuracy of their monitoring and intervention activities.

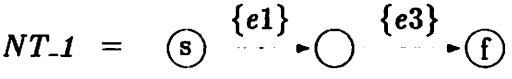
§3. Future Work

The high-level debugging techniques reported in this dissertation automate user modelling of errorful systems with the goal of aiding the discovery of similarities and differences between intended and actual system behaviors. To accomplish this, methods are provided to describe behaviors and to observe their occurrence in a system. The directions for future work will enhance the current methods and explore advanced levels of debugging tool automation. There are three areas in which the *EBBA* research should proceed. First is the addition of methods that permit description and comparison of behavior patterns that are not easily described now. Second is a need to provide some high-level style of interaction with the distributed debugging toolset. Finally, the real potential of the behavior monitor component needs to be explored.

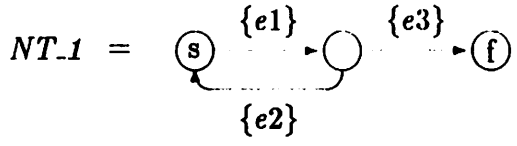
A pattern element that can only be discussed with great difficulty is so-called negative events. A negative event occurs when some specified event or event subexpression does not occur. Consider the following *EDL* description (supplemented with a negative event operator)

event *NT_1* **is**
    e1 ∘ ¬*e2* ∘ *e3*
end

The intent of the model specified by *NT_1* is that *e1* and *e3* events occur without an intervening *e2* event. This seems simple enough to describe. How should such an event expression be recognized? The simple recognizer



is not adequate because it does not take into account the possible occurrence of an *e2* event. A more satisfactory recognizer is described



This recognizer is capable of recognizing the *e1* and *e3* events without the intervening *e2* event. Important here is the context supplied by the *e3* event which determines when the *e2* is a problem. But the event description

event *NT_2* **is**
    $e1 \circ \neg e2$
end

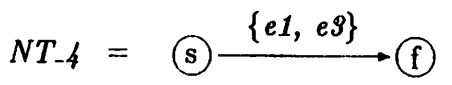
is a bit harder. Here there is no context to signal when *e2* has not occurred. It might be possible to impose a time constraint on this event such as in the following event description

event *NT_3(epsilon)* **is**
    $e1 \circ \neg e2$
cond
    $e2.time - e1.time > epsilon$
end

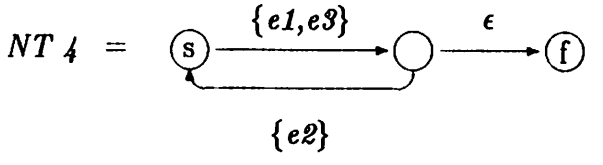
Here the intent is to describe a time interval during which the negative *e2* event must be observed. A general solution might be one in which a "time stamp" event is implicitly created and used by the recognizer as the context for the missing event (replace *e3* with the time stamp event). To illustrate why such a general solution is hard, consider the following shuffle expression which involves a negative event.

event *NT_4* **is**
    $e1 \triangle \neg e2 \triangle e3$
end

More than context is needed in this case. The recognizer



ignores the *e2* as did the first attempt for *NT_1* and is thus unsatisfactory. Like the second recognizer for *NT 1* using the *e2* event to restart the recognition is a better solution for some cases. That is



However, the concurrency of the shuffle operator is no longer being observed and in an environment with a parallel event stream, this description is not guaranteed to work.

It seems that the negative event should be treated as a "fence" or "fail" operator as in SNOBOL4 [Griswold71] or Prolog [Clocksin81]. In these pattern matching implementations, encountering the fail operator causes alternative patterns to be tried. As for pattern matching in support of *EBBA*, the current description and monitoring schemes would need to be altered considerably to accomodate negative events.

The most immediate need for the toolset is addition of a meaningful user interface. The existing tools are cranky and prone to proceeding for long periods without informing the user that anything useful is taking place. Each component must be manually started in its proper order and imprecise user commands can result in loss of control for an extended time. The only real outputs are detailed textual displays. A good user interface would consist of several interactive displays which contain the status of the viewpoints in use and the status of user requests for model recognition. A properly annotated display for a behavior model, similar to the structure dia-

grams used by chapters II-V, possibly enhanced with color, would be an important component of the interface.

The use of individual model displays would be the focus of all interaction with the debugging tools. Currently popular interactive environments [Barstow84] feature a mode-less style of user/tool interaction. Through the interface a user selects an object and indicates what operation to perform on the object. The range of operations depends on the type of the object, thus eliminating the possiblity of performing nonsensical operations on objects. In the envisioned debugging environment, the viewpoint display would list the event classes which are already available in the environment. A user could select an event class and request that the system be observed for its occurrence, display its structure, or replace it with another event definition. All interaction with the tools would be through the displays representing a behavior model.

Finally, a longer term project to follow this work is one that involves explicit use of pattern analysis techniques to more fully automate discovery of differences between actual and assumed behaviors. The differences between models explain what parts of the models are inaccurate. One of the real goals of high-level programming techniques is to minimize a user's ability to create a program that is different from their model of what the program is to do. Sophisticated debugging techniques need to help a user explore the differences between what they meant and what they actually expressed as a program. In the prototype toolset, the behavior monitor component is intended to be the repository for these difference detecting components. For example, a simple tool to help point up differences would work in a manner similar to minimum distance error correcting parsers (MDECP) [Aho72] or error-correcting tree automata (ECTA) [Lu78]. The tool would be able to create its own behavior models described in user supplied models by applying MDECP transformations to the pattern structure. Alternatively, an MDECP-style tool could use its transformations to reorder the event stream or influence the recognition process. The purpose of these actions would be to demonstrate to a user that specified

patterns might match the event stream if altered or viewed in specific, well-defined ways.

Expert systems and other "intelligent" methods are also candidates for inclusion in the behavior monitor. These AI techniques could bring problem domain-specific knowledge to bear on a system or assist the tool user with general debugging practices and thus greatly expand the role of the behavior monitor. Ideally, the Behavior Monitor could serve in an advisory role by helping the user focus attention in consistent directions and provide knowledge of previously solved or recognized anomalies in system behavior. A behavior monitor so formed, would be capable of requesting recognition of behavior patterns in response to its own perceived state of the problem and modifying its view according to the responses it receives. These behavior monitor models of system activity might form the basis for the advisory role to be played by the system. User interaction with the behavior monitor advisor would help both the user and the advisor to adjust their model of system activity with the view to accurately locating error causing behaviors.

A clairvoyant debugging tool is perhaps a long, long way in the future. However, a realistic goal for high-level debugging tools is, in the least, to encourage the machines to do more of the work in understanding the problems that they have.

BIBLIOGRAPHY

[Aho72] A.V. Aho and T.G. Peterson, "A Minimum-Distance Error-Correcting Parser for Context Free Languages," *SIAM Journal of Computing,* Vol 4, pp. 305-312, Dec. 1972.

[Aho77] A.V. Aho and J.D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading, Massachusetts, 1977.

[Andler79] S. Andler, *Predicate Path Expressions: A High-Level Synchronization Mechanism,* PhD. Thesis, Carnegie Mellon University, Dept of Computer Science, 1979.

[Balzer69] R.M. Balzer, "EXDAMS – Extendable Debugging and Monitoring System," in proceedings *AFIPS Joint Spring Computer Conference,* 1969, pp. 567-580.

[Barstow84] D.R. Barstow, H.E. Shrobe, and E. Sandewall, *Interactive Programming Environments,* McGraw-Hill, 1984.

[Bates78] M. Bates, "The Theory and Practice of Augmented Transition Network Grammars," in *Natural Language Communication with Computers,* ed. L. Bolc, Lecture notes in Computer Science #63, Springer Verlag, 1978.

[Bates82] P.C. Bates, J.C. Wileden, "Event Definition Language: An Aid to Monitoring and Debugging of Complex Software Systems," in proceedings *Fifteenth Hawaii International Conference on System Sciences,* Jan. 1982.

[Bates83] P.C. Bates, J.C. Wileden, "High Level Debugging of Distributed Systems: The Behavioral Abstraction Approach," *Journal of Systems and Software,* vol. 3, #4, 1983.

[Birrell82] A.D. Birrell, R. Levin, R. M. Needham, M.D. Schroeder, "Grapevine: An Exercise in Distributed Computing," *Communications of the ACM,* vol. 25, #4, April 1982.

[Brinch-Hansen73] P. Brinch-Hansen, *Operating Systems Principles,* Prentice-Hall, Englewood Cliffs, NJ, 1973.

[Brinch-Hansen77] P. Brinch-Hansen, *The Architecture of Concurrent Programs,* Prentice-Hall, Englewood Cliffs, NJ, 1977.

[Bruegge83] B. Bruegge, *User Manual for KRAUT - The Interim Spice Debugger,* Spice Document S156, Carnegie-Mellon University, Shenley Park, Pittsburg, PA 15213, April 1983.

[Campbell74] R. H. Campbell, A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," in *Lecture Notes in Computer Science,* editor G. Goos and J. Hartmanis, Springer-Verlag, 1974.

[Campbell79] R.H. Campbell, I.B. Greenberg, and T.J. Miller, *Path Pascal User Manual,* UIUCDCS-R-79, University of Illinois at Urbana-Champaign, 1979.

[Clocksin81] W.F. Clocksin and C.S. Mellish, *Programming in Prolog,* Springer Verlag, 1981.

[Cohen77] J. Cohen, N. Carpenter, "A Language for Inquiring about the Runtime Behavior of Programs," *Software Practice and Experience,* vol. 7, pp. 445-460, 1977.

[Curtis82] R. Curtis, L. Wittie, "BUGNET: A Debugging System for Parallel Programming Environments," in proceedings *Third Internatioinal Conference on Distributed Computing Systems,* Oct. 1982, pp. 394-399.

[Dahl72] O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare, *Structured Programming,* Academic Press, London, United Kingdom, 1972.

[Deitel84] H.M. Deitel, *An Introduction to Operating Systems,* Addison-Wesley, Reading, Massachusetts, 1984.

[Elliot82] B. Elliot, "A High-level Debugger for PL/1, Fortran and Basic," *Software Practice and Experience,* vol. 12, pp. 331-340, 1982.

[Enslow78] P. H. Enslow, "What is a "Distributed" Data Processing System," *IEEE Computer,* vol. 11, #1, pp. 13-21, Jan 1978.

[Ferrante82] J. Ferrante, *High Level Language Debugging with a Compiler,* RC-9210 (#40367), IBM T.J. Watson Research Center, Yorktown Heights, N. Y., Jan. 1982.

[Fu82] K.S. Fu, *Syntactic Pattern Recognition and Applications,* Prentice-Hall, 1982.

[Garcia81] H. Garcia-Molina, F. Germano, W.H. Kohler, *Debugging a Distributed Computing System,* DEC/TR-118, Digital Equipment Corporation, Corporate Research Group, Maynard, MA, Aug 1981.

[Graham75] R.M. Graham, *Principles of Systems Programming,* John Wiley & Sons, 1975.

[Griswold71] R.E. Griswold, J.F. Poage, and I.P. Polonsky, *The SNOBOL4 Programming Language,* Prentice-Hall, 1971.

[Guarino78] L.R. Guarino, *The Evolution of Abstraction in Programming Languages,* CMU-CS-78-120, Carnegie-Mellon University, Schenley Park, Pittsburg, PA, May 1978.

[Holt70] A.W. Holt, F. Commoner, "Events and Conditions," in proceedings *Record Project MAC: Conference on Concurrent Systems and Parallel Computation,* Dec. 1970.

[Hopcroft69] J.E. Hopcroft, J.D. Ullman, *Formal Languages and Their Relation to Automata,* Addison-Wesley, 1969.

[Horowitz83] E. Horowitz, *Fundamentals of Programming Languages,* Computer Science Press, Rockville, MD, 1983.

[Johnson81] M.S. Johnson, "DISPEL: A Run-Time Debugging Language," *Computer Languages,* vol. 6, pp. 79-94, 1981.

[Kenah84] L.J. Kenah and S.F. Bate, *VAX/VMS Internals and Data Structures,* Digital Press, Digital Equipment Corp., Bedford, MA, 1984.

[Kernighan78] B.W. Kernighan and D.M. Richie, *The C Programming Language,* Prentice-Hall, 1978.

[Lamport78] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Communications of the ACM,* vol. 21, #7, July 1978.

[Lauer75] P. Lauer, R. H. Campbell, "Formal Semantics of a Class of Primitives for Coordinating Concurrent Processes," in *ACTA Informatica,* Springer-Verlag, 1975, pp. 297-332.

[Lausen79] S. Lausen, "Debugging Techniques," *Software Practice and Experience,* vol. 9, pp. 51-63, 1979.

[Lesser81] V. R. Lesser, D. D. Corkill, "Functionally Accurate, Cooperative, Distributed Systems," *IEEE Transactions on Systems, Man and Cybernetics,* vol. SMC-11, #1, pp. 81-95 Jan 1981.

[Lesser83] V.R. Lesser, D.D. Corkill, "The Distributed Vehicle Monitoring Testbed: A Tool for Investigating Distributed Problem Solving Networks," *AI Magazine,* vol. 4, #3, pp. 15-33, Fall 1983.

[Lu78] S.Y. LU and K.S. Fu, "Error-Correcting Tree Automata for Syntactic Pattern Recognition," *IEEE Transactions on Computers,* Vol. C-27, Nov. 1978.

[McDaniel77] G. McDaniel, "METRIC: a Kernel Instrumentation System for Distributed Environments," *Proceedings of Sixth ACM Symposium on Operating Systems Principles,* Nov. 1977, pp. 93-99.

[Model79] M. L. Model, *Monitoring System Behavior in a Complex Computational Environment,* CSL-79-1, XEROX Palo Alto Research Center, Palo Alto, CA, Jan. 1979.

[Needham82] R. M. Needham, A. J. Herbert, *The Cambridge Distributed Computing System,* Addison-Wesley, 1982.

[Poole73] P. C. Poole, "Debugging and Testing," in *Advanced Course on Software Engineering,* editor F. L. Bauer, Springer-Verlag, 1973, pp. 278-318.

[Riddle76] W.C. Riddle, *An Approach to Software System Modelling, Behavior Specification, and Analysis,* RSSM/25, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, July 1976.

[Schiffenbauer81] R. D. Schiffenbauer, *Interactive Debugging in a Distributed Computational Environment,* MIT/LCS/TR-264, Massachusetts Institute of Technology, Sept. 1981.

[Satterwaithe75] E. H. Satterwaithe, *Source Language Debugging Tools,* STAN-CS-75-494, Stanford University, Palo Alto, CA, May 1975.

[Snodgrass82] R. Snodgrass, "Monitoring Distributed Systems: A Relational Approach," Ph.D. Dissertation, Computer Science Dept., Carnegie-Mellon University, Dec. 1982.

[Schwartz71] Jacob T. Schwartz, "An Overview of Bugs," in *Debugging Techniques in Large Systems,* editor Randall Rustin, Prentice-Hall, 1971, pp. 1-16.

[SIGPLAN83] SIGPLAN/SIGSOFT Symposium on High-Level Debugging, Asilomar Conference Center, Pacific Grove, CA, March 1983.

[Smith81] Edward T. Smith, *Debugging Techniques for Communicating, Loosely-Coupled Processes,* TR 100, University of Rochester, Department of Computer Science, Rochester, NY, Dec 1981.

[SWAT82] *SWAT Debugger User's Manual,* Data General Corporation, 1982.

[Swinehart74] D. C. Swinehart, *CoPilot: A Multiple Process Approach to Interactive Programming Systems,* Stanford AI Laboratory Memo AIM-230, Stanford University, Palo Alto, CA, Jul 1974.

[Teitelman77] W. Teitelman, *A Display Oriented Programmer's Assistant,* CSL-77-3, XEROX Palo Alto Research Center, Palo Alto, CA, March 1977.

[Teitelman78] W. Teitelman, *Interlisp Reference Manual,* XEROX Palo Alto Research Center, Palo Alto, CA, Oct. 1978.

[Teitlebaum81] T. Teitlebaum, T. Reps, "The Cornell Program Synthesizer: A Syntax Directed Programming Environment," *Communications of the ACM,* vol. 24, #9, pp. 563-573, Sept 1981.

[VAXDEBUG82] VAX-11 Symbolic Debugger Reference Manual, Digital Equipment Corporation, 1982.

[VanTassel78] D. VanTassel, *Program Style, Design, Efficiency, Debugging and Testing,* Prentice-Hall, 1978.

[Weiser79] Mark Weiser, *Theoretical Foundations of Program Slices,* RSSM/69, University of Michigan, Jan. 1979.

[Weiser82] Mark Weiser, "Programmers Use Slices When Debugging," *Communications of the ACM,* vol. 25, #7, pp. 446-452, July 1982.

[Wileden78] J.C. Wileden, *Modelling Parallel Systems with Dynamic Structure,* COINS Technical Report 78-4, University of Massachusetts, Amherst, Jan 1978.

[Wileden83] J. C. Wileden, J. H. Sayler, W. E. Riddle, A. R. Segal, A. M. Stavely, "Behavior Specification in a Software Design System," *Journal of Systems and Software,* vol. 3, #2, 1983.

[Wirth73] N. Wirth, *Systematic Programming: An Introduction,* Prentice-Hall, 1973.

[Wittie80] L. Wittie, A. van Tilborg, "MICROS, A Distributed Operating System for MICRONET, A Reconfigurable Network Computer," *IEEE Transactions on Computers,* vol. C-29, #12, pp. 1133-1144, Dec. 1980.

[Woods70] W.A. Woods, "Transition Network Grammars for Natural Language Analysis," *Communications of the ACM,* Vol. 12, #10, pp. 591-606, Oct. 1970.

[Zobrist77] A.L. Zobrist and F.R. Carlson Jr., "Detection of Combined Occurrences," *Communications of the ACM*, Vol. 20, #1, Jan. 1977.

APPENDIX

BNF DESCRIPTION OF *EDL*

event_defs	::=	event description | event_defs event_description
event_description	::=	**event** event_heading is_clause cond_clause with_clause **end**
event_heading	::=	identifier | identifier (arglist)
arglist	::=	identifier | arglist , identifier
is_clause	::=	**is** event_expression
event_expression	::=	re_expr | **primitive** value
re_expr	::=	re_sexpr | re_expr | re_sexpr
re sexpr	::=	re term | re_sexpr ∘ re_term
re_term	::=	re_factor | re term △ re factor
re_factor	::=	constituent_event | re_factor repetition | (re_expr)
constituent_event	::=	identifier elist event_index
elist	::=	(expr list) | ϵ
event_index	::=	[number] | [identifier] | ϵ
repetition	::=	* | $^+$

with_clause	::=	**with** attribute_list | ϵ
attribute list	::=	attribute | attribute_list ; attribute
attribute	::=	attribute_name | attribute name := expression
attribute_name	::=	identifier
expression	::=	primary | − expression | ! expression | ~ expression | expression binop expression
primary	::=	value | qualified name | identifier | (expression) | identifier (expr_list)
expr list	::=	expression | expr list , expression
value	::=	number | string | boolean
boolean	::=	**true** | **false**
binop	::=	* | / | % | + | - | << | >> | < | <= | >= | > | == | != | & | | | && | ||
qualified name	::=	identifier event index . identifier
cond_clause	::=	**cond** boolean_exprlist | ϵ
boolean_exprlist	::=	expression | boolean_exprlist ; expression
string	::=	" characters "
characters	::=	character | characters character

character	::=	ASCII-character | \number
number	::=	bin_num | oct_num | decimal | hex_num
bin_num	::=	**0b** bin string
bin_string	::=	bin_digit | bin_digit bin_string
bin_digit	::=	**0** | **1**
oct_num	::=	**0o** oct_string
oct_string	::=	oct_digit | oct_digit oct_string
oct_digit	::=	**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7**
decimal	::=	dec_string | **0d** dec_string
dec_string	::=	dec_digit | dec_digit dec_string
dec_digit	::=	**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
hex_num	::=	**0o** hex_string
hex_string	::=	hex_digit | hex_digit hex_string
hex_digit	::=	**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **A** | **B** | **C** | **D** | **E** | **F**