Process Discovery and Validation through Event-Data Analysis

Jonathan E. Cook

Software Engineering Research Laboratory
Department of Computer Science
University of Colorado
Boulder, CO 80309 USA

jcook@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-817-96 November 1996

© 1996 Jonathan E. Cook

This work was supported in part by the National Science Foundation under grant CCR-93-02739 and the Air Force Material Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract Number F30602-94-C-0253. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

PROCESS DISCOVERY AND VALIDATION THROUGH EVENT-DATA ANALYSIS

by

Jonathan E. Cook

B.S., Case Western Reserve University, 1988

M.S., Case Western Reserve University, 1991

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

1996

ABSTRACT

Software process is how an organization goes about developing or maintaining a software system. It is the methodology employed when people use machines, tools, and artifacts to create a product. Recent work has applied formal modeling to software process, with the hope of reaping the benefits of unambiguous and analyzable formalisms. Yet industry has been slow to adopt formal model technologies. Two reasons are that it is costly to develop a formal model and, once developed, there are no methods to ensure that the model indeed reflects reality.

This thesis develops techniques for process event data analysis that help solve these two problems, which are termed process discovery and process validation.

For process discovery, event data captured from an on-going process is used to generate a formal model of process behavior. To do this, results from the field of grammar inference are applied, and a new method is also developed. The methods are shown to be efficient and practical to use in an interactive tool that is developed in the course of this work.

For process validation, event data is used to measure the correspondence between existing process models and the actual process, yet allowing discrepancies to exist. A paradigm based on string distance metrics is developed, and several validation metrics in this paradigm are described. How these metrics can be calculated is then shown, and a tool set for doing process validation is provided.

In implementing these methods, a framework is developed, called Balboa, for managing process data and facilitating the construction of analysis tools. This framework serves to unite the variety of collection mechanisms and tools by providing consistent data manipulation, management, and access services, and assistance in tool construction.

Finally, the techniques developed in this thesis are applied in an industrial study. This study provides concrete results showing that one can relate the quality of a process as prescribed by a model to the quality of the product. In doing so, it also shows that the discovery and validation techniques are able to capture important aspects about software process, and can be applied in the real world.

DEDICATION

I hereby dedicate this thesis to a brown trout in the South Platte somewhere in South Park, a rainbow trout in the Roaring Fork just above Glenwood Springs, a cutthroat trout in Spruce Lake in Rocky Mountain National Park, and a brook trout in Lefthand Reservoir, way up at treeline near the Indian Peaks. If you see any of them, tell them I said "Thanks".

ACKNOWLEDGEMENTS

I would like to thank my advisor, Alexander Wolf, for guiding me in this thesis, and for his direction in my academic and professional growth. Without his mentoring I would be much less than I am.

Thanks also goes to my family, and especially my wife Jeanine, who put up with the many odd things and stresses that go with the pursuing of a Ph.D. degree.

Many thanks goes to the SERL and other CS students, who made my stay at the University of Colorado a memorable one. Thanks especially to David Johnson (in the early years) and André van der Hoek (in the later years) for making sure I never did too much work in one day.

The work described in Chapter 6 was performed in part while I was a visiting researcher at Bell Laboratories, Lucent Technologies. I would like to thank Dr. Larry Votta for his guidance during the study, and the Autoplex organization of Lucent Technologies for giving us access to their projects and data. David Christiansen, Donna Lockwood, Michael Carper, Carol Ferguson, and many others were patient with our requests for information.

Last, but certainly not least, I would like to thank the taxpayers of the United States of America for supporting my studies through the National Science Foundation under grant CCR-93-02739 and the Advanced Research Projects Agency. I hope that I can make your investment worthwhile.

Contents

Chapter 1

Introduction

Software process research is a relatively young field; because of this, most of the work has focussed, rightly, on methods and paradigms for modeling and implementing processes. Only recently has there been serious efforts to apply and analyze processes and process models. Just as software construction is no longer an art, in that its engineering has matured to the point such that it is common practice to collect and analyze data on software products, so must software process follow the same course—for the research to be applied, it will be necessary to collect and analyze data on software processes.

This thesis is a step in that direction, specifically contributing two important analysis methods, providing an in-depth statistical study that validates the two methods and begins to show evidence that process affects product, and developing a framework in which future analysis methods will be developed.

1.1 Basic Terminology

A *software process* is how an organization goes about developing and/or maintaining a software system. It includes the people, machines, resources, tools, and artifacts involved in producing the software system. A *process description* is any sort of written description of the intended (or predicted) behavior of the process.

A *process model* is a process description that is in some formal language. By formal we mean that it is mathematically unambiguous, and we add the requirement that it be executable, in the sense that the model provides a rigorous specification of the dynamic behavior of the process.

A *process execution* is an instance of actually doing the process; each time a set of people, machines, and resources are working on a software system, a process execution takes place.

In a sense, the process is what the organization thinks they are doing, the formal model is a description of what the organization should be doing, and the execution is what they are really doing.

1.2 A Tour of Software Process Research

In order to set the stage for the work in this thesis, a background in software process research and application is presented here. We divide software process research into four main areas—modeling, execution, improvement, and measurement—and describe each in turn.

1.2.1 Modeling

Modeling research includes the formulation of languages and paradigms for formally describing software processes. Paradigms that have been used in formal process modeling include programming [115], graph-based control such as state machines [80] and Petri nets [11, 12, 40], rule-based [14, 96], and planning [68]. Huff [67] gives a good overview of modeling and languages.

Each paradigm has its own strengths and weaknesses; a rule-based model will be able to easily incorporate exceptions into its behavior, but will not be able to give a process engineer a good overall picture of the model's likely behavior, while a graph-based model will be exactly the opposite, allowing easy human understanding but only elegantly modeling the more rigid control flow structures in processes.

Process models have been mainly used for two things: process execution (enactment), described below, and formal analyses, such as deadlock detection, resource usage and conflict detection, simulation, and other traditional system analysis goals.

Modeling research can also include the definition of "canonical" process models, though our focus is on the formal modeling described above. Canonical models tend not to be very specific and only attempt to give a framework for more specific activities. Representative examples include the venerable waterfall model, prototyping, the risk-based spiral model, and the cleanroom process model [99]. Some canonical models involve only specific sub-processes such as analysis, design, and testing. Code inspections and design reviews are perhaps the most studied and formalized of sub-processes [49]. Other work in this vein includes SA/SD, Booch Object-Oriented Design, and regression testing [99].

1.2.2 Execution

Execution research is involved in creating process execution environments in which a formal, executable process model drives the activities that take place in the environment. Thus, this area is an application and continuation of the modeling efforts. This work in some sense is closely related to workflow, since both try to use a model of the work to drive the type and order of activities.

Work in this area includes environments based on process programming [116], rule-based systems such as Marvel [14], and other formalism-based environments, such as the Petri net-based SPADE [12]. Garg and Jazayeri [53] give a good overview of process-centered environments and their issues.

Many environments take the approach of providing the developer with an agenda of tasks to be done, thus giving the control of task execution to the developer. When a task is selected, executed, and completed, the process engine then advances the process model (or program), decides which new tasks are to be instantiated, and places these on the corresponding developers' agendas.

Additionally, when the model reaches a state where some set of steps can be done automatically, process execution environments "take over" and perform the necessary steps. Their hope is that this not only reduces the work that humans do, but also lessens the chance for human error in the process.

Research in workflow has shown the need to allow exceptions to the process [62], and it is here that many process execution environments are the weakest, in that they require strict adherence to the process model. This problem is even worse, because execution requires an organization to model their whole process up front, and introduces problems when the model evolves.

1.2.3 Improvement

Improvement efforts can be classified as industry based efforts to define specific quality measurements of a company's or group's software process and to define specific goals for improving their process. The most notable of these efforts is the SEI's Capability Maturity Model (CMM) [69, 94, 95], but other work, not necessarily specific to software, includes the ISO 9001 (9000-3) certification [74], and the Baldridge Quality Award. A European initiative centered around augmenting ISO 9001 is the *TickIt* process [117].

The CMM rates the goodness of a process by providing a checklist of activities that processes should include, and then provides a path of maturing the process by adding these activities. The ISO 9000 certification basically states that an organization has written down their process, and that they actually follow that process. It does not specify what steps a good process contains. The Baldridge award is more focussed on quality control methods.

Improvement efforts have been well-received throughout the industry, as evidenced by AT&T [9], IBM [72], and DEC [43] all having a special issue of their technical journals on process improvement. Many organizations, especially those that are government-related, have taken a keen interest in being certified at a certain rating under both the CMM and ISO 9003, although there is not yet much evidence that this leads to better products.

Most recently a new effort has been Humphrey's Personal Software Process (PSP) [70], which is directed at improving the software engineering process at the level of the individual. It is centered around getting a software engineer to track his/her own work, analyze their own defect data, and to give the individual an understanding as to how their personal process can affect their quality of work.

Unfortunately, there has been almost no overlap between these industry efforts and formal modeling and execution efforts. One reason for this is that the improvement efforts offer a stepwise approach, whereas one must invest heavily up-front to apply formal model-based technologies.

1.2.4 Measurement

Measurement efforts have begun but are still limited in nature. Many efforts use product measurements to infer some process qualities, such as defect production, change data, and other product measurements [16, 21, 35, 110]. Some efforts are trying to use measurements to help characterize the process; for example, Wolf and Rosenblum collected event-based data and used it to do some temporal and visualization analysis [121], and Bradac, Votta, and Perry use actual measurements to define the parameters to a queueing model of a process [23].

A body of work has looked at experimental measurement of specific process techniques. One example of this is the detailed study of code inspections by Votta, Porter, and Siy [98]. This work used an experimental setting to determine the actual effects of code inspections on the quality of the product. Another example shows how process exceptions might correlate to defects in the product [119].

A special issue of IEEE Software [73] had a theme of measurement-based process improvement; most of the articles dealt with questionnaire-based improvement, with two articles reporting on statistical and experimental improvement. None of the articles, however, were concerned with how formal models might apply.

1.3 The Future of Process Research

As can be seen from the background given, most work until now can easily be categorized into one or another of these divisions. Unfortunately, this separation has not been beneficial to the widespread application of process research. Industry, for example, has focussed heavily on the improvement efforts, while mostly ignoring the modeling and execution work. Measurement efforts have also not yet taken advantage of the formal modeling research.

What is needed now is work that combines several or all of these areas into comprehensive process analysis and improvement techniques. For example, tools that use both process data and process models to quantitatively and qualitatively understand a process, and research in applying process modeling technologies to industry improvement efforts would contribute to unifying these areas of process research. It is clear that in order to combine these areas, methods and tools that provide a feedback path from process execution (through data collection) to process modeling and analysis must be constructed.

From this overview of process research, then, it can be seen that the analysis of process data in conjunction with formal models is a useful addition to the ongoing process research, and will facilitate the migration of the ideas of research into the industrial domain.

1.4 This Thesis

This section details the specific problems that this thesis solves, its approach, solutions, and scope, and finally the contributions that this thesis makes to software process research in specific, and computer science in general.

1.4.1 The Problem of Using Formal Process Models

As has been shown, efforts have been put forth to rigorously model software processes in order to gain understanding of the process and to be able to analyze the model for desirable properties. But two problems have hampered the widespread application of these efforts:

1. Creating a process model, by first understanding the current process that is being used, is difficult, time-consuming, and expensive.

2. Ensuring that the ongoing process execution actually follows, to some degree, a desired process (embodied in a model) is troublesome at best and impossible at worst.

Currently, to create a model of one's process, the typical approach is to interview the people involved, spend some time observing the process, read the existing process documentation, if any, and finally translate all of this into some formal model. This method is expensive and time consuming; for example, one organization has spent over $1 million dollars on their process definition, which was not even formalized.[1]

This presents a high-cost barrier to those that want or need to use formal model technologies to improve or control their process. The irony is that the more a project exhibits problems, the more it can benefit from these technologies, but also the less its managers may be willing or able to invest resources in new methods and tools. Providing methods that support the engineer in understanding the current process and in building a process model can lower this barrier and further the spread of process technologies.

Regarding the second problem, if any measure of how close a process model and execution agrees is attempted today, it is without the support of tools. The most common way is to interview people or use questionnaires to determine if they are following the process; the problem with this is that the data is weak and only partially quantitative—it has already passed through one "interpretation" step, only representing what the people "remembered" or thought they did, which might be different than what actually happened. Without confidence in its accuracy, practitioners are hesitant to invest effort in defining the model or to make critical decisions based on the model. Giving process engineers quantitative assurance that the model does in fact reflect reality would further the application of formal models and their analyses in the software process realm.

1.4.2 Solutions

Our solution to these problems is to use data analysis techniques in conjunction with formal process models. Specifically, we target the analysis of event data, which describes the sequencing of activity in a process execution, to solve the above problems.

We state our thesis in two parts, reflecting the two problems above:

---

[1] Personal communication with an industrial process engineer.

1. Understanding what the process is that is currently happening is difficult. Event data from process executions can be used to discover models of the software process. This lowers the effort needed to create formal process models.

2. People cannot (and should not) follow a formal process model exactly. Methods can be devised to measure the correspondence between a formal process model and a process execution, as recorded in event data. These methods enable the realistic application of process modeling work.

We call these two *process discovery* and *process validation*, respectively. Process discovery is the act of devising a suitable model of a process execution or set of executions. Process validation is the act of determining the level of correspondence between an executing process and a process model.

This thesis shows that methods for process discovery and validation can be realized, and that they indeed solve real-world problems associated with software process. Additionally, we have constructed a general framework for process data analysis tools, called BALBOA, in which the discovery and validation methods reside.

1.4.3 Approaches

In process discovery, we cast the problem as one of grammar inference, and apply two methods previously developed in this area to that of process discovery, extending an implementation of one, and theoretically extending the other, while implementing it from scratch. We also invented a new method for process discovery (and more general, grammar inference) that inherently is able to handle noisy data, and is faster than the other techniques that were applied. While the grammar inference methods assume a sequential process, we also discuss how our results might relate to concurrent processes, and what extensions might be able to help us discover concurrency in a process execution.

For process validation, the paradigm that was devised was one of viewing the problem as a string distance measure. In this manner, we devised two metrics that capture the correspondence of the process execution with the process model in terms of functions based on string distance measures. While computing the distance between two strings, or process executions, is straightforward, computing the distance of a process execution from a process model is a very hard problem. In this thesis we have implemented an efficient heuristic method for calculating the distance, and show that its output is close to that of the global optimum distance calculation. This method does not assume a specific modeling formalism, but is extensible in that almost any modeling formalism can be plugged into it.

In building the discovery and validation tools, called DISCOVERY and VALIDATION respectively, we took a step back and devised a framework, called BALBOA, that provides for the easy construction of process data analysis tools, and that helps manage process data collections. This framework will allow for the rapid construction of future process analysis tools, and will enable the integration of multiple data collection methods in a single environment.

1.4.4 Scope of this Thesis

In using event data, the scope of this thesis is behavioral analysis, in the sense of looking at the sequencing of actions in the process. We do not look at real-time issues in the behavioral realm, or at the correctness of what artifacts or actors are associated with the actions. We see these as future analysis techniques, applicable after the behavior is validated.

We also are not concerned with the static aspects of the process, such as the consistency of the relationships between the artifacts, or the performance of roles by valid actors. Analyses such as these are closely related to areas of database research that have already been studied in depth.

1.4.5 Validating the Results

We undertook an extensive industrial study in order to validate our methods in this thesis. We show that these methods do in fact capture important aspects of software process, and that both process discovery and validation can work in a real-world setting. In addition, we showed that greater process deviations (as measured by the validation methods) positively correlated with the product defect metric, and that the validation methods can show specific areas of the process model that may be responsible for the defect production.

The validation methods were also applied to a model discovered from the data itself, and these results also showed that the discovery methods captured the important process patterns. This model, when used in the validation methods, was also able to statistically differentiate the defect-producing processes from the rest.

1.4.6 Contributions

In summary, the contributions of this thesis are as follows In particular, we:

- applied previously developed discovery techniques to the domain of software process, and extended these techniques;
- invented and implemented a new and promising technique for process discovery;
- devised a paradigm for process validation, and defined quantitative metrics in that paradigm;
- proposed and implemented an efficient method for computing the process validation metrics;
- verified through an extensive real-world study that both the discovery and validation techniques are applicable and usable in the real-world of software process; and
- proposed and implemented an extensible framework that manages process data and provides a uniform platform for creating process data analysis tools.

Additionally, the techniques developed in this thesis are applicable in a broader scope than just software process. Other areas that can characterize a system's execution with event data (sometimes called trace data) can potentially benefit from the process discovery techniques, and when the system's behavior must be verified against an abstract representation of the system, then the process validation methods can be applied.

For example, in reengineering a message-based system, the discovery techniques could help in understanding the message protocol between modules, and in verifying an implementation to a specification of behavior, the validation techniques could both measure how closely the implementation matches the specification and provide information as to which parts of the system do not.

1.5 Road Map

This thesis has the following organization:

- Chapter 2 describes and justifies the framework in which this thesis is constructed. The framework is one of using event data to capture the behavior of process executions, and then using techniques to discover models of the behavior from the event data, and comparing the event data to a model for validation. A motivating scenario for the use of process discovery and validation concludes this chapter.

- Chapter 3 presents our work in process discovery methods. Three methods, ranging from purely statistical (neural net) to purely algorithmic are presented and compared. While the neural net method was prohibitively slow and complicated, two of the methods prove to be practical to the problem of process discovery, and fast enough to be used in an interactive, visual tool. Performance analysis of the methods is also given.

- Chapter 4 presents the work in process validation. A paradigm for quantitative validation is first described. This paradigm views comparing process executions to process models as a string distance problem, and formulas for specific metrics are given in this paradigm. The problem of computing these metrics is then discussed, and a model-generic method and implementation is described for efficiently computing these metrics. Performance analysis of the methods is also given.

- Chapter 5 describes a system built as a framework for process data analysis tools, in which both the discovery and validation tools exist, but that is a general foundation for future process analysis tools to be built upon. The framework has a client-server architecture, where the analysis tools are clients that request data and meta-data from a server that manages the data collections.

- Chapter 6 presents an industrial process study that made use of both the discovery and validation tools. This study provides strong evidence that the methods devised in this thesis are applicable to real software engineering practices. This study also is among the first to show a distinct relationship between the process and the product, and to provide evidence that formal-model technologies can improve the software process.

- Chapter 7 concludes with some closing thoughts and a look at the applying our techniques in domains other than software process, and directions that future work can take from this thesis.

Chapter 2

Framework

The foundation on which this thesis rests is a view of processes as a sequence of actions performed by agents, either human or automaton, possibly working concurrently. With this, we are taking a decidedly behavioral view of processes, because we are interested in the dynamic activity displayed by the processes, rather than, say, the static roles and responsibilities of the agents or the static relationships among components of the products. This does not mean that other aspects of a process are not worthy of study; it is just that the issues we have chosen to investigate are those having to do with behavior rather than structure.

This chapter details the definitions of important terms and discusses the framework and type of data that this thesis' analyses are built upon.

2.1 Events

Following Wolf and Rosenblum [121], we use an event-based model of process actions, where an *event* is used to characterize the dynamic behavior of a process in terms of identifiable, instantaneous actions, such as invoking a development tool or deciding upon the next activity to be performed. 'Instantaneous' is relative to the time granularity that is needed or desired; thus, certain activities that are of short duration relative to the analyses that are being done can be represented as a single event.

An activity spanning some period of time is represented by the interval between two or more events. For example, a meeting could be represented by a "begin-meeting" event and "end-meeting" event pair. Similarly, a module compilation submitted to a batch queue could be represented by the three events "enter queue", "begin compilation", and "end compilation".

For purposes of maintaining information about an action, events are typed and can have attributes; one attribute is the time the event occurred. Generally, event attributes would be items such as user(s) and artifact(s) associated with the event, result values (e.g., pass/fail from a code inspection, or errors/no-errors from a compilation), and other values that give the detail about the specific occurrence of that event type.

The overlapping activities of a process, then, are represented by a sequence of events, which we

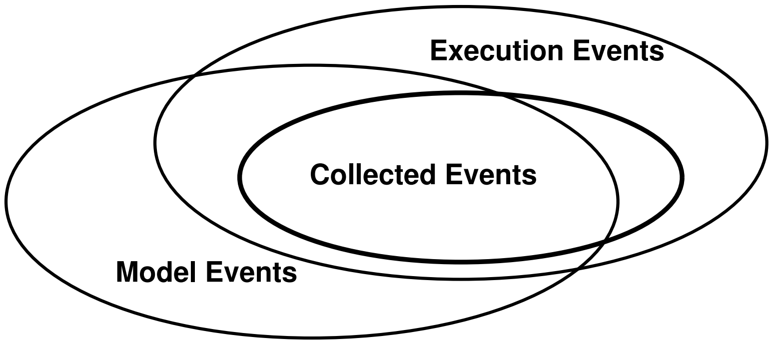

Figure 2.1: Venn Diagram of Event Types.

refer to as an *event stream*. For simplicity, we assume that a single event stream represents one execution of one process, although depending on the collection method and the organization, this assumption may have to be violated.

Using event data to characterize behavior is widely accepted in other areas of software engineering, such as program visualization [88], concurrent-system analysis [10], and distributed debugging [19, 38]. We feel it is applicable to software process as well.

2.2 Relating Models and Events: Event Sites

For our work in this thesis, we focus on behavior formalisms in process modeling languages. Examples are the Petri net foundation in Slang [12], state machines, and Statecharts [64]. In viewing a process execution as (part of) an event stream, we are assuming that a process model has some way of producing an event stream if it was "executed". If it cannot, then there is no way to relate a process execution to a process model using events, and analyses could not be constructed.

For this reason, we put a dependency on a modeling formalism: it must have or be able to have places in its behavior where a specific event can be produced. We call these places *event sites*. A state machine, for example, has state transitions as event sites, where a transition is labeled with the event it produces. A Petri net also naturally has its transitions as event sites—a firing sequence has a one-to-one mapping to an event stream. In reality, this dependency is not very constraining, in that, practically speaking, all modeling languages can be augmented with event sites.

Event sites are typed—that is, they do not produce any event, but only a specific type of event. For each event type, then, there is a set of event sites in a process model that produce it.

2.3 Event Domains

Given the distinction between the process model and the process execution, there are really two universes of event types. One universe is the set of event types associated with the model of a process, while the other is the set of event types associated with the execution of the process. Although one would expect a high degree of overlap between these two sets, in general they are not equivalent. Their relationship is depicted in Figure 2.1. For example, consider a project that executes a process and performs occasional code inspections as part of that process; a formal model of that process might not account for such an activity. Conversely, a model adapted from another project might include a subprocess for design reviews, but the current project may not do that.

A third set of event types is the set of those event types that are actually collected as data, also shown in Figure 2.1. Since this set must necessarily be a subset of the execution events (one cannot record something unless it is actually done), it can be viewed as a window into the actual process execution. This window may not show the whole execution, because there might be some activities for which no event data are collected. There are several reasons why this might occur, but two obvious reasons are that data about a particular event type might be considered inconsequential or the data might be considered too expensive to collect. For instance, events that occur off the computer are likely to be more expensive to collect than events that occur on the computer because they would require manual, as opposed to automated, collection techniques.

For process validation purposes, this means that only the part of the model that overlaps the collected events can be validated, while for process discovery it implies that only this same portion of a potential model can be inferred.

2.4 Event Data Collection

Techniques for process validation and discovery clearly depend on an ability to collect data about an executing process, but we do not address this topic in this thesis, because a variety of methods for collecting process execution data have already been devised:

- Basili and Weiss [16] describe a method for manual, forms-based collection of data for use in evaluating and comparing software development methods. Their methods would be a good place to begin for creating a system for manual event data collection.

- Selby [110] built a system, Amadeus, for automated collection and analysis of process metrics. It acts as an event monitor, allowing the triggering of actions based on certain events. Our use would simply collect the events into an event stream.

- Wolf and Rosenblum [121] use a hybrid of manual and automated collection methods to collect event data from a build process.

- Bradac, et al. [23], provided the user with a menu-based tool that collects sampled task and state information. This work could help ease the burden of collecting off-computer, manual events.

- Krishnamurthy and Rosenblum [84] built a system event monitor, Yeast, that can record events that occur on computer, and can react to those events. It also provides functionality for reporting any type of off-computer event as well.

- Barghouti and Krishnamurthy [15] describe a process enactment system that alternatively could be used to collect event data. The system is based on watching for events and matching the events and their contexts with the current state of the process. Lacking a process model, their infrastructure could be used to simply collect the event data.

Most process enactment systems such as Oz [20] and SPADE [12] could easily be instrumented to collect event data. Process integration components, such as ProcessWall [66], are likewise a natural focal point for such instrumentation.

Many software engineering environments, even those not process-based, can supply event data relatively easily; message-based systems, such as Field [100], Softbench [58], and ToolTalk [112], provide a natural framework from which to collect data about system-based activities.

Of the existing tools that developers already use, many naturally provide some event data collection as well. For example, all configuration management systems provide a history of the actions taken on each artifact under control, and many organizations have in-house tools, such as problem tracking systems, that track development or maintenance activity. These existing sources, when merged, can provide a comprehensive, though tool-based, event history of the process.

Any non-system-based data collection will have to have some sort of manual recording component, but some of the systems described above allow for some method of recording or announcing this information to the collector. For example, one can use the `announce` mechanism in Yeast to collect data about meeting and phone communication events.

2.5 Noisy Data

We do not address the issue of data integrity in this thesis. In this we assume that, prior to the application of our analyses, the data has been massaged, if necessary, so that it is "as clean as possible". This is not to say that there is no anomalous noise in the data, but that it has been eliminated as much as possible.

Eliminating noise is a general concern that begins with the collection methods and continues through possible consistency analyses on the data.

Although we assume that the data are mostly correct (i.e., the events that are collected have actually occurred) and consistent (e.g., all "begin" events for an activity have a corresponding "end" event), the methods developed in this thesis take into account the presence of noise in the data, and offer techniques for dealing with this.

2.6 The Big Picture

With this framework, then, the formulation of process discovery and process validation are as shown in Figure 2.2. They are centered around the use of event-based data collected from a process

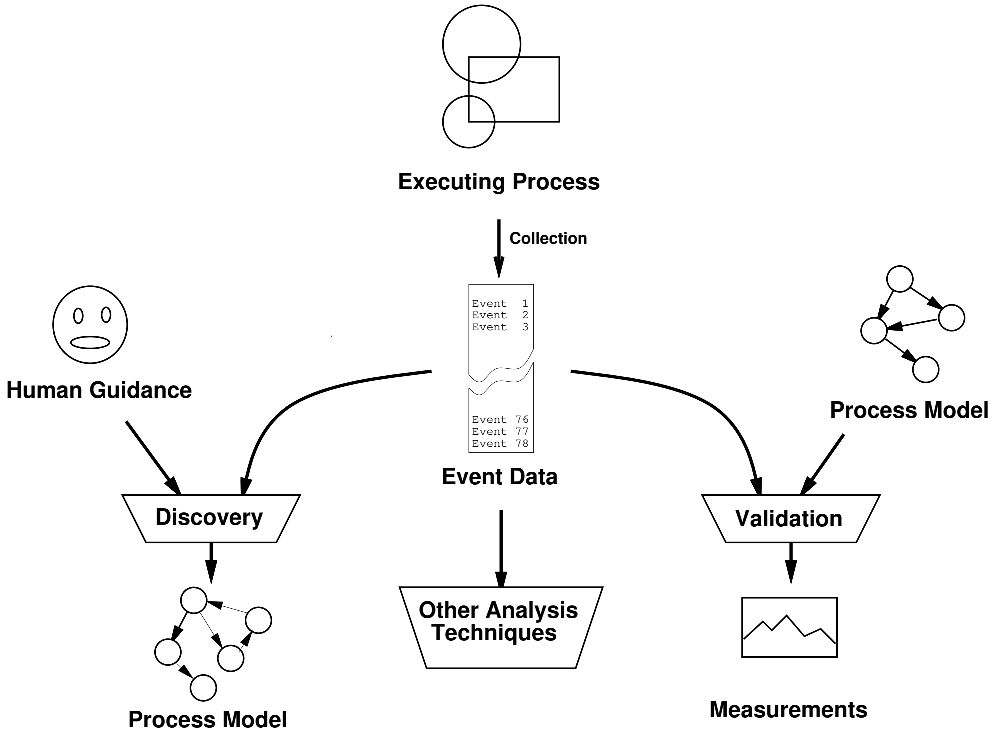

Figure 2.2: Process Validation and Discovery.

execution, with validation additionally using an existing process model, and discovery using human guidance (e.g., in the form of tuning or accuracy parameters to the algorithms, or model patterns to use).

This framework allows for other analysis methods to be devised for the event data as well, as shown in the figure.

2.7 Motivating Scenario

As a motivating example for how our methods might be applied, consider a development group building a large piece of software. The system is far enough along so that they do an integration build once a week, including all the additions and changes completed up to that point. The group feels that they are getting too many errors in the system at each integration build, and, looking at the errors, cannot point to something in the product (e.g., the specification or design) that is causing this. They decide that a more detailed look at their process is in order. Up to this point, they have relied on an informal textual document that described their process. The group decides to use Balboa to help them with this examination, and appoints a process engineer to coordinate and perform the examination.

The first step the process engineer makes is to instrument their site for data collection. The engineer uses Yeast [75] to watch a directory structure for file modifications, and output these as events. The engineer wraps the basic commands in shell scripts to first record the command as an event and then execute the command. The process engineer adds an email repository account for everyone to `cc:` to when communicating about the project, and adds a simple utility program to allow people to record phone or other communication events, and events such as meetings, inspections, and reviews.

With this infrastructure in place, the process engineer then collects data through a week's cycle of the process. At this point the process engineer will begin to use that data, while still collecting another week's worth of data. After that, the process engineer will continue collecting the automatically generated data, but will not ask the group to record other events. This way they know that the intrusiveness of the data collection will only last a short time.

With a little processing, the event data the process engineer has collected is put into a single event stream with a common format, having fields of event type, time, date, generating user, receiving user(s), and products (files) involved.

Since the process engineer knows that each developer works relatively independently, the first discovery experiments the engineer tries involve discovering individual processes. For this, the engineer extracts from the event stream just those events that are generated by a single person. The process engineer does this for each person.

For each of these extracted event streams, then, Discovery[1] is used to discover a model of that person's process. In doing this, it is found that some people are making last-minute changes during the regression testing of the piece they are working on, but do not re-start the regression testing. It is felt that this could be causing some of the integration problems.

As a second view to the process, the process engineer also extracts from the event stream just the events pertaining to a single product. The process engineer does this for several products that the process engineer feels are central and give a good experimental mix. Discovery is used here to see the process that a product goes through, including being checked out, modified, read, and checked in. In this analysis, it is seen that some products are being checked out in read-only while it is also checked out to be modified. The read version does not have the changes being made on the "locked" version. Although the group thought that the rules for using RCS specified a strict lock when modifying files, in reading their informal process specification they now see an ambiguity in it.

With these two problems identified, that of lax regression testing on components, and the non-strict use of RCS, the process engineer now goes about fixing the process. From the pieces that Discovery has created, the process engineer refines the discovered models into complete models that Validation[2] can use to validate their process. The process engineer also publishes a report

[1] Discovery is our process discovery tool.

[2] Validation is our process validation tool.

to the project group detailing their findings and recommendations. The group agrees that these changes should be enforced.

With the changes in the process being followed, the process engineer continues to collect the automatic data, since the extra off-computer events are not central to the problems the process engineer is trying to fix at the moment.

With another week's data, the process engineer uses VALIDATION to compare the new process execution with the model the process engineer now has of the desired process. The process engineer finds a good correspondence, and in a detailed look at the regression testing finds a much stricter testing process than what the the group did have. And indeed, in that integration build, and subsequent ones, the number of errors has decreased dramatically.

Chapter 3

Process Discovery

The issues of managing and improving the process of developing and maintaining software have come to the forefront of software engineering research. In response, new methods and tools for supporting various aspects of the software process have been devised. Many of the technologies, including process automation [14, 40, 96, 116], process analysis [61, 63, 80, 103], and process evolution [11, 76], assume the existence of some sort of formal model of a process in order for those technologies to be applied.

The need to develop a formal model as a prerequisite to using a new technology is a daunting prospect to the managers of large, on-going projects. The irony is that the more a project exhibits problems, the more it can benefit from the process technologies, but also the less its managers may be willing or able to invest resources in new methods and tools. Therefore, if we intend to help on-going projects by promoting the use of technologies based on formal models, we must seriously consider how to lower the entry barriers to those technologies.

In that vein, we have explored methods for automatically deriving a usable, formal model of a process from basic event data collected on the process. We term this form of data analysis *process discovery*, because inherent in every project is a process (whether known or unknown, whether good or bad, and whether stable or erratic) and for every process there is some model that can be devised to describe it. The challenge in process discovery is to use those data to describe the process in a form suitable for formal-model-based process technologies, and in a form that allows an engineer to understand the model, and thus the process.

In general, this is a very difficult challenge to meet. To scope the problem somewhat, we have concentrated our efforts on models of the behavioral aspects of a process, rather than, for example, on models of the relationships between artifacts produced by the project, or on models of the roles and responsibilities of the agents in the process. Any complete modeling activity would have to address those other aspects of the process as well.

In this chapter we present techniques that we invented and/or adapted for process discovery. They range in approach from the purely algorithmic to the purely statistical. The methods have been implemented as a set of tools operating on process data sets. While the methods are automated, they all require guidance from someone knowledgeable in software processes and at least somewhat familiar with the particular process under study. This guidance comes in the form of tuning parameters built into the methods, and from the selection and application of the event data. The results produced by the methods are initial models of a process that can be refined by a process engineer. Indeed, the initial models may lead to changes in data collection to uncover greater detail about particular aspects of the process. Thus, the discovery methods complement a process engineer's knowledge, providing empirical analysis in support of experience and intuition.

Our process discovery methods need not only be applied with the goal of creating a formal model of the process. The methods themselves can be used as process visualization tools, discovering and displaying the patterns of behavior in the process executions, with the goal of simply better understanding what is currently happening.

This chapter proceeds as follows. The next section (3.1) discusses related software process work. Section 3.2 gives a statement of the discovery problem and outlines our approach to solving it. Section 3.3 then describes in depth the work our approach builds on. The methods themselves are described and illustrated using a simple example in Section 3.4. Use of the methods on a more complex and significant example is presented in Section 3.5. Section 3.6 describes the implementation, and Section 3.7 evaluates the performance of the tools. Section 3.8 then talks about the problem of concurrency. Finally, we conclude in Section 3.9 with a summary of our results.

3.1 Related Process Work

In the software process domain, there has been little work that deals with techniques for discovering process models from current process execution behavior. Three pieces of work that are related are:

1. Garg and Bhansali [54] describe a method using explanation-based learning to discover aspects and fragments of the underlying process model from process history data and rules of operations and their effects. This work centers on using a rule base and goals to derive a generalized execution flow from a specific process history. By having enough rules, they showed that a complete and correct process fragment could be generated from execution data.

2. Garg et al. [57] employ process history analysis, mostly human-centered data validation and analysis, in the context of a meta-process for creating and validating domain specific process and software kits. This work is more along the lines of a process post-mortem, to analyze by discussion the changes that a process should undergo for the next cycle.

3. Madhavji et al. [47] describe a method for eliciting process models from currently executing processes. The method is basically a plan for how a person would enter a development organization, understand their process, and describe the process in a formal way. There is no notion of automation or tool support in any way.

Thus, the discovery problem has not yet been addressed in depth. The techniques we developed will be a beginning for assisting the process engineers in discovering, building, and evolving process models. Other techniques that look at different types of process data will ultimately be useful as

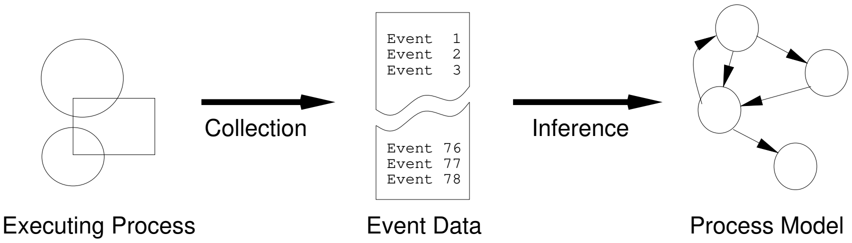

Figure 3.1: Process model inference framework.

well. For example, the area of data mining (e.g., [1]) may have useful techniques for discovering software product relationships.

3.2 Problem Statement and Approach

Our goals in this work are to take a trace of the process execution, in the form of an event stream, and deduce (partial) formal models of the behavior of the process. This framework is shown in Figure 3.1. We do not envision being able to deduce fully complete and correct process models; this is something akin to automatic programming, which has never been successful. Rather, we aim to provide a process engineer with some formal description of the patterns of behavior that exist in the currently executing process, as seen through the collected execution event stream. An engineer can then use this information as a basis for creating a complete process model, for evolving an existing model, or for correcting some aspects of the execution.

As mentioned in Chapter 2, we assume that event data are being collected on the executing process. Once collected, the data can be viewed as a window onto the execution. In general, this window will not show the whole execution, since there will be some activities for which no event data are collected at all. Hence, just as for any other data analysis technique, the results obtained by discovery methods strongly depend upon the content and quality of the data that are collected.

So, what we seek from the data are recurring patterns of behavior. For our purposes, finite state machines (FSMs) provide a good starting point for expressing those patterns. While FSMs may be somewhat deficient for prescribing software processes, they are quite convenient and sufficiently powerful for describing historical patterns of actual behavior.

In this thesis we do not present a formal description of finite state machines. A good reference for this background is Carrol and Long [29]. We use the term FSM to refer to nondeterministic, transition-labeled state machines.

Why not choose a more expressive representation than finite state machines, such as push-down automata? One reason is that the more powerful the representation, the more complex the discovery problem, and it is not clear that we need that power. In particular, we are aiming to infer only the structure of the behavior and not the values controlling that structure. Thus, while our methods could not discover a pattern such as $\mathtt{A}^n\mathtt{B}^n$ (the canonical example that demonstrates the superior power of push-down automata), they would certainly discover and describe the sequenced looping structure of an `A` loop followed by a `B` loop. Additionally, there are clear migration paths from FSMs to other modeling paradigms, whereas something like push-down automata are more specialized.

The drawback to FSMs is that they have no inherent ability to model concurrency; however, we address this issue in Section 3.8.

To develop our initial set of methods, we have cast the process discovery problem in terms of another, previously investigated FSM discovery problem. That problem is the discovery of a grammar for a regular language given example sentences in that language [7]. This area of research historically uses the term grammar inference for this problem. If one interprets events as tokens and event streams as sentences in the language of the process, then a natural fit between these areas is seen. There are, however, several important differences, as we discuss in the next section.

Techniques for state machine inference have been applied in many diverse areas, from DNA/RNA and protein understanding to speech/text recognition, and other scientific sequence data. Thus, there is historical reason to believe that these techniques will be applicable to software process discovery.

3.3 Grammar Inference

In this section we present a formal definition of the problem of grammar inference, its theoretical complexities, and a detailed look at work in this field that is applicable to the process discovery problem.

3.3.1 A Formal Definition

The grammar inference problem is usually informally stated as, given some sample sentences in a language, and perhaps some sentences specifically not in the language, infer a good grammar that represents the desired language. But what is a good grammar? Occam's razor would suggest that the smallest grammar that includes all sentences in the language and excludes all sentences not in the language would be the best, by the measure of complexity. Our goal of discovering and presenting the patterns in the event stream to an engineer would also lead us to say that the smallest grammar, in the above sense, would be the easiest for an engineer to work with. We will see later that this definition of good is, however, problematic.

Formally, we define this problem with the following terms.[1] Let Σ be the alphabet of tokens, and Σ^* be all possible sequences (sentences) over Σ. Then, we define a language L as $L \subset \Sigma^*$, that is, L is a subset of sentences from Σ^*.[2] A grammar G is a formalism of some class (e.g., regular, context-free) that describes L; L is said to be in that class of languages.

[1]This presentation is consistent with others in the field [7, 97, 105].

[2]$L = \Sigma^*$ is not a very interesting language.

We define Θ_L as the infinite set of all pairs $\langle s, l \rangle$ from $\Sigma^* \times \{0, 1\}$, where $l = 1$ if $s \in L$ and $l = 0$ otherwise. Thus, Θ_L is just Σ^* tagged with whether each sequence belongs to L or not. We call the sequence s in $\langle s, l \rangle$ positive if $l = 1$ (i.e., it is in L) and negative otherwise. Π_L is defined as the possibly infinite set of all positive sequences, and Γ_L is defined as the possibly infinite set of all negative sequences.

For learning purposes, an algorithm needs some presentation of data. A complete presentation is just a presentation of Θ_L. A positive presentation is just a presentation of Π_L. A negative presentation is just a presentation of Γ_L. Note that any of these sets are or can be infinite, so that an algorithm cannot use the whole presentation; yet, we cannot beforehand eliminate any single sequence from the presentation.[3] A partial presentation is a finite subset of a complete, positive, or negative presentation.

Now, the grammar inference problem can be phrased as, given some presentation of all or part of Θ_L, can one infer a good grammar G describing L?

3.3.2 Complexity Results

Gold's "identification in the limit" was the first framework for analyzing the grammar inference problem [59]. Identifying in the limit frames the problem in terms of looking at the complete presentation of Θ_L. If, after some finite presentation of pairs from Θ_L, the algorithm guesses G correctly, and does not change its guess as the presentation continues, then the algorithm is said to identify G in the limit. Gold showed [60] that finding a DFA with a minimum number of states with a presentation of Θ_L is NP-hard, and also that it is impossible with just a positive presentation (Π_L). The problem with only positive examples is that the algorithm has no way to determine when it is overgeneralizing.

Thus, this early result shows that learning the best grammar of L, even for relatively simple classes (regular languages), is very difficult, and that learning from only positive examples is strictly weaker than learning from both positive and negative examples.

The PAC model for learning concepts from examples was proposed by Valiant [118]. PAC stands for "Probably Approximately Correct", which may not sound rigorous but is: probably is defined as within some probability $1 - \delta$, and approximately is defined as within some $1 - \epsilon$ of the correct solution. If a polynomial-time algorithm can be constructed for some learning problem that, within some likelihood (δ) produces a solution that is close (within ϵ) to the correct one, then that domain is PAC-learnable. Classes of boolean formulas and decision trees, and some geometric and algebraic concepts have been shown to be PAC-learnable. Results for grammars, including DFAs, are more negative, but less definite. As Pitt [97] reports,

> If DFAs are polynomially approximately predictable, then there is a probabilistic polynomial time algorithm for inverting the RSA encryption function, for factoring Blum integers, and for deciding quadratic residues.

[3] Actually, since some of the sequences can be infinite themselves, any theoretical analysis assumes only a presentation of finite sequences.

Angluin [6] phrases the learning problem in terms of an oracle. An algorithm can make a fixed set of queries to the oracle, the basic two being "Is this string accepted by the correct grammar?" and "Is this grammar equivalent to the correct grammar?". This active model of learning is different from the passive presentations above; the algorithm has a certain amount of control over the information it receives and uses, rather than just being given a set of data. Specifically, the algorithm can ask questions to elucidate conflicts in the grammar. With this framework, Angluin gave a polynomial-time algorithm for learning DFAs, and others have extended this to other classes of languages.

There is much work describing the computational complexities of various learning paradigms and classes of languages or formulas. The survey by Angluin and Smith [7] is a broad look at inductive inference learning. Pitt's survey [97] is a very good starting point for understanding the theoretical complexities involved in learning regular grammars. Further theoretical work on language learning is found in [4, 77, 86, 87, 104].

Angluin and Smith, in their survey, acknowledge the gap between the theoretical results that have been achieved and the practical application of inference methods. In their conclusion they state:

> The most significant open problem in the field is perhaps not any specific technical question, but the gap between abstract and concrete results. It would be unfortunate if the abstract results proliferated fruitlessly, while the concrete results produced little or nothing of significance beyond their very narrow domains.

Since their survey, however, research efforts have continued to produce significant and practical results, as we discuss in the next section.

3.3.3 Practical Inference Techniques

In this section we survey previous work that has devised or used practical algorithms for grammar inference in the analysis of real-world problem domains.

One class of techniques represents a common paradigm in learning from positive examples. The basic idea behind this class is to build a prefix tree machine from the input data, where each sentence has a linear state-transition sequence that produces it. Each linear sequence starts at the same start state, and shares states until that sequence diverges from every other example sentence. Thus, the machine is a tree, the root being the start state, and each state that has more than one out transition being a point where example sentences no longer share a prefix.

For a set of data that covers a language (has examples of every behavior), any grammar for the language can be shown to be contained in the prefix tree machine.

This machine is then merged according to some rules until a final state machine is output as the learned language. Examples are:

- Ahonen et al. [3] describe a prefix tree method where states are merged based on previous contexts. That is, if two states are entered from the same k-length contexts, then they are merged.

- Angluin [5] merges states of a prefix tree based on a notion of k-reversibility, which restricts the class of languages her algorithm infers. While 0-reversible languages can be inferred in near-linear time, higher k values cause the algorithm to be cubic in the length of the input.

- Bierman and Feldman [22] describe a prefix tree method where states are merged based on k-length future behaviors. Although their method is not directly described as prefix tree merging, it does fall in this class.

- Carrasco and Oncina [28] describe a statistical method for learning stochastic regular languages, based on state merging from a prefix tree, where states are merged based on statistical likelihood calculations. Running time is maximally n^3; however they claim actual times are near-linear.

- Castellanos et al. [31] apply a method of this class to learning natural language translators.

Another class of techniques is directed more towards iteratively building up a regular expression from the sequences, and then translating that into an FSM (if necessary). In this sense, these algorithms make a pass through a sequence, find repetitive patterns, replace all occurrences with the regular expression, then start all over until it is decided that they are done. Two examples are:

- The work of Brāzma et al. [25, 24, 26] is most representative of this work. They have been able to construct fast algorithms that learn restricted regular expressions. The restrictions they place on the inferred regular expressions are that both the *or* operator and the **-nesting* are limited and fixed. Although the algorithm is in worst case cubic, in practice they report near-linear times.

  They have extended this work to handle noisy strings by treating the noise as an edit distance problem, although currently they only allow for a single-token edit, not multi-token gaps. Their work is geared towards analyzing biosequence (DNA/RNA, protein) data.

- Miclet [91, Section 3.3.3] describes a method he calls the uv^kw algorithm, where each pass looks for repetitions of substrings (the v^k), chooses the best candidate, and replaces with a **-expression*; then the next pass is started on the reduced string. This method is well-adapted for loops, but has problems with the *or* operator as well.

Miclet's survey [91] describes several other techniques for inferring (N)DFAs from positive samples, and is, in general, a good reference for practical inference techniques.

3.3.4 Statistical Model Inference Techniques

Statistical models are models of languages that incorporate some notion of the likelihood of a sequence or subsequence occurring in the language. Discovering a statistical model can provide the user with more feedback about the relative probability of the discovered sequence patterns happening. For example, discovering that a loop in the process is 80% likely to iterate rather than exit would be important for the user to see. While the methods we explore in this thesis do not initially support statistical models, we present this background here and later discuss how the methods can be extended in this direction.

Hidden Markov Models (HMMs) are the base representation for statistical grammars. HMMs are finite state machines with token-labeled transitions, where each transition has an associated probability of occurring from its source state. Additionally, HMMs are usually defined as allowing λ-transitions, which are transitions from one state to another that do not produce a token.

The problem of inferring HMMs is usually, in the literature, meant to mean finding the probabilities on some given model's structure. The most widely used algorithm for this is the forward-backward algorithm, also known as the Baum-Welch formula; this algorithm, being a gradient descent method, converges on a local maxima, but is somewhat slow; the Viterbi algorithm is a faster approximation to this.

Some reference to inferring the structure of the HMM is also made. Usually the problem of inferring the structure is cast as a subproblem of inferring the probabilities of the transitions, in this manner: start with a fully connected model, infer the probabilities based on the input data, and then remove all transitions below some threshold level of probability. In practice, this approach has been too time-consuming to be practical, and also requires an assumption on the number of states in the model [91].

The next level above HMMs is Stochastic Context-Free Grammars (SCFG). An SCFG is just a CFG where each production rule has some associated probability of firing. As with HMMs, inferring an SCFG usually means inferring the probabilities on the given production rules. The inside-outside algorithm is the most widely used algorithm for doing this.

The algorithms mentioned above for parameter estimation on both the HMMs and SCFGs are types of expectation maximization algorithms. All of them are gradient descent techniques that find a local maximum. Hence, initializations are important for them. A good reference on these algorithms is Charniak's book [32].

To solve the structure inference problem for statistical models, the standard algorithmic techniques for inferring an FSM or CFG can be applied to finding a structure, and then the well-known algorithms can be used to assign the correct probabilities to the transitions or production rules. This has been done in several pieces of work.

- Stolke [111] using a prefix tree method of FSM inference to provide an input HMM to a parameter estimation routine. His results show a good improvement over parameter estimation beginning with a fully connected model.

- Sánchez and Benedí [106] use a specific inference mechanism (that only finds loop-free automata) to provide a model structure and initial transition probabilities to a forward-backward parameter estimation algorithm.

- Chen [33] describes a corpus-based SCFG inference method using bayesian probabilities and compares it to ngram models.

- Casacuberta [30] provides a transformation algorithm that takes a CFG and translates it to a CFG in Chomsky Normal Form, which is required for parameter estimation using the inside-outside algorithm.

3.3.5 Inference Using Neural Nets

The neural network community has also looked at the problem of grammar inference from positive samples. Recent representative work in this field is [39, 124]. The various methods all consist of defining a recurrent network architecture, and then analyzing the hidden neuron activity to discover the states and transitions for the resulting grammar. The difference between these methods is how they inspect the hidden neurons to infer state information. The most advanced techniques apply clustering to the neuron activations during training, to help induce better state-machine behavior.

These techniques are somewhat controversial, because many neural network researchers believe one should not look into the net to try to learn anything, but should only be concerned with the input/output performance of the net. Nevertheless, there is a thread of research that is applicable to our investigations.

3.3.6 Limitations of Grammar Inference

The grammar inference methods presented here look only at the tokens that make up the language. For our purposes, this means that they are limited to operating on event types (being mapped to tokens), and ignore the potential information in event attributes. Thus, causal relationships between event attributes cannot be deduced using grammar inference techniques.

Another limitation is that the methods generally do not support seeding the algorithm with pre-known portions of a model (or grammar). Often a researcher will know something about the process under study, and may be able to easily formulate some portions of a desired model. It would be useful to have techniques that could take that and build upon it by discovering the patterns surrounding the given portion.

A final restriction is that the methods assume a single state machine. In the typical process, there are generally concurrent activities going on, which produce an event stream that may have non-deterministic orderings of events. In fact, an event stream could have interleaved executions of the same process model, and separations of these executions might only be done by relating the event attributes.

Despite these restrictions, we feel that these methods are a strong starting point for providing techniques to assist the process engineer in developing his/her process model. Also, we discuss the problem of discovering concurrent behavior in Section 3.8 of this chapter.

3.4 Methods for Process Discovery

In this section we describe three inference methods that we have adapted or invented for process discovery. The methods represent three points on the spectrum from algorithmic to statistical techniques. Since we are dealing with data collected from process executions, we have only positive

`ABCABCBACBACABCBACBACABCBACABCABCBACBA`

Figure 3.2: Simple Event Stream Example.

samples with which to work. The methods can be characterized by the method in which they examine those samples.

1. The RNet method is a purely statistical (neural network) approach that looks at the *past* behavior to characterize a state. For this method we have extended an implementation by Das [39]. Our extensions allow this method to handle more event types (theirs was restricted to two), and enable the easier extraction of the discovered model from the net.

2. The KTail method is a purely algorithmic approach that looks at the *future* behavior to compute a possible current state. We modified the theoretical description given in [22] to make it less dependent on multiple sequences, and extended it to allow for handling noisy data. In addition, we added some post-analysis steps that remove some common overly complex constructs that the basic algorithm tends to leave in a discovered model. Our implementation was from scratch.

3. Finally, the Markov method is a hybrid statistical and algorithmic approach that looks at the *neighboring* past and future behavior to define a state. This method is a new method that was invented and implemented solely by us.

Since we only have positive samples of the target language, the definition of what is a good discovered model as given at the beginning of Section 3.3.1 cannot be applied, since we would need negative samples to determine the bounds of the language. Thus, for our purposes, a good result from a discovery method is one that distinctly identifies patterns made up of sequencing, selection, and iterations (loops); one that does not needlessly complicate the patterns identified, and one that sensibly shares states when applicable.

In some sense, we are saying that "goodness is in the eye of the beholder", which in our case is the engineer using the methods. To this end, we have purposely pursued methods that are naturally parameterized for the amount of complexity they allow to be inferred.

To explain and illustrate the three methods in this section, we use the simple event stream of Figure 3.2. The stream is a sentence in a three-token language and is used as sample input for each of the methods. All three methods typically can work on multiple samples to refine their results, but in this simple example we just give them the one stream shown. Of course, there are many different FSMs that could generate this sentence. What is important is that the methods produce an FSM model that reflects the structure inherent in the sample—that is, the `A-B-C` and `B-A-C` loops.

In this chapter, all of the graphical representations presented have been automatically generated by the *dot* directed-graph drawing tool [83] from the output of the discovery tools.

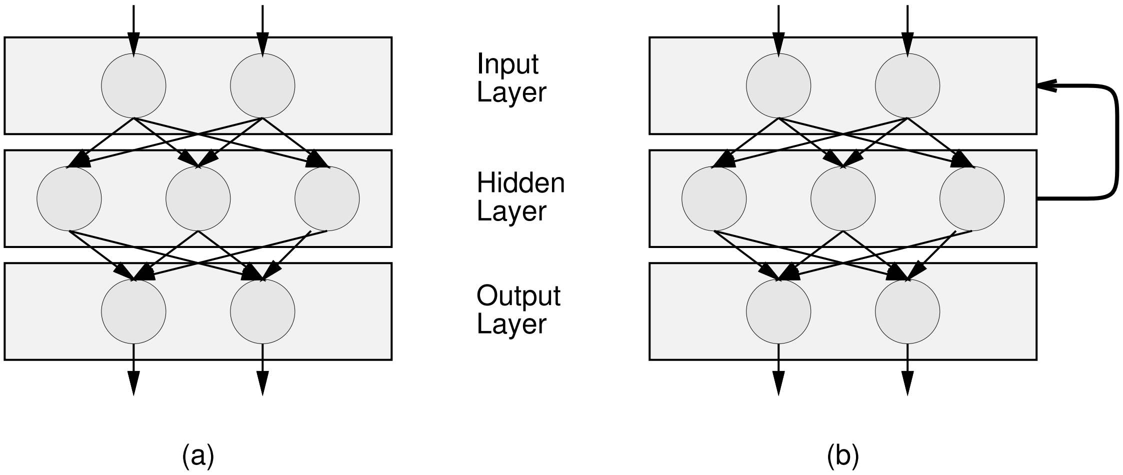

Figure 3.3: Standard (a) and Recurrent (b) Neural Network Architectures.

In Section 3.5, we demonstrate the methods on a more complex example, exposing the relative performance and possible strengths and weaknesses of each method.

3.4.1 RNet

The first method we describe comes from the neural network community and is a purely statistical approach to inference. This recently developed method is due to Das and Mozer [39]. We refer to it here as the RNet method.

In a standard feed-forward neural network, neurons are split into layers, with all the outputs of the neurons in one layer feeding forward into all the neurons of the next layer (see Figure 3.3a). Typically, there is a layer of input neurons, at least one layer of internal or hidden neurons, and a layer of output neurons. For a given input to the input neurons, activation flows forward until the output neurons are activated in some pattern. The network is trained by propagating the difference between actual and desired outputs backwards through the network. We refer to this difference as the learning error.

To this basic architecture, Das and Mozer have added a recurrent connection, which feeds the results of the hidden layer directly back to the input layer (see Figure 3.3b). This lets the network model an FSM by providing the "current" state—in the form of the activation pattern of the hidden layer—as an input to the network. Now the output activity (i.e., the "next" state) is determined, not just by the input, but also by the current hidden neuron activity. This recurrent network is the inference mechanism of RNet.

Training takes the form of presenting a window of a specified length to the network, and having it attempt to predict the next token. Learning error is only back-propagated through the network after the whole window is presented. By sliding the window forward over the sample stream one token at a time, each position in the stream is used in training. RNet therefore takes a historical view of the sample stream, since the window focuses attention on events that precede the current event.

Once the network is trained, the FSM representation is extracted from the network. This is done by presenting the same or different strings to the network and observing the activity of its hidden neurons. Activation patterns that are closely related are clustered into the same state. Transitions are recorded by noting the current activation pattern, the input token, and the next activation pattern. This information, collected over all input patterns, then represents the FSM that the network has inferred.

The parameters to RNet are numerous and varied, and the realistic application of RNet requires some knowledge of how neural nets work and what can be expected from them. The parameters are:

- the window size that is presented; this controls the size of the locality that is used in learning,
- the number of hidden neurons in the net; this controls how detailed of information (and thus an FSM) the net can learn,
- the weight of the recurrent connection with respect to the forward connections; this controls the importance of the history of the net,
- the learning rate and momentum; these control how fast the algorithm learns and how important the previous direction in learning is.
- the weight of the clustering method; this controls how strongly the net tries to cluster similar states into one, and
- the number of training iterations, or the error threshold to stop at.

It is beyond the scope of this thesis to evaluate the full dimensions of a neural net and provide meaningful assistance. Many neural net practitioners still refer to it as an "art". Our experience is that as one uses RNet, its behavior with respect to parameters can be learned to some extent.

For our experiments we started with an implementation by Das. In their use, they restricted themselves to two-token languages, and thus the implementation was quite limited. We extended the implementation to allow for many token types, and also enhanced the output of the net to easily extract the more complex models that result from having more token types.

Figure 3.4 shows the inferred FSM that RNet produces from the example stream of Figure 3.2. The method successfully produces a deterministic FSM that incorporates both the `A-B-C` and `B-A-C` loops. But it also models behavior that is not present in the stream, such as an `A-A-C` loop and a `B` loop. This shows the inexactness of the neural network approach; even with a known perfect sample input, one cannot direct it to produce a machine just for that stream.

An advantage of the RNet method is that, since it is statistical in nature, it has the potential of being robust with respect to input stream noise (e.g., collection errors or abnormal process events).

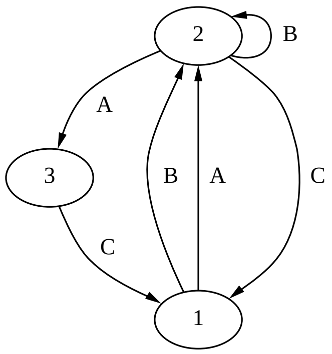

Figure 3.4: FSM Inferred by the RNet Method for the Example Event Stream of Figure 3.2.

Its disadvantages are that it is in general very slow because of the training time required, and the size of the net grows rapidly with the number of token types and the expected size of the resulting model. The plethora of tuning parameters and the lack of guidelines in setting them are a drawback as well.

3.4.2 Ktail

The next method is purely algorithmic and based on work by Biermann and Feldman [22]. Their original presentation was formulated in terms of sample strings and output values for the FSM at the end of the strings. Our formulation of this algorithm does not make use of the output values and is thus presented as just operating on the sample strings themselves. In addition, our method is less dependent on having multiple strings, we have enhanced their algorithm to reduce the number of states in the resulting FSM, and we have introduced a threshold parameter for dealing with noisy data.

The notion that is central to Ktail is that a state is defined by what future behaviors can occur from it. Thus, for a given history (i.e., token string prefix), the current state reached by that history is determined by the possible futures that can occur from it. Two or more strings can share a common prefix and then diverge from each other, thus giving a single history multiple futures. The "future" is defined as the next k tokens, where k is a parameter to the algorithm. If two different histories have the same future behaviors, then they reside in the same equivalence class; equivalence classes represent states in the FSM.

This equivalence class is called the *Nerode* relation for the prefixes in the class, and the extension that Ktail uses is to limit the Nerode relation to a fixed k-length future. Additionally, the original definition by Biermann and Feldman only looked at the k-length behavior at the end of the string (presumably because they were interested in a large set of short strings) and the output function of the state machine. We define the k-length behavior at any point in an input string to be independent of their requirements.

Formally, KTAIL is defined as follows. Let S be the set of sample strings and let A be the alphabet of tokens that make up the strings in S. Let P be the set of all prefixes in S, including the full strings in S. Then $p \in P$ is a valid prefix for some subset of the strings in S. Let $p \cdot t$ be the prefix p appended with the token string t. We call t a tail. Finally, let T_k be the set of all strings composed from A of length k or less. An equivalence class E is a set of prefixes such that

$$\forall (p, p') \in E,\ \forall t \in T_k,\ \ p \cdot t \in P \longleftrightarrow p' \cdot t \in P$$

This means that all prefixes in E have the same set of tails of length k or less.[4] Thus, all prefixes in P can be assigned to some equivalence class. It is these equivalence classes that are mapped to states in the resulting FSM.

Transitions among states are defined as follows. For a given state (i.e., equivalence class) E_i and a token $a \in A$, the destination states of the transitions are the set D of equivalence classes

$$D = \bigcup \mathcal{E}[p \cdot a],\ \ \forall p \in E_i$$

where $\mathcal{E}[p \cdot a]$ is the equivalence class of the prefix $p \cdot a$. Intuitively, this says that to define the transitions from E_i, take all $p \in E_i$, append a to them, and calculate the equivalence classes of these new prefixes, which are the destination states of token a from state E_i. If $|D| = 0$, then this transition does not exist; if $|D| = 1$, then this transition is deterministic; and if $|D| > 1$, then this transition is nondeterministic. The transitions, if any, are annotated with the token a in the final FSM.

Our Enhancements to the Basic Algorithm

This is where the algorithm, as Biermann and Feldman define it, stops. While it produces an FSM that is complete and correct, the algorithm has certain tendencies to produce an overly complicated FSM. Figure 3.5 shows the FSM that is produced by the algorithm, with $k = 2$, when given the sample data of Figure 3.2. The complexity is that a loop in the data will be unrolled to a length of k. The unrolling arises because the algorithm sees a change in the tail of the loop body at the exit point of the loop. This change causes the last iteration of the loop to be placed in a separate equivalence class. Consider the `A-B-C` loop. In the figure, this loop is represented by the path $< 1, 2, 3, 1 >$ in the FSM. But the last iteration through the loop has a separate representation as the path $< 1, 2, 4, 5 >$. A similar effect occurs with the `B-A-C` loop.

We can improve the basic algorithm by automatically merging states in the following manner. If a state S_1 has transitions to states $S_2 \ldots S_n$ for a token t, and if the set of output transition tokens for the states $S_2 \ldots S_n$ are equivalent, then we merge states $S_2 \ldots S_n$. Intuitively, this procedure assumes that if two (or more) transition paths from a state are the same for a length of more than one transition, then the internal states of those paths should be assumed to be the same and thus

[4] Tails of length less than k, down to zero, occur at the ends of strings.

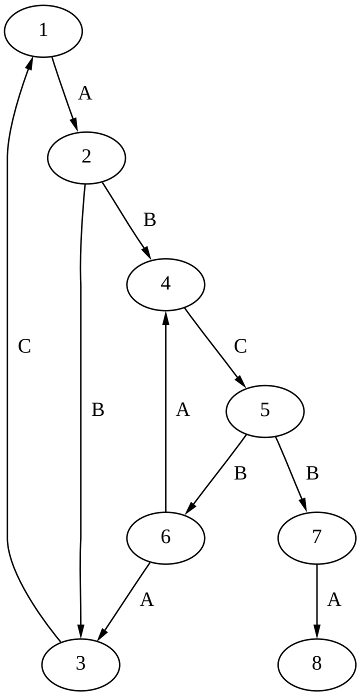

Figure 3.5: FSM Inferred by the Basic Biermann-Feldman Algorithm ($k =$ 2) for the Example Event Stream of Figure 3.2.

merged. In Figure 3.5, states 6 and 7 can be merged, states 3 and 4 can be merged, and 8 can be merged with the merged states 3/4, once 6 and 7 are merged.

This improvement, implemented in KTAIL, results in a much cleaner FSM than that which results from the basic Biermann-Feldman algorithm. The FSM produced by KTAIL from the example stream of Figure 3.2 is shown in Figure 3.6. Clearly, KTAIL does a good job of discovering the underlying loops in the behavior. The `A-B-C` loop is completely embodied in the transition path $< 1, 2, 3/4/8, 1 >$, while the `B-A-C` loop is completely embodied in the path $< 5, 6/7, 3/4/8, 5 >$. Note that there is a point of nondeterminism in state 3 upon the occurrence of a `C`. This is because the algorithm sees two different behaviors (i.e., tails) from the `C` event in the sample data.

We can add some ability of ignoring noise in the data to the KTAIL method as described. Assuming that noise is low-frequency event sequences, then an equivalence class with very few members may be interpreted as a place where the k-length futures are erroneous. Thus, we could decide that this equivalence class does not represent proper behavior and throw it away, not allowing it to appear as a state in the final FSM. This formulation suggests setting a threshold on the size of the equivalence class (in terms of the number of members), and then pruning all classes less than that size. Pruned equivalence classes will not show up in the resulting state machine, and potential

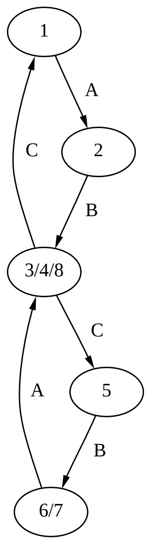

Figure 3.6: FSM Inferred by the KTAIL Method ($k = 2$) for the Example Event Stream of Figure 3.2.

transitions to them will also not show up. This pruning mechanism is implemented in the KTAIL method.

Evaluation

KTAIL, with thresholds set to 0, necessarily produces an FSM that can recognize all sample input data. This is because every position in the set of input strings belongs to some equivalence class, and is thus mapped to some state in the state machine. Since equivalence classes share k-length subsequent behaviors, the transitions from each state are guaranteed to satisfy each position in the set of input strings that is in that equivalence class. Moreover, the sequence of equivalence classes for the positions in an input string is the production path in the FSM for that string.

Since we only merge states that have the same output behavior, both in terms of the tokens their output transitions accept and the resulting states those transitions go to, this transformation has no effect on the correctness of the resulting FSM. It only (possibly) reduces the number of spurious states that the FSM contains.

If the threshold is non-zero, then we cannot guarantee the correctness of the resulting FSM with respect to the input data.

KTAIL is controlled in its accuracy by the value of k; the greater the value, the greater the length of the prefix tails that are considered, and thus the more differentiation that can occur in states and transitions. Note that as k increases, this algorithm monotonically adds states, and complexity, to the resulting FSM. If k is as long as the longest sample string, then the resulting

FSM is guaranteed to be deterministic, but deterministic FSMs can result from much smaller values of k, depending on the structure of the sample strings. It is not always the case that the most interesting and informative FSM will result from k being large enough to generate a deterministic FSM. To the contrary, points of nondeterminism in the resulting FSM might signify important decision points in the process, where the decision will determine the path of execution.

An advantage of KTAIL is that it is parameterized by the simple value k, so the complexity of the resulting FSM can be controlled in a straightforward manner. A disadvantage is that its simple thresholding may not be tunable enough to be robust in the presence of input stream noise.

3.4.3 MARKOV

The third method is one that we invented, and is a hybrid of algorithmic and statistical techniques. This method uses the concept of Markov models to find the most probable event sequence productions, and algorithmically converts those probabilities into states and state transitions. Although our method is new, there is previous work that has used similar methods. For example, Miclet and Quinqueton [92] use sequence probabilities to create FSM recognizers of protein sequences, and then use the Markov models to predict the center point of new protein sequences.

A discrete, first-order Markov model of a system is a restricted, probabilistic process[5] representation that assumes that:

- there are a finite number of states defined for the process;
- at any point in time, the probability of the process being in some state is only dependent on the previous state that the process was in (the Markov property);
- the state transition probabilities do not change over time; and
- the initial state of the process is defined probabilistically.

In general, the definition of an n^{th}-order Markov model is that the state transition probabilities depend on the last n states that the process was in.

The basic idea behind the MARKOV method is to use the probabilities of event sequences. In particular, it builds event-sequence probability tables by tallying occurrences of like subsequences. The tables are then used to produce an FSM that accepts only the sequences whose probabilities are non-zero or, more generally, that exceed a probability threshold that is a parameter to the method. MARKOV thus proceeds in four steps.

1. The event-sequence probability tables are constructed by traversing the event stream.[6]

[5] The use here of the word "process" does not refer to "software process", but to the generic definition in the terminology of Markov models.

[6] In the current version of our MARKOV tool, only first- and second-order probability tables are constructed. A future version of the tool will accept the maximum order as a parameter.

	A	B	C
A	0.00	0.50	0.50
B	0.54	0.00	0.46
C	0.42	0.58	0.00

	A	B	C
AA	0.00	0.00	0.00
AB	0.00	0.00	1.00
AC	0.50	0.50	0.00
BA	0.00	0.00	1.00
BB	0.00	0.00	0.00
BC	0.33	0.67	0.00
CA	0.00	1.00	0.00
CB	1.00	0.00	0.00
CC	0.00	0.00	0.00

Table 3.1: First- and Second-order Event-sequence Probability Tables for the Example Event Stream of Figure 3.2.

Table 3.1 shows first- and second-order probability tables for the event sequence of Figure 3.2. For instance, as given by the third row of the second-order table, the event sequence `A-C` is equally likely to be followed by an `A` or a `B`, but is never followed by a `C`.

2. A directed graph, called the *event graph*, is constructed from the probability tables in the following method. Each event type is assigned a vertex. Then, for each event sequence that exceeds the threshold probability, a uniquely labeled edge is created from an element in the sequence to the immediately following element in that sequence.

   Consider event sequence `A-C-B`, whose entry in the second-order table is 0.50. For a threshold less than 0.50, edges are created from vertex `A` to vertex `C` and from vertex `C` to vertex `B`.

3. The previous step can lead to over-connected vertices that imply event sequences that are otherwise illegal. To correct this, over-connected vertices are split into two or more vertices.

   Consider vertex `B`. After the previous step, edges exist from `B` to `C` and from `C` to `B`. However, sequence `C-B-C`, permissible under this connectivity, has a zero probability (see row 8 of the second-order table). Thus, vertex `B` is split into two `B` vertices, one having an edge to `C` and the other having an edge from `C` to it. This avoids the illegal sequence `C-B-C`.

   The general definition of this step works by finding disjoint sets of input and output edges for a vertex that have some non-zero sequence probability, and splitting the vertex into as many vertices as there are sets.

4. The event graph G is then converted to its dual G' in the following manner. Each edge in G becomes a vertex in G' marked by the edge's unique label. For each in-edge/out-edge pair of a vertex in G, an edge is created in G' from the vertex in G' corresponding to the in-edge to the vertex in G' corresponding to the out-edge. This edge is labeled by the event type.

   In Figure 3.7, vertex 5 and its edges are constructed from an edge labeled "5" in the event graph that connects vertex `B` to vertex `C`.

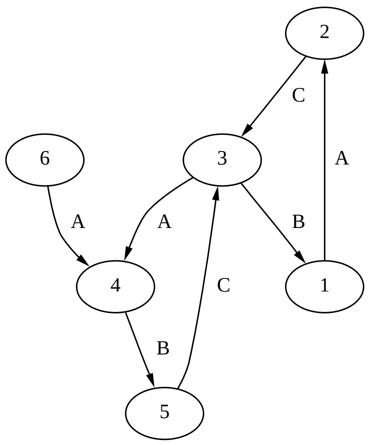

Figure 3.7: FSM Inferred by the Markov Method for the Example Event Stream of Figure 3.2.

The graph G' constructed in the last step is an FSM. This FSM can be further reduced using some techniques implemented in the Markov tool that merge nondeterministic transitions. Specifically, Markov can produce states where more than one transition from the state is annotated with the same event, but where this nondeterminism is not useful because the next states' behaviors are non-conflicting anyways. Nondeterministic transitions only make sense if they resolve a conflict in some way. Thus, for the nondeterministic transitions from a state, if their destination states' output transitions are non-conflicting (that is, they accept a disjoint set of events), then these states are merged and the set of nondeterministic transitions becomes one transition.

Figure 3.7 shows the inferred FSM that Markov produces from the example stream of Figure 3.2. As with Ktail, Markov infers an FSM with exactly two loops: the `A-B-C` loop as the path $< 3, 4, 5, 3 >$ and the `B-A-C` loop as path $< 3, 1, 2, 3 >$. The difference is that Markov produces a deterministic FSM. Notice, too, the existence of what can be interpreted as a start state (6) in the FSM produced by Markov. This is a result of using the single sample input that begins with the token `A`.

A Bayesian Extension to the Markov Method

Markov uses the frequencies of event sequences directly in its tables as forward probabilities of these sequences occurring. But there is some argument that Bayesian probabilities, which reverse the probabilities, would be better. Using the frequencies directly as probabilities can be affected by the occurrences of the events; for example, if the second event of a two-event sequence occurs very often in an event stream, then the frequency of that sequence might be high by chance; a Bayesian calculation can account for this.

In process event streams, we would not expect that certain event types would overwhelmingly populate the event stream to the point where MARKOV would be seriously affected, and thus one would suspect that the forward and Bayesian probabilities would not be radically different. Still, the Bayesian formulation might provide better thresholding cutoff effects than just using the forward probabilities.

We have implemented a Bayesian transformation in MARKOV that is selectable as an option. This transformation takes the forward probability tables built in the standard MARKOV method, and converts them in place using Baye's Law:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Note that all components of the right-hand side are in the forward probability tables. The Bayesian probabilities replace the forward probabilities, so that when MARKOV looks up the probability of the sequence `AB`, for example, it is now using the Bayesian probability instead.

We have not intensively explored its behavior with thresholding, but have not seen radically different results from the regular MARKOV to date. It is available, however, for those users who feel that they are not getting good results or who firmly believe that Bayesian probabilities are better.

Evaluation

MARKOV, with thresholds set to 0, will create an FSM that recognizes all of the input streams' behavior. Consider in the input stream S the subsequences S_i^N of length N at position i, and S_{i+1}^N of length N at position $i+1$, where N is the Markov order of the MARKOV algorithm producing M. The algorithm specifically includes production paths for all sequences of length N or less, so S_i^N and S_{i+1}^N are necessarily produced by M. Thus, S_i^{N+1}, by the union of the two above, is also produced by M. Induction proves that all of S is then producible by M.

Setting thresholds above 0, however, defeats any assurances that the resulting FSM will completely model all of the behavior in S.

MARKOV is more robust in the presence of noise in the event stream than the KTAIL method. Moreover, the level of this robustness can be easily controlled by the process engineer through the probability threshold parameters. These parameters can also be used to control the complexity of the discovered model from large amounts of data, ignoring low-probability sequences, even if they are not the result of noise.

3.5 Example Use of Discovery Methods

In this section we show how the three methods perform on a rather more complex process than the simple example of the previous section. The example in this section is taken from the ISPW 6/7 process problem [81]. We describe the process using an FSM that is based on Kellner's Statemate solution [80]. Our version is shown in Figure 3.8. The idea is to see how well the methods perform at reproducing this FSM and, thereby, discovering the process.

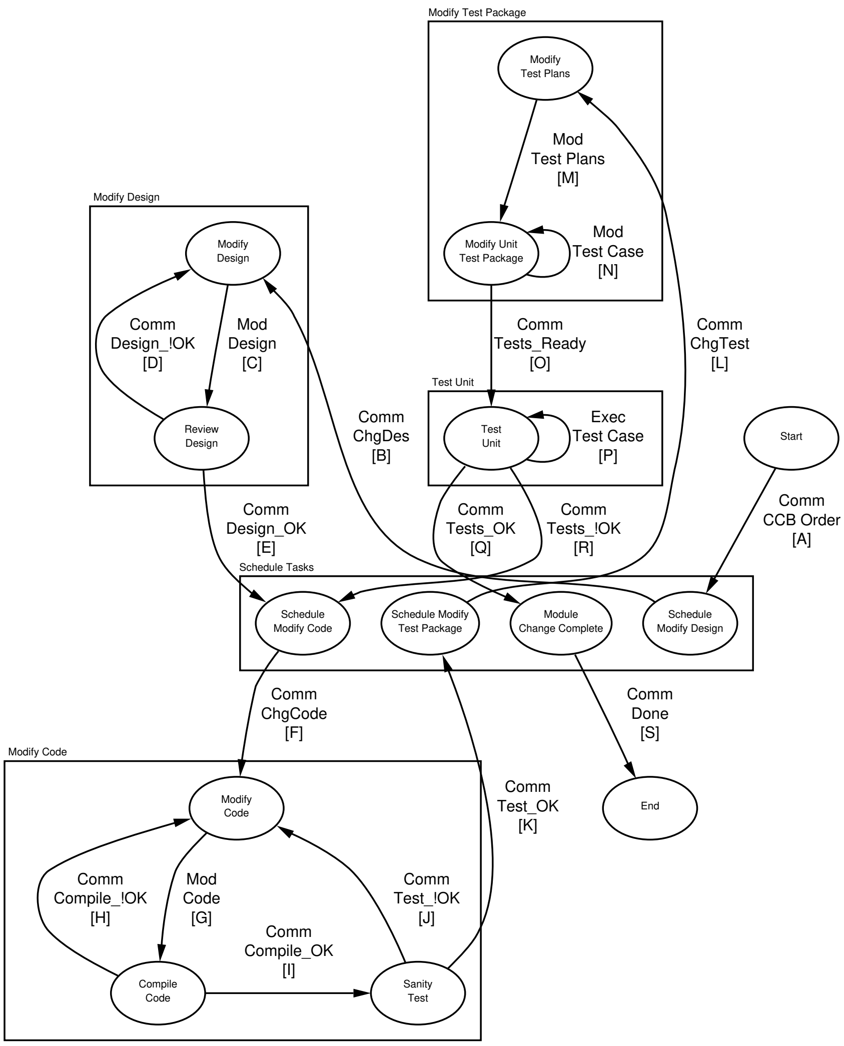

Figure 3.8: ISPW 6/7 Process.

At a high level, the ISPW 6/7 process proceeds as follows. A change order from the Change Control Board (CCB) triggers the start of the process. The design modification subprocess is scheduled and performed, leading to the scheduling and performing of the code modification subprocess. After code modification, the test package modification subprocess is scheduled and takes place, followed directly by the unit testing subprocess. If unit testing fails, the process loops back to the code modification subprocess (since the design modification is assumed to be correct) to redo the previous few process steps. This loop continues until unit testing succeeds and the process completes.

As stated in Chapter 2, an activity that spans time is represented by the interval between two or more events. Here we simplify the presentation by collapsing an activity into a single event. Transitions in the FSM of Figure 3.8 are labeled with the events that occur as the process executes. Many of the events are communication events, as indicated by "Comm" in their labels. Others are modification events, indicated by "Mod". Finally, there is one program execution event in the process and it is labeled with "Exec". Along with each label is a letter enclosed in brackets (e.g., `[A]`). The letter denotes the token that is used to identify the corresponding event in the sample streams fed to the methods. In the figures that show the FSMs produced by the methods (i.e., figures 3.9, 3.10, and 3.11), the tokens are used to identify events; the reader can refer to Figure 3.8 to relate a token to an event.

The boxes in Figure 3.8 serve to pictorially group individual states into subprocesses. The labels on the boxes correspond to the activity titles given in the ISPW 6/7 problem statement. In the figures for the inferred FSMs, below, we also group states into subprocesses in order to aid in the explanation of the results. Instead of using boxes with solid borders, however, we use boxes with dotted borders to emphasize the fact that we have placed the boxes into the figures by hand; the methods do not automatically group states into subprocesses.

We use the FSM of Figure 3.8 to generate sample event streams for the methods. Three different event streams were generated and used as input to the KTAIL and MARKOV tools. One of them, for example, is the following.

$\boxed{\texttt{ABCDCEFGHGIJGIKLMNOPRFGIKLMNOPQS}}$

RNET was given only one event stream for this example. The reason is that, since RNET is purely statistical in nature, results for a small example such as the ISPW 6/7 process can vary significantly on different sample input streams. To show the best obtainable results for RNET, this one stream artificially contained a balanced proportion of different event sequences.

Figure 3.9 shows the FSM that was discovered by the RNET neural network method. For this example, we use a network of 20 input neurons, 30 hidden neurons, and 20 output neurons. We also use an event window of size 3. The number of input and output neurons are constrained by the number of event types, the number of hidden neurons allow for a good discretization of the state space, and the window of size 3 is comparable to what the other algorithms use.

Looking at the results for RNET, we see that the *Modify Design* subprocess (represented by states 2, 3, and 4) and the *Modify Code* subprocess (represented by states 5 through 9) are evident in the FSM. Overall, however, the results are fairly poor, even given an artificially balanced sample. In particular, RNET has failed to differentiate the state transitions of some important event sequences. For example, state 14 has recurrent edges for three different events, which is not very informative. A similar problem arises with state 13. The root of the problem is that, since RNET looks at the statistical history, it does poorly at differentiating loop exits. That is, upon seeing the exit token, for example the `E` in `C-D-C-D-C-E` (states 3 and 4), it still activates the state in the loop that can accept a `D` (state 4). Thus, the real exit of the loop does not happen until one token later (`F`). This is true for all loops in this example; `O` continues the loop at 13, `R` and `Q` at 14, and `K` at 8. One effect

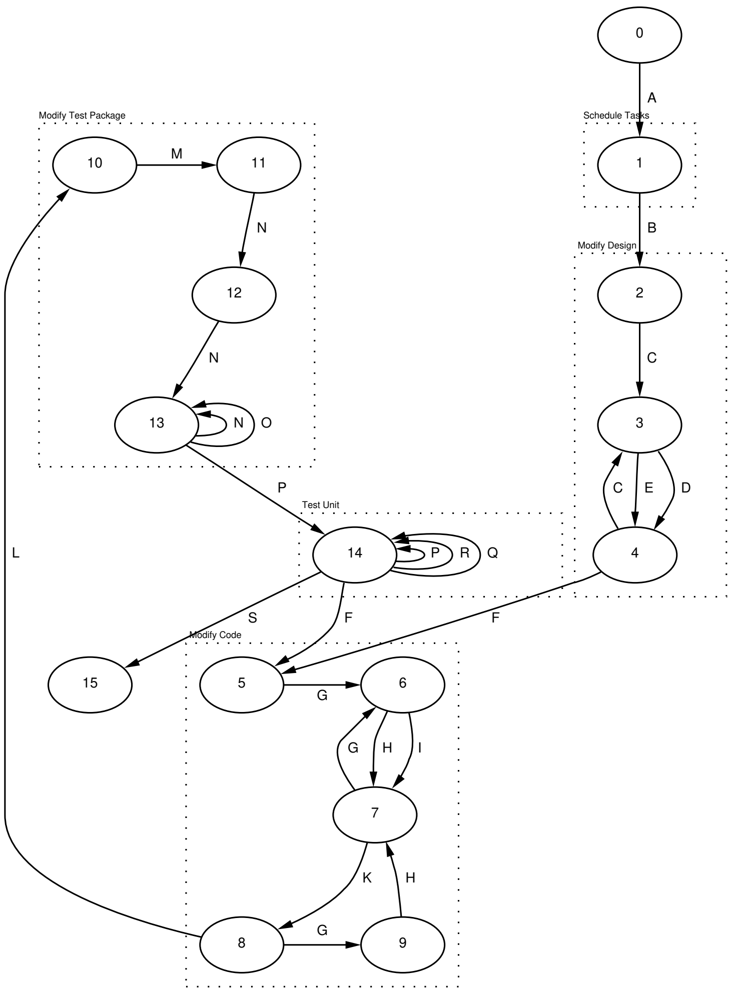

Figure 3.9: RNET **Discovery of Process in Figure 3.8.**

of this is that a *Schedule Tasks* subprocess is hard to locate in the FSM, since the intermediate scheduling states are not distinct.

Figure 3.10 shows the FSM that was discovered by the MARKOV method. For this example, since we know that the data are exactly correct (i.e., contain no noise), the probability threshold is set to zero, so that no non-zero transition is ignored.

One can see that this method produces a good model that is very close to the one in Figure 3.8; it is equivalent in almost all of its structure. The subprocesses are easily located in this model, and the loop paths, though usually having one extra state in them, are easily distinguished. The extra state for each loop results from the fact that the algorithm creates an entry state, such as 3, for

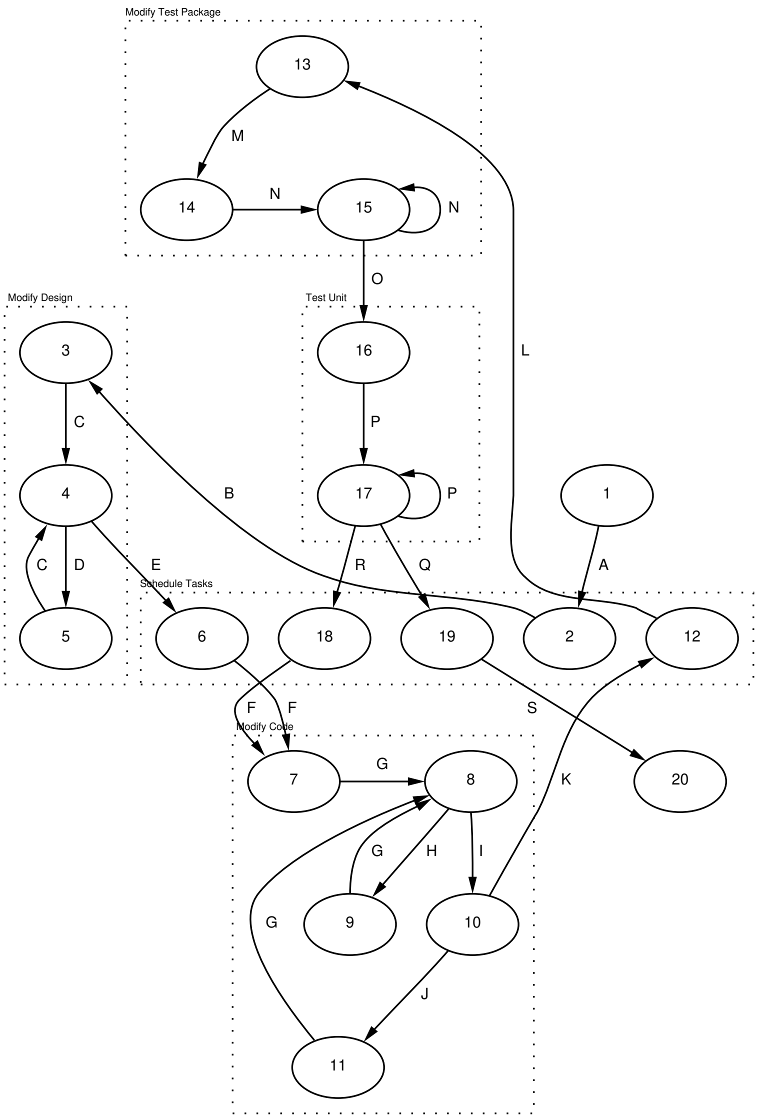

Figure 3.10: MARKOV **Discovery of Process in Figure 3.8.**

each loop. However, MARKOV failed to merge the states 6 and 18 in *Schedule Tasks*, both of which provide an entry into the subprocess *Modify Code*. All in all, however, the method provides a clean and understandable representation of the process.

Figure 3.11 shows the FSM discovered by the KTAIL method. For this example, we use a value of 2 for the tuning parameter k. Other values of k were explored, and $k = 2$ gave the cleanest

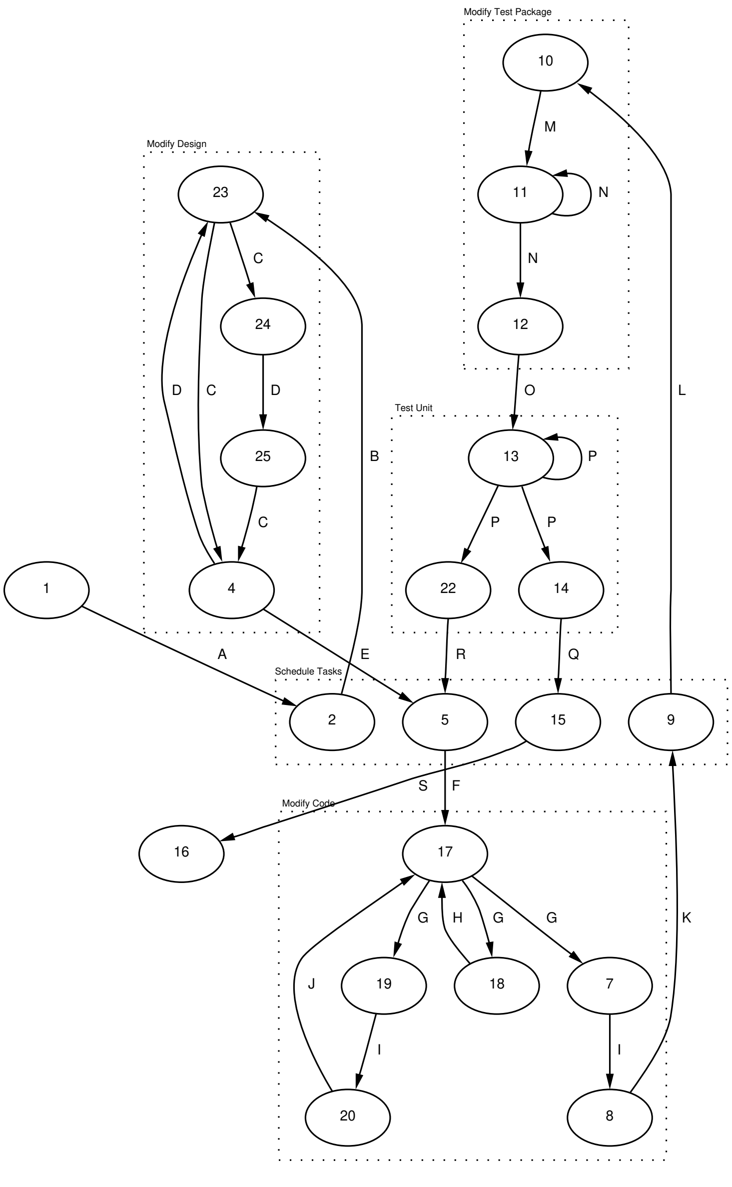

Figure 3.11: Ktail **Discovery of Process in Figure 3.8.**

result.

The FSM successfully captures all the behavior of the example process. It has the correct number of states in the *Schedule Tasks* subprocess, finds the loops, and implements a correct machine that allows no anomalous behavior. However, as explained in Section 3.4.2, the algorithm

Tool	Language	NC-LOC
RNet	C	1060
KTail	C++	1010
Markov	C++	950
Common	C++	1100
UI	Tcl/Tk	840

Table 3.2: Non-Commented Lines of Code for the Discovery Tools.

consistently unrolls loops. The $(11, 12)$ and the $(13, 14, 22)$ subgraphs both have the same feature; their loops are unrolled by one state and, in the case of node 13, two exit paths are produced. Notice that the *Modify Code* subprocess shows distinct execution paths for each possible loop and for the exit path in that subgraph. With a straightforward merging of states 7, 18, and 19, and then a consequent merging of 8 and 20, the subgraph becomes identical to the corresponding subgraph in Figure 3.8.

In contrast to MARKOV, the KTAIL method correctly produces a single state (5) in the *Schedule Tasks* subprocess for the two possible paths into *Modify Code*, thus showing that it can recognize a shared subprocess in its entirety. This fact, along with the fact that most extra loop states can be algorithmically removed, causes us to view KTAIL as producing the best results of the three methods for this example.

3.6 Implementation of the Discovery Tools

The methods described in this chapter all have been implemented and experimented with as part of this thesis. RNET was adapted from an implementation by Das [39], while the others were implemented from scratch. Additionally, a common user interface, other common code (startup and graph operations), and an interface to the BALBOA framework (Chapter 5) are all implemented. Table 3.2 shows the size (LOC) of the various discovery tools.

The discovery methods output their discovered FSM in a format compatible with the *dot* graph layout program [83]. This leveraging of an existing graph layout tool is central to the successful application of the discovery tools. It provides an end-to-end solution, going from event data directly to a visual display of the discovered process model.

DISCOVERY, seen in Figure 3.12, is the process discovery tool of BALBOA. It allows one to select event streams, specify discovery method parameters, and run the methods to produce discovered process models.

With DISCOVERY, one just specifies the source event collection (with its mapping specification), the discovery method, and the parameters for that discovery method. DISCOVERY then spawns off a discovery process, and in turn spawns a DOTVIEWER window to display the results. Each time a discovery process is run a new DOTVIEWER window is created. Once the DOTVIEWER display

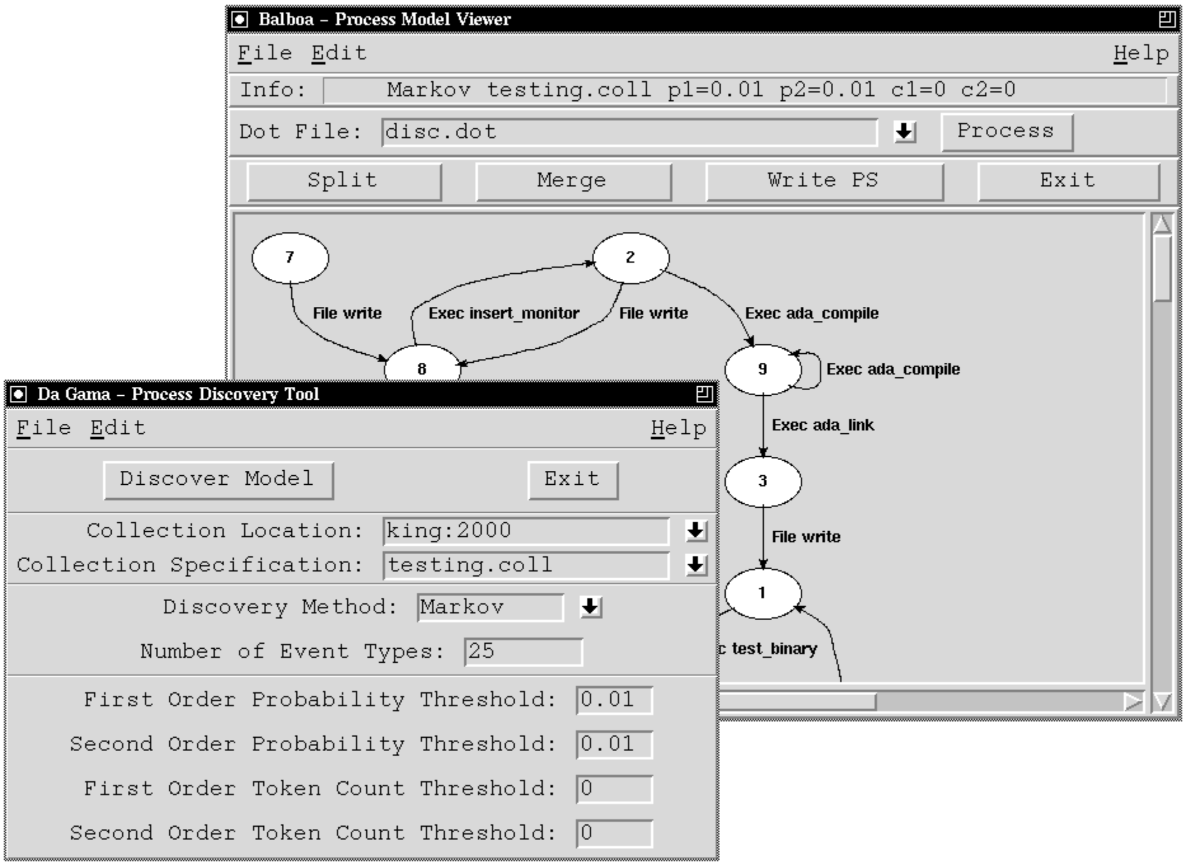

Figure 3.12: The Discovery tool.

window is spawned, it is completely independent of Discovery; thus, multiple methods and/or parameter variations can be run and compared side-by-side on a large display.

DotViewer is the tool that allows one to view an FSM, specified in *dot* format, and then edit the FSM. It has several features that are directly specific to dealing with process models from Discovery. Previously, we said that we intended Discovery to produce process models that need completing by human beings. To this end, the DotViewer tool not only presents the discovered process models visually, but allows for the modification of the discovered process model. The operations that it supports are: merging nodes, creating/deleting nodes, creating/deleting edges, naming/renaming nodes, and splitting nodes. It uses *dot* to lay out the graph automatically when first loading the graph, and after each edit operation. Thus, there are no commands for moving nodes and edges, only the high-level operations listed above.

Feature	KTAIL	MARKOV	RNET
Correct Model	Possibly	Possibly	Unlikely
Speed	10^1 sec	10^1 sec	10^3 sec
Tunability	two params	few params	many params
Usability	easiest	easy	hard
Robustness to Noise	moderate	good	good

Table 3.3: Comparison of Discovery Methods.

3.7 Evaluation of the Discovery Methods

In this section we evaluate and compare the three methods that have been presented in our work. We defer evaluating the usefulness of the methods on real-world data to Chapter 6, where an in-depth study is presented, and just concentrate on the performance of the algorithms here.

Table 3.3 shows a comparison of the three methods with respect to several characteristics. An in-depth description of those characteristics follows.

Correct Model. Both KTAIL and MARKOV can generate a correct model for the data they are given, but also could potentially not generate a complete and correct model for the data given, since thresholds can be set to ignore low-probability event sequences. If the thresholds are set to 0, the methods will always generate a complete and correct model. RNET, being purely statistical, cannot be configured to guarantee the generation of a correct model.

Speed. The numbers in Table 3.3 are approximate, intended to show relative speeds of the methods. They were taken from the performance of the methods on the ISPW 6/7 example. KTAIL and MARKOV are comparable, but RNET is considerably slower. This slowness is due mainly to the large amount of required training.

Tunability and Usability. These two characteristics are quite related, because often the more tunable something is, the more complicated is its interface. In this respect, it seems that RNET is overly tunable, or at least it takes too much knowledge of neural networks to establish a reasonable set of initial settings. KTAIL and MARKOV, being only tunable in a few dimensions, might be overly restrictive, but using them is straightforward.

Robustness to Noise. Both the MARKOV and RNET methods have the potential of being good at inferring models in the presence of collection errors and abnormal process events. KTAIL has some ability, in its equivalence class size threshold, but we regard it as less configurable to handle noisy data streams.

The next two tables and discussion compare KTAIL and MARKOV more closely. Because RNET is not comparable in performance to the other two, we do not explore its performance space. In all dimensions, it seems to be much worse than the other two.

Length of Event Stream	MARKOV		KTAIL	
	Time (s)	Size (KB)	Time (s)	Size (KB)
312	4.61	670	1.07	386
625	4.74	701	2.10	432
1250	5.07	780	5.26	533
2501	5.62	917	16.30	736

Table 3.4: Time and Space Requirements Versus Event Stream Lengths.

Number of Events	MARKOV		KTAIL	
	Time (s)	Size (KB)	Time (s)	Size (KB)
20	1.03	432	1.51	426
40	4.74	701	2.10	432
60	13.90	1304	2.24	433

Table 3.5: Time and Space Requirements Versus Number of Events.

Table 3.4 shows the performance of MARKOV and KTAIL over different event stream lengths, but with the same number of event types. Both show a linear time relationship to the stream length, but MARKOV's coefficient is so low that the increase in running time is almost negligible, whereas KTAIL shows significant increase in running time. Still, both of the methods show good performance.

As for space usage, both increase slightly for longer stream lengths, yet neither show drastic space requirements.

Table 3.5 shows the performance of MARKOV and KTAIL over different numbers of event types, keeping the stream length constant (625 events). MARKOV displays non-linear time and space relationships to the number of event types processed, while KTAIL does not.

From the two tables above, KTAIL is affected by event stream length, while MARKOV is affected by the number of event types. This is what we would expect, since KTAIL processes each event position in the stream as being in some equivalence class, and MARKOV process tables of probabilities of event type sequences.

3.8 Handling Concurrency

Suppose that the event stream data represent the activity of more than one process execution happening concurrently. These could be the same process or different processes. Is there any hope for analyzing the data and discovering the concurrency? This section describes some approaches that may help to further this end.

P1	P2
.	.
.	event(A)
receive(X)	.
.	send(X)
event(B)	.
.	.

Figure 3.13: Two Processes Producing Events A and B.

Concurrency is manifested as interleaved event streams. Concurrency in a (given) model is not the same as concurrency in the event stream; consider the two processes in Figure 3.13: although the model is formulated in terms of processes P1 and P2, events A and B are strictly ordered. So, concurrency in the event stream does not necessarily reflect the separation of the discovered model.

We can view concurrency as randomly (by the definition of 'concurrent') interleaved sets of event streams. The concurrency then manifests itself as 'noise', in the sense that the sequencing of the events will contain some randomness. If the noise is too strong, then we cannot find anything. If it is not, however, a probabilistic analysis is our only hope, since any algorithmic approach cannot ignore the random sequences.

While our methods discover finite state machines, given the above view, they are not necessarily unable to discover any concurrency in the event stream. Specifically, the structure in the resulting state machine describes the pattern of event sequences, but it does not imply a control model behind the machine. One could imagine more than one thread of control active in the state machine at one time, resulting in interleaved but constrained event sequences that are described by the patterns in the model.

If the concurrency manifests itself as low-probability sequences, where one process switches to another, then with the proper setting of thresholds in the algorithms, these sequences will be ignored. The resulting FSM may be disjoint, actually consisting of separate state machines for each process. If one interprets each disjoint machine as having its own thread of control, then the algorithm has, in fact, discovered concurrent processes from the input data.

Other approaches might deal with avoiding the concurrency. We might be able to preprocess the event stream to separate unrelated events based on their attributes, or we might be able to modify the data collection method to generate separate event streams for different threads.

Another approach is to look for places in the event stream that point to possible concurrent actions. For instance, assume that if B is causally related to A, then it will happen quickly after A. We define quickly as within some window of size w on the event stream, after A. We now define $P_w(B|A)$ = probability of B occurring in the window w after A; this is then interpreted as the probability that B is causally related to A over some window of size w.

Suppose that events B and C happen concurrently after A, and that this is manifested in the event stream as sometimes seeing ABC and other times seeing ACB. On average, if one looks at just the event following A, which is a window of size 1, then $P_1(B|A) = 0.5$ and $P_1(C|A) = 0.5$, since they will both follow A half the time, on average. However, looking at a window of size 2 gives $P_2(B|A) = 1.0$ and $P_2(C|A) = 1.0$, because B and C always occur within that window.

Thus, looking at a window larger than 1 after a certain event may show more deterministic behavior than just looking at the next event. This is an indication that that location in the event stream is manifesting concurrent behavior.

The problem in applying this idea is how to select the window size for all locations in the event stream. Simply choosing a fixed window for the whole event stream does not capture the notion of concurrency only happening at certain points. It seems very likely that the best window size to use after some event E will be specific to that event type—assuming that an event type occurrence indicates some potential concurrency.

A method to answer this question is to calculate the entropy of an event stream as seen through the windows; An entropy calculation should have a maximum value of 1, which represents complete distribution of probabilities. With N choices, entropy is

$$E = \sum_{i=1}^{N} P(i) log_N P(i)$$

For us, N is the number of event types, or more specifically the number of token types (after the mapping of events).

The entropy calculation tells us how the probabilities are distributed. If $P(B|A) = 1.0$ and $P(E|A) = 0.0$ for all other event types E, then the behavior after A is perfectly deterministic and thus the entropy is 0. As the probabilities become more distributed, entropy increases.

Thus, if we calculate E_w for window sizes $1 \leq w \leq W_{max}$, and for each token type, pick the window size that minimizes E, we can use that window size (for each token type) to build the probability tables, and find which event types signal possible concurrent behavior after them. For an event that is using a window greater than size 1, all significant events in that window will get transitions made for them in the final CFSM—this branching area in the state machine would be interpreted as a fork rather than a selection.

While this method may detect forks, detecting joins, or the termination of concurrency, may be more difficult. At some point the discovery methods will naturally begin to see a single pattern sequence, but it might be up to the engineer to identify the correct point at which concurrent behavior stops.

3.9 Summary of Discovery Work

We have described three methods of process data analysis that can be used to produce formal models corresponding to actual process executions. This analysis supports the process engineer in constructing initial process models. Based on our early experience with these methods, we conclude that the KTAIL and MARKOV methods show the most promise, and that the neural network based RNET method is not sufficiently mature to be used in practical applications. Of course, more experiments are needed to fully explore the strengths and weaknesses of all three methods.

Process discovery is not restricted to creating new formal process models. Any organization's process will evolve over time, and thus their process models will need to evolve as well. Methods for process discovery may give a process engineer clues as to when and in what direction the process model should evolve, based on data from the currently executing process.

While we have focused here on the use of these techniques for generating formal models, we also believe that they are useful in simply visualizing the data collected on a process. An engineer may simply be interested in a way to better understand the current process, as captured by the event data; discovering patterns of behavior as those represented by a finite state machine can be of help.

We do not see our discovery techniques as the only available tool to help process engineers build process models. Rather, we believe that more than just one technique will be useful in helping process engineers to discover, build, and evolve process models. For example, the area of data mining may have useful techniques for discovering software product relationships.

Future work for process discovery will be:

- Implementing and experimenting with the ideas for discovering concurrency in event streams that were discussed in Section 3.8.

- Seeding discovery methods with potential models or pieces of models would take advantage of an engineers existing knowledge of a process. Both KTAIL and MARKOV, at first thought, would seem to naturally allow this, just by initializing them with equivalences classes and probabilities, respectively—although this may not translate to exactly the same model pieces that were seeded. A related issue is making the methods interactive, allowing the user to dynamically control the result.

- Exploring the extension of the discovery methods to other models. Much inference work includes learning grammars, but this does not naturally fit software process. Petri nets or rule bases are more appropriate extensions for this domain.

- Investigating the difference in behavior of the Bayesian versus forward probabilities for the MARKOV algorithm. Some measure of goodness for deciding which is better would need to be devised, and then an exploration across different data and threshold parameters could help answer whether or not the Bayesian extension is useful.

- Visualization improvements for the output of the discovery methods may help in the understanding by the engineer. For example, drawing transitions with a thickness relative to their probability in the MARKOV-discovered model, or drawing states with a shading relative to the size of the equivalence class in KTAIL, may help the user in understanding the discovered model better.

- The MARKOV method really discovers a Hidden Markov Model, with the probabilities on transitions. However, this is not well integrated into the method's implementation.
- Better assistance in parameter selection for the methods would reduce the trial-and-error recursion that a process engineer must go through to apply the discovery methods. We have implemented a graphical viewer of the probabilities in the MARKOV method to help guide parameter selection, and a similar tools would be useful for KTAIL.

Chapter 4

Process Validation

Anytime a formal specification of a system is created, the question arises of whether or not the specification actually describes the real system, regardless of whether it is itself formally consistent, complete, and correct (which is presumably why the formal specification was created in the first place). Where an automatic translation of the specification to the system takes place (e.g., a program compiled into an executable), this question is often irrelevant, but if the specification goes through a human-interpreted step, the problem of whether the behavior predicted (or prescribed) by the specification agrees with the actual behavior of system is a serious one. Software process modeling as a formal domain, where the model is a specification and the system is the process as it is executed by people, is not exempt from this problem, as Figure 4.1 shows.

Software process researchers have historically attacked this problem through prevention; they make the formal model be the process. In other words, the formal model is used to enforce the process execution and, therefore, the model and process are necessarily in sync. Enforcement is typically done by making the model executable and embedding that execution within the automated software engineering environment used by the project. This approach, however, suffers from a fundamental flaw. In particular, it assumes that virtually the entire process is executed within the context of the automated environment. In fact, critical aspects of the process occur off the computer and, therefore, not under the watchful eye of the environment [113, 121, 122]. That being the case, there is no effective way to enforce the process using this approach nor to guarantee the mutual consistency of a process model and a process execution.

In addition, the workflow community has long recognized the need to allow exceptions to the prescribed process [62]; so, in reality, there is a strong need to allow for deviations from the model.

Even if one could completely enforce a process, there still remains the issue of managing change in a process, which might lead to a discrepancy between the model and the execution. There has, in fact, been considerable recent work that addresses process evolution [11, 76]. Commensurate with the historical approach mentioned above, that work is concerned more with the problem of effecting changes to a process model used for automation, than it is with the problem of uncovering inconsistencies between the model and the execution.

This thesis has developed techniques for detecting and characterizing differences between a

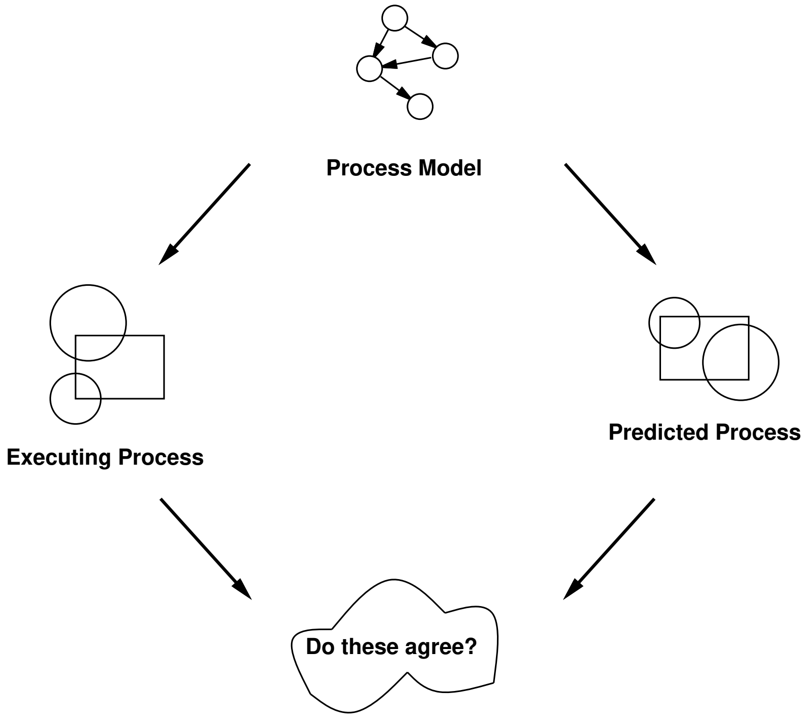

Figure 4.1: The Process Validation Problem.

formal model of a process and the actual execution of the process. We refer to the detection and characterization of differences as *process validation*. Process validation serves several purposes. For one, confidence in a formal process model is raised when it can be shown that the process execution is consistent with the behavior predicted by the model. This, in turn, raises confidence in the results of any analyses performed on the formal model. For another, process validation can be used as a process enforcement tool, uncovering differences between intended behavior and actual behavior. It is potentially a more flexible enforcement tool than others proposed, since it can accommodate the unavoidable, yet necessary, local perturbations in a process. Finally, process validation can reveal where a process may need to actually evolve to accommodate new project requirements and activities.

Note that process validation techniques themselves are neutral with respect to both the correctness of the model ("Does our model reflect what we actually do?") and the correctness of the execution ("Do we follow our model?"). The software process engineer has ultimate responsibility for making the appropriate determination of whether a problem lies within the model or within the execution, based on the particular inconsistency uncovered.

In this chapter we describe the techniques we have developed for process validation. The techniques borrow from other areas of computer science, including distributed debugging, concurrency analysis, and especially pattern recognition. The techniques go further than simply detecting an inconsistency; they provide a measure of that inconsistency. We believe that developing metrics for process validation is critical because the highly dynamic and exceptional nature of software processes means that simple yes/no answers carry too little information about the significance of any given inconsistency. Managers need to understand where an inconsistency occurs and how severe that inconsistency might be before taking any corrective action. We view the metrics presented here as a first step toward a suite of useful metrics for process validation.

The next section presents the background and justification for the methods we use. Section 4.1 describes the background and related work. Section 4.2 presents the definitions of our validation metrics, Section 4.3 presents example uses of the metrics, Section 4.4 discusses possible extensions, and Section 4.5 describes how one can compute the metrics. Section 4.6 describes the implementation of the validation tools, and Section 4.7 evaluates the same. Section 4.8 then summarizes the work in validation metrics.

4.1 Discussion and Background

In our framework of process validation, we have an executing process that is producing an event stream and, on the other side, we have a model that can produce a desired or prescribed event stream.

Thus, we can cast the validation problem as quantitatively measuring how close the event stream of the executing process is to an event stream that could be produced by the model. We call these two event streams the *execution event stream* and the *model event stream*. This solution paradigm is shown in Figure 4.2. In describing this paradigm, we will put off discussing how to generate the model event stream until Section 4.5.

There are several methods for doing a measurement such as this, but one that seems most widely applied is string distance metrics [85]. The string distance method counts the number of token (symbols that make up the string) swaps, insertions, and deletions needed to transform one string into the other. By applying various mathematical transformations, this method becomes a family of metrics. These methods have been used in fields as various as DNA/RNA matching ([120]), substring matching ([79, 108]), spelling correction ([45]), syntax error correction ([2, 52, 102]), and even the well-known Unix *diff* program. A good reference to the general area of sequence comparison is [85].

Other methods of doing this measurement do not offer the versatility that the string distance metrics do. Hamming distance, for example, is the count of the number of tokens that differ, but this assumes that either the streams are the same length or that they can be suitably matched and padded. In actuality, this is subsumed by string distance methods, by removing the insertion and deletion operations, and just tallying swaps.

Generally speaking, string distance metrics have become the standard method in any domain

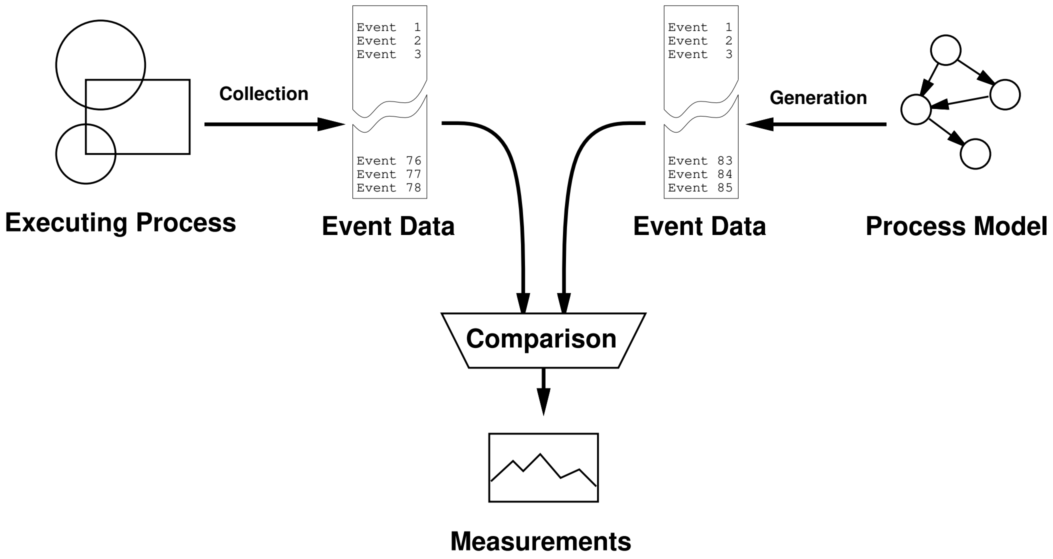

Figure 4.2: The Process Validation Abstract Solution.

where symbolic sequence comparison is taking place.[1] String distance methods are also generalizable to tree and graph distance measurements [90].

Methods that can compare a model directly with some event stream are also available, but our abstraction away from this is more generalizable. Error-correcting parsers [2], for example, have techniques in which they add error productions to the regular grammar production rules to catch deviations from the expected grammar's syntax. This could be instrumented with measurements, but it is specific to grammar formalisms. In our work we would like to avoid specific formalisms as much as possible, because the software process modeling domain has many different modeling languages, and other tools may drive the selection of the modeling language. Section 4.5 discusses the requirements that we place on a modeling language.

4.1.1 Related Process Work

There has been significant previous work that uses data to characterize processes, but none that uses the data in a process validation activity. In particular, most previous work has used product data metrics to guide process changes that refocus effort onto specific problem areas of a project. Below, we summarize some of this work.

[1]Numeric sequences, which are really a representation of some function (e.g., a time series of stock value), is a different topic altogether.

- Chmura et al. [35] and Bhandari et al. [21] try to deduce problems in the process by looking at defect data in the products. Specifically, they statistically analyzed change data and effort data to determine the behavior of the process. For example, they saw ripple effects from interface changes, saw high percentages of fix-on-fix changes, and proposed two measures of process progress that required less data than previous measures.

- Selby et al. [110] take the approach of providing automated support for empirically guided software development. Their system, Amadeus, can automatically collect measurement data (currently focused primarily on product data) that can then be used to guide development efforts. They applied Amadeus to automatically building classification trees that describe what properties of a source code module are likely to lead to faults in the module. However, Amadeus itself is a generic data collection and activity automation system.

- Basili and Weiss [16] describe a methodology for selecting metrics and data collection techniques based on the goals that are desired of the measurement activity. Their work also focuses on using product data, such as code modifications and change classification. This work has become known as the GQM, or goal-question-metric, paradigm.

- Kellner [80] shows the usefulness of simulation and "what-if" analyses in forecasting the schedule and outcome of a specific execution of a process. He uses deterministic and stochastic modeling, along with resource constraints, to derive schedule, work effort, and staffing estimations. Though there is no attempt to relate this to real data, "what-if" analyses are powerful tools in their own right.

Some recent efforts have begun to look at process data itself, but still not for the purpose of process validation.

- Garg and Bhansali [55] describe a method that uses explanation-based learning to discover aspects and fragments of the underlying process model from process history data and rules of operations and their effects. This work centers on using a rule base and goals to derive a generalized execution flow from a specific process history.

- Garg et al. [56] employ a manual process history analysis in the context of a meta-process for creating and validating domain-specific process models and software toolkits.

- Bradac et al. [23] describe the beginnings of a process monitoring experiment in which their goal is to model the process as a queuing network, and use actual data about the time spent by the agents in specific tasks and states to determine the real parameters (i.e., service times and probabilities, and branch path probabilities) of the queuing network that can then be analyzed.

- Wolf and Rosenblum [121] demonstrate how to collect event-based process data and use basic statistical and visual techniques to find interesting relationships among the data in order to uncover possible areas of process improvement.

- Porter et al. [98] use a well constructed experimental setting to assess the benefits of code inspections, in terms of group size and meeting gains.

We feel that the work in this thesis effectively complements these other approaches to process improvement by raising confidence in the correspondence between formal models and executions of processes.

4.1.2 Related Event Work

In using event-based data to compare an execution with a formal model, the most closely related work to ours is in the areas of distributed debugging and history checking.

- Bates [17, 18, 19] uses "event-based behavioral abstraction" to characterize the behavior of programs. He then attempts to match the event data to a model based on regular expressions. However, he only marks the points at which the data and model did not match, not attempting to provide aggregate measures of disparity.

- Cuny et al. [38] builds on the work of Bates, attempting to deal with large amounts of event data by providing query mechanisms for event relationships. They assume that there is some problem somewhere in the event stream and that one is trying to locate that problem.

- Felder et al. [50, 51] describe a method and tool by which one can compare an execution history against a temporal logic specification to decide the correctness of that execution with respect to the model.

Our immediate goal is to quantify discrepancies, with correctness being a subsumed issue. In fact, we assume that the execution event stream is incorrect with respect to the model, so that these techniques would not give us much information.

4.2 Validation Metrics

In this section we introduce three metrics for determining the correspondence between a formal model of a process and an execution of the process. The metrics are successively more refined and, not surprisingly, successively more complex. They share the characteristic that they compare the event stream produced by a process execution to an event stream representing a possible behavior predicted by the process model. The issue of how the second of these event streams is constructed is an important one and is discussed in Section 4.5. We defer detailed examples of applying the metrics to Section 4.3.

We make no assumptions about the formalisms used to model processes, other than that they must have well-defined behavioral semantics permitting simulation of the specified behavior. By simulation, we simply mean the ability to generate (pseudo-) execution paths through the specification. Along with this, it must be possible to identify locations in the specification where the simulation could produce an event of a specified type. We refer to such a location as an event site. Thus, possible paths through the specification—that is, possible behaviors specified by the model—can be represented by event streams produced by a simulation.

Several formal models suitable for our analyses have been used to describe software processes. These include models based on state machines (e.g., Statemate [65]), Petri nets (e.g., Slang [13] and FUNSOFT Nets [63]), and procedural languages (e.g., APPL/A [114]).

4.2.1 Recognition Metric

The first metric is a very straightforward one that has just two values, true or false. The value is true if the event streams exactly match and is false otherwise. We refer to this metric as the *recognition* (REC) metric because we imagine an implementation of the measurement operating as a simple recognition engine in the style of a grammar checker, one that terminates by returning the value false at the first error or by completing the recognition and returning the value true. An implementation of the measurement can also easily point out the event at which the two streams diverge. For example, consider the two event streams shown in Figure 4.3(a), where the lettered boxes indicate events. The recognition metric applied to the two streams would result, as shown in Figure 4.3(b), in the value false because they diverge after the third event. Notice that, with REC there is no information on how the rest of the event streams correspond.

4.2.2 Simple String Distance Metric

For the second metric, we view event streams as strings, where event types appear as distinct tokens in the strings. We can then apply a well-known method for calculating the distance between strings [85] and use distance as the metric of difference between the process model and process execution. String distance metrics have been used profitably as measures of correspondence in a wide variety of other domains, including parsing, DNA/RNA sequencing, and text recognition [107].

The basic Levenshtein distance between two strings is measured by counting the minimal number of token insertions, deletions, and substitutions needed to transform one string into the other. Figure 4.3(a) shows two event streams, represented as strings, and one possible correspondence between their events. For our purposes in computing a distance between execution and model event streams, we choose the execution event stream as the one to which the operations are applied.[2] With this choice, insertions represent missed activities (the model expected them but the execution did not perform them), and deletions represent extra activities (the model did not call for them, but they were performed in any case). Figure 4.3(c) shows how SSD might transform the execution stream into the model stream; we could delete a C, substitute a D for a C, and insert an E, a D, and another E, resulting in a distance of 5. This happens to be the minimal transformation required.

To strengthen the metric, weights can be assigned to each of the operation types (insertion, deletion, and substitution), giving a relative cost to each operation. Then, instead of minimizing the number of operations to calculate the distance, the goal would be to minimize the total cost

[2] The operations are isomorphic, so choosing one event stream over another does not change the resulting measurement, it just reverses the senses of insertion and deletion.

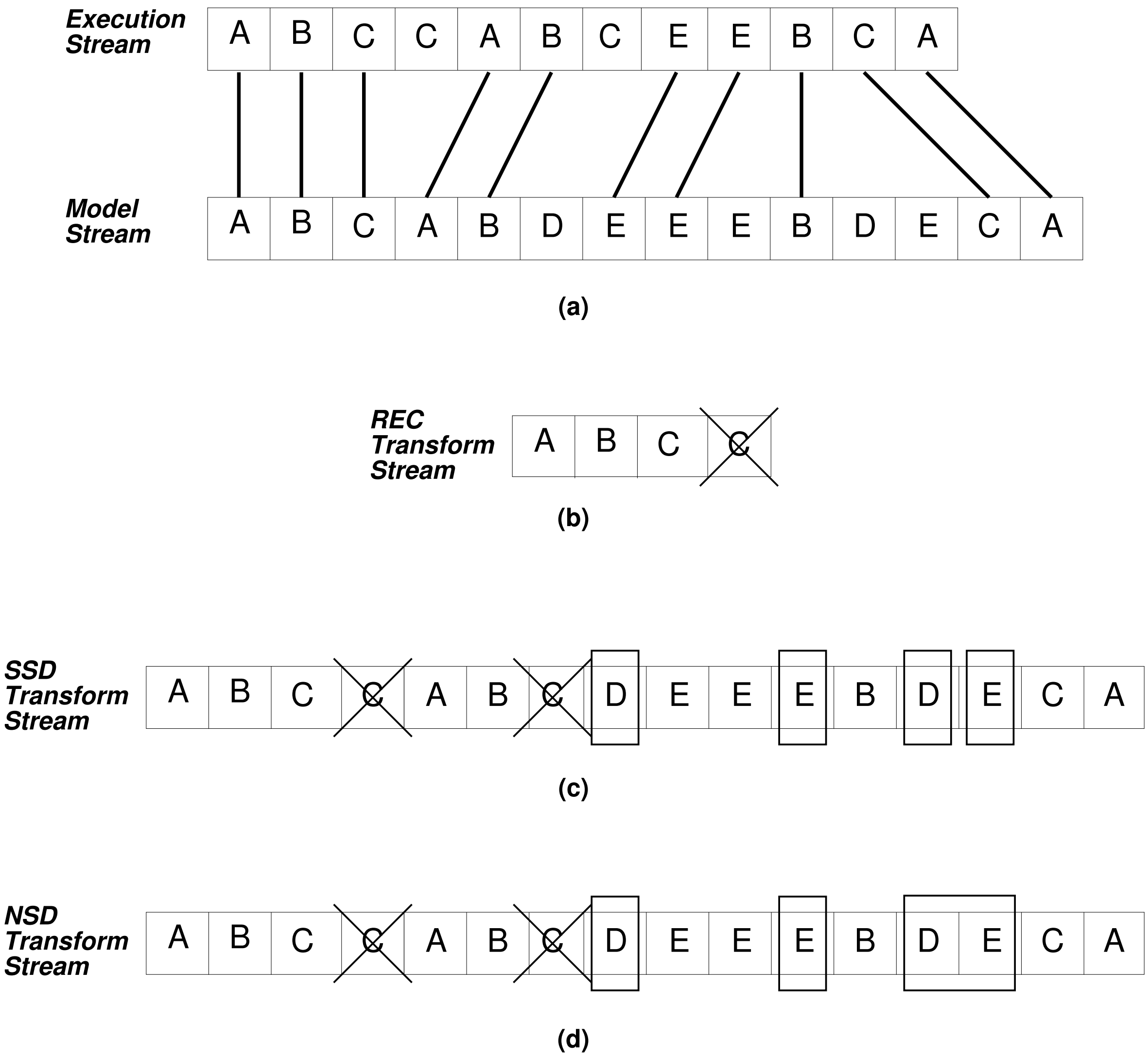

Figure 4.3: Two Event Streams and One Possible Correspondence of Events.

of the operations. Given two strings, one of length n and the other of length m, the minimal total cost of operations can be computed in $O(nm)$ time using a well-known dynamic program [85].

In some applications of this method, such as DNA/RNA sequencing or text recognition, token substitution in the string distance metric makes sense. For process validation, however, it is not clear that a substituted event should contribute in any way to the goodness of the correspondence. To account for this, we can set the weight of substitution to be greater than the sum of the insertion and deletion weights, so that substitution is never applied, since it would then be less costly to apply a deletion and insertion pair at the potential substitution point. We will not consider substitution further in this thesis.

The *simple string distance* (SSD) metric is then formulated as the following equation

$$SSD = \frac{W_I N_I + W_D N_D}{W_{max} L_E}$$

where W_I and W_D are the weights for the insertion and deletion operations, N_I and N_D are the number of insertion and deletion operations performed on the execution event stream, W_{max} is the maximum of W_I and W_D, and L_E is the length of the execution event stream. The divisor in the equation normalizes the value to the size of the input and the maximum weight used.

The weights W_I and W_D act as tuning parameters for the metric and can be used to highlight different properties of the process. For example, one could argue that insertions into the execution event stream are more costly than deletions, since they inherently represent missed activities in the process execution. Conversely, deletions from the execution event stream in some sense represent extra work that was performed (from the perspective of what is predicted by the formal model) and extra work probably does not affect the correctness of the process execution. Thus, we can set $W_I \gg W_D$ to reflect this property.

The values of the metric are, for all intents and purposes, bounded between 0 and 1.0; although technically a value greater than 1.0 could appear (e.g., if all events are deleted and some others are inserted), this is highly unlikely. Thus, one might pick the standard statistical correlation rules of thumb [42] and say that any measurement less than 0.2 is a strong correspondence, less than 0.5 is a moderate correspondence, and greater than 0.5 is a weak correspondence. (Actually, these are inversions of the standard statistical rules of thumb, but their effect is the same.)

4.2.3 Non-linear String Distance Metric

A characteristic of the SSD metric is that it is focused narrowly on the cost of individual operations. The *non-linear string distance* (NSD) metric is an enhancement of the SSD metric based on the notion of a sequence of insertions or a sequence of deletions. Such sequences can represent a more significant discrepancy between the model and the execution than can be indicated by a simple count of insertions and deletions.

A sequence of insertions or a sequence of deletions is called a block. By sequence we mean an unbroken series of like transformation operations. In Figure 4.3(d), for example, NSD recognizes the consecutive D and E insertions required at the end of the streams as an insertion block of length 2. All other blocks in the figure are of length 1.

The NSD metric uses block lengths to calculate values. The distance equation then becomes

$$NSD = \frac{\sum_{j=1}^{N_I^B} W_I f(b_j) + \sum_{k=1}^{N_D^B} W_D f(b_k)}{W_{max} L_E}$$

where N_I^B and N_D^B are the numbers of insert and deletion blocks, b is a particular block length, $f(b)$ is a cost function applied to a block length b, and all other terms are the same as in the SSD metric. Note that the weights W_I and W_D could be pulled into the cost function f, but we have left them outside to more easily compare the NSD and SSD metric equations.

The definition of the cost function f is an additional tuning parameter in the NSD metric. A rather natural function to use would be an exponential one, such as

$$f(b) = e^{k(b-1)}$$

where k is a constant and is the actual tuning parameter. This equation yields 1.0 for a block length of 1, so if all blocks are kept to a length of 1, then the NSD equation reduces to the SSD equation, as expected. The cost function yields exponentially increasing values for blocks greater than 1. Notice that for $k < 0.7$ and a block length of 2, the function would cause the distance value to be less than the corresponding value given by the SSD metric, which is not what we want. For this reason, we only consider $k > 0.7$ so that the value produced by the NSD metric is always greater than the value produced by the SSD metric for blocks of length greater than 1.

An important question to ask about an NSD measure is whether it results from many short blocks or a few long blocks. We could interpret many short blocks as meaning that there are mostly localized discrepancies between the model and the execution, whereas a few long blocks means that there are some major differences. To answer this question, we make two calculations, one with the tuning parameter k small and the other with k large, and view the ratio between the results as a measure of the relative number of longer or shorter blocks contributing to the distance measurement. This works because if most of the blocks are small, the difference between the measurements with the two values of k would also be small.

Unlike the SSD metric, the NSD metric is unbounded on the high end, although bounded by 0 at the low end. Thus, it is harder for us to say what value might represent a good correspondence between model and execution and what might represent a bad correspondence. We can, however, derive some values from the rules of thumb we used for the SSD metric (i.e., the 0.2 cutoff for good correspondence and 0.5 for moderate correspondence). What is needed for the NSD rules of thumb is a notion of the average block length that could be expected in an event stream with good correspondence to the model. With this defined as B_{avg}, our derived cutoff for good correspondence for the NSD metric is

$$C = \frac{0.2e^{k(B_{avg}-1)}}{B_{avg}}$$

This takes the SSD good correspondence cutoff of 0.2 and weights it according to the exponential weight of the average expected block length, taking into account the tuning parameter k. For example, if one sets $B_{avg} = 2.5$ and $k = 1.5$, then the cutoff value for good correspondence would be $C = 0.76$. This value nicely reduces to the SSD cutoff value for $B_{avg} = 1$. For the moderate cutoff value, we would use 0.5 in place of 0.2.

4.3 Example Use of the Metrics

To illustrate the various metrics introduced above, we use the Test Unit task from the ISPW 6/7 process problem [81]. This is a very simple and small process fragment, but it should give the reader

Example 1		Example 2		Example 3		Example 4		Example 5	
execution	model	execution	model	execution	model	execution	model	execution	model
co	co	co	co	co	co	co	co	co	co
make	make	make	make	make	make	make	make	make	make
exec	exec	exec	exec	make		exec	exec	exec	exec
diff	diff	diff	diff	make		diff	diff	diff	diff
exec	exec	exec	exec	exec	exec	exec	exec		exec
diff	diff	diff	diff	diff	diff	diff	diff	diff	diff
tcov	tcov	exec	exec	exec	exec	exec	exec	exec	exec
ci	ci		diff	diff	diff	diff	diff	diff	diff
mail-m	mail-m	tcov	tcov	tcov	tcov	tcov	tcov	exec	exec
		mail-t	mail-t		ci	mail-d		diff	diff
				mail-m	mail-m		ci		tcov
							mail-m		ci
								mail-d	mail-d

Table 4.1: Example Pairs of Execution and Model Event Streams.

a feeling for how the metrics are applied to a process. In this task, a developer and a tester are involved in testing a module that has undergone some change. They are to retrieve the test suite from configuration control, build the test executable, run all the specified tests, and make sure that at least a 95% code coverage has been achieved by the tests. If a failure occurs, either because the new module has an error or the test suite needs updating, then they are to notify the module developers or test developers, as appropriate. On a successful completion of the tests, they are to store the test results under configuration control and alert the manager to the new status of the module.

Figure 4.4 shows a colored Petri net model of this process. Circles denote places and rectangles denote transitions. Tokens have attributes (i.e., are colored) and those attributes are used by transition predicates to deterministically control the transition firing. Thick rectangles correspond to transitions that are event sites in the model and are labeled with the event that is produced at that site. Thin rectangles correspond to (internal) transitions used to control the model but that are not themselves event sites. To keep the figure simple, we collapse the begin/end event pairs of an activity into one (pseudo) event type; each event site can be thought of as a two-transition sequence with the first producing the begin event and the second producing the end event. We use familiar Unix command names as the names of event types.[3]

Table 4.1 shows five example pairs of execution and model event streams. A blank space in an execution event stream is a point at which the model has predicted that a particular event should have occurred, but in fact that event did not occur. Similarly, a blank space in a model event

[3]For those unfamiliar with the Unix command names appearing in the figure, "co" and "ci" are the check-out and check-in commands for a configuration management tool, "make" is a build tool, "exec" stands for the running of an executable (i.e., a test run, in this example), "tcov" is a test coverage tool,"diff" is a text differencing tool, and "mail" is an electronic mail tool.

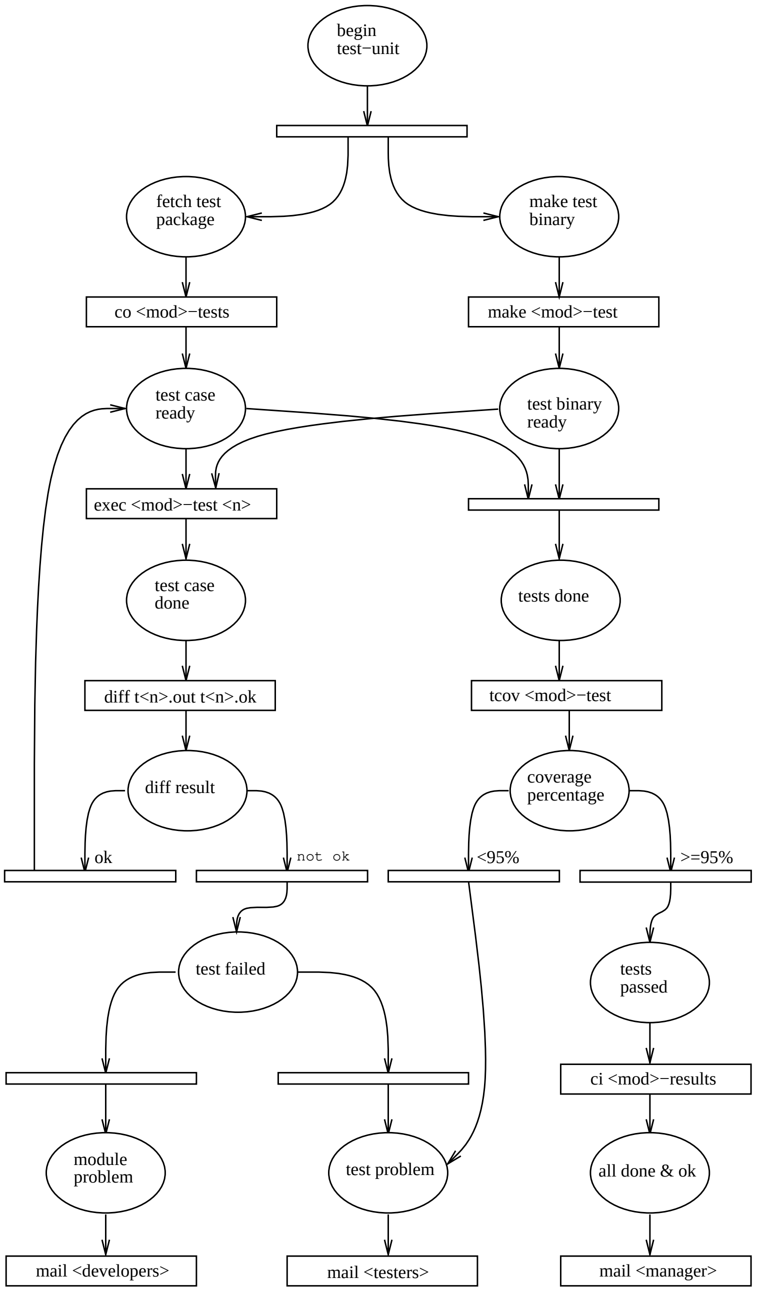

Figure 4.4: Petri Net Model of the ISPW 6/7 Test Module Task.

Stream #	# Ins	# Del	REC	SSD $W_I = 1$ $W_D = 1$	NSD $W_I = 1$ $W_D = 1$ $k = 1.5$	NSD $W_I = 1$ $W_D = 1$ $k = 3$	SSD $W_I = 4$ $W_D = 1$	NSD $W_I = 4$ $W_D = 1$ $k = 1.5$	NSD $W_I = 4$ $W_D = 1$ $k = 3$
1	0	0	Yes	–	–	–	–	–	–
2	1	0	No	0.11	0.11	0.11	0.11	0.11	0.11
3	1	2	No	0.30	0.55	2.11	0.15	0.21	0.60
4	2	1	No	0.30	0.55	2.11	0.23	0.47	2.03
5	3	0	No	0.30	0.55	2.11	0.30	0.55	2.11
Good Cutoff Values				0.20	0.45	2.01	0.20	0.45	2.01

Table 4.2: Example Event Stream Measurements for the Streams of Table 4.1.

stream is a point at which an event occurred in the execution that was not predicted by the model. Intuitively, such blanks correspond to either missed or extra activities in the process execution or to a mis-specification in the model. For example, in stream 3, the execution involves three consecutive invocations of the "make" tool, perhaps as a result of some problem performing the build, while the model predicts that only one should have occurred.

The SSD and NSD metric calculations require transformation of an execution event stream into the corresponding model event stream by means of insertion and deletion operations.[4] We apply an insertion operation at a blank in the execution event stream, and we apply a deletion operation at a blank in the model event stream.

Table 4.2 shows various validation measurements for the event streams in Table 4.1. Each row contains measurements for the correspondingly numbered example event stream. The first two columns give the raw number of insertions and the raw number of deletions needed to transform the execution event stream into the model event stream. The third column gives the results of the simple REC metric; only the first pair of streams yields "yes", since they are identical.

The last six columns of Table 4.2 give the results of the parameterized string distance calculations. We vary the relative weights of W_I and W_D for both the SSD and NSD metrics, and vary the exponential constant k for the NSD metric. We present cases where the weights are equal ($W_I = W_D = 1$) and cases where the insertion cost is weighted heavier ($W_I = 4W_D$) to highlight missed events in the execution.[5] The exponential constant k for the NSD metric is given values 1.5 and 3 to show the magnitude of change and the unboundedness of the metric. The last row of the figure shows the cutoff values for the "good" correspondence rules of thumb for each metric; values in a column that are less than the bottom row fall into what we would call a good correspondence between the model and the execution event streams. For the NSD metric cutoffs, the average

[4]Recall that we chose to use the execution event stream as the one to which the transformation operations are applied (see Section 4.2.2).

[5]The cost ratio given by $W_I = 4W_D$ was chosen arbitrarily for this example. In practice, a process engineer would explore various ratios that might best reflect the actual situation they are analyzing.

expected block length, B_{avg}, is taken to be 2.

There are several interesting things to see in the measurements presented in Table 4.2. The first observation is the similarity of values for the three columns with $W_I = W_D = 1$; for streams 3, 4, and 5, which all have one block of length 1 and one of length 2 (though in different operation combinations), these measurements do not differentiate between these errors. On the other hand, if one looks at the measurements with $W_I = 4$ and $W_D = 1$ (the last three columns) for event streams 2 and 3, one can see the effect of weighting insertions heavier; for the SSD metric, the measurement only changes by 0.04 with the addition of the two deletes (in stream 3), and still remains well within the good correspondence range for the NSD metrics. For stream 3, the SSD with $W_I = 4$ also produces a measurement that is in the good correspondence range, whereas the SSD with $W_I = 1$ is in the moderate range (0.2–0.5). Since stream 3 has just one insertion, like stream 2, this weighting better reflects the correspondence of the execution streams than does the $W_I = 1$ weighting.

For event stream 4, where the insertions have a block of length 2 rather than the deletions as in stream 3, the last three measurements (with $W_I = 4$ and $W_D = 1$) are significantly greater than for event stream 3. This shows how the metrics can be tuned to place importance on insertions—that is, on missed events in the process execution. Event stream 5 also shows this in the last two measurements; it has all insertions, and the difference in the weighting of insertions is evident in comparison to event streams 3 and 4.

The example presented here illustrates how the metrics we have defined can be used to quantify the correspondence between the process execution and the process model. We have also shown how the parameters of the distance metrics can be used to tune the measurements that result.

4.4 Extensions to the Metrics

In this section we present several important extensions to the metrics described earlier. These extensions enhance the usefulness of the metrics in terms of practical considerations, and are part of the implementation of the methods, as will be described in Section 4.6.

4.4.1 Extensions to the Weighting Model

We have presented these metrics with weights for the separate transformation operations, namely insertion and deletion. This was because the two operations represent the very different concepts of missed and extra activities, respectively.

However, event types also represent different things, and counting each event type as the same does not reflect reality. For example, a code inspection event may be very important for a small-change process; the organization that requires this action before a change is accepted would like to have a mechanism for easily detecting when it has not happened correctly.

Event type weights, separate for insertions and deletions, solve this problem. The global insertion weight and deletion weight is still specified, but only as the default weights to use. For any event type, one can then override the default with its own specific weights for one or both operations.

In the above example, the process engineer could specify a very high insertion weight for the code inspection event type. This would, in effect, make it more likely for transformations to take place around a code inspection event in the execution event stream, in order to make sure that that event corresponded to the model event stream, rather than to simply treat it like any other event. If there was no code inspection event, or if it was very far from its correct place, then it would end up getting inserted, and the measured correspondence would be low because of the high insertion cost for this event, and the process engineer would see this in the measurement.

Previous work in other areas using string distance measures have also found the need for per-symbol weights [8, 79].

4.4.2 Usability Enhancements

The simple and non-linear string distance metrics are naturally decomposable in a hierarchical fashion, to be able to give more information about the measurement than just one number for the whole process model. There are also "side effect" calculations which can be made to help interpret the distance metrics.

The easiest calculation to show that would help in understanding the distance metric is a simple count of the number of event insertions and deletions that were calculated. This simple count would help in understanding the distance calculation, especially when the weights W_i and W_d are different.

For the exponential string distance, an accompanying histogram of the block sizes of the operations would also be a useful tool in understanding the dynamics behind the final measurement.

One metric decomposition is over time; a graphical representation of the distance errors over the event stream could show in detail the areas which were most in error; for very long event streams this could be condensed using data visualization techniques (like using one pixel-line for each event, and displaying the event stream with the differences vertically on a screen).

In addition to simply counting the insertion and deletion operations, and their blocks, each operation can be related to some event site in the model. By keeping track of the operations' positions in the model, one can relate the areas of the model to the correspondence of the event stream.

4.5 Producing a Process Model Event Stream

Our validation metrics, as presented so far, assume the existence of two event streams, the execution stream and a model stream. However, we really have one stream (the process execution stream) and a formal model. Thus, we must derive an event stream from the formal model. However, a formal model of anything but the most trivial process likely leads to a large, if not infinite, number of such event streams.

Moreover, because we are measuring correspondence, we need to derive a model event stream that most closely approximates the execution event stream, in order to get as accurate a measure as possible. By 'closely' we mean the model stream that produces the lowest validation measurement. In addition, this closeness is not static, but depends on how the SSD and NSD measurement equations are weighted; different weights on the insertion and deletion operations (and on the event types) will affect which model event stream is the closest. In fact, the model streams may be different for the two metrics; for example, the NSD with a high block penalty may favor a correction of five disjoint events rather than a block of four events.

Thus, in general, producing a model stream to measure correspondence against is a hard problem. There are several areas which have addressed similar but not exactly the same problems; we discuss these in the next section.

4.5.1 Related Work

In this section we explore three areas of computer science that have dealt with similar problems to our model stream generation problem.

Parsing

Programming language parsing encounters a similar problem when trying to recover from syntax errors. When compiling a program and encountering a syntax error, the compiler must do two things: produce a reasonable error message by locating the error, and recover from the error so that it can continue to parse the rest of the program. This research has produced several methods:

- The simplest recovery technique is a panic; in this mode, the parser skips forward in the input token stream until it finds one of a special set of key symbols, such as `begin`, `end`, or a semicolon. Once this symbol is found, the current parse stack is popped until that symbol can be accepted. This method assumes the language has such a set of special symbols, and it makes no attempt at a minimal recovery, only deleting symbols until it can recover.

- Aho and Peterson [2] show a cubic algorithm for performing global minimum cost error correction in terms of token insertion and deletion. They augment the language grammar with error productions, and modify the parse algorithm to do corrections. They do not expect that their algorithm to be used, because of its high cost; they only propose it as a baseline for other methods to compare against.

- Röhrich [102] describes an error correction method biased towards insertion of symbols, arguing that as little of the program text should be skipped (deleted) as possible. This method is based on the ideas of minimum distance corrections, but makes the assumption that one never needs to back up in the input stream to find a good correction.

- Fischer and Mauney [52] describe a method for locally least cost error correction. They are also biased towards insertions, but include deletions. Their method will back up in the current parse stack, and uses a local search with a priority queue to find a locally minimum cost fix. Additionally, their method uses per-symbol insertion and deletion costs, so that the correction behavior of the parser can be fine-tuned for the input language. They show that their method is fast enough to reasonably implement in a compiler.

These techniques have been specifically developed for programming language parsing, and for the grammars that are used in this domain. Some of the critical assumptions, such as not backing up in the input stream (or only looking at the parse stack) do not always hold up in the software process domain; in fact, many modeling languages have no notion of a parse stack. Thus, in general, these techniques are not extendible to our general validation problem, although they demonstrate two things: optimal solutions are cost-prohibitive and heuristics can be effectively employed in practice.

Model Checking

Another body of research called model checking has created several methods that enable checking very large models (in terms of their state space) for inherent properties, including whether a behavior is allowed by the model.

In one example [27], Burch, et.al. describe a model checker based on binary decision diagrams that is able to check models with 10^{20} states, where previous work had only handled 10^{8} states. While this is impressive, the models they were analyzing were of a pipelined ALU, which has many self-similar states because of the width of bits that make a value. Their model checker directly represents this regularity in the state space, so that they avoid, to a large extent, the state explosion. If models do not have regularity in the state space, they admit that their techniques will not provide much leverage.

Another example that more directly is concerned with matching a behavior to a model is Constrained Expressions [10, 44]. This is an elegant method that, given a model, a current simulation state of that model, and a desired event, can answer the question "Can this event be produced in the future?". The model is stated as a system of event sequences, specified by extended regular expressions (though the body of CE work provides automatic mappings from several standard modeling languages to these representations). This representation, along with the desired event or event sequence to find in the behavior, is reduced to a system of inequalities that are fed into an integer linear solver. The ILP system produces a binary answer, and some parameters when it finds a solution.

If the answer is "yes", heuristics are used along with the parameters from the ILP system to produce a plausible behavior that leads to the event [37]. This behavior constitutes the next sequence of model events. Unfortunately, if the answer is "no", Constrained Expressions do not help in determining a correction to the event stream or model state to continue the analysis of the rest of the event stream.

Our problem of validation needs techniques that analyze the model in the continual presence of deviations from the model; thus, it appears that techniques like Constrained Expressions are not applicable. The problem is, while they leverage system transformations to gain speed and scalability, these transformations make the system inherently uninspectable and their analysis methods only produce binary results.

Regular Expression Matching

In [82, 93], Myers, Miller, and Knight describe algorithms for approximately matching a string to a regular expression, using the insert, delete, and substitute operations. These methods build on the dynamic programming techniques of the string to string comparison algorithms, and extend this to regular expressions. For a string S of length M and a regular expression R of length N, they make M copies of a finite automaton implementing R, and connect these together in a correspondence graph that allows them to define a dynamic programming recurrence relation. For simple operation and symbol weightings, equivalent to our SSD metric, their algorithms operate in $O(MN)$ time.

However, dealing with multi-symbol blocks (or gaps), as our NSD metric requires, complicates matters significantly. In general, for both string to string comparisons [48] and string to regular expression comparisons [93], arbitrary cost functions for blocks require at least $O(MNmax(M,N))$, or cubic time. Better results can be obtained if one assumes concave block costs, where $F(B_i) - F(B_{i-1}) \geq F(B_{i+1}) - F(B_i)$, i.e., the difference between the cost of a block of length $i+1$ and i is non-increasing as i increases. With this assumption, the matching with regular expressions take $O(MN(logM + log^2 N))$ time, but which also takes $O(MN + Nlog^2 N)$ space. Our NSD metric, in general, does not have concave block costs; in fact, in our presentation we use convex costs because a longer block represents a more serious deviation from the process model.[6]

The regular expression algorithm takes advantage of the simplicity of its modeling paradigm. In general, constructs used in process modeling languages are not reducible to regular expressions. More powerful, yet still restricted, constructs have been looked at. Context free languages, for example, are thought to have high-order polynomial time algorithms for solving approximate matching [82].

In general, these super-quadratic to cubic solutions, while providing optimal answers, are impractical unless one requires an optimal solution. They also generally have large constants in the actual running times.

4.5.2 Solving the Model Stream Generation Problem

This survey of techniques leads one to the conclusion that the general validation problem has no known efficient, optimal solutions; in fact, some formulations of the problem are known to not have optimal solutions that are efficient.

Fortunately, the problem is not quite so bad as it seems. We can use the execution event stream to guide derivation of the model event stream, thus significantly cutting down on the required search. And we expect that the execution stream does in fact correspond to some degree to the model. In particular, we can traverse the execution event stream and incrementally derive events for the model event stream by consulting the model. Moreover, the distance calculation can be performed

[6] In other areas a block cost function is naturally concave. For example, in DNA matching, the high cost of physically breaking the sequence means that as a block gets longer that breaking cost can be amortized over the length of the block. This results in a concave cost function.

along with the production of the model event stream, making it unnecessary to separately produce the model event stream and then feed it to an algorithm to find the measured distance.

This approach implies some kind of state-space search method, and implies using a heuristic-driven method to control the state explosion. AI research has provided several search methods that seem applicable; two that we describe here are *best first search* and *A^* search* [71, 101].

While the standard depth first and breadth first searches of a tree of states are exhaustive in a single dimension, best first search is a heuristic-driven search that determines its search path by following the lowest cost path in the state space. For each state S with a parent S_p, a cost is estimated by

$$EstimatedCost(S) = Cost(S_{start}, S_p) + Cost(S_p, S) + Estimate(S, S_{goal}),$$

that is, the total estimated cost is the known cost of getting from the start state S_{start} to S_p, plus the new known cost of getting from the parent S_p to the new state S, plus an estimate of the cost getting from S to a goal S_{goal}. The heuristic comes in estimating the cost of going from S to a goal state S_{goal}.

The best first search uses a priority queue of states to be evaluated, and always evaluates the lowest cost state on the priority queue. When it reaches a goal state, one can either stop or go through one more iteration of states to make sure that the found goal is not likely to be usurped by a lower-cost goal. This method is not guaranteed to find a minimum cost solution unless the heuristic estimator can be proven to always underestimate the true cost. If this is true, then the first goal found will always be a lowest-cost goal. Without adding estimations of the cost to a goal state, this method is called uniform cost; it also always find a lowest cost goal. Unfortunately, both of these methods that find lowest cost goals are guaranteed to inspect every state that is lower cost than the goal.

The A^* algorithm is similar to best first search, but it operates on a graph of states rather than a tree of states. This property makes its interface to the priority queue more complex, since if a new parent of an existing child has a lower cost than the old one, the child's cost must be updated.

Because we need to maintain a state sequence that is a unique set of event stream operations and model states that produce the closest model stream for the given execution stream, our state space is a tree of states. Thus, the best first search method is the most applicable one to solving our problem, and it is the one that is used in this thesis.

Since we are trying to calculate minimum cost distance metrics, it is natural to use them to assign actual costs to the states during our state search. In fact, we must use the metrics as actual costs in order to guarantee we are minimizing the correct function. But the question of how to estimate the cost of reaching a goal remains.

In general, we will not know the total length of the event stream[7], thus we cannot rely on knowing what proportion of the event stream we have seen so far. The SSD and NSD metrics are

[7] For example, a real-time stream coming from a currently executing process does not have a known length, or even a defined end.

already normalized with respect to the length of the event stream, so that the position in the event stream is factored out of the cost assigned to a state. Thus, we might say that the estimated total cost of reaching a goal state is just the value of the SSD (or NSD) metric that has been calculated so far, i.e., the cost of the current state. This estimation assumes that the event stream processed so far is representative of the total stream, and that the metric calculation will not change too much.

4.5.3 Pruning

The technique of pruning the state space has been proven to be useful in reducing the cost of finding a low-cost goal. Pruning is the act of throwing away sections of the state space that look unpromising, and never going back to look at them, no matter what happens to the outcome. By pruning, one cannot guarantee a lowest cost goal, but smart pruning, in some domains (such as game playing), has shown that it has negligible effects on the outcome of the search while dramatically reducing search costs [101]. Pruning can take many forms, and can use vastly different methods and heuristics.

One pruning method is to throw away any newly generated state that has an estimated cost higher than some threshold relative to the current best-looking state. The hypotheses behind this are that the estimated costs are fairly accurate, or at least predictable, and that a state's actual cost is not likely to be vastly better than its estimate. Additionally, it assumes that one can set a single threshold for the whole state space, and that this will work consistently.

In reality, our initial observations showed that the variability in costs assigned to states changes over time; in the beginning, especially, there is much larger variability. Thus, for this pruning mechanism, we set a threshold as a fraction of the current standard deviation of state costs. This lets the threshold account for some of the variability during the state space search.

This method could, and probably should, also be applied periodically to the old but not-yet-examined states on the priority queue.

Another method is to prune any state (new or old, as above) that is some specified distance behind in the execution event stream from the current farthest state. This heuristic assumes that the most likely paths to the lowest cost goal state will be examined in an interleaved fashion; i.e., their costs will not fluctuate too widely from each other, and thus the paths will be expanded close to each other. Then, any unexamined state that is far enough behind in the event stream is thrown out as not likely to be on a path to a good goal state.

We have examined both of these pruning techniques for our validation problem, and evaluate them in Section 4.7.

4.6 Implementing the Validation Methods

In the scope of this thesis, it is important to actually understand the practical uses of the proposed validation methods. To this end, we have implemented the work described in this chapter, and describe it here.

```
object PriorityQueue of State;
object EventStream isa ProcessEventStream;
set NewStates of State;
object CurState isa State;
object Model isa ProcessModel;
object CostModel isa OperationCostModel;
object Pruner isa PruningMethod;
boolean Done ← False;

PriorityQueue.AddSet (Model.InitialStates (EventStream));
while (not Done)
{
    CurState ← PriorityQueue.BestItem();
    if (Pruner.ShouldPrune (CurState))
    {
    }
    else if (Model.IsFinalState (CurState))
    {
        Done ← True;
    }
    else
    {
        NewStates ← Model.Successors (CurState,EventStream));
        foreach State in NewStates
            State.Cost ← CostModel.MakeCost (State);
        NewStates ← Pruner.PruneSet (NewStates);
        PriorityQueue.AddSet (NewStates);
    }
}
MakeModelStream (CurState);
```

Figure 4.5: Validation Calculation Algorithm.

4.6.1 The Validation Engine

Our method, then, is to use a pruning, best-first search engine to find the model stream using the execution stream, and also calculating the metrics at the same time.

Figure 4.5 shows the psuedocode algorithm for the validation engine, and Figure 4.6 shows the data flow for the algorithm. The algorithm is described in terms of a generic object-based imperative language. This algorithm is just a pruning, best-first search. A priority queue of generated but not

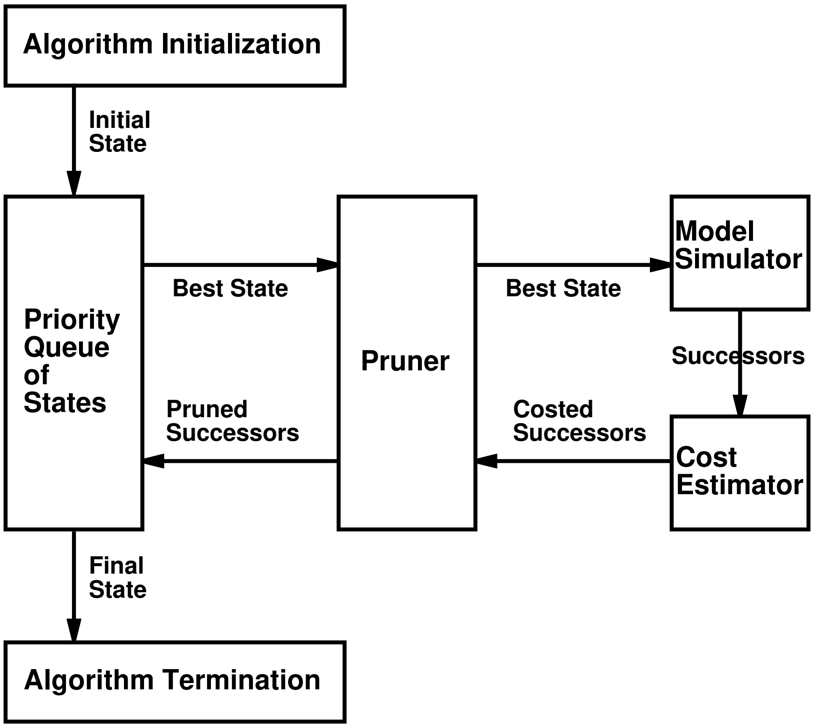

Figure 4.6: Data Flow in the Model Simulator.

yet evaluated states is maintained such that the lowest cost state is at the top; the algorithm takes the lowest cost state, has the model generate its set of successor states, prunes them, and puts the successors on the priority queue. It finishes when the state it pulls off the priority queue is a final state, or when it processes the last event.[8] Pruning takes place both when the state is pulled off the priority queue and when the successors are generated.

One of our goals was to provide a mechanism whereby the modeling paradigm and language are abstracted from the validation method. To do this, we created an abstract model interface, for which a specific modeling paradigm can be plugged in. The only requirements are that:

1. The model can specify a compact representation of its state, and
2. Given a state and a position in the execution event stream, the model can produce a set of successor states that are constructed by applying one of match, insert, or delete to the event stream.

[8]The model state may be arbitrarily far from a final state, so always searching for a final state may be very costly. In practice, this should be an option for the user to control.

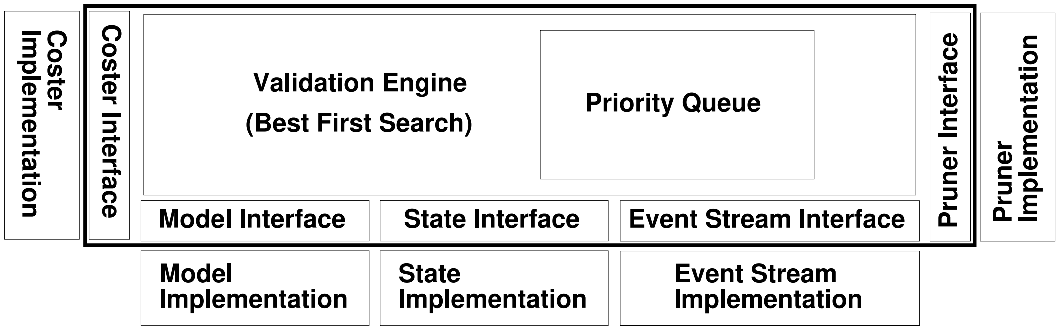

Figure 4.7: Architecture of the Model Simulator.

Virtually any modeling paradigm can satisfy these two requirements, though those that are control-flow-oriented, such as state machines, statecharts, and Petri nets would be easier to fit in.

The only question is a compact representation of state; still, for complex models such as Petri nets, where the state is the marking of the net, a state could be stored as a difference from its parent. In reality, this compactness is only required because the search algorithm will maintain a possibly large set of outstanding states to look at; however, current hardware platforms often can provide hundreds of megabytes of memory, so that in affect this compactness is not a strong requirement. Additionally, we explore the needs of the search engine in terms of space usage in Section 4.7.

4.6.2 Implementation of the Validation Engine

We have implemented this validation method in both a command-line and a window-based tool, in **C++** and **Tcl/Tk**. Figure 4.7 shows the architecture of the implemented validation engine. The control is centralized around a best-first search loop, built on top of a priority queue of states. The main goal of the architecture is to have a flexible, replaceable, component-based system. The modeling paradigm, represented by the *Model* and *State* objects, is particularly important to have replaceable. However, we also wanted flexibility with the event stream, pruning method, and cost model. To this end, this implementation uses (sometimes virtual) base classes as abstract interfaces to these components. An implementation of any of these components is subclassed from the interface, and the instantiated component is then used in the validation engine.

As example interfaces, we show the most important ones in Figure 4.8, the Model and State classes. These two classes are the interface to a modeling paradigm that is instantiated for this tool. The State base class is not abstract, in that it implements standard requirements of the validation engine, such as maintaining a parent pointer, a stream position pointer, and a cost. A subclass of State only needs to implement is its own state information and a method to access the event associated with that state. The Model base class is abstract, however, because there is no general information needed by the validation engine that is common across models. The interface to a model is simple, and is mostly in terms of states and sets of states: it needs to identify initial and final states, and, given a state, produce a set of successor states for the validation engine.

VALIDATION is the process validation tool of BALBOA. It lets one quantitatively and qualitatively compare an execution (collected) event stream to a process model and see how many and where deviations occur between the execution stream and what the model expects. Figure 4.9 shows a snapshot of VALIDATION.

As can be seen in the figure, VALIDATION gives the user both the quantitative metrics view of the validation, and also a detailed view in the scrolling areas showing each event stream. In the metrics view, the upper portion of the window, there are three levels of detail: the count of the individual insertions and deletions (in the upper middle); the count of the blocks of insertions and deletions (in the lower middle); and finally the calculated distance measurements (on the right side). VALIDATION also makes available the parameters used in the distance calculations (on the left side of the window), so that the user can interactively change the parameters and see the change in the calculations. The lower portion of the tool shows the detailed differences between the execution and what the process model expects. The execution stream is displayed on the left and the generated model event stream on the right. Events that need to be inserted into the execution stream to have it correspond to the model are colored blue (light grey bars), and those that need to be deleted are colored red (dark grey bars).

In order to do the validation, VALIDATION is given an event stream from BALBOA, and a process model specified by the user. VALIDATION then spawns a validation process that analyzes the model using the event stream, and produces a model stream that corresponds to the validated event stream. Additionally, VALIDATION can spawn a DOTVIEWER window for the convenience of viewing a process model during the validation exercise.

Additionally, VALIDATION allows the user to control the validation engine through selection of the cost model and pruning methods and parameters.

Overall, the validation engine is about 2100 lines of non-commentary lines of C++ source code and the interface is about 700 lines of non-commentary Tcl/Tk source code. This does not include standard code to connect to a BALBOA server or read from local event streams, and to interpret events according to their mapping.

```
class State
{
  public:
   State();
   virtual ~State();
   // standard, already implemented methods
   virtual State *Parent(State *P=0);
   virtual Operation CreateOp(Operation Op=0);
   virtual int OpSequence(int *OpSeq=0);
   virtual Cost CostOf(Cost *C=0);
   virtual StreamPosition StreamPosOf(StreamPosition *Sp=0);
   virtual operator<(State &S2);
   virtual operator>(State &S2);
   virtual operator==(State &S2);
   // model and state need to implement access to the event
   virtual EventPattern AssocEvent(class Model *M,
                                   EventServer *Es=0)=0;
  protected:
   State *MyParent;            // parent state pointer
   StreamPosition MyStreamPos; // execution event stream position
   Operation MyOp;             // op (A/D/I) that created this state
   int MyOpSequence;           // count of sequence of same op
   Cost MyCost;                // total cost from initial to this state
   // add modeling-specific state data here (protected or private)
};

class Model
{
  public:
   Model() {};
   virtual ~Model() {};
   // all methods are virtual, and need instantiated
   virtual int InitializeFromFile(char *Filename) = 0;
   virtual SetOf<State*> *SimulationStep(State *S,
                                         EventServer *Es) = 0;
   virtual SetOf<State*> *InitialStates() = 0;
   virtual int FinalState(State *S) = 0;
   virtual int InitialState(State *S) = 0;
   virtual EventPattern EventOfTransition(State *From, State *To) = 0;
  protected: // no default data; this base class is abstract
};
```

Figure 4.8: Modeling Interface Based on Model and State Base Classes.

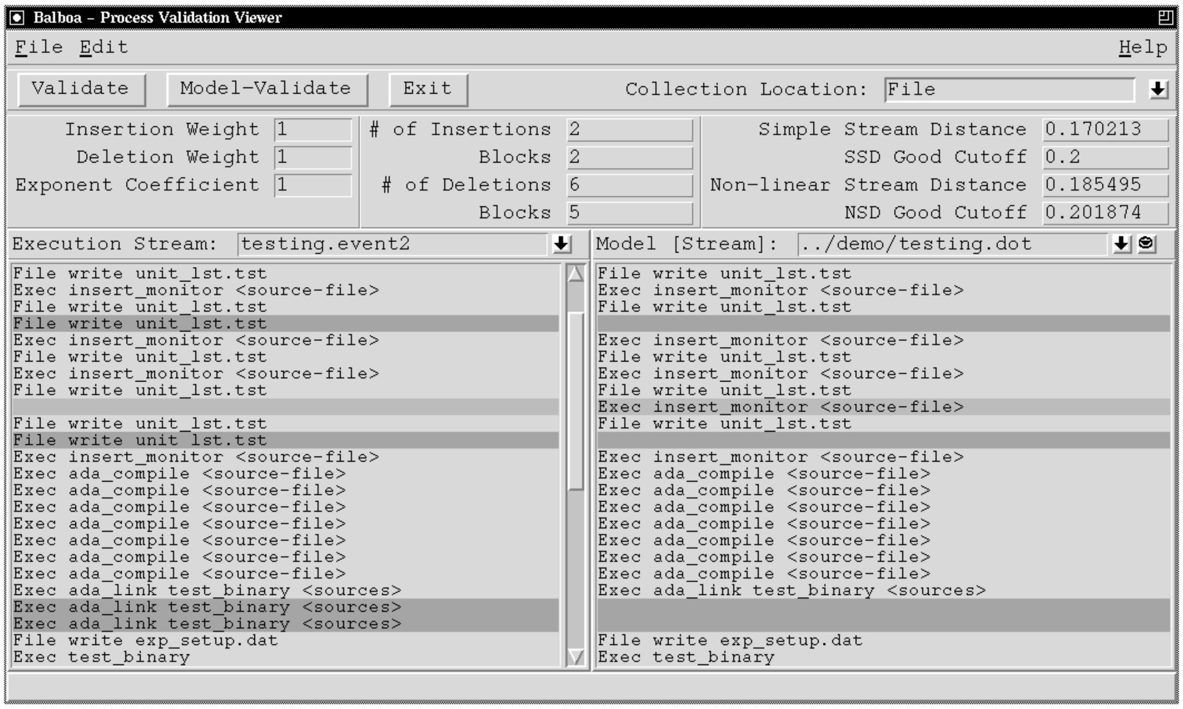

Figure 4.9: The VALIDATION tool.

Pruning Method	Parameter (variable)	SSD Metric	Time (Sec)	Size (KByte)	States Examined	States Pruned
None	-	0.058	100.0	10343	248049	0
Position	20	0.058	11.0	1818	29638	1178
Position	10	0.058	4.5	1181	13763	1394
Position	5	0.058	3.1	976	8828	1591
Cost	60	0.058	107.5	9912	248049	10631
Cost	40	0.061	82.0	7502	200008	24905
Cost	20	0.104	17.9	2409	45523	2124

Table 4.3: Performance of Pruning Methods on a 1250-event Event Stream.

4.7 Evaluation of Validation Metrics

In this section we evaluate the performance of our validation methods. Particularly, we look at the how performance of the search, the pruning, and the cost model affect the results in terms of speed, space, and correctness. We look at the two pruning methods that were proposed in Section 4.5.3, cost-based and position-based pruning. The cost pruning is specified as a parameter that is a percentage of the best state's cost, and position pruning is specified as a maximum look-back amount from the furthest state.

Table 4.3 shows the performance of VALIDATION on a stream of 1250 events being validated against a model with 53 states, shown in Figure 4.10. This model is not meant to represent a specific software process, but is complex and large enough to be used to explore the performance aspects of VALIDATION. We defer the analysis of VALIDATION on real-world process data to Chapter 6.

The table shows the performance of VALIDATION using no pruning, position pruning with 5, 10 and 20 position look-back, and cost pruning with 20, 40, and 60 percent cost thresholds. As the SSD metric shows, even the position pruning of 5 finds the optimal model stream, while the cost pruning performs very poorly, not finding the optimal model stream until it searches the same amount of states as no pruning. In terms of time and space performance, position pruning does very well, whereas cost pruning does poorly in all dimensions.

This appears to support the hypothesis that is behind position pruning, which is that the search paths that contain good solution states are expanded concurrently, and pruning based on position in the event stream is effective at both controlling the search and finding a good solution.

During our experimentation, cost pruning was never shown to perform well. This is because it has pruned the state space subtree that contains the optimal solution, and gets lost in flat areas of the state space, where all states have similar costs, and thus will not be pruned. This leads us to question the premise that the variability of state costs is similar from the beginning of the search to the end. Our cost pruning results seem to say that setting a single threshold is not effective at controlling the search or in finding good solutions. From here on we explore the behavior of only

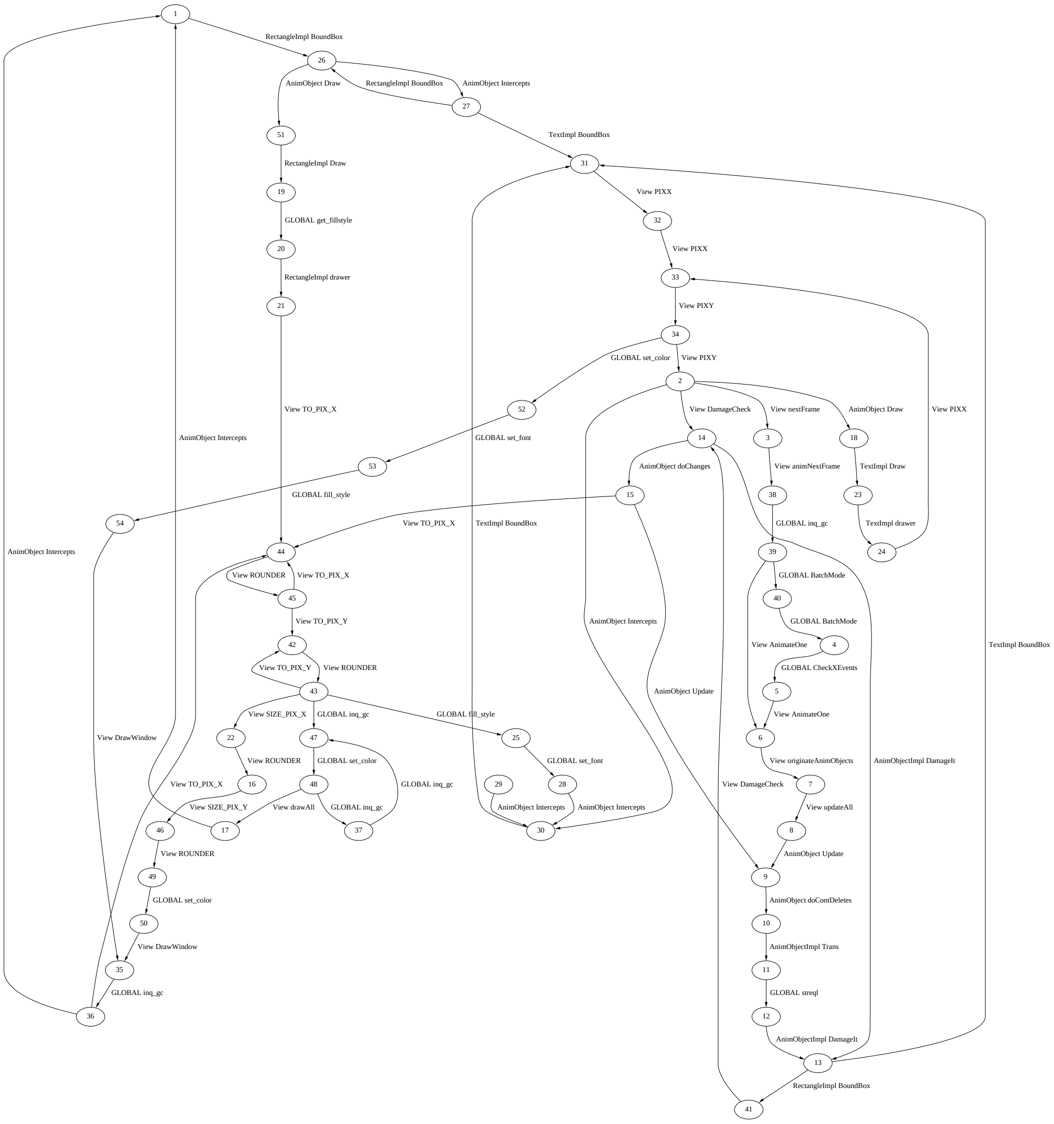

Figure 4.10: Model Used for Performance Evaluation of VALIDATION**.**

position pruning.

We expect that the more distance there is between the execution event stream and what the model expects, the harder it will be to find the optimal model stream, since the algorithm needs to do more model searching. Table 4.4 shows the performance of VALIDATION with position pruning across streams that are successively less correspondent to the model. As can be seen on the

Event Stream	Pruning Parameter	SSD Metric	Time (Sec)	Size (KByte)	States Examined	States Pruned
Closest	20	0.058	11.0	1818	29638	1178
Medium	20	0.149	93.7	8666	242357	47251
Closest	10	0.058	4.5	1181	13763	1394
Medium	10	0.149	18.3	2411	53911	11483
Furthest	10	0.305	66.2	7219	176380	15068
Closest	5	0.058	3.1	976	8828	1591
Medium	5	0.165	6.5	1402	24211	6032
Furthest	5	0.340	13.2	2053	45066	11668

Table 4.4: Performance of Position Pruning Methods Across Higher Deviating Event Streams.

Stream Length	SSD Metric	Time (Sec)	Size (KByte)	States Examined	States Pruned
635	0.051	1.5	814	4086	657
1250	0.058	3.1	976	8828	1591
2501	0.058	5.7	1259	17720	3234
5002	0.058	12.3	1855	37325	8275

Table 4.5: Performance of 5-Position Pruning Across Event Stream Lengths.

SSD metric, the 5-position look-back does not find the optimal model stream for the two less correspondent ones, but it does not significantly err on the SSD metric, staying within 15% of the best calculations. Pruning with a 20-position lookback on the furthest deviating stream did not contain the state-explosion, and it failed to complete on the machine used for these tests.

In running these tests, a guideline for quickly deciding whether or not the resulting measurement can be expected to be close to the truely optimal one has been discovered. If the blocks of deviations between the model stream and the execution stream are often larger (or just sometimes much larger) than the look-back parameter for position pruning, then it may be that the measurement is not close to optimal; if, however, the deviations are smaller than the look-back parameter, then it is likely that the measurement is optimal or close to it.

The above observation points to the fact that position pruning is well-behaved, in the sense that by looking at the output (i.e., the sizes of the deviations), one can understand approximately how well the algorithm performed. This makes it very usable, unlike cost pruning, where the pruning parameter has no direct relation to the result.

Table 4.5 shows the performance of VALIDATION over different event stream lengths, using 5-

position pruning. The event streams are all compared against the same model, and essentially are the same, the longer ones just being concatenated copies of the shorter ones. Position pruning shows linear performance over the different stream lengths, which is what one would expect since it limits how far back it will look.

We conclude from this evaluation that position pruning is effective, and allows VALIDATION to be used on significantly large inputs, while still performing well.

4.8 Summary of Process Validation Work

In summary, we have proposed and demonstrated several metrics for process validation. These metrics range from simple exact matching, to a linear distance measure in terms of event insertions and deletions, and finally to a non-linear distance measure that takes a broader view of event comparison. The metrics are independent of the specific process modeling paradigm, and thus can be generally applied throughout the software process field. We believe that to truly have software process improvement, analysis techniques such as the ones we have implemented are needed.

Chapter 6 describes an industrial study that shows the usefulness of these metrics in a real world setting. This study shows that the validation metrics as proposed can in fact capture important information about how a process is behaving.

Computing these metrics is a non-trivial task; we have described a model-independent method for computing these metrics, and have shown that it is both practical and accurate.

Future work for process validation includes:

- Adding control over starting states and ending states to the validation engine. A user may want to validate a piece of behavior that neither starts at an initial state nor ends at a final state. The final state situation is currently handled, but the validation engine currently always starts at an initial state as specified by the model.

- Finding places in the model where an event stream can be pinned down could help reduce searches on large models. For example, a key event might happen at a specific site, where all previous behavior can then be ignored. This is similar to the concept of trace change points in [46].

- Implementing other modeling paradigms, such as Petri nets, for the metric-calculation engine.

- Developing techniques for better visualizing the measurements. For example, overlaying the differences onto the process model rather than onto the model event stream may help a process engineer in understanding the problems in the process.

- Identifying other properties of process models that can be exploited in doing validation calculations. For example, points in a model where one can fix the execution stream could help further reduce the search cost.

- Investigating other analyses for process executions and process models. Time-oriented metrics, for example, would be very a useful extension to execution stream analysis. Real-time systems analysis techniques could be useful here [51, 109]. Methods for measuring the efficiency of a process would be another useful analysis method. Both of these would help in the optimization of a process that has already been behaviorally validated.

- Formal models are often useful because they can detect conflicts and consistency violations. But with a sample behavior that does not exactly match the model, but has been validated, can we decide if the desired properties of the model still hold? This would be important future work, to show what properties hold under what deviations from a model.

Chapter 5

A System Framework for Process Analysis

Software process engineering has an advantage over other disciplines in that much of its activity takes place on computers; thus, it is more amenable to reducing the effort needed for process tracking and analyses. Indeed, in the past several years, there have been efforts to collect process data and analyze it to improve the process. This work, so far, has seen the creation of single tools that access process data in an ad hoc manner. Several methods for collecting process data have been proposed and constructed (e.g., [16, 23, 110, 121]); however, there has not been a significant effort to propose a coherent framework for tools and methods with which to do analysis from process data.

Such a framework would be useful in many ways. From the data access perspective, a framework that isolated the tools from the variety of data formats and provided a consistent access method to all of the data would reduce the effort needed to make a tool widely-usable with multiple data formats. Indeed, by providing access methods in libraries, tools could be constructed quite easily. From the data management perspective, a framework that provided for the management of data would eliminate the need to provide such (redundant) facilities in each and every tool, and for each and every data format.

This chapter describes such a framework, called Balboa, that supports the analysis of process executions and process models through the use of event-based data. We concentrate on event data because it supports a wide variety of behavioral and temporal analyses. Balboa supports the use of event data in two ways. The first is support for the managing and interpreting of data that has been collected. The second is support for the building of tools that fit into the Balboa framework.

This chapter has the following organization. Section 5.1 describes the motivation and related work, Section 5.2 gives a presentation of the high-level architecture of the Balboa framework and its platform of implementation. Sections 5.3, 5.4, and 5.5 describe the data management, data collection, and client tool interfaces to Balboa, respectively. Finally, Section 5.6 concludes with some insights and ideas for future work on Balboa.

5.1 Motivation

In [89], Lott gives an extensive summary of process support and measurement support in seventeen software engineering environments. Most process execution/guidance systems have little or no measurement support, and those that do tend to concentrate heavily on product measures to infer process characteristics; a few make use of aggregate process statistics, such as cumulative personnel effort and computer usage time. In most systems the type of data and how it was used was fixed; in general, no effort was made to allow extensions to the data collection and analysis methods.

The one system that can come closest to claiming the position of a framework is Amadeus [110]. It is flexible in the specification of what data to collect and what to do with the data. It is positioned as a supporting unit to a process-guidance system, so it is meant to support other tools. However, it is essentially a system for registering process events and triggering actions on the occurrence of those events. It does not provide further assistance in maintaining event collections, interpreting events, and providing those event collections to multiple tools. Thus it cannot be considered a framework for collecting, managing, and providing event data to a variety of analysis tools.

In other areas of event data application such as real-time, concurrent systems analysis, tools have had complete control over the environment, from the compiler and linker used to instrument the system to the tools that are used to analyze the data. For example, the PICL trace format standardizes the data format for tools that analyze parallel, message-passing systems [123]. Thus, they have not needed a general framework. Tools in the software process arena, on the other hand, must deal with event data from multiple, heterogeneous sources, and also with varied formats of the events. In this climate a framework such as BALBOA is needed.

In the BALBOA framework, we do not address the issue of providing mechanisms for collecting the event data, for two reasons. One is that there exists a variety of mechanisms for collecting such data already, as described in Chapter 2. The second is that the mechanisms for collecting such data are often site-specific, so that it is more flexible to allow event data to be sent to BALBOA from various sources. It also may be the case that a single project will have to use several different collection mechanisms; for example, a combination of automated and manual collection. It is because of this variety and heterogeneity that a framework such as BALBOA is needed, so that tools have consistent access to all kinds of data.

We also do not address the issue of data integrity; we assume that the data are correct (i.e., the events that are collected have actually occurred) and consistent (e.g., all "begin" events for an activity have a corresponding "end" event). If needed, tools could be built to help clean up raw event data.

From this summary, then, it can be seen that a framework for the analysis of process data is a useful addition to the ongoing process research, and will facilitate the application of the ideas of research in the industrial domain.

5.2 The BALBOA System Framework

BALBOA is a framework that meets the requirements above, providing for the management of

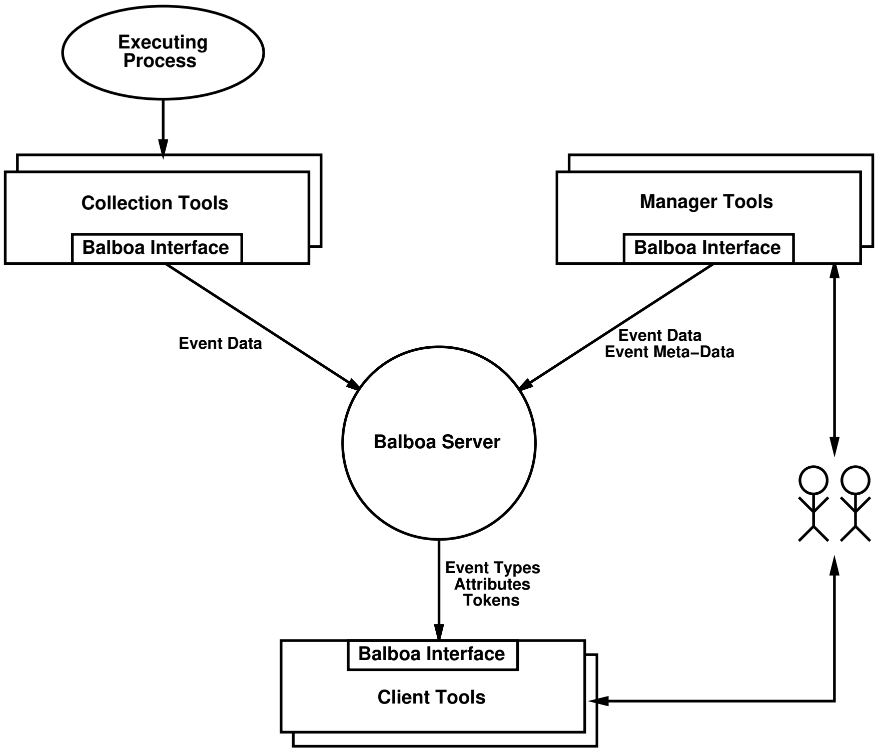

Figure 5.1: The Balboa Framework.

process data and for the facilitation of building analysis tools that use the data. Balboa provides a client-server foundation for tools, where a server provides client tools a uniform access interface to heterogeneous event data collections. Figure 5.1 shows a high-level view of the Balboa framework. Three access channels are provided to a Balboa server:

- A *data collection* interface allows site-specific collection tools to submit collected event data to a server.
- A *data management* interface allows management tools to manage event data and create meta-data for describing the make-up of event data.
- A *client tool* interface allows client analysis tools to access event data in a consistent manner.

Balboa is built on the Unix (Posix) platform, with a Balboa server being a daemon process. Balboa relies on TCP sockets as the communication channel for the tool interfaces. It uses the socket interface to communicate ASCII string (newline-terminated) messages between the tools and the server. Messages sent from clients are commands to the server, and the server replies with one or more data and/or status messages. Data messages are also sent to the server in the data collection and management interfaces. Using strings for messages removes the concern about different data type implementations on different machine architectures (i.e., endianness), but does introduce performance concerns. We do not, however, envision event streams that are extremely large and produced in a very short time, such as the hundred-megabyte or gigabyte traces that large parallel programs can generate in a short time (minutes or hours).

Balboa provides support for client tools in both the C++ and Tcl/Tk programming languages, thus allowing both the construction of robust, "polished" tools and the fast exploration of ideas using an interpreted prototyping language. Other languages, such as Perl, that have access to a socket interface could also be easily integrated into the Balboa framework. Both Discovery and Validation, the tools that implement the analysis methods in this thesis, are Balboa client tools. Another person is currently building a third client tool that will perform time and ordering queries on event data.

As shown in Figure 5.1, the client tools do not normally access raw event data; rather, they see *interpreted* data: event types, attributes and values, and tokens (event types mapped to integer identifiers). The manner in which the interpretation takes place is specified by the meta-data that the data management tools allow the user to create. This interpretation frees the tools from the specific format of each data source, and lets the tool builders concentrate on building the analysis methods rather than data manipulation.

Figure 5.2 shows the data flow in the Balboa system, from collecting event data from an executing process and creating meta-data describing the event data using data management tools, to providing interpreted data to client analysis tools.

On the user side, Balboa also provides the LaunchPad, a tool that acts as a central execution point for the various manager pieces of Balboa, and for individual analysis tools. Figure 5.3 shows a snapshot of LaunchPad. As can be seen, this tool is a simple button-oriented interface to launch the various management and analysis tools of Balboa. The LaunchPad is extensible in that analysis tools can be installed onto it; thus Balboa helps to manage the tools that use it as well as the data. The LaunchPad can specify a Balboa server as a default that is then inherited by all the tools as they are started up. This makes the interaction with one Balboa server transparent across all tools.

5.3 The Data Management Interface of Balboa

In order to make use of the event data, tools and users have to be able to manage and describe the data. Balboa provides a message interface and support tools for this purpose.

5.3.1 The Management Message Interface

The data management interface to Balboa supports creating, modifying, and removing event data collection specifications and event mapping specifications.

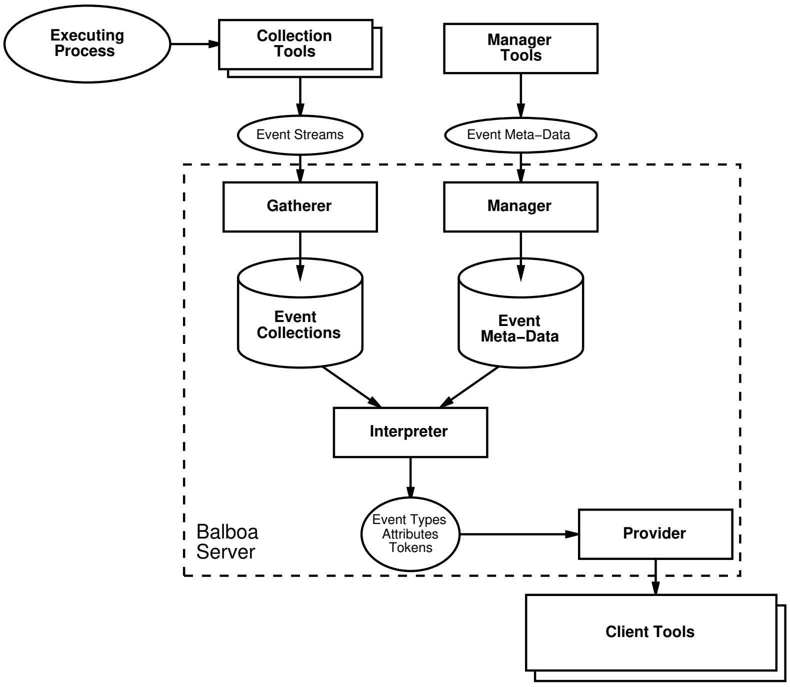

Figure 5.2: Dataflow in the BALBOA Framework.

The messages that implement the management interface are:

Message	Description
MsgMakeCollection	Create an event data collection
MsgRemoveCollection	Remove a collection
MsgGetCollectionList	Return the list of event collections
MsgGetCollection	Return the specification of a collection
MsgMakeEventMap	Create an event mapping specification
MsgGetEventMapList	Return the list of event maps
MsgGetEventMap	Return the event map specification

This interface has two basic parts, that of creating and controlling event collections, and creating and controlling event mapping specifications. The event maps are specifications that tell BALBOA how to interpret the raw event data. This interpretation maps each event into an event type and individual attributes with values.

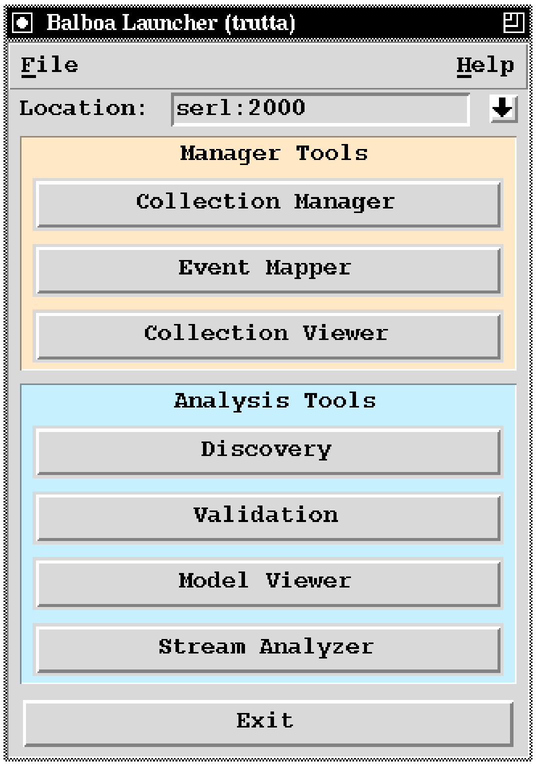

Figure 5.3: The LaunchPad Tool.

5.3.2 Specifying Event Maps

As described in Chapter 2, events collected in an event stream are complex data that have attributes. Tools will want to understand event types and the attribute values rather than just use the raw events. Thus, to make use of the event data, one must specify mapping criteria to map the events to specific event types, and to handle attribute matching. These mapping specifications are then used by an interpreter to understand the raw event data and extract event type and attribute values from it.

We do not assume that event streams are homogeneous, but rather that it is likely that they are non-homogeneous, coming from a variety of sources. The mechanism to interpret events, i.e., map them into event types and set attribute values, is a two-tiered one based on regular expressions and attribute values.

For a given event stream, an ordered set of regular expressions are specified for describing the events in that event stream. Each event is expected to match one regular expression. If it matches more than one, the first one it matches is used; if it matches none, then that event cannot be interpreted, which is an error.

The regular expression is piecewise bracketed by '{}' braces in order to recognize attributes of a matching event. Each piece of the regular expression inside the braces represents one attribute, and each part of the regular expression outside the braces represents the white space between the attributes. The attributes are then named in a naming specification rather than referring to them positionally. To specify a unique event type, a subset of the attributes of that regular expression are flagged as defining the event type. The unique values of the cross product of those flagged attribute types determine the event types in the event stream.

For example, suppose we have an event stream that has the event

```
FILE-WRITE code io.c jcook 15:34:21 09/11/93
```

in it. This event can be mapped by providing the specification

```
{.*}[ ]*{.*}[ ]*{.*}[ ]*{.*}[ ]*.*
1=sysop, 2=doctype, 3=doc, 4=user
type=sysop+doctype
```

where the first line specifies a simple pattern of four attributes separated by spaces, and ignoring all of the rest (the time and date), the second line names these attributes (`sysop`, `doctype`, `doc`, and `user`, respectively), and third line specifies the event type as the attributes `sysop` plus `doctype`. Thus, an event with `sysop==FILE-WRITE` and `doctype==design` would be a different event type from the one shown.

We represent the event type as the spaced concatenation of the flagged attributes; thus, the event shown above would have an event type of `FILE-WRITE code`. The attributes `doc` and `user` are variable within an event type; These are accessible to an analysis tool, and might be used, for example, to analyze a particular software module.

Our event mapping also supports an encoded version of the event type, called a token. A token is a single character or integer encoding of the event type. This encoding reduces a large event stream to a simple, small stream of characters (integers). Accessing a stream in this manner is useful because, for tools that might do pattern matching or something similar, the need to look at large amounts of data in a simple form is paramount. Rather than have the tool do the mapping, the server provides the mapping for the tool, and also remembers the reverse mapping from token to event type, so that user I/O can be accomplished using the understandable event type, and not the externally meaningless token.

5.3.3 Data Management Tools

In order to make use of the event data, tools and users have to be able to manage and describe the data. BALBOA provides several user tools to perform these functions:

COLLECTIONMANAGER is a tool that lets one specify a data source, its parameters, and other information. Once this is done, that data is considered a collection by BALBOA.

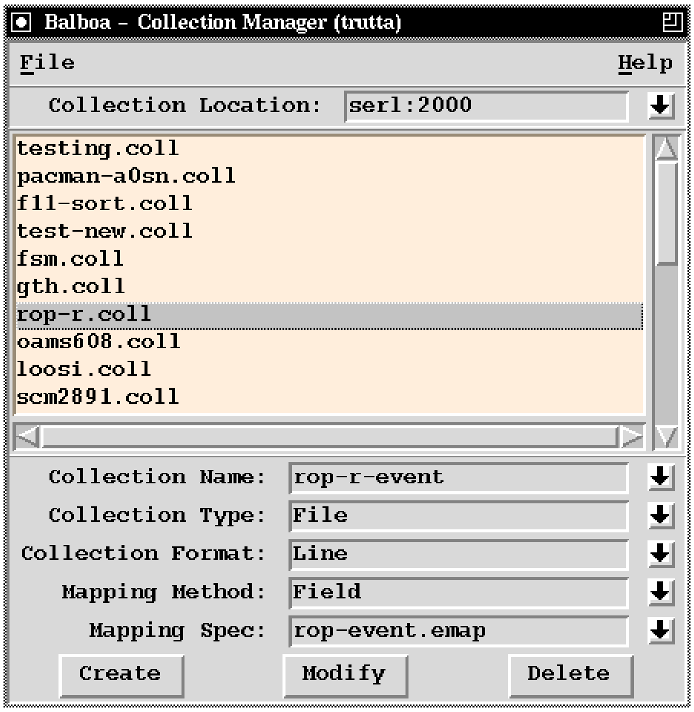

Figure 5.4: The CollectionManager Tool.

EventMapper is a tool that lets one specify how to interpret the event data. It enables the mapping of events into event types, and the mapping of event attributes.

CollectionViewer is a tool that lets one view a data source, either through some event mapping or in a raw form. This allows one to sanity check an event mapping and to simply visually inspect the data collection.

Figure 5.4 shows the CollectionManager. This tool provides a mechanism for the user to view, create, and modify the descriptions of collections of events. It provides a direct-manipulation list of event collections. The lower part of the screen is a field-oriented form for modifying or creating specifications. To modify a collection, clicking on it in the list will bring its data into the fields, which can then be individually modified and finally written back with the modify button. To create a new specification, the fields are filled out and the create button is pressed. Note that an existing collection description can be used as a template simply by selecting it in the list, thus bringing in its data to the fields.

When a new specification is created, if it is a file-based stored collection, the data can be immediately imported into Balboa. A dialog box appears after the create button, allowing the

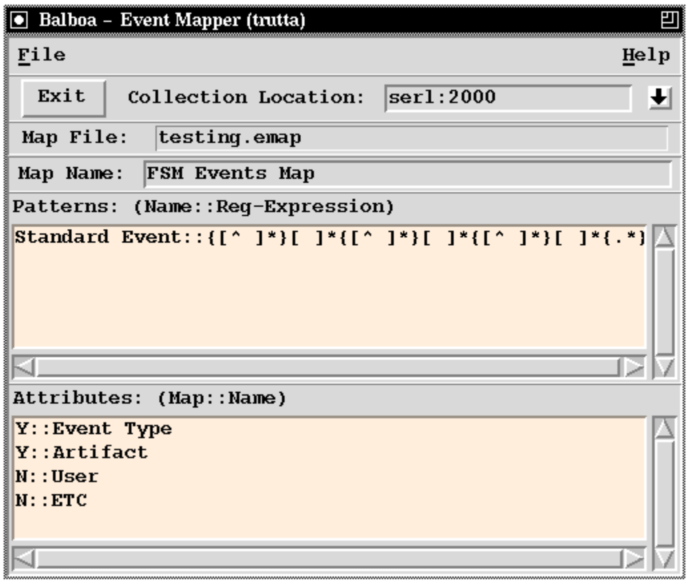

Figure 5.5: The EVENTMAPPER **Tool.**

user to specify a file to import the data from. This file is imported into BALBOA as the raw event data. If it is an ongoing collection, events will be imported as they occur in the executing process and appended to the collection.

Figure 5.5 shows a snapshot of EVENTMAPPER. This tool is used to create and modify the event mapping specifications used in BALBOA. As described in Section 5.3.2, event mapping is done using regular expressions, braced to specify event attributes, and a flagged subset of the event attributes to specify event types. EVENTMAPPER, then, provides the user with a two-list view of an event mapping specification. The upper list is the set of regular expressions specified in the mapping, and the lower list is the set of attributes for the currently selected regular expression.

Figure 5.6 shows a snapshot of COLLECTIONVIEWER. This tool provides a simple browser for event collections. Whether constructing an event mapping specification or simply wanting to see the raw data, there are times when a browser is indispensable.

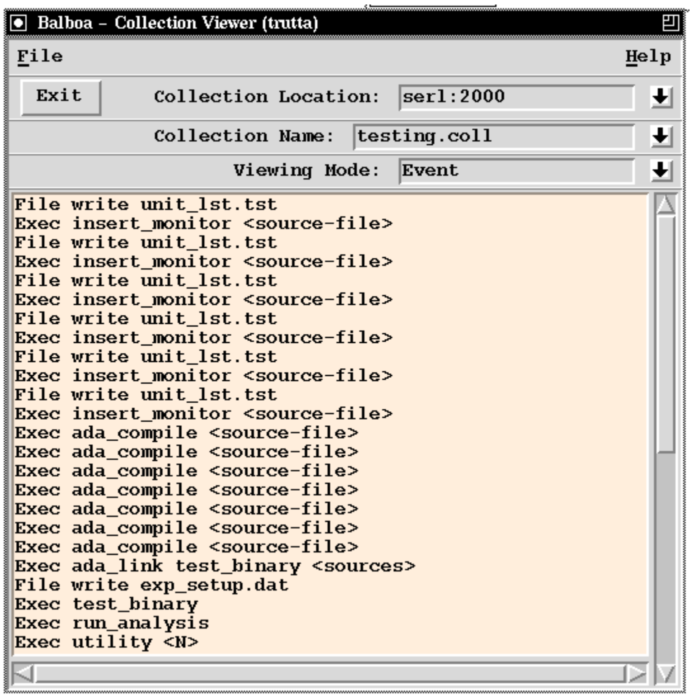

Figure 5.6: The CollectionViewer Tool.

5.4 Balboa's Data Collection Interface

The data collection interface to BALBOA is fairly simple, in that it just supports creating and adding to event data collections. The messages that implement these server access functions are:

Message	Description
MsgMakeCollection	Create an event data collection
MsgAppendToCollection	Add events to an existing (ongoing) collection
MsgDone	Signals completion of a sequence of event data messages

Each of these message has one parameter, that being the collection name. Both MsgMakeCollection and MsgAppendToCollection are followed with data messages from the client to the server, each message being an event that is to be stored in the collection. This stream of messages is terminated with a MsgDone message, signaling the server to complete the creation of (addition to) the event collection.

For data collection tools, our assumption is that in some way they can access sockets, either directly or indirectly. The most likely method in which this access will be provided is through having the tools use an "exec" interface to execute commands that provide this access. BALBOA itself provides a Tcl messaging interface built on Tcl 7.5 sockets, and Perl provides raw socket capabilities. Most collection tools, such as Yeast and Amadeus, provide the ability to execute scripts or external commands when an event is processed.

As an example, consider sending an event from Yeast to BALBOA. Our Yeast specification for watching a particular file executes a Tcl script called "sendevent" when the named file changes. Both are shown in Figure 5.7. The Tcl script justs connects to the proper BALBOA server, and appends the event to the appropriate event collection. Thus, this simple method of providing access through a scripting language allows an extensible collection tools such as Yeast to interface with the BALBOA framework.

Process-centered environments, such as as Oz [20] and Spade [12], have the ability to invoke and communicate with external tools during the enactment of the process, so they would not have a problem connecting to BALBOA. Additionally, if they produced a log of events, a periodic job could be set up to send the log to BALBOA in a batch mode.

```
addspec repeat file /project/src/Interface.C changed do
   'sendevent serl:2000 Project.1 FILE-CHANGE /proj/src/Interface.C'
```

Yeast Specification

```
#!/tools/balboa/bin/btclsh

set auto_path [linsert $auto_path 0
               /home/serl/jcook/projects/balboa/lib/tcl]
source "/home/serl/jcook/projects/balboa/src/Balboa.tcl"

set BServer   [lindex $argv 0]
set CollName  [lindex $argv 1]
set YeastType [lindex $argv 2]
set FileName  [lindex $argv 3]
file stat $FileName FileStat

BSetCurLocation $BServer
socket send $BSocketId "AppendTo Collection: $CollName"
socket send $BSocketId
            "$YeastType $FileName $FileStat(mtime) $FileStat(uid)"
socket send $BSocketId "Done: "
BCloseLocation

exit
```

Tcl Script

Figure 5.7: Yeast and Tcl Code to Register an Event with BALBOA.

5.5 The Client Interface of BALBOA

In this section we describe the specifics of the interfaces that client analysis tools use to access data from a BALBOA server. The next section describes the low-level message interface, followed by sections describing the higher-level C++ and Tcl/Tk interfaces.

5.5.1 The Messaging Interface

The base message interface that client tools use to access data is composed of the set of messages shown in Table 5.1. Basically, a client tool needs to use MsgSetCollection to specify a data collection to access, and then use the appropriate MsgGet[Event,EventPattern,Token] to access the events in

Message	Description
MsgSetCollection	Set the collection for accessing
MsgGetEvent[At]	Get next event (or specific)
MsgGetEventPattern[At]	Get next event pattern (or specific)
MsgGetToken[At]	Get next token (or specific)
MsgGetEventAttribute[At]	Get event attribute(s)
MsgMapEvent	Map a raw event and return the event pattern
MsgGetPatternOfToken	Return the event pattern of a token
MsgSetTokenParams	Set parameters for token processing

Table 5.1: Basic Balboa Message Interface

the desired mode. Attribute values are also available through the MsgGetEventAttribute message. The managerial messages MsgGetCollectionList and MsgGetEventMapList are available to client tools as well, for uses such as presenting the user with a selection list of valid data collections to use.

5.5.2 The C++ Interface

The C++ interface that a client uses to access an event stream from a Balboa server is comprised of types and classes defined in a `Balboa.h` header file, and a library that is linked in when the executable is created.

The main interface is composed of event and token server classes, which allow the access of an event collection as a sequence of events, event types, or tokens. These classes are defined with the public methods shown in Table 5.2.

The TokenServer class is a subclass of the EventServer class, so that it offers all of the functionality of the EventServer, adding to it the token interface. It is separate so that an application that does not need access tokens does not pay the overhead for maintaining token to event type mapping information.

For applications that have direct access to event collections in files (e.g., the Balboa server and its repository is not remote), and that need the fastest access possible to the data, the Balboa interface library provides subclasses of the EventServer and TokenServer classes that implement a file-based interface. Thus, client tools can be written to the Balboa interface, but not be dependent on always using a Balboa server.

Figure 5.8 shows an example minimal C++ client that simply connects to a server and prints out the named event collection in the selected mode (event, event type, or token).

```
//----------------------------------------------------------------------
//  C++ Minimal Event Stream Lister    Copyright (C) 1996 Jonathan E. Cook
//----------------------------------------------------------------------
#include <stdio.h>
#include <Event.h>
#include <SocketTokenServer.h>

enum Mode {m_event, m_pattern, m_token};

int main(int argc, char **argv)
{
   Mode mode; Event event; Token token;
   SocketTokenServer *TS; char tokenstr[10];
   if ((--argc)<4)  {
      fprintf(stderr,"Usage: %s <host> <port> <collection> <mode>\n", argv[0]);
      return 1;
   }
   if (!strcmp(argv[4],"Pattern"))
      mode = m_pattern;
   else if (!strcmp(argv[4],"Token"))
      mode = m_token;
   else
      mode = m_event;
   TS = new SocketTokenServer(argv[1],atoi(argv[2]),argv[3],(int) 'A', 100);
   while (1)  {
      switch (mode)  {
       case m_event:
         event = TS->GetEvent();  break;
       case m_pattern:
         event = (Event) TS->GetEventPattern();  break;
       case m_token:
         token = TS->GetToken();
         if (ValidToken(token))  {
            sprintf(tokenstr,"%d",token);
            event = tokenstr;
         } else  event = 0;
         break;
       default: event = 0;
      }
      if (!event) break;
      printf("%s\n",event);
   }
   delete TS;
   return 0;
}
```

Figure 5.8: A Simple C++ Event Stream Printer.

Class	Method
EventServer	Event GetEvent(StreamPosition P=0)
EventServer	Event CurEvent()
EventServer	EventPattern GetEventPattern(StreamPosition P=0)
EventServer	EventPattern CurEventPattern()
EventServer	AttributeList GetEventAttributes(char *Attribute=0, StreamPosition P=0)
EventServer	StreamPosition CurStreamPosition()
TokenServer	Token GetToken(StreamPosition P=0)
TokenServer	Token CurToken()
TokenServer	EventPattern PatternOfToken(Token T)
TokenServer	Token LookupEventPattern (EventPattern Ep)
TokenServer	int SetTokenParams(Token Base, int Arity)
TokenServer	Token TokenBase()
TokenServer	int TokenArity()

Table 5.2: Basic C++ Balboa Client Interface

Command	Description
BSelectLocation	Select a Balboa server location and connect (Tk)
BSetCurLocation	Connect to a specified location
BCloseLocation	Close connection to current location
BSelectCollection	Select an event collection for processing (Tk)
BGetCollectionList	Return a list of Balboa server locations
BSelectEventMap	Select an event map specification (not needed) (Tk)
BGetEvent	Get the next event in the specified collection
BGetEventPattern	Get the next event as an event pattern (type)
BGetToken	Get next event as a token
BGetEventAttribute	Get specified attribute's value for current event
BGetEventAttributes	Get all atttribute values for current event
BGetPatternOfToken	Get the event pattern (type) of a specified token

Table 5.3: Basic Tcl/Tk Balboa Client Interface.

5.5.3 Tcl/Tk Interface

The Tcl/Tk interface is composed of Tcl procedures that are built on top of our extension to the Tcl interpreter that provides TCP socket capabilities. The main procedures that make up the interface are shown in Table 5.3. More detailed and specific commands are available when an application needs to get beyond this basic connect-getstream-close interface. Those that are marked '(Tk)' pop up a dialog window, while those unmarked are just simple Tcl commands, and would be valid in a Tcl interpreter that does not have Tk.

Figure 5.9 shows an example minimal Tcl client that simply connects to a server and prints out the named event collection in the selected mode (event, event type, or token).

```
#-----------------------------------------------------------------------
#  Tcl Minimal Event Stream Lister    Copyright (C) 1996 Jonathan E. Cook
#-----------------------------------------------------------------------
source "$env(BALBOAHOME)/lib/tcl/Socket.tcl"
source "$env(BALBOAHOME)/src/BUtil.tcl"
if {$argc<2} {
   puts "Usage: $argv0 <location> <collection> \[viewmode\]"
   exit 1 }

set Location   [lindex $argv 0]
set Collection [lindex $argv 1]
if {$argc >= 3} {
   set Mode       [lindex $argv 2]
} else {
   set Mode "Event"
}

if {[BSetCurLocation $Location]} {
   puts "Error: No Balboa server found at $Location"
   exit 2 }
if {[BSetCurCollection $Collection]} {
   puts "Error: $Collection not found at $Location"
   exit 2 }

while {1} {
   if {$Mode == "Event"} {
      set event [SGetEvent $BSocketId]
   } elseif {$Mode == "Pattern"} {
      set event [SGetEventPattern $BSocketId]
   } elseif {$Mode == "Token"} {
      set event [SGetToken $BSocketId]
      if {$event == "\377"} {
         set event ""
      }
   } else {
      puts "Error: Mode $Mode undefined"
      break
   }
   if {$event == ""} {
      break
   }
   puts $event
}

BCloseLocation
exit
```

Figure 5.9: A Simple Tcl Event Stream Printer.

5.6 Conclusion

In this chapter we have presented BALBOA, a framework for consistently managing, interpreting, and serving process event data to analysis tools. The advantages it gives are that both the data collection and the tool construction are decoupled from having to worry about the format and access method of the data, how to interpret the data, and the managerial aspects of handling the data. This separation of tools from data format facilitates the construction of tools and allows them to access data from a wider variety of sources.

The usefulness of BALBOA has been demonstrated with the construction of the process discovery and validation tools, presented in Chapter 3 and Chapter 4, respectively. These tools are integrated into the BALBOA framework, and the cost of their construction was reduced by utilizing this common interface.

A number of possible enhancements to BALBOA can be explored as future work:

- While event data supports many varieties of analyses, certainly there are other forms of valid data to be used. A major direction to take BALBOA in the future would be to incorporate other types of process, and perhaps product, data into it.

- A tool authorization mechanism, where a server has knowledge of who is allowed to connect to it, and from where. This information could be kept at a per-collection detail, allowing certain collections to be made public, while keeping others private.

- In the data collection interface, it would seem probable that events may not be reported right when they occur. Thus, building into BALBOA some knowledge about the timestamps of events would allow it to ensure that an event stream is properly ordered, and would allow time-oriented queries on the client tool end.

- Hierarchical event collections would be useful when, for example, separate subprocesses are collected into separate collections, but there is a need to view the whole process as well.

- Sets of event collections, related closely to hierarchical collections, would be useful when analyzing many executions from a single process.

Chapter 6

A Case Study with Balboa

A significant goal of this thesis is to show that the discovery and validation methods proposed are applicable in a real world setting, and that they can operate on more than just "toy" problems. To this end, we pursued an industrial study, in which process event data was collected, and the discovery and validation tools were used to analyze the process and its execution. Our study was undertaken at an AT&T software lab, under the guidance of Dr. Larry Votta, a researcher with Bell Laboratories.[1]

This study has a broader focus than just showing that our techniques are useful; it provides the first statistical proof that formal-model process technologies can lead to better products. Thus, this study is important in its own right, and includes more than just an analysis of the methods in this thesis.

In this chapter we present this study and its results. Section 6.1 motivates and discusses the issues behind the study. Section 6.2 details the experimental design, including an overview of the process under study. Section 6.3 presents the results of our analyses. We conclude in Section 6.5 with an evaluation of the discovery and validation methods, and some general observations on the significance of historical-data-based studies such as ours.

6.1 Discussion

The last decade has seen much effort (and a lot of money) put into creating, implementing, and using good practices in the software process. But does having a good process lead to a good product? While some process improvements are relatively obvious, much of what is obtained from redesigning the process is only a vaguely comfortable feeling that one must be doing things better. Is there some way that we can actually measure the effect of process on product?

The question of whether good process leads to good product may seem obvious. After all, something is done right when a good product is produced and, similarly, something is done wrong

[1] Dr. Votta and the organization that this study was done in are now with Lucent Technologies, formerly part of AT&T.

when a defect is introduced into a product. But in the framework of formal process modeling and process improvement, the "good process" is embodied in a formal specification; in this case, it is not necessarily true that good process (i.e., the model) will in fact lead to good product. The agreed upon activities found in the process model may be unnecessary structure imposed on the developers. Or worse, they may create a complex set of interactions that in the end serves to completely demotivate people, as shown in Deming's red and white beads experiment [41]. Indeed, the current wisdom is that it is possible, for example, to achieve a high CMM maturity level (i.e., use good practices) and still have trouble producing quality products. So the question remains.

It is clear that somewhere within the process lay the mechanisms that cause defects to be introduced into the product. These mechanisms can be quite diverse. Some might be simple and unavoidable occurrences of human error, such as a coding mistake not caught by testing. Others might be institutionalized in the process by the organization, such as lax enforcement of source code access control. The question, then, is whether or not the process model identifies the mechanisms and prevents them from occurring. If it does not, then the process model may not be useful in describing what is good. The key to answering this question lies in the collection and analysis of reliable data.

There have been numerous studies that have analyzed product data to identify patterns of defects (e.g., [16, 21, 34, 110]). From these patterns, the process mechanisms behind them are inferred, but this last step of inference is purely speculative and not backed directly by hard data. We have taken the approach of analyzing process data directly, with the aim of directly identifying statistically significant process patterns that may cause defects.

While our primary purpose in this work is to determine whether we can relate good product to good process, we have a secondary goal that has to do with identifying sources of reliable process data. In particular, we are interested in whether we can make effective use of the process data already collected routinely by an organization for purposes other than our own. The issue here is finding ways to minimize the cost and intrusion incurred when new process data analyses are called for.

Our approach is to view a software process together with its existing repositories of data as a naturally occurring experiment. By looking backwards in time, we already know the outcome of the process execution and can examine the data to see what caused, or at least what factors are correlated with, that observed outcome. A similar kind of historical data analysis was successfully employed in a recent study conducted by Votta and Zajac [119].

In this chapter we present a study that analyzes a repetitive process to determine if good process does in fact lead to good product. In the course of performing the study, we take advantage of the process data that were previously collected, both manually and automatically, as part of an industrial software development effort. The organization under study is responsible for performing customer-initiated updates to a large software product. The readily available data collected by the organization form the basis for a naturally occurring experiment in which to frame our study.

As a first part of the study, simple aggregate metrics, such as the number of source lines changed and the elapsed time of the process, are examined to see if they can identify the presence of any defect-producing mechanisms. Next, the process validation methods of Chapter 4 are used to measure the correspondence between the process as executed and the process as prescribed by two different process models, one of which was created using the process discovery methods of Chapter 3. Our results indicate differences that may point to mechanisms causing the unsuccessful processes to be unsuccessful. We found two aggregate metrics that correlated with the defect metric: the delay between the appearance of a customer problem report and the beginning of the activity to make the fix, and the developer who performed the fix. We also found significant differences between how the successful and unsuccessful processes followed the prescribed process. Thus, we have demonstrated through this study that good product can be measurably related to good process and that taking advantage of historical data is an effective means for analyzing processes.

6.2 Experimental Design

Designing an experiment to answer our two questions required the careful selection of a suitable organization to study, one that already collected process data, not just product data, and that was willing to invest a small amount of time to work with us. We also needed to understand the process to a sufficiently detailed level, so finding an organization that had already documented their process would greatly help us in getting started with the experiment.

In this section we describe the process followed by the organization with which we worked, our method for selecting, collecting, and analyzing their process data, and our method for comparing their conception of the process to actual executions of that process. We also consider various threats to the possible validity of the experiment.

6.2.1 Overview of the Subject Process

In this study we performed a post-mortem examination of a customer-initiated software update process for a large telecommunications software product. We targeted a small, repetitive process responsible for identifying and solving customer-reported software problems. The prescribed steps in the process are depicted informally and at a high level in Figure 6.1. We are interested only in those instances of the process that involved making actual changes to the software. Data about the other instances were ignored.

Any problem in the field that causes the customer to call for assistance is recorded in a customer assistance database and identified by a so-called CAROD ticket. Most reports are not specifically software problems, so they can be resolved by performing some other simple process, not part of our study, such as supplying the customer with the documentation they need to solve their own problem or helping them with some confusion about the configuration of their system.

Some number of customer reports, however, are identified as problems in the software. If there already exists a fix for the problem, then it is released to the customer as a software update. If not, then the problem is assigned to a specific developer who assumes responsibility for generating a fix. We use this assignment to indicate an instance of the process to study. Performing the fix involves opening a modification request (MR) for that fix, employing the source code control system to gain access to the code, and subjecting the changed code to various levels of quality assurance.

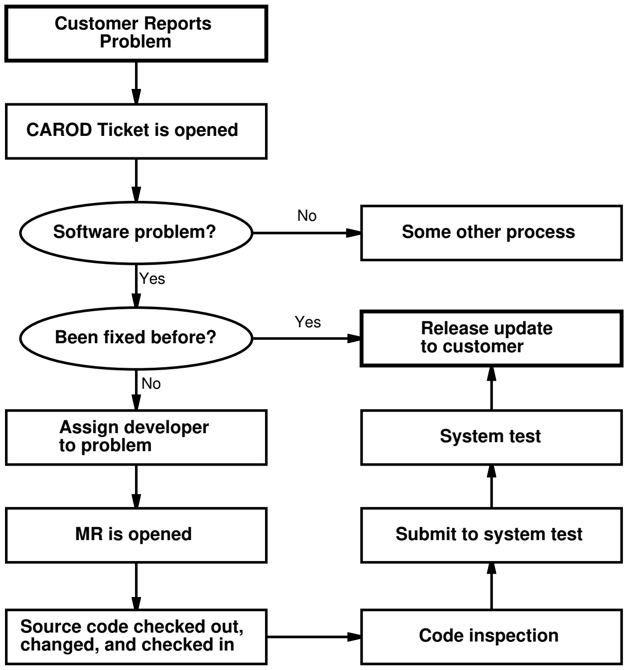

Figure 6.1: Basic Structure of the Process Under Study.

Once a fix is completed, it is released to the customer as a software update. When a customer applies the fix, they may find that the fix does not in fact solve their original problem. We consider this to be a failure in the process; some unknown mechanism was at work to introduce a defect into the software. For our purposes, the accept/reject judgment by the customer terminates an instance of the process. Of course, the organization will act to resolve the problem, and in the end most such rejected fixes are eventually corrected and accepted. But what is important for our study is that the first attempt to fix the problem was not successful.

6.2.2 Sources of Data

As mentioned above, the approach taken in this study was one of analyzing process data previously collected by the organization. Therefore, we could not specify the data we wanted; rather, we could only view the process through the data that existed. The danger in such an approach is that the data only reflect what the organization felt was important. Parts of the process for which no data are collected will be invisible to our analyses. On the other hand, no additional data collection costs are required of the organization and we can examine large amounts of data in a non-intrusive way. While certainly not true of all processes, in our subject process we were able to find enough data of high quality for valid statistical analysis.

Event	Definition
carod-abstract	create an abstract of the customer's problem
carod-closed	close the CAROD ticket (i.e., mark as complete)
carod-create	create a CAROD ticket for a customer problem
carod-custdue	promised due date for a customer solution
carod-dcreated	description of ticket created
carod-delivered	customer solution is delivered
carod-dupdated	update problem description
carod-inhouse	problem brought in-house
carod-response	initial response to the customer
carod-solved	customer problem has been solved
carod-update	update of CAROD ticket
code-checkin	source code module check-in
code-inspect	source code inspection occurred
mr-acceptmr	developer accepts MR assignment
mr-acptsys	system test accepts MR for a test build
mr-asndue	assign due date to MR
mr-asnmymr	assign my MR (to myself)
mr-asnprior	assign a priority to an MR
mr-asntarg	assign target to MR
mr-assign	assign MR to developer
mr-bwmbuilt	test build has been built for MR
mr-create	create an MR
mr-createmymr	create an MR (for myself)
mr-defer	defer the MR
mr-featchg	MR is a feature change
mr-integrate	MR is integrated (into a build)
mr-killmdmr	kill an MR
mr-rejectbwm	MR is rejected from system build
mr-rejectmr	MR is rejected
mr-smit	submit MR to system test
mr-smitsys	submit MR to system test
mr-test-plan	test plan written for MR

Table 6.1: Definitions of the Event Types in the Data.

There are many kinds of data we could examine, but we chose to look at event data [23, 121] because they neatly characterize the dynamic behavior of the process in terms of the sequencing of its major activities. The event data come from several sources. The customer database gives us events concerning the interaction with the customer, including the opening of a CAROD ticket (customer assistance request), each update to the status of the ticket, and when the problem was solved, the fix delivered, and the ticket closed. The source code control system gives us every check-in event associated with changes to individual source files. Each process instance is tracked by an MR (modification request) number, and this tracking database gives us the events of opening an MR, assigning an MR to a developer, accepting an MR by the developer, generating a test plan, submitting an MR solution to system test, having an MR accepted by system test, and eventually closing an MR. Lastly, a database of inspection information gives us the date of the code inspection and its result. All but the last source of event data involved automatic collection through tools; inspection dates and results were recorded manually. Table 6.1 defines the event types that occur in the data.

By merging the events for a particular instance of the process from each of the data sources, we were able to create an event stream that represented the sequence of activities occurring in a process execution. We collected 159 such event streams. The event streams were then separated into two populations: one for which the customer accepted the fix (141) and one for which the customer rejected the fix (18). Because there is no source of data that directly records the presence of defects in a released software update, this partitioning of the fixes had to serve as our defect metric. Of course, the validity of the metric is based on the assumption that the rejection of a fix indicates the presence of some defect.

6.2.3 Methods of Analysis

With this separation of populations, we then performed analyses on a variety of metrics in order to discover process characteristics that correlate with the acceptance or rejection of a fix. Our analyses centered on performing statistical significance tests for each metric that was calculated.

We looked first at simple aggregate metrics, including the number of source code lines changed and the elapsed time of the process, to see if some readily available metric could begin to explain the defect metric. We then took a deeper look at the prescribed process itself, using the process validation method discussed in Chapter 4 to measure the correspondence between process executions and the process model. These measurements give a detailed picture of where and by how much the process execution deviates from the organization's prescribed sequence of activities. Process validation is described in the next section.

Most of the analyses were performed using metrics whose values are numeric. For those metrics, we used the Mann-Whitney significance test [42, Chapter 15], which does not assume an underlying distribution of the data but is still nearly as powerful as standard significance tests that do assume a distribution. The premise behind this test is that if there is no difference between the two populations, then when the data values from the two populations are merged and sorted, each population should be distributed approximately the same in the merged ranking. For this reason, it is also called the Wilcoxon Rank-Sum test.

Other metrics we use are two-valued, such as "an event in the event stream is either matched or deleted when it is validated with respect to a model". These metrics produce binomially distributed populations, and imply a test over the population proportions. The standard significance test is

$$Z = \frac{p_1 - p_2}{\sqrt{pq(\frac{1}{m} + \frac{1}{n})}}$$

where m and p_1 are the size and proportion of one population, and n and p_2 are the size and proportion of the other population. p and q are defined as

$$p = p_1 \frac{m}{m+n} + p_2 \frac{n}{m+n} \quad , \quad q = 1 - p$$

that is, p is the weighted sum of the two proportions and q is its inverse.

For each application of the Wilcoxon Rank-Sum (W) and the proportional (Z) significance tests, the two-tailed p-value is calculated. We interpret any p-value less than 0.05 as strongly signifying that the two populations differ on the metric, and any p-value less than 0.1 as weakly signifying a difference.

Metrics that significantly differentiate the populations were then examined and interpreted. In addition to presenting the p-values in our results, we also present the means and standard deviations, or proportions, of these metrics, and a plot of each metric's distribution. This, should provide an understanding of the general range and makeup of the data, and thus the process being studied.

6.2.4 Comparing Executing and Prescribed Processes

Measuring the correspondence between the process executions and the prescribed process is done using the validation techniques described in Chapter 4. The study discussed here uses a non-interactive, batch version of the validation tools.

In addition to the SSD and NSD metrics, we look at per-event-type and per-model-state information. For each type of event (e.g., a check-in event of the source code control system), we record the total number of matches, insertions, and deletions for each process execution. This allows us to calculate, for each event type, the number of events of that type that occurred correctly according to the model (matches), the number that were missed (insertions), and the number that were extra or at the wrong time (deletions). Similarly, for each state in the model, we also record the total number of event matches, insertions, and deletions that occur at that state in the model. These counts are then combined into meaningful metrics, as follows.

1. *matches/(matches+deletions)* gives, for each event type, the proportion of event occurrences in the event stream that are matched by the model;

2. *matches/(matches+insertions)* gives, for each event type, the proportion of events predicted by the model that are matched in the event stream;

3. *matches/(matches+deletions)* gives, for each state in the model, the proportion of event occurrences in the event stream that are matched by the model; and

4. *matches/(matches+insertions)* gives, for each state of the model, the proportion of events predicted by the model that are matched in the event stream.

In effect, the first and third metrics represent the proportion of matches from the perspective of the event stream, while the second and fourth metrics represent the proportion of matches from the perspective of the model. Further, the first and second metrics provide an understanding of locality based on the event type, while the third and fourth metrics provide the same for locality in the model.

6.2.5 Threats to Validity

As with any experimental study, validity concerns must be explicitly addressed. Here we discuss threats to the construct, internal, and external validity of our results. We use the definitions of validity given by Judd, Smith, and Kidder [78].

Construct validity is concerned with how well the metrics used in the study faithfully and successfully reflect real-world attributes and values. In this study, we are using the customer's acceptance or rejection of a fix as the metric for the success or failure of the process; it is on this basis that we separate the populations. But one could imagine that there would be other reasons for a customer to reject a fix, not necessarily related to whether or not the developer fixed the problem as they understood it. This metric, however, was the closest we could come to a direct measure of success and failure. It is important to note that an inability to directly measure something of interest occurs often in experimental studies.

In contrast to the success/failure metric, most of the other metrics that we used measure attributes directly, such as the number of source lines or the elapsed time of portions of the process. These direct measures are only threatened by the possibility of false or inaccurate data. As mentioned above, most of the data were automatically collected, so inaccuracies are unlikely. During our assembly of the data, and in interacting with those providing the data, there was no indication that people were engaged in purposely falsifying the data; since the data were not particularly used before our study, there was not even a motivation for them to do so.

The remaining set of metrics are those for measuring how closely the process models are followed, using the validation metrics discussed above. The application of these metrics are to date largely untested, and this study is part of an evaluation of whether they do measure something useful. However, they are based on widely-used methods for measuring differences in similar types of data, so there is good reason to expect that the measurements are accurate.

Internal validity is concerned with how well the experimental design allows for conclusions of causality among the constructs under study. Conclusions about causality comes from being able to control the experimental setting and randomizing independent variable assignments. Since ours is a historical study examining processes that already occurred, we cannot randomize variables and so cannot conclude causality from any statistically significant measures that we might obtain. This does not mean, however, that we cannot learn anything from the results.

External validity is concerned with how well the study's results can be generalized. That is, if we answer the question "Does good process lead to good product?" in this case, does that mean the same answer holds for other cases? On the negative side, the process we studied is more of a maintenance process than a development process, so the results may be biased towards maintenance kinds of processes. It is also fairly small, although the software it manages is very large. On the positive side, this is a real-world industrial process that is repeatable in its execution. It is also a process that is ubiquitous in industry; almost all organizations have a problem reporting and fault fixing process. Thus, while the results probably do not generalize to all processes, they are likely to shed light on many processes that are in use in industry today.

6.3 Results

We now present the results of our analyses on the subject data. We begin with the aggregate metrics and follow those with the correspondence metrics.

6.3.1 Aggregate Metrics

We calculated several simple aggregate metrics and measured their statistical significance in separating the accepted fix population from the rejected fix population. These metrics were the following.

ncsl	Number of source lines of the fix, including new, changed, and removed lines. This is calculated directly from the source code control system.
nfiles	Number of source files modified for the fix. This is calculated from the source code control system.
nevents	Total number of events for each process execution. This represents a simplistic count of the number of steps executed in a particular execution of the process.
ctime	Total time, in days, from the customer ticket open to close. This is the total elapsed time of the process execution.
mtime	Total time, in days, from the MR open to close. This is the total elapsed time of the development subprocess.
dtime	Delay time, in days, from customer ticket open to MR open. This is the interval between the time a problem is reported and the time a developer begins to fix the problem.
developer	The developer who performed the fix, as recorded in the MR database.

For each of the metrics, a statistical test was performed to determine if the populations were significantly different for that metric. The results for the first six metrics are shown in Table 6.2. Figure 6.2 shows distribution plots for each of the aggregate metrics, except *developer*. The results for *developer*, which uses a nominal scale, are shown in Figure 6.3. Five of the metrics were not statistically significant in separating the populations. We now examine the two metrics that were significant.

The first significant metric is *dtime*, the delay between the time a customer reports a problem and the time a developer starts working on the problem. This correlation is quite interesting, and could have several explanations. One explanation might be that the developer's understanding of the problem degrades during the delay, and thus the developer encounters a harder time fixing the problem. Another might be that it takes longer for customer support, working with the customer, to understand the problem in enough detail to relay it the developer. This delay could be due to the fact that the problem is simply a difficult one to understand, and thus less likely to be fixed correctly. In either case, this result would warrant a closer examination of the relationship.

Measure	P-value (2-tailed)	Sig Test (W)	Accept Pop.(N=141) Mean	Accept Pop.(N=141) Std Dev	Reject Pop.(N=18) Mean	Reject Pop.(N=18) Std Dev
ncsl	0.23	1.20	217.22	554.31	166.22	275.62
nfiles	0.32	1.00	2.75	4.00	2.94	2.67
nevents	0.25	1.15	30.34	18.55	35.22	16.28
ctime	0.19	1.32	165.94	141.10	180.83	89.91
dtime	0.00	3.13	18.29	31.22	68.17	75.41
mtime	0.72	-0.36	96.89	110.76	87.94	85.56

Table 6.2: Aggregate Metrics.

The other significant metric is *developer*, the developer working on the fix. For each developer, we calculated the ratio of rejected fixes to total fixes as a measure of that developer's failure rate. To determine whether or not the failure rates are significantly different, we estimated the standard deviation for each ratio using $\sqrt{p(1-p)/t}$, where p is the failure rate, and t is the total number of fixes. In Figure 6.3 we show the data with error bars. Since non-overlapping error bars are significant at about the 0.1 level, *developer* is clearly a significant metric.

One possible explanation for why certain developers have more fixes rejected than others is that the fixes are assigned to developers who have an area of expertise that matches the suspected problem. Some developers may be working on rather simple peripheral code, while others may be engrossed in the internals of a large subsystem that is difficult to change.

A further examination of the event streams is shown in Table 6.3, where each event type is counted for the number of times it occurs in each process execution. One can see that several event types have significantly different average counts between the populations.

One of the most significant is *carod-dupdated*, which is an instance of someone updating the description of the customer's problem. The greater value of the rejected fix population would seem to imply that it takes more effort to figure out the problem. This could be due to the problem being more difficult, or that communication with the customer is breaking down and requires more iterations.

The other significant CAROD event types indicate that there are slightly more CAROD records per MR in the rejected fix population, as can be seen by the mean values of the event counts. This occurs when more than one customer reports the problem while it is open and being fixed. If the fix was already performed for the first customer, then it would be a simple software update for the next; if the fix is not yet completed, and another customer reports the same problem, then both of the customer CAROD tickets are associated with the same MR number and thus the same fix process. This dual association could cause more customer rejections because, for example, the fix might be directed more towards the first customer, and not quite fix the problem reported by the second customer. This increase in rejections could also happen if the association was erroneous—that is, the person who associated the two problems with the same fix may have been wrong.

In general, the data reveal the presence of one or more defect-producing mechanisms at work, and that the mechanisms are somehow associated with the delay in beginning a fix and the developer

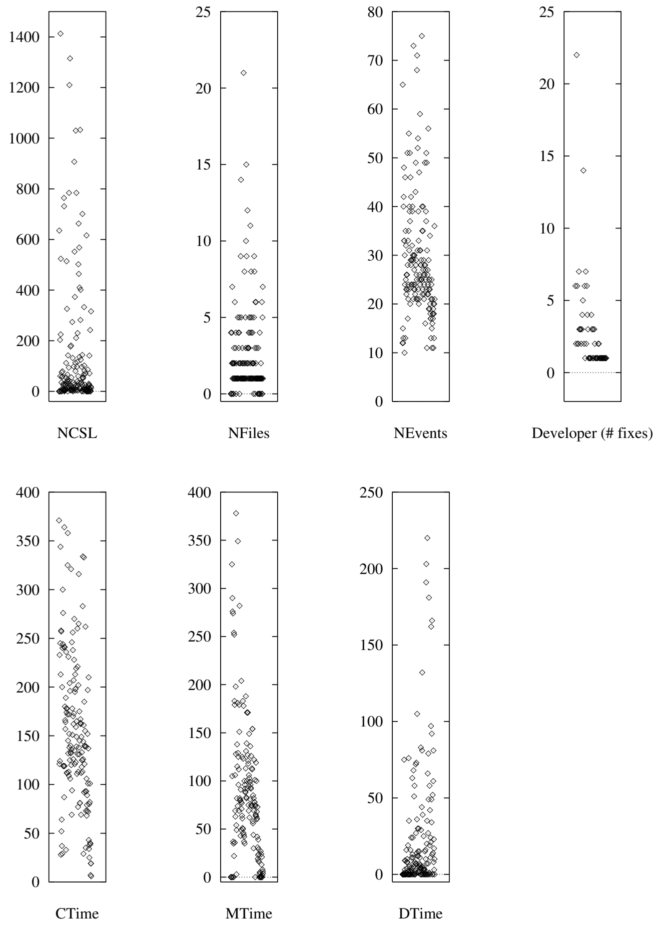

Figure 6.2: Distribution Plots of the Aggregate Metric Data.

performing the fix. The results do not identify the mechanisms themselves, but rather point the organization in possible directions to look for improvements.

6.3.2 Correspondence Metrics

We now analyze the process executions with respect to the prescribed process model. Using the validation tool discussed in Section 6.2.4, we measure the correspondence of the process executions, as represented by event streams, to the organization's prescribed process, as represented by a finite-

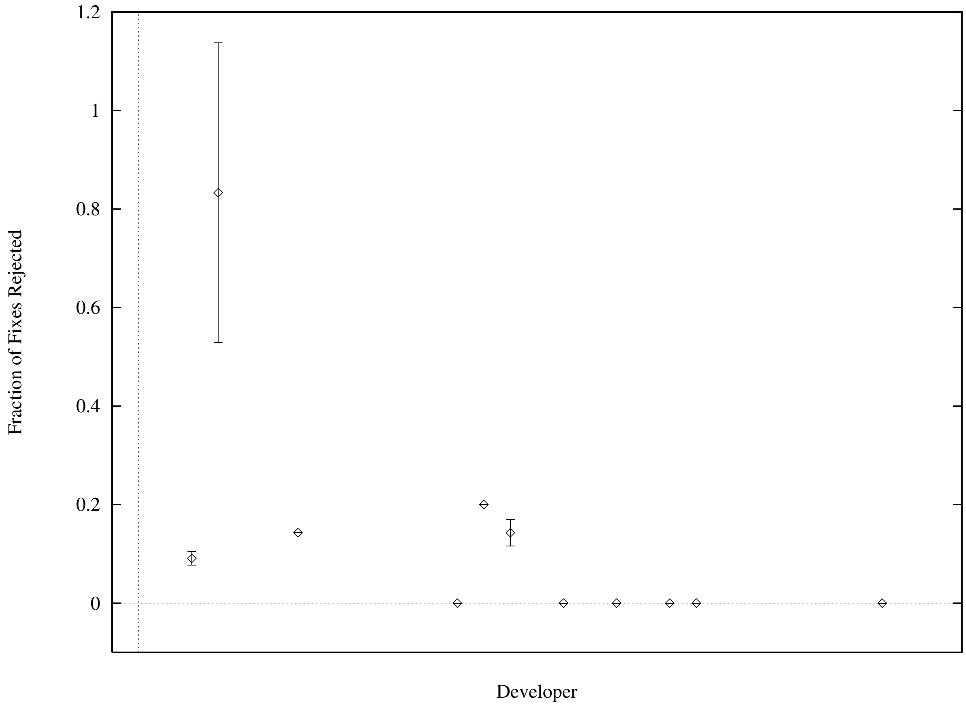

Figure 6.3: Fraction of Rejected Fixes Per Developer. Only those developers who performed four or more fixes are shown. Non-overlapping error bars are significant at approximately the 0.1 level.

state machine model of the process. The model, shown in Figure 6.4, is based on the organization's (paper) documentation of the process, interviews with several members of the organization, and a cursory look at the data. Thus, it faithfully represents the idealized view of the prescribed process.

Using this model and the 159 event streams, we calculated the SSD and NSD metrics, weighting insertions and deletions equally. We also took simple counts of the number of matches, insertions, and deletions of events, as well as of insertion and deletion blocks. Table 6.4 gives the results of our metric calculations for both the accepted fix and rejected fix populations, and Figure 6.5 shows distribution plots for each metric.

The first thing to notice is that, based on the number of event matches and the average number of events in an event stream (from Table 6.2), only about 45-55% of the total events are matched by the model. Evidently, the model in Figure 6.4 is not good at describing the actual behavior of the processes. This is also reflected in the SSD and NSD metrics, which have quite large values.

Next, notice that while the SSD metric fails to differentiate the populations, the NSD metric succeeds. In particular, the deviation from the prescribed process is significantly greater for the

Measure	P-Value (2-tailed)	Sig Test (W)	Accept Pop. N	Accept Pop. Mean	Accept Pop. Std Dev	Reject Pop. N	Reject Pop. Mean	Reject Pop. Std Dev
carod-dupdated	0.01	2.79	141	3.04	3.42	18	6.00	5.38
carod-abstract	0.01	2.67	141	1.18	0.58	18	1.56	0.92
carod-create	0.01	2.55	141	1.18	0.58	18	1.56	0.92
carod-custdue	0.01	2.58	141	1.16	0.61	18	1.56	0.92
carod-inhouse	0.01	2.55	141	1.18	0.58	18	1.56	0.92
carod-response	0.01	2.55	141	1.18	0.58	18	1.56	0.92
carod-update	0.01	2.55	141	1.18	0.58	18	1.56	0.92
mr-rejectbwm	0.01	2.71	141	0.05	0.22	18	0.22	0.43
carod-dcreated	0.08	1.75	141	1.26	0.63	18	1.61	0.98
mr-acptsys	0.10	-1.65	141	0.86	0.59	18	0.61	0.50
mr-acceptmr	0.13	1.53	141	0.96	0.51	18	1.11	0.32
mr-smitsys	0.17	1.37	141	1.20	0.74	18	1.39	0.61
mr-createmymr	0.18	1.35	141	0.74	0.44	18	0.89	0.32
mr-rejectmr	0.21	1.25	141	0.04	0.20	18	0.11	0.32
carod-closed	0.23	-1.21	141	0.80	0.76	18	0.78	1.22
mr-asnmymr	0.24	1.17	141	1.01	0.71	18	1.11	0.32
mr-smit	0.29	1.06	141	1.23	0.76	18	1.39	0.61
mr-create	0.31	-1.01	141	0.21	0.41	18	0.11	0.32
carod-delivered	0.39	-0.87	141	0.81	0.76	18	0.83	1.20
code-checkin	0.45	0.75	141	5.96	15.63	18	4.44	5.64
mr-assign	0.52	0.65	141	1.11	0.77	18	1.11	0.32
code-inspect	0.61	-0.51	141	0.93	0.31	18	0.89	0.32
mr-asnprior	0.61	-0.51	141	0.01	0.12	18	0.00	0.00
mr-featchg	0.64	0.47	141	0.09	0.30	18	0.11	0.32
mr-killmdmr	0.66	0.45	141	0.10	0.36	18	0.11	0.32
mr-asntarg	0.72	-0.36	141	0.01	0.17	18	0.00	0.00
mr-defer	0.72	-0.36	141	0.01	0.08	18	0.00	0.00
mr-integrate	0.72	-0.36	141	0.01	0.08	18	0.00	0.00
mr-killmrnm	0.72	-0.36	141	0.01	0.08	18	0.00	0.00
mr-killok	0.72	-0.36	141	0.01	0.08	18	0.00	0.00
carod-solved	0.85	-0.19	141	0.91	0.78	18	1.06	1.21
mr-test-plan	0.89	0.13	141	0.94	0.48	18	1.00	0.59
mr-bwmbuilt	0.91	-0.12	141	0.82	0.56	18	0.83	0.79
mr-killmr	0.92	0.10	141	0.06	0.32	18	0.06	0.24
mr-killmymr	0.98	-0.03	141	0.06	0.27	18	0.06	0.24

Table 6.3: Event Type Counts Per Event Stream. All events above the horizontal line are at least weakly significant.

rejected fix population than it is for the accepted fix population. If we look at the constituents of the NSD metric, we see that it is the number of deletions that is significant, but then only weakly. The NSD metric focuses attention on blocks of insertions and deletions, so the fact that the NSD metric detects a strongly significant difference in the populations indicates that there were larger

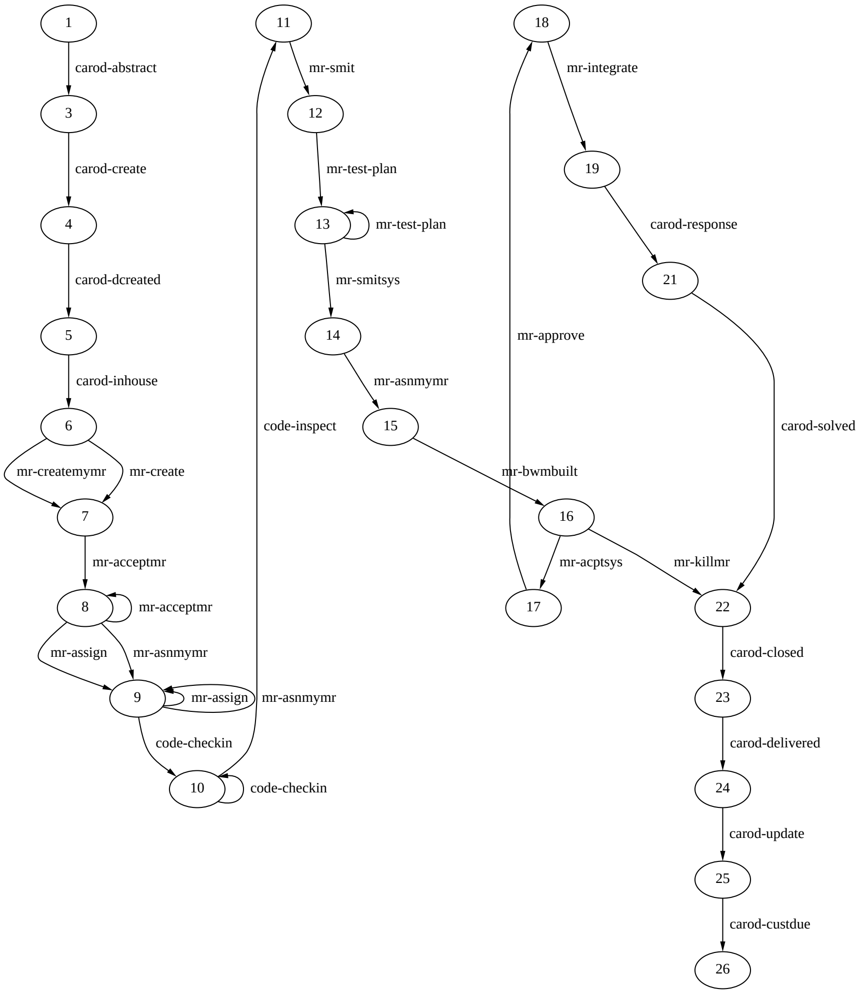

Figure 6.4: Finite-State Machine Model of the Subject Process.

areas of deviation from the process model in the rejected fix population than in the accepted fix population.

At this level, then, we see that the model describes only about half of the behavior, and that there is some significant difference in how the populations relate to the model. We can see the difference in more depth by looking at the proportions of matched events, shown per event type and per model state in tables 6.5 and 6.6, respectively (see Section 6.2.4). In these tables, only the event types and model states that did in fact have significant differences are shown.

The first significant difference is the check-in event near state 10 in the process model. Since this is the only place in the model where that event appears, and the difference is significant in

Measure	P-value (2-tailed)	Sig Test (W)	Accept Pop.(N=141) Mean	Accept Pop.(N=141) Std Dev	Reject Pop.(N=18) Mean	Reject Pop.(N=18) Std Dev
SSD	0.19	1.32	0.65	0.24	0.72	0.21
NSD	0.01	2.72	29.75	72.59	88.58	111.67
Matches	0.62	-0.50	17.93	15.78	16.11	5.28
Insertions	0.80	-0.26	6.96	2.86	7.33	3.38
Deletions	0.09	1.72	10.77	7.32	14.56	8.99
Insertion blocks	0.22	-1.24	3.30	1.24	3.17	0.99
Deletion blocks	0.45	0.75	4.88	2.12	5.22	1.96

Table 6.4: Correspondence Metrics for the Model of Figure 6.4.

Event Type	P-value (2-tailed)	Sig Test (Z)	Accept Pop. # Events Occurring	Accept Pop. Proportion Matched	Reject Pop. # Events Occurring	Reject Pop. Proportion Matched
code-checkin	0.01	2.55	833	0.87	80	0.76
mr-bwmbuilt	0.06	1.91	112	0.85	14	0.64
mr-test-plan	0.06	1.87	129	0.41	17	0.18

Event Type	P-value (2-tailed)	Sig Test (Z)	Accept Pop. # Events Predicted	Accept Pop. Proportion Matched	Reject Pop. # Events Predicted	Reject Pop. Proportion Matched
carod-closed	0.02	2.30	127	0.57	18	0.28
mr-bwmbuilt	0.06	1.90	132	0.72	18	0.50
mr-test-plan	0.06	1.85	136	0.39	18	0.17
mr-acptsys	0.09	1.70	107	0.89	10	0.70
mr-acceptmr	0.10	-1.66	146	0.88	20	1.00

Table 6.5: Per-Event-Type Metrics for the Model of Figure 6.4. Only those event types that had significant measures are shown.

both tables, we can conclude that check-in occurs with less regularity in the rejected fix population, which may lead to erroneous inspections or untested pieces of code.

Another significant point of difference is the *mr-test-plan* event near state 12, with both populations having low match rates, but with the rejected fix population significantly lower. This may point to a problem in the timely creation of test plans to test the fix, which could have an affect on the success of the fix.

Lastly, the area in the model around states 16 and 22 shows significant differences in both the per-event-type and per-model-state numbers. States 16 and 22 have much fewer matches of predicted events in the rejected fix population, and state 16 also has fewer matches of events that occurred. This may point to problems in controlling the system test or the delivery of the fix to the customer.

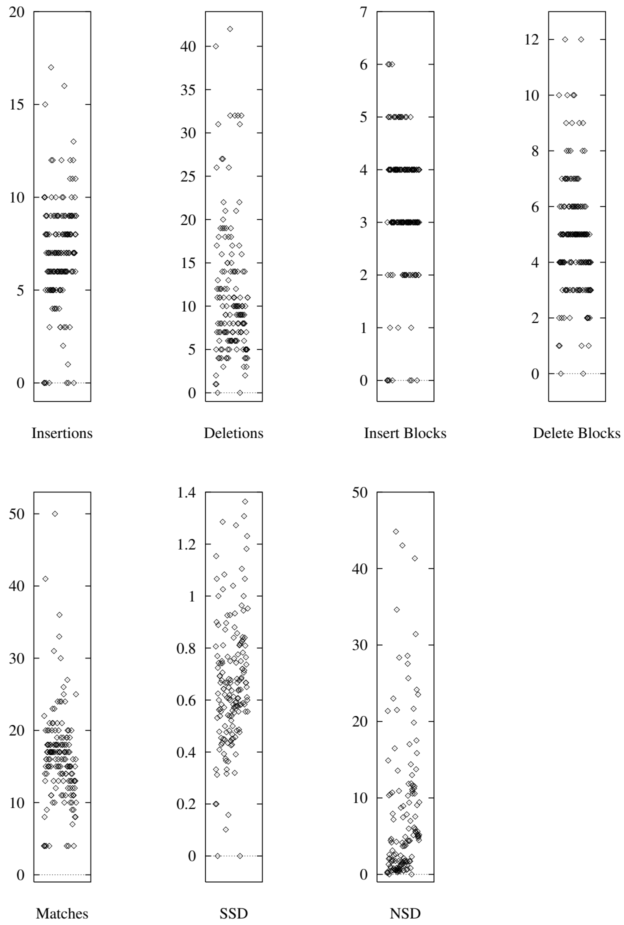

Figure 6.5: Distribution Plots of the Correspondence Metric Data for the Model of Figure 6.4.

Metrics Based on an Alternative Process Model

The model we have been using up to this point reflects the intended behavior of the process. But what if this view of the process is out of date with the current practice in the organization? This is certainly implied by the fact, seen above, that roughly half the predicted events are not matched in the actual execution. Is there an alternative model, still representing the same abstract process described in Section 6.2.1, that we could use as the basis for the correspondence metrics,

Model State	P-value (2-tailed)	Sig Test (Z)	Accept Pop. # Events Occurring	Accept Pop. Proportion Matched	Reject Pop. # Events Occurring	Reject Pop. Proportion Matched
16	0.00	3.34	176	0.54	32	0.22
24	0.01	2.52	183	0.63	39	0.41
21	0.02	2.30	78	0.59	15	0.27

Model State	P-value (2-tailed)	Sig Test (Z)	Accept Pop. # Events Predicted	Accept Pop. Proportion Matched	Reject Pop. # Events Predicted	Reject Pop. Proportion Matched
16	0.00	2.82	132	0.72	18	0.39
22	0.02	2.30	127	0.57	18	0.28
10	0.03	2.14	742	0.93	62	0.85
15	0.06	1.90	132	0.72	18	0.50
12	0.08	1.75	133	0.38	18	0.17
7	0.10	-1.66	133	0.86	18	1.00

Table 6.6: Per-Model-State Metrics for the Model of Figure 6.4. Only those model states that had significant measures are shown.

one that is perhaps more indicative of the organization's activities?

Figure 6.6 shows such an alternative. Rather than being based on manuals and interviews, this model is generated directly from the data using the process discovery methods discussed in Chapter 3. For this application, we used both the MARKOV and KTAIL discovery methods, and applied them to all of the 159 event streams together, since they are all executions of the same process. As was shown by the first model, the event data is highly variable; by setting thresholds for the discovery methods to only recognize highly probable recurring activity patterns in the data, the resulting model represents a more objective view of the process that is normalized over numerous executions of the process.

The model shown was composed from the patterns that the discovery methods revealed in the data; given the amount of data, it would have been very difficult to produce a model such as this without the discovery tools. By using this model instead of the previous one, we are in a sense shifting the source of process prescription from external mandate to emergent organizational behavior. Nevertheless, the question remains the same: is there a correlation between adherence to a process and the quality of the product?

Table 6.7 gives the results of our basic metric calculations, and Figure 6.7 shows the distribution plots. Overall, we can see a somewhat better correspondence between the model and the executions, with an average of about 65% of an execution's events being matched. While greater than for the previous model, this average indicates a significant amount of variation among the individual executions, considering that the model is derived from the collective data describing those executions. As with the first model, the SSD metric fails to differentiate the populations, and the NSD metric succeeds, with the rejected fix population showing the greater deviation. But

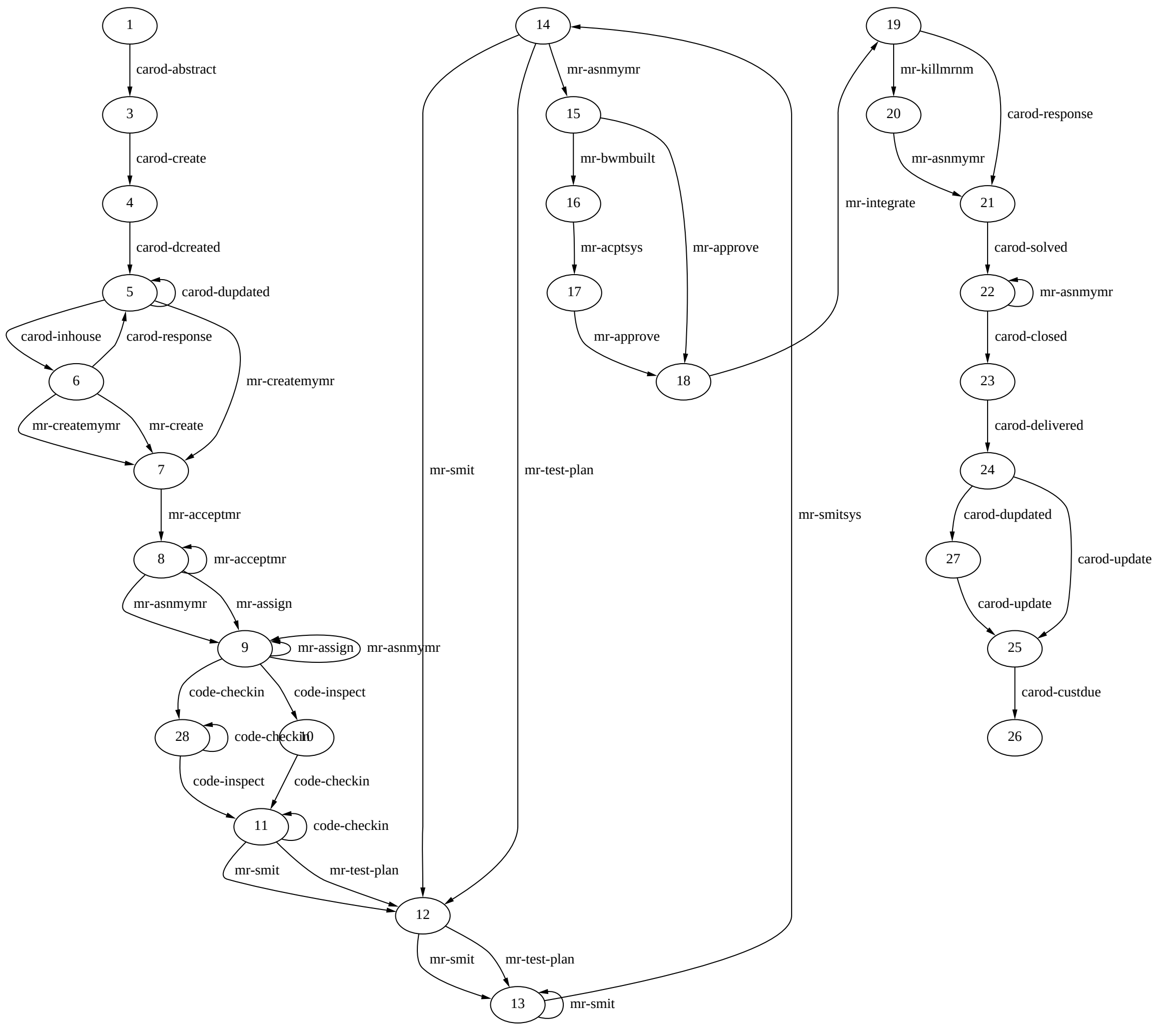

Figure 6.6: Alternative Finite-State Machine Model of the Subject Process.

in this case, it is insertions, rather than deletions, that are the drivers.

Looking at the proportions of matched events, shown per event type and per model state in tables 6.8 and 6.9, we gain interesting and significant insights. The check-in events, associated with state 11 in the second process model, are again significantly different in the two populations. However, because this model has more complex behavior at this point than the previous one, it is less clear what the root of the problem may be. Nonetheless, this result reaffirms our suspicion gained using the first model that there may indeed be a problem associated with check-ins that warrants further investigation.

An even stronger difference in the populations can be seen near the end of the process, in states 21 through 23 in Table 6.9, as well as in the various CAROD events in Table 6.8. Specifically, both tables indicate that the proportion of matches for the rejected fix population is less than half that for the accepted fix population. This difference may suggest that the accepted fix population carefully follows a more structured customer delivery mechanism.

6.4 Summary and Evaluation

6.4.1 Summary of Results

To summarize the experiment itself, our analysis of the data has revealed several interesting things about the subject organization and its process.

- The documented model of the process does not adequately capture what the members of the organization actually do to successfully carry out the process.
- There is a large variance in the structure of activities among individual executions of the process.
- The success of the process is highly dependent on the person responsible for making the required fix.
- A long delay in beginning a fix is an indication of a likely failure in the process.
- Missed or irregular code check-in is an indication of a likely failure in the process.

While we cannot point to specific defect-producing mechanisms from these results, the organization can use this information to focus their process improvement efforts. For example, they can better document their process in order to communicate good practices and potential pitfalls to new members; they can establish an alarm system that monitors the delay between accepting a problem report and beginning the corrective action, and have managers automatically notified when the

	P-value	Sig Test	Accept Pop.(N=141)		Reject Pop.(N=18)	
Measure	(2-tailed)	(W)	Mean	Std Dev	Mean	Std Dev
SSD	0.41	0.82	0.56	0.22	0.61	0.23
NSD	0.00	3.65	24.49	83.90	59.96	95.82
matches	0.79	0.27	21.15	16.25	21.39	8.55
insertions	0.10	1.63	7.18	2.90	8.28	3.16
deletions	0.27	1.11	7.71	5.74	10.17	7.77
insertion-blocks	0.49	-0.68	3.48	1.51	3.22	1.52
deletion-blocks	0.74	-0.34	4.20	1.91	4.22	2.34

Table 6.7: Correspondence Metrics for the Model of Figure 6.6.

Event Type	P-value (2-tailed)	Sig Test (Z)	Accept Pop. # Events Occurring	Accept Pop. Proportion Matched	Reject Pop. # Events Occurring	Reject Pop. Proportion Matched
carod-closed	0.01	2.63	126	0.52	17	0.18
carod-delivered	0.01	2.69	126	0.52	17	0.18
carod-solved	0.07	1.82	126	0.40	17	0.18

Event Type	P-value (2-tailed)	Sig Test (Z)	Accept Pop. # Events Predicted	Accept Pop. Proportion Matched	Reject Pop. # Events Predicted	Reject Pop. Proportion Matched
carod-delivered	0.01	2.74	91	0.73	10	0.30
code-checkin	0.01	2.50	836	0.97	80	0.91
carod-update	0.02	2.31	128	0.88	19	0.68
carod-closed	0.04	2.10	89	0.73	8	0.38
carod-response	0.05	-1.93	165	0.60	25	0.80
carod-solved	0.06	1.88	112	0.46	15	0.20

Table 6.8: Per-Event-Type Metrics for the Model of Figure 6.6. Only those event types that had significant measures are shown.

delay exceeds some significant threshold; and they can consider better ways to structure access to code so that members of the organization are motivated to use the check-in mechanism. In fact, the organization has already begun to make improvements based on our study.

6.4.2 Evaluation of the Validation and Discovery Methods

We conclude that this study provides significant evidence that the discovery and validation methods are both useful and practical in a real-world setting.

The validation metrics played a central role in the study, being used to statistically analyze the process executions against their resulting outcomes (i.e., accepted or rejected product). This study showed that the validation metrics can and do capture important information about the process behavior, since they were successful in separating the two populations.

Furthermore, the extensibility of the metrics was shown in the detailed per-event-type and per-model-state measurements. These uses show the ability of the validation metrics to enable more detailed, situation-specific analyses that can further elucidate the processes' behavior.

The discovery tools, while not playing as central a part in this study, showed their usefulness in providing a good model for the process data. Since the prescriptive model was not very good at matching the event data, using the discovery tools allowed us to create a process model that was descriptive of the actual behavior of the process. This model better matched what was actually done, giving a clear and understandable visualization of the real process. Additionally, with the validation tools, it too was able to statistically distinguish between the accepted and rejected populations. Thus, it captured the important behavioral aspects of the process.

Given the large amount of data (159 event streams), and the high variability of the event data

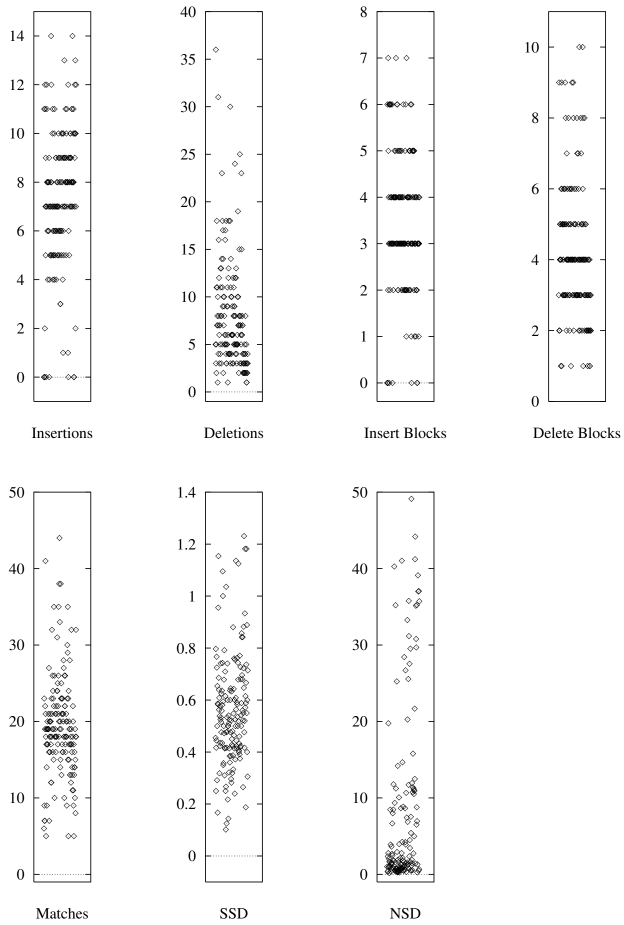

Figure 6.7: Distribution Plots of the Correspondence Metric Data for the Model of Figure 6.6.

(even the discovered model only matched 65% of it), without the discovery tools it would have been very difficult to write down a model that captured even some of the behavior of the event streams. This application of the discovery methods also shows the need for, and the usefulness of, parameters to the methods to control the complexity of the generated models and ignore noise in the event streams.

Model State	P-value (2-tailed)	Sig Test (Z)	Accept Pop. # Events Occurring	Accept Pop. Proportion Matched	Reject Pop. # Events Occurring	Reject Pop. Proportion Matched
16	0.00	3.19	174	0.53	34	0.24
27	0.03	2.17	76	0.46	27	0.22
11	0.04	-2.01	203	0.82	19	1.00
24	0.05	1.95	126	0.95	18	0.83

Model State	P-value (2-tailed)	Sig Test (Z)	Accept Pop. # Events Predicted	Accept Pop. Proportion Matched	Reject Pop. # Events Predicted	Reject Pop. Proportion Matched
23	0.01	2.69	126	0.52	17	0.18
22	0.02	2.33	126	0.52	18	0.22
6	0.05	-1.94	127	0.68	19	0.89
11	0.07	1.84	218	0.77	31	0.61
21	0.07	1.82	126	0.40	17	0.18
15	0.10	1.64	130	0.72	17	0.53

Table 6.9: Per-Model-State Metrics for the Model of Figure 6.6. Only those model states that had significant measures are shown.

6.5 Conclusion

Process research has assumed that an organization will be led to a good product by creating and following a prescribed process. Actually demonstrating this is critical to the acceptance of process research results in industry. The work described in this chapter is a significant first step toward that goal. We presented the results of a study examining the relationship among process models, executions, and outcomes for a real-world industrial process. Using both simple aggregation metrics and sophisticated process data analysis tools, we were able to show correlations between deviations from the prescribed process and the presence of defects in the product. Moreover, the results were gained at a sufficiently detailed level to permit recommendations of relatively specific places to improve the process.

The question of how widely the results of this study can be applied is important. While our subject process, the management of customer-initiated software updates, is fairly common in industry, the answer will come only with more studies like this one. Fortunately, our study also successfully demonstrates that one major cost of performing such studies, namely data collection, need not be so high as might be expected. In particular, we showed that it is feasible to perform a process study by using historical data that are readily available—that is, already collected manually or automatically during the regular course of the process for purposes other than the study at hand, without the addition of new and specific instrumentation.

The main disadvantage of this approach to data gathering is that the study effectively becomes a naturally occurring experiment whose subject cannot be statistically controlled or varied in order to test hypotheses. Thus, substantive conclusions about causality are limited.

Nevertheless, there are two benefits of this approach that should not be undervalued. First, large amounts of data can be collected without disturbing the process and the people involved. This lack of intrusion is particularly important for researchers who are interested in studying real-world organizations. Second, using previously collected, readily available data is much less costly than instrumenting to collect new data. The results from a study such as this can then be used to focus subsequent instrumentation, providing a cost-effective path to data-driven process improvement. This two-step approach overcomes the problem of convincing an organization to implement a potentially costly instrumentation of their process, providing a solid foundation from which to argue its benefits.

This study has also evaluated the techniques for process discovery and validation that have been developed in this thesis. We have shown that the process validation techniques in this thesis do in fact capture important qualities about the process execution, and that the metrics can be correlated to the product quality produced by the process. Also, we demonstrated that the process discovery techniques significantly reduced the effort in understanding the process as it had been executed. Indeed, the variability in the event data and the amount of event data would have made it very difficult to find the important process patterns without the help of the discovery tools.

Thus, this study gives significant reason to believe that the techniques developed in this thesis will be useful in a real-world setting, and can affect software processes, and more importantly products, in a positive manner.

Chapter 7

Conclusion

In this thesis, we have explored two problems in software process, these of process discovery and process validation. We have devised and implemented methods that solve these two problems, and have validated the work by showing that the methods work in a real-world setting.

In process discovery, we extended an implementation of a neural net inference method, enhanced an algorithmic method and implemented it, and invented and implemented a probabilistic method. The neural net method, RNet, did not prove to be practical to use. However, both the algorithmic method, Ktail, and our probabilistic method, Markov, have been demonstrated to be able to infer patterns in process event data, and to be efficient in doing so.

In process validation, we devised a paradigm for validation in the context of process models and event data, that of string distance measurements, and specified quantitative metrics in that paradigm. We showed that, while computing these metrics can be very hard, a heuristic method for this calculation performs very well both in terms of speed and the quality of its results.

An industrial study was undertaken to determine if our methods can operate on real-world data. We showed that both the discovery and validation methods can be successfully applied, and that they can capture the important features in a process. Furthermore, we also showed that following a process can be statistically related to producing a good product.

Finally, we have implemented an extensible framework for building process data analysis tools, called Balboa, that eases the burden of managing event data and building tools to access the data. The discovery and validation tools are built on this framework, and other tools have been and are being built as well.

This thesis, then, has contributed analysis methods and tools to the body of computer science research. The rest of this chapter describes how these methods can be applied outside of software process, and directions for future work along the lines explored in this thesis.

7.1 Other Applications of this Work

The techniques developed in this thesis are applicable in a broader scope than just software process. Other areas that can characterize a system's execution with event data (sometimes called trace data) can potentially benefit from the process discovery techniques, and when a system's behavior must be verified against an abstract representation of the system, then the process validation methods can be applied.

This section presents thoughts on the application of our techniques outside of software process, and some potential domains of interest.

7.1.1 Process Discovery

What process discovery does is essentially just provide a model of the patterns that occur in some sequence. Thus, wherever a trace of some system can be collected, process discovery can potentially help understand the recurring patterns that that system undergoes.

The need to understand a system has widespread application in software engineering, including debugging (especially distributed debugging), reengineering of existing legacy systems, maintenance of a poorly documented system, and other similar activities. We feel that our process discovery techniques can be successfully applied to these areas.

An important role that the discovery tools can play is that of visualization tools. That is, not only do they provide discovery of the behavior, but they provide visual representation of it as well. We have not explored this aspect heavily, but simply providing pictures of the behavior of a system can help in understanding it.

Applying the discovery methods with low threshold parameters will show detailed patterns in a system, while using high threshold parameters would allow an engineer to extract the "big picture" patterns from the event stream. This could, in effect, highlight the architecture of a system.

7.1.2 Process Validation

A model is essentially an abstract specification of a system. So, when there is a need to answer the question. "How close does the behavior match the specification?", our process validation methods can be applied. Thus, other areas in computer science can take advantage of our techniques, especially in the area of formal methods for specification and design.

Checking a system for adherence to a requirements specification or design document, for example, can be done using the validation methods. At first glance, one might think that adherence must be 100%, but in reality, since a complex specification or design might have conflicting, incorrect, or simply abstracted, parts, it is rare that the implementation of a large system exactly agrees with the specification. Validation can help measure its agreement, and highlight areas of the system that diverge from the specification, thus focussing the effort in building the system to the points where close agreement with the specifications is lacking.

In software architecture, which describes the high-level structure of a system, it is often accepted that the low-level details of the implementation may have to bend the rules of the architecture to successfully or satisfactorily implement the needed functionality. Callbacks in a client-server framework are an example of this. So, an important question is, how closely does an architecture specification actually describe the real system? Our techniques in validation can help answer the behavioral side of this question. Applying validation through the life of the product may also help identify architectural drift and assist in combatting this problem.

7.1.3 Systems with Trace Data

In this section we outline several potential domains where systems can provide trace data, and speculate on how our methods can be applied in these areas. In each of these domains, we have some examples of data, and have experimented with the data enough to believe that our techniques can be successfully applied.

Program Traces A trace of the method invocations of an object-oriented program represents the pattern of communication between objects. While a call graph can show the static potential communication, the actual dynamic behavior might be substantially different for different inputs. The process discovery methods we have developed can be used to visualize and understand the actual behavior of the system. Validation can be applied where a design specification of the program needs to be compared to the implemented system.

Subsystem Protocols The interface between subsystems in a large software system can be viewed as a messaging protocol. Capturing a message trace and discovering the protocol that the interface is using would be a useful in understanding the behavior of the system. It could lead to identifying strongly coupled subsystems, erroneous connections between modules, and could help in re-architecting an old, heavily evolved system.

On the validation side, an existing model of the protocol, being an abstraction of the real world, may idealize the protocol but not capture its exact implementation. Validation measurements could give results that show how closely the real protocol matches the model.

Message Logs A large telecommunications product produces large amounts of *ROP*[1] log data when in testing, where the logs are informational messages about the state of the system, and event occurrences internal to the system. In debugging, experts in the system and in reading the ROP logs use the logs to try to understand what is wrong in the system. Due to the volume of data, this approach has been likened to a search for the "needle in a haystack". This method of extensive searching is probably the wrong paradigm for this technology.

Using the discovery methods to discover patterns in the ROP traces could assist the engineers in understanding the trace data. These tools would only effectively show general and common patterns. Still, the reduction to a model from a large ROP trace might be very useful to someone analyzing the data, and these models could be used by the validation methods to highlight areas of the traces that differ from the general patterns.

[1]Read Only Printer.

7.2 Future Work

In this section we present directions for future work in process discovery, validation, and finally for the Balboa framework.

7.2.1 Process Discovery

Process discovery has several areas of potential work, including extensions to the methods themselves, and enhancements to the tools that implement them.

- Discovering concurrency in event streams.

  As presented in Chapter 3, there are some promising techniques to identify points in the event stream where concurrent behavior might be happening.

- Seeding discovery methods with potential models or pieces of models.

  This would take advantage of an engineer's existing knowledge of a process. Both Ktail and Markov, at first thought, would seem to naturally allow this, just by initializing them with equivalences classes and probabilities, respectively—although this may not translate to exactly the same model pieces that were seeded. A related issue is making the methods interactive, allowing the user to dynamically control the result by, for example, seeing the model produced so far and constraining certain portions of it to be fixed while the method continues.

- Exploring the extension of the discovery methods to other models.

  Much inference work includes grammar learning, but this does not naturally fit software process. Petri nets or rule bases are more appropriate extensions for this domain.

- Investigating the difference in behavior of the Bayesian versus forward probabilities for the Markov algorithm.

  Whether the Bayesian extension to Markov is better than the original has not been explored well. Some measure of goodness for deciding which is better would need to be devised, and then an exploration across different data and threshold parameters could help answer whether or not the Bayesian extension is indeed useful.

- Integrating Hidden Markov Model information with the Markov method.

  Markov really discovers a Hidden Markov Model, with the probabilities on transitions. However, this is not well integrated into the method's implementation and output. Augmenting the output with the probability information would provide more information to the user.

- Improve and explore the visualization aspect of the discovery methods.

  Our methods can act as visualization tools, and enhancements to our implementation may help in the understanding by the engineer. For example, drawing transitions with a thickness relative to their probability in the MARKOV-discovered model or drawing states with a shading relative to the size of the equivalence class in KTAIL may help the user in understanding the discovered model better.

- Assist the user in parameter selection.

  Better assistance in parameter selection for the methods would reduce the trial-and-error recursion that a process engineer must go through to apply the discovery methods. We have implemented a graphical viewer of the probabilities in the MARKOV method to help guide parameter selection, and a similar tools would be useful and easily implemented for KTAIL.

7.2.2 Process Validation

Process validation also has areas of future work that span the range from enhancements to tools to theoretical extensions.

- Adding control over starting states and ending states to the validation engine.

  A user may want to validate a piece of behavior that neither starts at an initial state nor ends at a final state. The final state situation is currently handled, but the validation engine currently always starts at an initial state as specified by the model.

- Identifying other properties of process models that can be exploited in doing validation calculations.

  For example, places in the model where an event stream can be pinned down could help reduce searches on large models (e.g., a key event might happen at a specific site, where all previous behavior can then be ignored). This idea is similar to the concept of trace change points [46].

- Implementing other modeling paradigms, such as Petri nets, for the metric-calculation engine.

- Developing techniques for better visualization of the measurements.

  For example, overlaying the differences onto the process model rather than onto the model event stream may help a process engineer in understanding the problems in the process.

- Investigating other analyses for process executions and process models.

  Time-oriented metrics, for example, would be very a useful extension to our validation methods. Real-time systems analysis techniques could be useful here [51, 109]. Methods for measuring the efficiency of a process would be another useful analysis method, which we explored a little in [36]. Both of these would help in the optimization of a process that has already been behaviorally validated.

- Identifying what formal properties are maintained in a behavior that does not agree 100% with a model.

Formal models are often useful because they can detect conflicts and consistency violations. Can these questions be answered about a behavior that doesn't exactly match the model, and can the validation methods provide leverage to help answer this?

7.2.3 BALBOA Framework

A number of possible enhancements to BALBOA can be explored as future work:

- Add other types of data support.

  While event data supports many varieties of analyses, certainly there are other forms of valid data to be used. A major direction to take BALBOA in the future would be to incorporate other types of process, and perhaps product, data into it.

- Implement a level of security.

  A tool authorization mechanism, where a server has knowledge of who is allowed to connect to it, and from where, would help protect sensitive data. This could be at a per-collection detail, allowing certain collections to be made public, while keeping others private.

- Add an understanding of time.

  In the data collection interface, it would seem probable that events may not be reported at the same time they occur. Thus, building into BALBOA some knowledge about the timestamps of events would allow it to ensure that an event stream is properly ordered, and would better support time-oriented queries on the client tool end.

- Allow hierarchies and sets of collections.

  Hierarchical event collections would be useful when, for example, separate subprocesses are collected into separate collections, but there is a need to view the whole process as well. Sets of event collections, related closely to hierarchical collections, would be useful when analyzing many executions from a single process.

REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM-SIGMOD 1993 International Conference on Managment of Data*, pages 207–216. ACM Press, May 1993.

[2] A.V. Aho and T.G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computing*, 1(4):305–312, December 1972.

[3] H. Ahonen, K. Mannila, and E. Nikunen. *Grammatical Inference and Applications*, volume 862 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 153–167. Springer-Verlag, New York, 1994.

[4] D. Angluin. Inductive inference of formal languages from positive data. *Information and Control*, 45:117–135, 1980.

[5] D. Angluin. Inference of reversible languages. *Journal of the ACM*, 29(3):741–765, July 1982.

[6] D. Angluin. Learning regular sets from queries and counter-examples. *Information and Computation*, 75:87–106, 1987.

[7] D. Angluin and C.H. Smith. Inductive inference: Theory and methods. *ACM Computing Surveys*, 15(3):237–269, September 1983.

[8] A. Apostolico, M.J. Atallah, L.L. Larmore, and S. McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM Journal on Computing*, 19(5):968–988, 1990.

[9] *AT&T Technical Journal*, volume 70:2. AT&T Corporation, March/April 1991.

[10] G.S. Avrunin, U.A. Buy, J.C. Corbett, L.K. Dillon, and J.C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, November 1991.

[11] S. Bandinelli, A. Fuggetta, and C. Ghezzi. Software Process Model Evolution in the SPADE Environement. *IEEE Transactions on Software Engineering*, 19(12):1128–1144, December 1993.

[12] S. Bandinelli, A. Fuggetta, C. Ghezzi, and L. Lavazza. SPADE: An Environment for Software Process, Analysis, Design, and Enactment. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modeling and Technology*, pages 223–248. Wiley, 1994.

[13] S. Bandinelli, C. Ghezzi, and A. Morzenti. A Multi-Paradigm Petri Net Based Approach to Process Description. In *Proceedings of the 7th International Software Process Workshop*, pages 41–43, October 1991.

[14] N.S. Barghouti and G.E. Kaiser. Scaling Up Rule-based Development Environments. In *Proceedings of the Third European Software Engineering Conference*, number 550 in Lecture Notes in Computer Science, pages 380–395. Springer-Verlag, October 1991.

[15] N.S. Barghouti and B. Krishnamurthy. Using event contexts and matching constraints to monitor software processes. In *Proceedings of the 17th International Conference on Software Engineering*, pages 83–92. IEEE Computer Society Press, April 1995.

[16] V.R. Basili and D.M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, 10(6):728–737, 1984.

[17] P. Bates. EBBA modelling tool a.k.a. event definition language. Technical Report COINS-87-35, University of Massachusetts at Amherst, 1987.

[18] P. Bates. Shuffle automata: A formal model for behavior recognition in distributed systems. Technical Report COINS-87-27, University of Massachusetts at Amherst, 1987.

[19] P. Bates. Debugging heterogenous systems using event-based models of behavior. In *Proceedings of a Workshop on Parallel and Distributed Debugging*, pages 11–22. ACM Press, January 1989.

[20] I.S. Ben-Shaul and G.E. Kaiser. A paradigm for decentralized process modeling and it realization in the OZ environment. In *Proceedings of the 16th International Conference on Software Engineering*, pages 179–188. IEEE Computer Society Press, May 1994.

[21] I. Bhandari, M. Halliday, E. Tarver, D. Brown, J. Chaar, and R. Chillarege. A Case Study of Software Process Improvement During Development. *IEEE Transactions on Software Engineering*, 19(12):1157–1170, December 1993.

[22] A.W. Biermann and J.A. Feldman. On the Synthesis of Finite State Machines from Samples of Their Behavior. *IEEE Transactions on Computers*, 21(6):592–597, June 1972.

[23] M.G. Bradac, D.E. Perry, and L.G. Votta. Prototyping a process monitoring experiment. *IEEE Transactions on Software Engineering*, pages 774–784, October 1994.

[24] A. Brāzma. *Algorithmic Learning Theory*, volume 872 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 260–271. Springer-Verlag, New York, 1994.

[25] A. Brāzma and K. Čerāns. *Algorithmic Learning Theory*, volume 872 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 76–90. Springer-Verlag, New York, 1994.

[26] A. Brāzma, I. Jonassen, I. Eidhammer, and D. Gilbert. Approaches to the automatic discovery of patterns in biosequences. Technical Report TCU/CS/1995/18, City University (London), December 1995.

[27] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:141–170, 1992.

[28] R.C. Carrasco and J. Oncina. *Grammatical Inference and Applications*, volume 862 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 139–152. Springer-Verlag, New York, 1994.

[29] J. Carrol and D. Long. *Theory of Finite Automata*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[30] F. Casacuberta. *Grammatical Inference and Applications*, volume 862 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 119–129. Springer-Verlag, New York, 1994.

[31] A. Castellanos, I. Galiano, and E. Vidal. *Grammatical Inference and Applications*, volume 862 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 93–105. Springer-Verlag, New York, 1994.

[32] E. Charniak. *Statistical Language Learning*. The MIT Press, Cambridge, Massachusetts, 1993.

[33] S.F. Chen. Bayesian grammar induction for language modeling. Technical Report TR-01-95, Harvard University, Center for Research in Computing Technology, January 1995.

[34] L.J. Chmura, A.F. Norcio, and T.J. Wicinski. Evaluating software design process by analyzing change data over time. *IEEE Transactions on Software Engineering*, 16(7):729–739, July 1990.

[35] L.J. Chmura, A.F. Norcio, and T.J. Wicinski. Evaluating Software Design Processes by Analyzing Change Data Over Time. *IEEE Transactions on Software Engineering*, 16(7):729–739, July 1990.

[36] J.E. Cook and A.L. Wolf. Toward Metrics for Process Validation. In *Proceedings of the Third International Conference on the Software Process*, pages 33–44. IEEE Computer Society, October 1994.

[37] J. Corbett and A. Polk. A tool for automatic generation of behaviors for constrained expression analysis. Technical report, February 1993.

[38] J. Cuny, G. Forman, A. Hough, J. Kundu, C. Lin, L. Snyder, and D. Stemple. The adriane debugger: Scalable application of event-based abstraction. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–95. ACM Press, May 1993.

[39] S. Das and M.C. Mozer. A Unified Gradient-Descent/Clustering Architecture for Finite State Machine Induction. In *Proceedings of the 1993 Conference*, number 6 in Advances in Neural Information Processing Systems, pages 19–26. Morgan Kaufmann, 1994.

[40] W. Deiters and V. Gruhn. Managing Software Processes in the Environment MELMAC. In *SIGSOFT '90: Proceedings of the Fourth Symposium on Software Development Environments*, pages 193–205. ACM SIGSOFT, December 1990.

[41] W. Edwards Deming. *Out of Crisis*. MIT Press, Cambridge, MA, 1982.

[42] J.L. Devore. *Probability and Statistics for Engineering and the Sciences*. Brooks/Cole, Pacific Grove, California, 3rd edition, 1991.

[43] *Digital Technical Journal*, volume 5:4. Digital Equipment Corporation, Fall 1993.

[44] L.K. Dillon, G.S. Avrunin, and J.C. Wileden. Constrained expressions: Toward broad applicability of analysis methods for distributed software systems. *ACM Transactions on Programming Languages and Systems*, 10(3):374–402, July 1988.

[45] M.W. Du and S.C. Chang. A model and a fast algorithm for multiple errors spelling correction. *Acta Informatica*, 29:281–302, 1992.

[46] Z. K. F. Eckert and G. J. Nutt. Trace extrapolation for parallel programs on shared memory multiprocessors. Technical Report TR CU-CS-804-96, Department of Computer Science, University of Colorado, May 1996.

[47] K. El Emam, N. Moukheiber, and N.H. Madhavji. An evaluation of the G/Q/M method. Technical Report MCGILL/SE-94-11, McGill University, 1994.

[48] D. Eppstein. Sequence comparison with mixed convex and concave costs. *Journal of Algorithms*, 11:85–101, 1990.

[49] M.E. Fagan. Advances in Software Inspections. *IEEE Transactions on Software Engineering*, SE-12(7):744–751, July 1986.

[50] M. Felder, D. Mandrioli, and A. Morzenti. Proving Properties of Real-time Systems Through Logical Specifications and Petri Net Models. *IEEE Transactions on Software Engineering*, 20(2):127–141, February 1994.

[51] M. Felder and A. Morzenti. Validating Real-time Systems by History-checking TRIO Specifications. In *Proceedings of the 14th International Conference on Software Engineering*, pages 199–211. IEEE Computer Society, May 1992.

[52] C.N. Fischer and J. Mauney. A simple, fast, and effective LL(1) error repair algorithm. *Acta Informatica*, 29:109–120, 1992.

[53] P. Garg and M. Jazayeri. Process-centered software engineering environments: A grand tour. In *Software Process*, Trends in Software, pages 25–52. Wiley, 1996.

[54] P.K. Garg and S. Bhansali. Process programming by hindsight. In *Proceedings of the 14th International Conference on Software Engineering*, pages 280–293. IEEE Computer Society Press, May 1992.

[55] P.K. Garg and S. Bhansali. Process Programming by Hindsight. In *Proceedings of the 14th International Conference on Software Engineering*, pages 280–293. IEEE Computer Society, May 1992.

[56] P.K. Garg, M. Jazayeri, and M.L. Creech. A Meta-Process for Software Reuse, Process Discovery, and Evolution. In *Proceedings of the 6th International Workshop on Software Reuse*, November 1993.

[57] P.K. Garg, M. Jazayeri, and M.L. Creech. A meta-process for software reuse, process discovery and evolution. In *Proceedings of the 6th International Workshop on Software Reuse*, page [need pages], November 1993.

[58] C. Gerety. HP SoftBench: A New Generation of Software Development Tools. Technical Report SESD–89–25, Hewlett-Packard Software Engineering Systems Division, Fort Collins, Colorado, November 1989.

[59] E.M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[60] E.M. Gold. Complexity of automatic identification from given data. *Information and Control*, 37:302–320, 1978.

[61] R.M. Greenwood. Using CSP and System Dynamics as Process Engineering Tools. In *Proceedings of the Second European Workshop on Software Process Technology*, number 635 in Lecture Notes in Computer Science, pages 138–145. Springer-Verlag, September 1992.

[62] J. Grudin. Groupware and Cooperative Work: Problems and Prospects. In R.M. Baeker, editor, *Groupware and Computer-Supported Cooperative Work*, pages 97–105. Morgan Kaufmann, 1993.

[63] V. Gruhn and R. Jegelka. An Evaluation of FUNSOFT Nets. In *Proceedings of the Second European Workshop on Software Process Technology*, number 635 in Lecture Notes in Computer Science, pages 196–214. Springer-Verlag, September 1992.

[64] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[65] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. In *Proceedings of the 10th International Conference on Software Engineering*, pages 396–406. IEEE Computer Society, April 1988.

[66] D. Heimbigner. The ProcessWall: A Process State Server Approach to Process Programming. In *SIGSOFT '92: Proceedings of the Fifth Symposium on Software Development Environments*, pages 159–168. ACM SIGSOFT, December 1992.

[67] K.E. Huff. Software process modelling. In *Software Process*, Trends in Software, pages 1–24. Wiley, 1996.

[68] K.E. Huff and V.R. Lesser. A plan-based intelligent assistant that supports the software development process. In *Proc. 3rd ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 97–106. ACM Press, February 1989.

[69] W.S. Humphrey. *Managing the Software Process.* Addison-Wesley, Reading, Massachusetts, 1989.

[70] W.S. Humphrey. *A Discipline for Software Engineering.* SEI Series in Software Engineering. Addison-Wesley, 1995.

[71] A. Hutchinson. *Algorithmic Learning.* Graduate Texts in Computer Science. Oxford University Press, 1994.

[72] *IBM Systems Journal*, volume 33:1. IBM Corporation, 1994.

[73] *IEEE Software*, volume 11:4. IEEE Press, July 1994.

[74] ISO 9000-3 guidelines for the application of ISO 9001 to the development, supply, and maintenance of software. Technical report, International Standards Organization, 1991.

[75] P. Inverardi, B. Krishnamurthy, and D. Yankelevich. Yeast: A Case Study for a Practical use of Formal Methods. In *TAPSOFT '93: Proceedings of the 4th International Joint Conference CAAP/FASE*, number 668 in Lecture Notes in Computer Science, pages 105–120. Springer-Verlag, April 1993.

[76] M.L. Jaccheri and R. Conradi. Techniques for Process Model Evolution in EPOS. *IEEE Transactions on Software Engineering*, 19(12):1145–1156, December 1993.

[77] S. Jain and A. Sharma. *Algorithmic Learning Theory*, volume 872 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 349–364. Springer-Verlag, New York, 1994.

[78] C.M. Judd, E.R. Smith, and L.H. Kidder. *Research Methods in Social Relations.* Holt, Rinehart and Winston, Inc., Fort Worth, sixth edition, 1991.

[79] R.L. Kashyap and B.J. Oommen. The noisy substring matching problem. *IEEE Transactions on Software Engineering*, 9(3):365–370, 1983.

[80] M.I. Kellner. Software Process Modeling Support for Management Planning and Control. In *Proceedings of the First International Conference on the Software Process*, pages 8–28. IEEE Computer Society, October 1991.

[81] M.I. Kellner, P.H. Feiler, A. Finkelstein, T. Katayama, L.J. Osterweil, M.H. Penedo, and H.D. Rombach. Software Process Modeling Example Problem. In *Proceedings of the 6th International Software Process Workshop*, pages 19–29, October 1990.

[82] J.R. Knight and E.W. Myers. Approximate regular expression pattern matching with concave gap penalties. *Algorithmica*, 14:85–121, 1995.

[83] E. Koutsofios and S.C. North. Drawing Graphs with Dot. AT&T Bell Laboratories, October 1993.

[84] B. Krishnamurthy and D.S. Rosenblum. Yeast: A General Purpose Event-Action System. *IEEE Transactions on Software Engineering*, 21(10):845–857, October 1995.

[85] J.B. Kruskal. An Overview of Sequence Comparison. In D. Sankoff and J.B. Kruskal, editors, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, pages 1–44. Addison-Wesley, Reading, Massachusetts, 1983.

[86] S. Lange, J. Nessel, and R. Wiehagen. *Algorithmic Learning Theory*, volume 872 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 423–437. Springer-Verlag, New York, 1994.

[87] S. Lange and P. Watson. *Algorithmic Learning Theory*, volume 872 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 438–452. Springer-Verlag, New York, 1994.

[88] R.J. LeBlanc and A.D. Robbins. Event-Driven Monitoring of Distributed Programs. In *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pages 515–522. IEEE Computer Society, May 1985.

[89] C.M. Lott. Process and measurement support in SEEs. *SIGSOFT Software Engineering Notes*, 18(4):83–93, October 1993.

[90] S.Y. Lu and K.S. Fu. Error-correcting tree automata for syntactic pattern recognition. *IEEE Transactions on Computers*, C-27:1040–1053, November 1978.

[91] L. Miclet. *Syntactic and Structural Pattern Recognition: Theory and Applications*, volume 7 of *Series in Computer Science*, chapter 9: Grammatical Inference, pages 237–290. World Scientific, New Jersey, 1990.

[92] L. Miclet and J. Quinqueton. *Syntactic and Structural Pattern Recognition*, volume 45 of *NATO ASI Series F: Computer and Systems Sciences*, pages 153–171. Springer-Verlag, New York, 1988.

[93] E.W. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51(1):5–37, 1989.

[94] M. Paulk, B. Curtis, M. Chrissis, and C. Weber. Capability maturity model for software, version 1.1. Technical Report CMU/SEI-93-TR-24, Software Engineering Institute, February 1993.

[95] M. Paulk, B. Curtis, M. Chrissis, and C. Weber. Capability Maturity Model, Version 1.1. *IEEE Software*, 10(4):18–27, July 1993.

[96] B. Peuschel and W. Schäfer. Concepts and Implementation of a Rule-based Process Engine. In *Proceedings of the 14th International Conference on Software Engineering*, pages 262–279. IEEE Computer Society, May 1992.

[97] L. Pitt. *Analogical and Inductive Inference*, volume 397 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 18–44. Springer-Verlag, New York, 1989.

[98] A. Porter, H. Siy, C.A. Toman, and L.G. Votta. An experiment to assess the cost-benefits of code inspections in large scale software development. In *Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 92–103. ACM Press, October 1995.

[99] R.S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Reading, MA, third edition, 1992.

[100] S.P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, pages 57–66, July 1990.

[101] E. Rich. *Artificial Intelligence*. McGraw-Hill Series in Artificial Intelligence. McGraw-Hill, 1983.

[102] J. Röhrich. Methods for the automatic construction of error correcting parsers. *Acta Informatica*, 13:115–139, 1980.

[103] M. Saeki, T. Kaneko, and M. Sakamoto. A Method for Software Process Modeling and Description Using LOTOS. In *Proceedings of the First International Conference on the Software Process*, pages 90–104. IEEE Computer Society, October 1991.

[104] Y. Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97:23–60, 1992.

[105] Y. Sakakibara. Grammatical inference: An old and new paradigm. Technical Report ISIS-RR-95-9E, Institute for Social Information Science, September 1995.

[106] J.A. Sánchez and J.M. Benedí. *Grammatical Inference and Applications*, volume 862 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 130–138. Springer-Verlag, New York, 1994.

[107] D. Sankoff and J.B. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, Reading, Massachusetts, 1983.

[108] M. Schneider, H. Lim, and W. Schoaff. The utilization of fuzzy sets in the recognition of imperfect strings. *Fuzzy Sets and Systems*, 49:331–337, 1992.

[109] R.L. Schwartz, P.M. Melliar-Smith, and F.H. Vogt. In Interval Logic for Higher-level Temporal Reasoning. In *Proceedings of the Second ACM Symposium on Principles of Distributed Computing*, pages 173–186. Association for Computer Machinery, August 1983.

[110] R.W. Selby, A.A. Porter, D.C. Schmidt, and J. Berney. Metric-Driven Analysis and Feedback Systems for Enabling Empirically Guided Software Development. In *Proceedings of the 13th International Conference on Software Engineering*, pages 288–298. IEEE Computer Society, May 1991.

[111] A. Stolcke. *Grammatical Inference and Applications*, volume 862 of *Lecture Notes in Artificial Intelligence (subseries of LNCS)*, pages 106–118. Springer-Verlag, New York, 1994.

[112] *An Introduction to the ToolTalk Service.* Sun Microsystems, Inc., 1991.

[113] S.M. Sutton, Jr. Accommodating Manual Activities in Automated Process Programs. In *Proceedings of the 7th International Software Process Workshop*, October 1991.

[114] S.M. Sutton, Jr., D. Heimbigner, and L.J. Osterweil. Language Constructs for Managing Change in Process-Centered Environments. In *SIGSOFT '90: Proceedings of the Fourth Symposium on Software Development Environments*, pages 206–217. ACM SIGSOFT, December 1990.

[115] S.M. Sutton, Jr., D. Heimbigner, and L.J. Osterweil. APPL/A: A Language for Software Process Programming. *ACM Transactions on Software Engineering and Methodology*, 4(3):221–286, July 1995.

[116] S.M. Sutton, Jr., H. Ziv, D. Heimbigner, H.E. Yessayan, M. Maybee, , L.J. Osterweil, and X. Song. Programming a Software Requirements-specification Process. In *Proceedings of the First International Conference on the Software Process*, pages 68–89. IEEE Computer Society, October 1991.

[117] TickIT: A guide to software quality management system construction and certification using EN29001, issue 2.0. Technical report, UK Department of Trade and Industry, and the British Computer Society, London, 1992.

[118] L.G. Valiant. A theory of the learnable. *Communications of the ACM*, 27:1134–1142, 1984.

[119] L.G. Votta and M.L. Zajac. Design process improvement case study using process waiver data. In *Proceedings of the Fifth European Software Engineering Conference (ESEC'95)*, pages 44–58. Springer-Verlag, September 1995.

[120] M.S. Waterman. General methods of sequence comparison. *Bulletin of Mathematical Biology*, 46:473–501, 1984.

[121] A.L. Wolf and D.S. Rosenblum. A Study in Software Process Data Capture and Analysis. In *Proceedings of the Second International Conference on the Software Process*, pages 115–124. IEEE Computer Society, February 1993.

[122] A.L. Wolf and D.S. Rosenblum. Process-centered Environments (Only) Support Environment-centered Processes. In *Proceedings of the 8th International Software Process Workshop*, pages 148–149, March 1993.

[123] P.H. Worley. A new PICL trace file format. Technical Report ORNL/TM-12125, Oak Ridge National Laboratory, 1992.

[124] Z. Zeng, R.M. Goodman, and P. Smyth. Learning finite state machines with self-clustering recurrent networks. *Nueral Computation*, 5:976–990, 1993.