DRAFT: November 1, 1996 13:52

Software Process Validation: Quantitatively Measuring the Correspondence of a Process to a Model Using Event-Based Data

Jonathan E. Cook and Alexander L. Wolf

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

{jcook,alw}@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-8??-96 November 1996

To a great extent, the usefulness of a formal model of a software process lies in its ability to accurately predict the behavior of the executing process. Similarly, the usefulness of an executing process lies largely in its ability to fulfill the requirements embodied in a formal model of the process. When process models and process executions diverge, something significant is happening.

We have developed techniques for uncovering and measuring the discrepancies between models and executions, which we call process validation. *Process validation takes a process execution and a process model, and measures the level of correspondence between the two. Our metrics our tailorable and give the engineers control over determining the severity of different types of discrepancies. The metrics are also hierarchical, providing detailed information once a high-level measurement indicates the presence of a problem.*

We have applied our process validation methods in a real-world, industrial study, of which a small portion is highlighted in this paper. The success of our techniques lead us to view this work as a first step toward a suite of useful methods for process validation.

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.

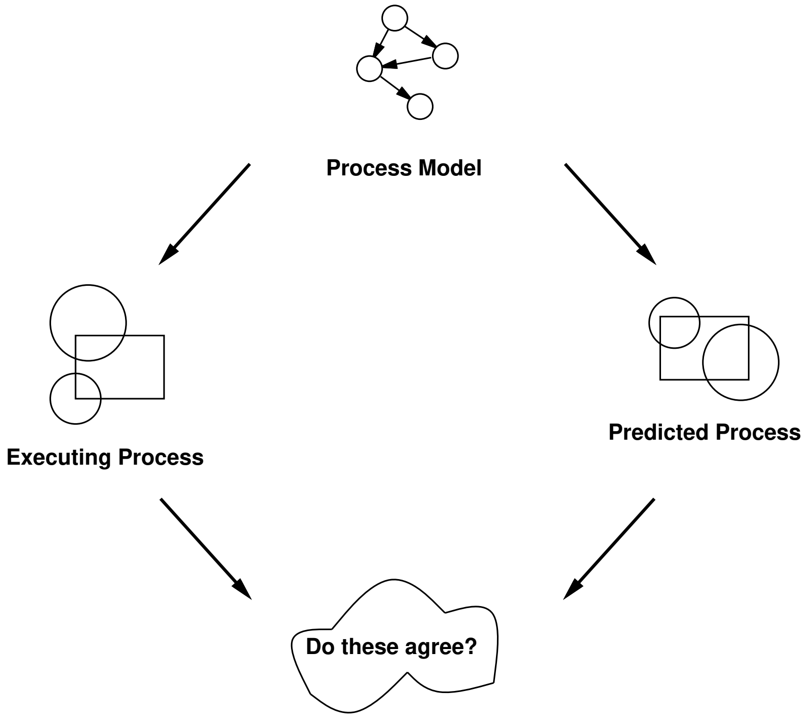

Figure 1: The Process Validation Problem.

1 Introduction

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 by the specification agrees with the actual behavior of system is a serious one. Software process modeling, where the model is a specification and the system is the process as executed by people, is not exempt from this problem, as Figure 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 [52, 55, 56]. 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 [30]; 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 [4, 35]. 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.

We have developed techniques for detecting and characterizing differences between a 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 paper 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 and methods presented here as a first step toward a suite of useful methods for process validation.

The next section presents the data analysis framework in which this work rests. Section 3 describes the background and related work. Section 4 presents the definitions of our validation metrics, Section 5 presents example uses of the metrics, Section 6 discusses extensions that make the methods more usable, and Section 7 describes how one can compute the metrics. Section 8 describes the implementation of the validation tools, and Section 9 evaluates the same. Section 10 presents a short overview of an industrial study that successfully applied the validation techniques presented here. Section 11 then summarizes our work in process validation, and gives ideas for future directions.

2 An Event-Based Framework for Process Validation

The foundation on which our process validation work 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.

Following Wolf and Rosenblum [55], 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 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 [42], concurrent-system analysis [3], and distributed debugging [10, 19]. We feel it is applicable to software process as well.

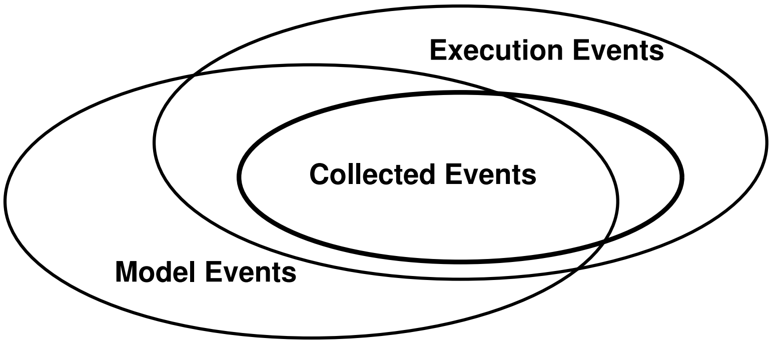

Figure 2: Venn Diagram of Event Types.

2.1 Relating Models and Events: Event Sites

For our work in process validation, we focus on behavior formalisms in process modeling languages. Examples are the Petri net foundation in Slang [5], the generic state machines, and the more specific state language Statecharts [32]. In viewing a process execution as 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, 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.

2.2 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. 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 step.

A third set of event types is the set of those event types that are actually collected as data, also shown in Figure 2. 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; nothing is known about how the execution corresponds to the rest of the model.

For a more detailed presentation of event data, systems that can collect event data, and using event data for analysis purposes, please see [17].

3 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 3. In describing this paradigm, we will put off discussing how to generate the model event stream until Section 7.

There are several methods for doing a measurement such as this, but one that seems most widely applied is string distance metrics [41]. A string distance metric 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 ([54]), substring matching ([37, 49]), spelling correction ([22]), syntax error correction ([1, 27, 47]), and even the well-known Unix *diff* program. A good reference to the general area of sequence comparison is [41].

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

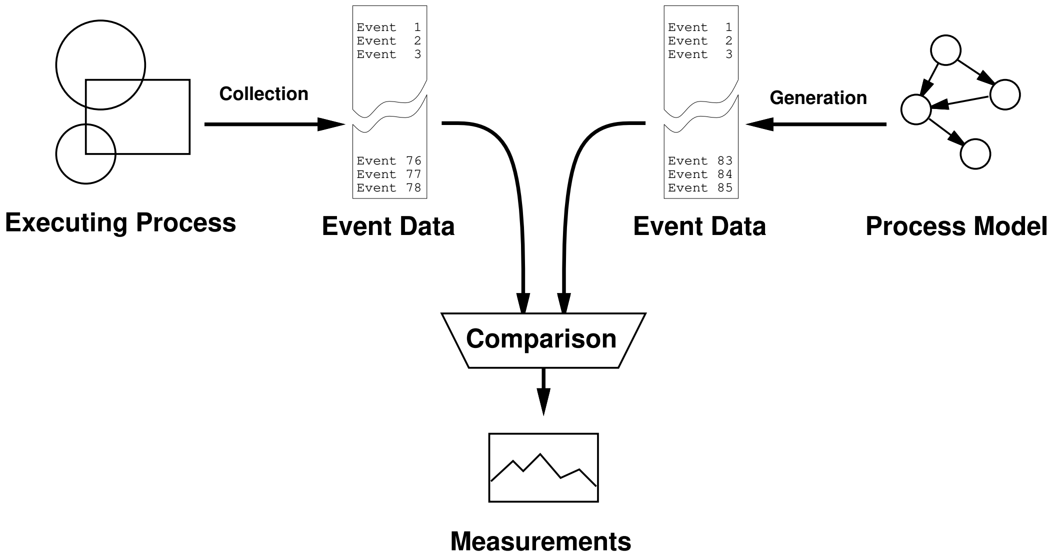

Figure 3: The Process Validation Abstract Solution.

where symbolic sequence comparison is taking place.[1] String distance methods are also generalizable to tree and graph distance measurements [43].

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 [1], 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 7 discusses the requirements that we place on a modeling language.

3.1 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 mathematical function (e.g., a time series of stock value), is a different topic altogether.

- Chmura et al. [14] and Bhandari et al. [11] 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. [51] 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 [7] 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 [38] 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 [28] 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. [29] 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. [12] 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 [55] 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. [45] use a well constructed experimental setting to assess the benefits of code inspections, in terms of group size and meeting gains.

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

3.2 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 [8, 9, 10] 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. [19] 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. [25, 26] 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 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 7. We defer detailed examples of applying the metrics to Section 5.

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 [33]), Petri nets (e.g., Slang [6] and FUNSOFT Nets [31]), and procedural languages (e.g., APPL/A [53]).

4.1 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(a), where the lettered boxes indicate events. The recognition metric applied to the two streams would result, as shown in Figure 4(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 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 [41] 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 [48].

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(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(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 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 [41].

[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.

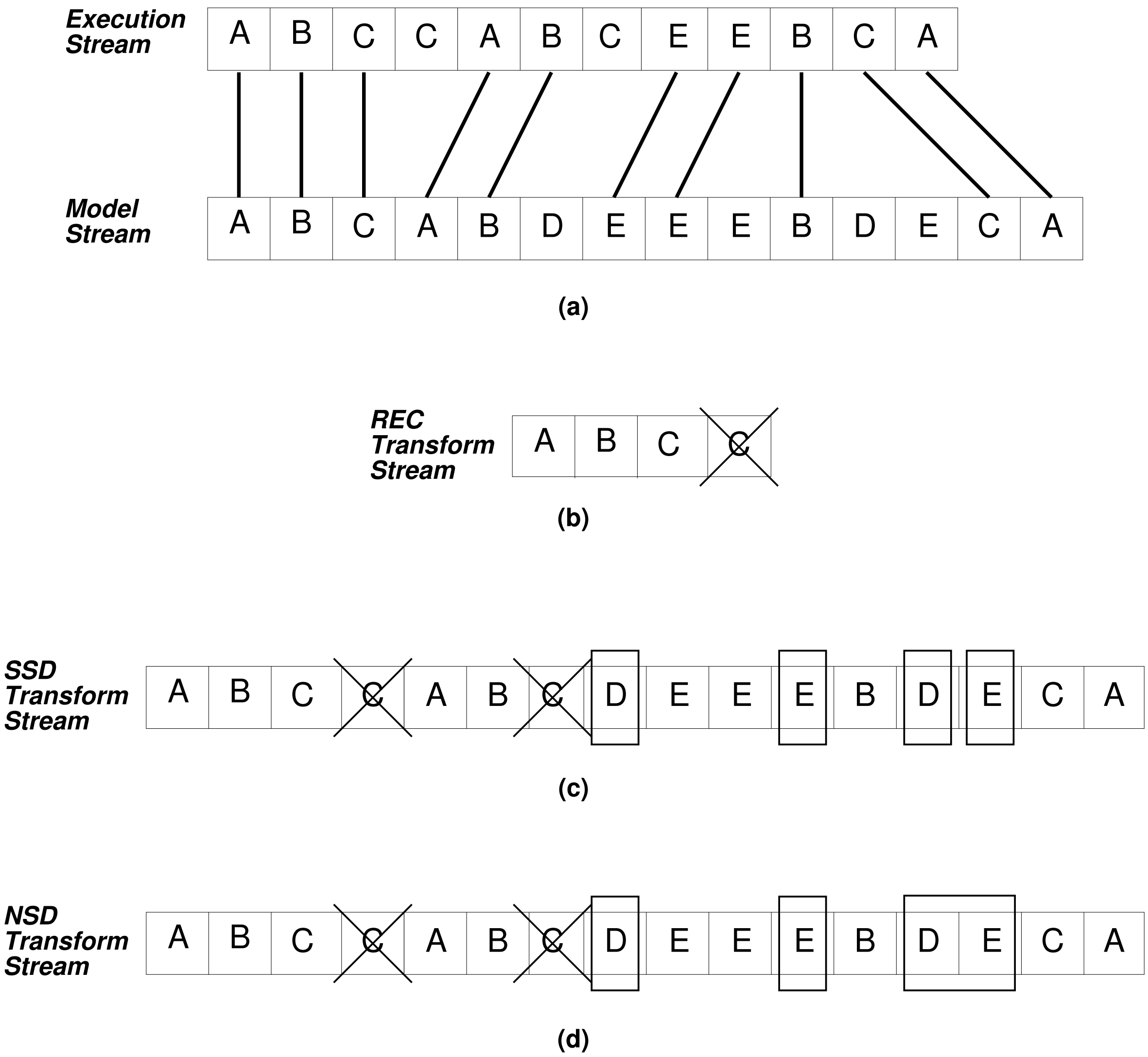

Figure 4: Two Event Streams and One Possible Correspondence of Events.

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 unlikely. Thus, one might pick the standard statistical correlation rules of thumb [20] 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.)

4.3 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(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.

5 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 [39]. 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 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 5 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 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.

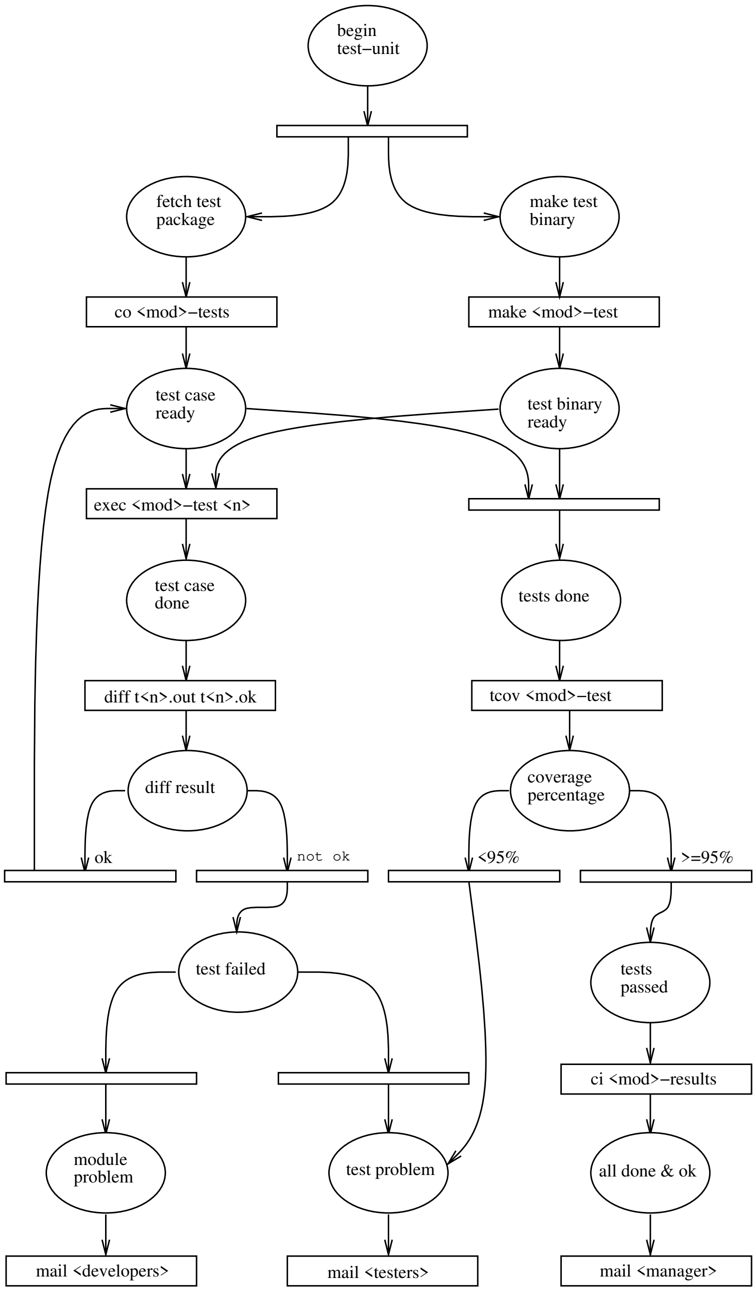

Figure 5: Petri Net Model of the ISPW 6/7 Test Module Task.

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 2: Example Event Stream Measurements for the Streams of Table 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 2 shows various validation measurements for the event streams in Table 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 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).

[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 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.

6 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 8.

6.1 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 [2, 37].

6.2 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 (such as 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.

7 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.

7.1 Related Work

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

7.1.1 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 [1] 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 [47] 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 [27] 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.

7.1.2 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 [13], 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 [3, 21]. 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 [18]. 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.

7.1.3 Regular Expression Matching

In [40, 44], 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 [24] and string to regular expression comparisons [44], 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 $O(MN(logM + log^2N))$ time, but which also takes $O(MN + Nlog^2N)$ 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 [40].

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.

7.2 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* [34, 46].

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 of 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 work.

Since we are trying to calculate minimum cost distance metrics, it is natural to use the metrics 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 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.

7.3 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 [46]. 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 9.

8 Implementing the Validation Methods

The implementation of the validation methods described so far has been done, and we 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 6: Validation Calculation Algorithm.

8.1 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 6 shows the psuedocode algorithm for the validation engine, and Figure 7 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

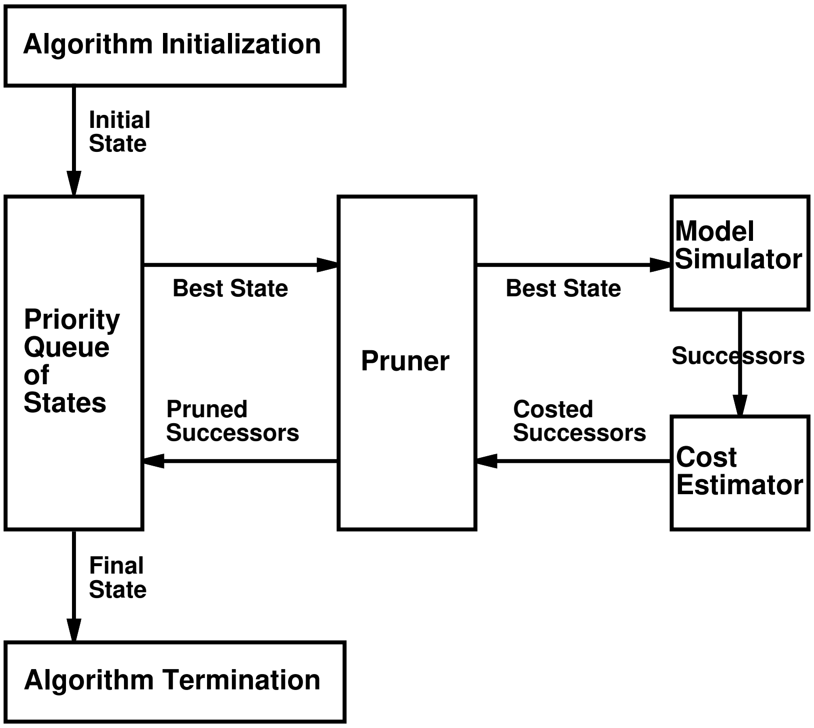

Figure 7: Data Flow in the Model Simulator.

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.

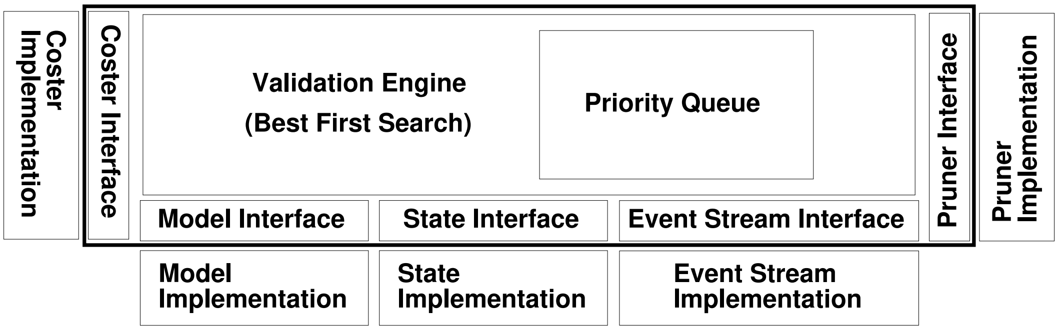

Figure 8: Architecture of the Model Simulator.

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 9.

8.2 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 8 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 9, 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 our process validation tool, built using the BALBOA process data analysis framework [15]. 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 10 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 code to interpret events according to their event-type 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 9: Modeling Interface Based on Model and State Base Classes.

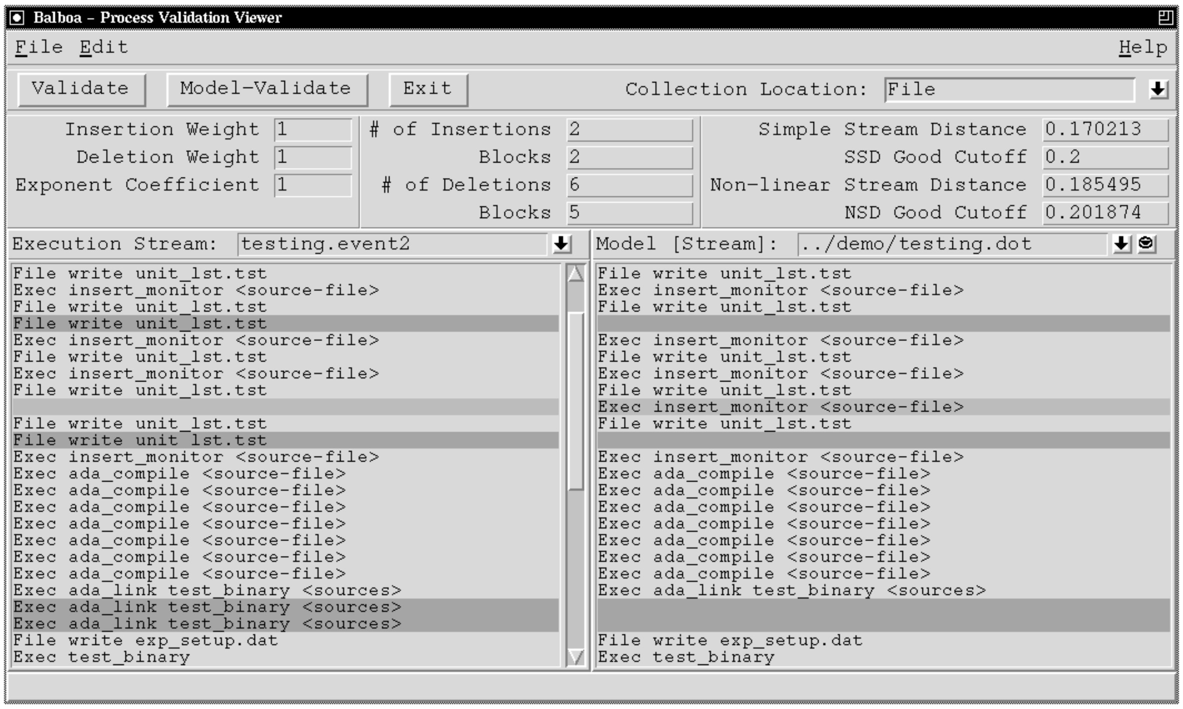

Figure 10: The VALIDATION tool.

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

9 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 7.3, 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 3 shows the performance of VALIDATION on a stream of 1250 events being validated against a model with 53 states, shown in Figure 11. 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 Section 10, where a summary of an in-depth study is presented.

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

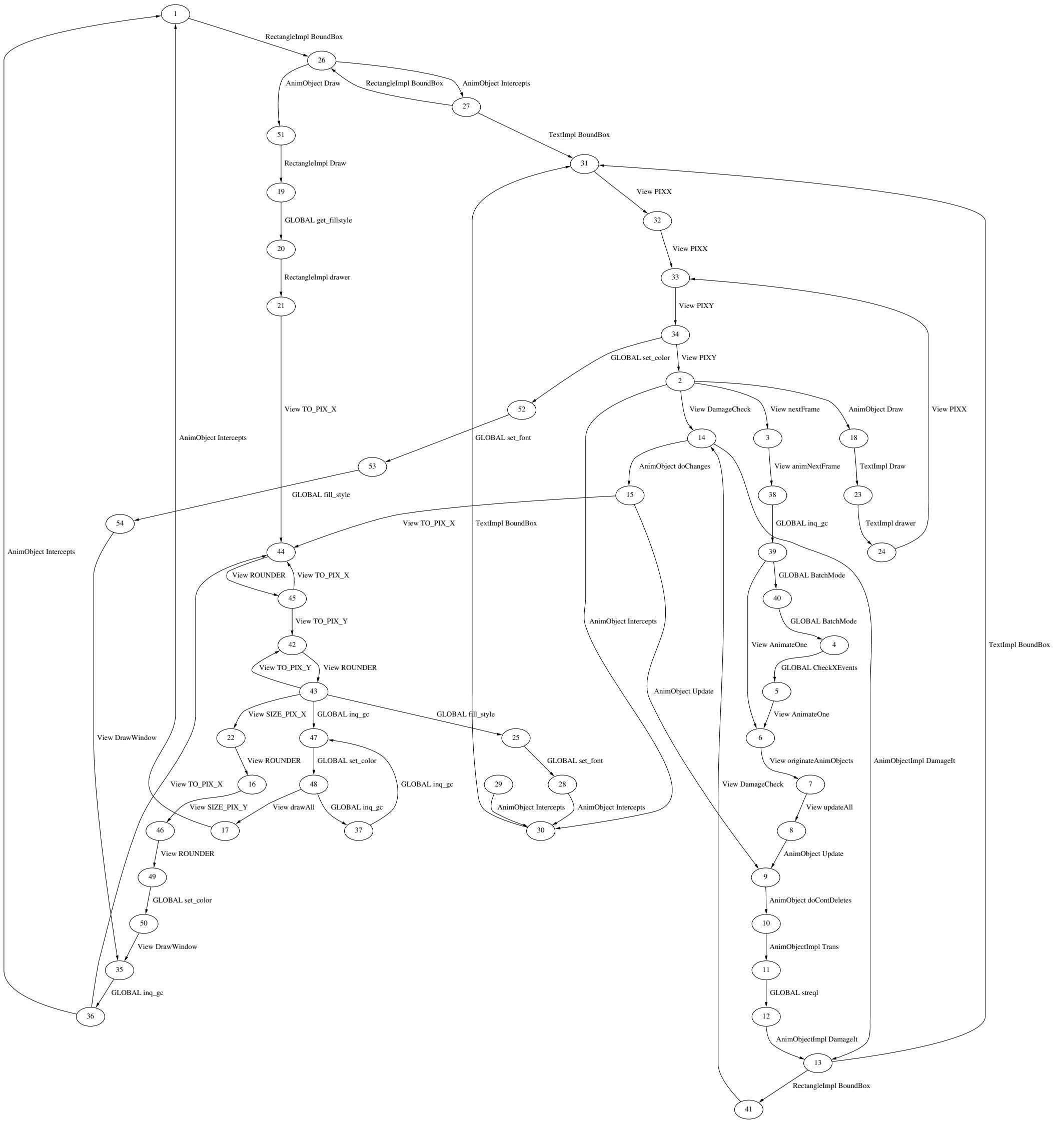

Figure 11: Model Used for Performance Evaluation of VALIDATION.

controlling the search or in finding good solutions. From here on we explore the behavior of only 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 shows the performance of VALIDATION with position pruning

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

across streams that are successively less correspondent to the model (all streams are about 1250 events long). As can be seen on the 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. The 10-position lookback does find the optimal model stream for all of the event streams, and does so in reasonable amounts of time. 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 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.

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

10 Validation in the Real World

The validation tools were applied in an industrial case study, where their usefulness was readily evident. An in-depth presentation of this study can be found in [36]; we will just highlight the use of the validation tools here.

The study focussed on a change request process for a large telecommunications software system. In this process, a customer reports a problem, the problem is handed to a developer, the developer completes and tests a fix, and the fix is sent out to the customer; a successful process results in the fix being accepted by the customer, while an unsuccessful one results in the fix being rejected. We were able, through gathering historical data from multiple sources, to reconstruct event streams for 159 executions of this process. The goal of the study was to statistically identify process behaviors that correlated with the acceptance or rejectance of the product.

The correlation of behaviors was performed by using our process validation techniques. We tested various aggregate measures, such as the number of source lines for a change, the number of files affecting each time, and the time it took for each change. The only significant aggregate metrics were the delay from first customer report to when the fix was started and the developer doing the fix. These did not preclude looking at the process behavior through the use of the validation techniques.

Validation was done using an existing process model derived from the (informal) process documentation, and using a model that was generated from the data itself using our *process discovery* methods (see [16]). In general, the process was highly variable and somewhat less than 50% of the behavior matched what the document's model predicted, and about 65% matched the discovered model. The NSD metric showed statistically significant differences between the successful (accepted product) and unsuccessful (rejected product) processes on both models, while the SSD metric did not on either model.

By using the detailed information that the validation methods provide, in particular the event type that was matched, inserted, or deleted, and the model state where each operation took place, we were able to localize the statistical differences between the two populations in terms of where in the model the differences were, and on what type of events. This detailed analysis shows the power of the information that the validation methods produce: once a gross metric (like the NSD) indicates a problem, the detailed information that the validation methods produce let one perform an in-depth analysis.

The success of the validation tools in this setting, where the process was highly variable and the data was of lower quality than would be desirable (the event streams were reconstructed from historical data not meant for this purpose), gives us the expectation that these methods will be highly applicable in many process settings.

11 Conclusion

In summary, we have proposed and implemented 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.

While Section 10 gave a short presentation of an industrial study in which the process validation techniques were applied, this study is presented in-depth in [36]. This study shows the usefulness of these techniques in a real world setting, and 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.

There are many directions that continuing and future work in process validation might take, some of these being:

- 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, points in a model where one can fix the execution stream and ignore previous behavior could help further reduce the search cost in a large model. This is similar to the concept of trace change points in [23].

- 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 better 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 execution stream analysis. Real-time systems analysis techniques could be useful here [26, 50]. 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.

We intend to explore some of these directions, and to continue improving the practicality and usefulness of our process validation methods.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] J.E. Cook and A.L. Wolf. Balboa: A framework for event-based process data analysis. Technical Report CU-CS-8??-96, Department of Computer Science, University of Colorado, November 1996.

[16] J.E. Cook and A.L. Wolf. Discovering models of software processes from event-based data. Technical Report CU-CS-8??-96, Department of Computer Science, University of Colorado, November 1996.

[17] J.E. Cook and A.L. Wolf. Process discovery and validation through event-data analysis. Technical Report CU-CS-817-96, Department of Computer Science, University of Colorado, November 1996.

[18] J. Corbett and A. Polk. A tool for automatic generation of behaviors for constrained expression analysis. Technical report, February 1993.

[19] 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.

[20] J.L. Devore. *Probability and Statistics for Engineering and the Sciences*. Brooks/Cole, Pacific Grove, California, 3rd edition, 1991.

[21] 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.

[22] M.W. Du and S.C. Chang. A model and a fast algorithm for multiple errors spelling correction. *Acta Informatica*, 29:281–302, 1992.

[23] 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.

[24] D. Eppstein. Sequence comparison with mixed convex and concave costs. *Journal of Algorithms*, 11:85–101, 1990.

[25] 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.

[26] 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.

[27] C.N. Fischer and J. Mauney. A simple, fast, and effective LL(1) error repair algorithm. *Acta Informatica*, 29:109–120, 1992.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[33] 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.

[34] A. Hutchinson. *Algorithmic Learning*. Graduate Texts in Computer Science. Oxford University Press, 1994.

[35] M.L. Jaccheri and R. Conradi. Techniques for Process Model Evolution in EPOS. *IEEE Transactions on Software Engineering*, 19(12):1145–1156, December 1993.

[36] L.G. Votta J.E. Cook and A.L. Wolf. Does following prescribed processes lead to better products? Technical Report CU-CS-8??-96, Department of Computer Science, University of Colorado, November 1996.

[37] R.L. Kashyap and B.J. Oommen. The noisy substring matching problem. *IEEE Transactions on Software Engineering*, 9(3):365–370, 1983.

[38] 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.

[39] 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.

[40] J.R. Knight and E.W. Myers. Approximate regular expression pattern matching with concave gap penalties. *Algorithmica*, 14:85–121, 1995.

[41] 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.

[42] 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.

[43] 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.

[44] E.W. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51(1):5–37, 1989.

[45] 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.

[46] E. Rich. *Artificial Intelligence*. McGraw-Hill Series in Artificial Intelligence. McGraw-Hill, 1983.

[47] J. Röhrich. Methods for the automatic construction of error correcting parsers. *Acta Informatica*, 13:115–139, 1980.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] S.M. Sutton, Jr. Accommodating Manual Activities in Automated Process Programs. In *Proceedings of the 7th International Software Process Workshop*, October 1991.

[53] 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.

[54] M.S. Waterman. General methods of sequence comparison. *Bulletin of Mathematical Biology*, 46:473–501, 1984.

[55] 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.

[56] 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.