Software Process Validation: Quantitatively Measuring the Correspondence of a Process to a Model

Jonathan E. Cook

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003 USA

jcook@cs.nmsu.edu

Alexander L. Wolf

Department of Computer Science
University of Colorado
Boulder, CO 80309 USA

alw@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-840-97 May 1997

© 1997 Jonathan E. Cook and Alexander L. Wolf

ABSTRACT

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 are tailorable and give process engineers control over determining the severity of different types of discrepancies. The techniques provide detailed information once a high-level measurement indicates the presence of a problem. We have applied our process validation methods in an industrial case study, of which a portion is described in this paper.*

This work was supported in part by the National Science Foundation under grant CCR-93-02739 and by the Air Force Material Command, Rome Laboratory, and the Defense 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.

1 Introduction

Whenever a model of a system is created, the question arises as to whether that model faithfully captures the system. In software process research, where the model is typically embedded and executed within an automated software engineering environment [25], this question is avoided; the model and process are necessarily in agreement because the model *becomes* the process.

When applied in software process practice, however, this approach suffers from a fundamental flaw. In particular, it assumes that virtually the entire process is executed within the context of the environment. In fact, critical aspects of the process occur off the computer and, therefore, not under the watchful eye of the environment [43, 46, 47]. 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. Moreover, deviations from the process are naturally to be expected [14, 20, 27].

Even if one could completely enforce a process, there still remains the issue of managing change in a process, which might also lead to a discrepancy between the model and the execution. There has, in fact, been considerable work that addresses process evolution [3, 30]. 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 this activity as *process validation* [12]. The techniques are neutral with respect to 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 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. The validation techniques have been implemented in a prototype tool, which has been used as part of an industrial process data analysis case study [11]

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 described 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.

The techniques borrow from various areas of computer science, including distributed debugging, concurrency analysis, and 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.

The next section presents the framework in which this work is cast. Section 3 then states the process validation problem and outlines our approach. Section 4 defines the validation metrics and Section 5 presents example uses of the metrics. The issue of deriving a characteristic behavior from a process model is discussed in Section 6. Section 7 presents a short review of an industrial case study in which the validation techniques were successfully applied. Section 8 summarizes work

related to process validation. Finally, we describe our implementation of a validation tool and discuss some ideas for future work in Section 9.

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.

2.1 Events

Following Wolf and Rosenblum [46], 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. The use of events to characterize behavior is already widely accepted in other areas of software engineering, such as program visualization [36], concurrent-system analysis [2], and distributed debugging [5, 15].

The "instant" of an event is relative to the time granularity that is needed or desired; thus, certain activities that are of short duration relative to the time granularity are represented as a single event. An activity spanning some significant 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, the other event attributes would be items such as the agents and artifacts associated with an event, the tangible results of the action (e.g., pass/fail from a design review; errors/no-errors from a compilation), and any other information that gives character to the specific occurrence of that type of event. In the work described here, we do not make use of attributes other than time.

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 data collection method, this assumption can be relaxed.

The ability to collect events is central to supporting event data analysis. Fortunately, the tools and environments of today provide strong support for logging the events that occur on a system. A version control system, for example, logs the accessing and modification of documents and code. Off-computer events, such as meetings or phone calls, can be logged manually. Email logs (e.g., using a project alias) can be a starting point for collecting communication events. A detailed discussion of the existing support for collecting events and previous studies that have made use of events is beyond the scope of this paper, but can be found elsewhere [10, 11, 13].

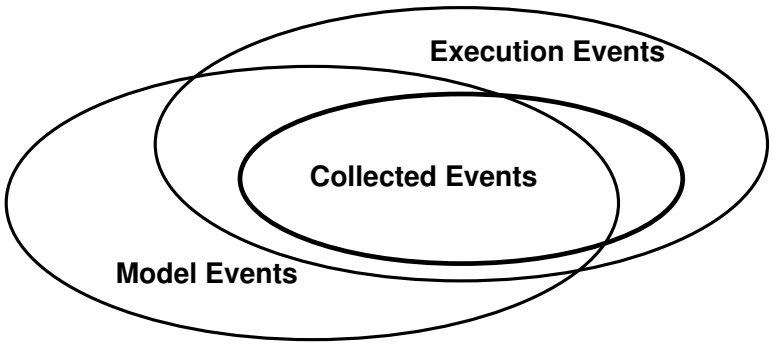

Figure 1: Venn Diagram of Event Types.

2.2 Relating Models and Events: Event Sites

For our purposes we focus on behavioral process modeling formalisms. These include models based on state machines (e.g., Statemate [29]), Petri nets (e.g., SLANG [4] and FUNSOFT Nets [28]), procedural languages (e.g., APPL/A [44]), and rule-based languages (e.g., Oz [6]). We assume that a model described in any such formalism induces one or more event streams, and thus it has places in its behavioral description where events can be recognized. 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 is, in effect, an event stream. A rule-based language would have rules as event sites. Note that not all transitions (or rules) must be event sites. There may be internal behavior in a model that does not need to be visible to event stream analyses.

Event sites are typed—that is, each event site produces a specific type of event. For each event type, then, there is a set of event sites in a process model that produce it.

2.3 Event Domains

Given the distinction between the process model and the process execution, there are really two universes of event types. One universe is the set of event types associated with the model of a process, while the other is the set of event types associated with the execution of the process. Their conceptual relationship is depicted in Figure 1, which indicates that the sets are not necessarily equivalent. For example, consider an organization that executes a particular process. Some members of that organization may informally and unilaterally decide to perform occasional code inspections; a formal model of that process might not account for such an ad hoc activity. Conversely, a model adopted from another organization might include a subprocess for design reviews, but the adopting organization might decide never to perform that activity.

A third set of event types, also shown in Figure 1, are those that are actually collected as data. Since this set must necessarily be a subset of the execution events (one cannot record something unless it actually occurs), it can be viewed as a window onto the actual process execution. This window might not show the whole execution, because there may be some activities for which no event data are collected. There are several reasons why this might occur, but two obvious ones 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, such as most

staff meetings, are likely to be more expensive to collect than events that occur on the computer, simply because off-computer events would require manual, as opposed to automated, collection techniques.

It is important to note that we concentrate here on event types, which are abstractions of the activities in a process. We assume that the model and the execution agree to a significant extent about the set of event types (i.e., activities) involved in the process, although they may not agree on the specific orderings and numbers of events of those types. If they did not largely agree on the basic sets of activities, then it would not be clear what it would mean for them to be relating to the same process. Indeed, modeling and data collection are often closely related, in the sense that models are used to frame the data collection activity, and vice versa.

We do not, however, assume that the particular names for the event types used in the model and found in the collected execution data are equivalent. Fortunately, this is a simple syntactic issue that can be easily dealt with through a name mapping applied to either event stream. In fact, the data analysis framework within which our validation tool is implemented provides a convenient mechanism based on regular expressions for creating and applying such mappings to the execution stream [13]. The mechanism additionally allows the analyst to extract events from the data and map them to event types at arbitrary levels of granularity. In particular, the granularities of event types derived from the data can be made to match the granularity of event types found in a model.

3 Problem Statement and Approach

In our framework for process validation, we have an executing process that produces an actual event stream and, on the other side, we have a model that induces 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 resembles an event stream induced by the model. We call these two event streams the *execution event stream* and the *model event stream*, respectively. Figure 2 depicts the process validation framework.

There are several methods for performing a measurement such as this, but one that seems most applicable is the *string distance metric* [35]. A string distance metric counts the number of token insertions, deletions, and substitutions needed to transform one string into the other. By applying various mathematical transformations, this method becomes a family of metrics. String distance metrics have been used in applications as varied as DNA/RNA matching [45], substring matching [32, 41], spelling error correction [18], syntax error correction [1, 24, 40], and text file differencing as in the UNIX tool *diff*. In general, string distance metrics have become the standard method in any domain requiring symbolic sequence comparison.[1]

Other methods that could be used to quantify the difference between two event streams do not offer the versatility of string distance metrics. Hamming distance, for example, is the count of the number of tokens that differ, but this method assumes that either the streams are the same length or that they can be suitably matched and padded. In fact, string distance metrics can subsume this method by ignoring the insertion and deletion operations, and just tallying substitutions.

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

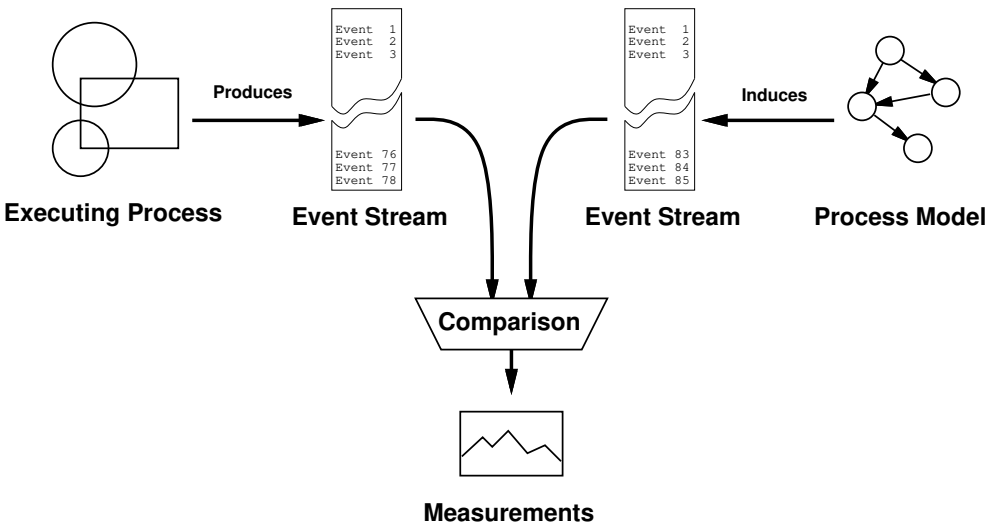

Figure 2: Process Validation Framework.

4 Validation Metrics

In this section we introduce two metrics for determining the correspondence between a formal model of a process and an execution of the process. They share the characteristic that they use string distance to 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 6. We defer detailed examples of applying the metrics to Section 5.

4.1 Simple String Distance Metric

The first metric uses a simple, direct approach to measuring string distance, and thus we refer to it as the *simple string distance* (SSD) metric. Under this method, the distance between two strings is measured by counting the minimum number of token insertions, deletions, and substitutions needed to transform one string into the other. For our purposes, 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 predicted them but the execution did not perform them), and deletions represent extra activities (the model did not predict them, but they were performed in any case).

As an example, consider the execution and model event streams shown in Figure 3a, where the lettered boxes represent events. The lines drawn between the two streams indicate one possible correspondence between their respective events. The transformation of the execution stream into the model stream is depicted in Figure 3b. In particular, we delete a C, substitute a D for a C, and insert an E, a D, and another E. The resulting value for the distance is then 5. This happens to be the minimum transformation required.

To strengthen the metric, weights can be assigned to each of the operation types (insertion,

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

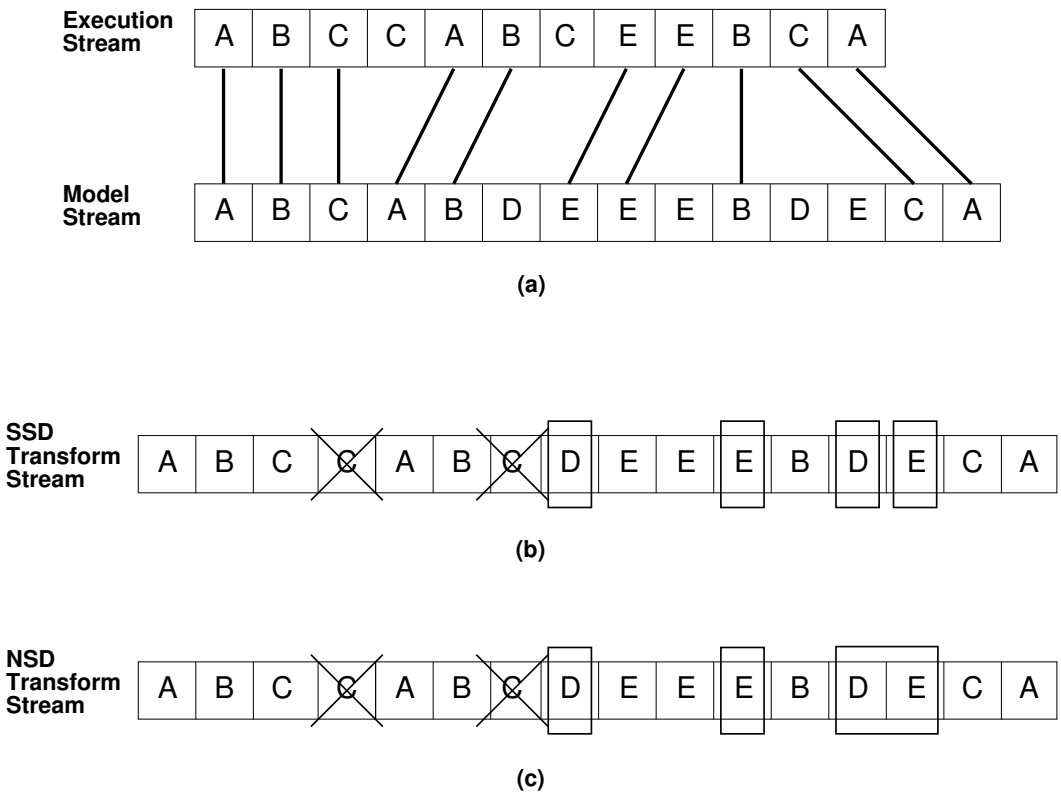

Figure 3: Example Execution and Model Event Streams (a), with Execution Stream Transformed for the SSD Metric (b) and the NSD Metric (c) Calculations.

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 M and the other of length N, the minimum total cost of operations can be computed in $O(MN)$ time using a well-known dynamic program [35].

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 measure 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 paper.

The SSD metric is 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 [16] 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.[3]

4.2 Non-linear String Distance Metric

A characteristic of the SSD metric is that it is focused narrowly on the costs of individual string transformation operations, since each operation is weighted separately. In terms of process behavior, though, a sequence of missed activities, for example, can be viewed as a single deviation from the expected (model) behavior, and might potentially be a more serious breech of desired execution than can be represented by a simple count of those missed activities. Our next metric accounts for this.

The *non-linear string distance* (NSD) metric is an enhancement to the SSD metric based on the notion of a sequence of insertions or a sequence of 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 3c, 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.

[3] Actually, these are inversions of the standard statistical rules of thumb, but their effect is the same.

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. Practical k values range from about 1 to 3, with larger values being used when blocks are small but important.

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 as follows.

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

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. Note that as a history of applying the NSD metric to a process is accumulated, the actual value of B_{avg} for that process will be known.

4.3 Event Type Weighting

We have included in our validation metrics a means to differentially weight the insertion and deletion transformation operations. We did this because they represent conceptually different kinds of deviations, namely missed and extra activities, respectively.

But, at a finer level of granularity, it would also be useful to differentiate the relative importance of specific types of events. For example, an event representing the conclusion of a regression testing activity might be considered more important than an event representing the conclusion of a weekly team training meeting. One would like to be able to distinguish the significance of missing each of kind of event as part of the process validation.

The SSD and NSD metrics easily incorporate such a weighting scheme. The process engineer can override the default insertion and deletion weights, and vary the operation weights according to the type of event involved in the operation. The default weights are still used in the normalization part of each metric's equation.

4.4 Auxiliary Measures

As presented, the metrics provide a single value as the measure of correspondence. This measure by itself is useful to indicate the presence of a potential problem. But a deeper understanding of the deviation is needed to uncover the source of the problem. The SSD and NSD metrics naturally provide auxiliary measures to the analyst, including:

- the number of events that match in the compared streams;
- the number of insertion and deletion operations used to calculate the metric;
- the number, size, and average size of blocks of operations;
- the number of matches, insertions, and deletions per event type;

- the locations in the execution event stream where deviations occur; and
- the locations in the model where deviations occur.

These auxiliary measures enhance the usefulness of the basic metrics. In Section 7 we show how they contribute detailed and important information to an understanding of deviations in an industrial case study.

5 Example Use of the Metrics

To illustrate the two validation metrics introduced above, we use the Test Unit task from the ISPW 6/7 process problem [33]. This is a simple and small process fragment, but it should give the reader 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. Upon a successful completion of the tests, they are to store the test results under configuration control and alert the manager to the new status of the module.

Figure 4 shows a colored Petri net model [31] 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.[4]

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 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 an error 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.[5] 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.

[4]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 case), "tcov" is a test coverage tool, "diff" is a text differencing tool, and "mail" is an electronic mail tool.

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

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.

Stream #	# Ins	# Del	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	–	–	–	–	–	–
2	1	0	0.11	0.11	0.11	0.11	0.11	0.11
3	1	2	0.30	0.55	2.11	0.15	0.21	0.60
4	2	1	0.30	0.55	2.11	0.23	0.47	2.03
5	3	0	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.

Table 2 shows 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 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.[6] 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 table 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 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,

[6]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.

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 among the discrepancies. 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 than deletions. 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 metric. 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, where it has all insertions, and the difference in the weighting of insertions is evident in comparison to event streams 3 and 4.

6 Deriving the Model Event Stream

Section 4 presents the process validation metrics assuming the existence of two event streams, an execution stream and a model stream. However, we are given only one stream, the execution stream. Instead of a model stream, we have a model that describes a set of streams. To perform the measurement, we must induce a model stream from the model.

The challenge, of course, is that a formal model of any but the most trivial process likely leads to a large, if not infinite, number of possible event streams. How do we choose one? Because we are measuring correspondence, we need to derive a model event stream that most closely matches the execution event stream, in order to get as useful a measure as possible. By "most closely" we mean the model stream that gives the minimum distance measurement. Any other model stream would imply a greater discrepancy than necessary. It is important to note that closeness is not a fixed property of the relationship between an execution event stream and a model, but also depends on the validation metric being applied and on the weightings used with the metric; different weights on the insertion and deletion operations (and on the event types) will affect which model stream is the closest.

Deriving a model stream does not have to be a blind generate-and-test effort. For example, if the execution stream exactly matches the model, then one wants to choose the model stream that *is* the execution stream. If there are small differences, one still wants to choose a model stream that is almost like the execution stream. Thus, the model stream derivation problem can be restated as finding the smallest changes to the execution stream that make it a valid model stream. These changes are exactly the insertions and deletions that contribute to the distance metrics. Clearly, both the execution stream and the model itself can and should be used to derive the model stream used in the metric calculation.

6.1 Background

There are three main areas of related work that have addressed a problem similar to that of our model stream derivation problem: error-correcting parsing, behavior searching, and regular expression matching. All of the approaches use both a given "stream" and a model. We review the

work in these areas to place our approach in context.

6.1.1 Parsing

When a syntax error is encountered, a modern compiler for a programming language is expected to report the error and then recover in some way so that it can continue to parse the rest of the program. In essence, the compiler must find a correction between a given string of tokens (the program) and a model (the syntax of the language), so that the model can continue to match the rest of the tokens. A minimal correction is desired to allow the compiler to process as much of the program as possible, and thus this problem is similar the one we face in process validation.

Compiler research has produced several methods of interest here.

- Aho and Peterson [1] show a cubic algorithm for performing globally minimum cost error correction in terms of token insertion and deletion. They do not expect their algorithm to be used, however, because of its high cost. Rather, they propose it as a baseline against which to compare other methods.

- Röhrich [40] 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 idea of minimum distance correction, but makes the assumption that one never needs to back up in the input stream to find a good correction.

- Fischer and Mauney [24] describe a method for locally least cost error correction. They are also biased towards insertions, but include deletions. Their method uses a local search with a priority queue to find a locally minimum cost fix. 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 that domain. Some of the critical assumptions, such as not being able to back up in the input stream, are not necessarily valid in the software process domain. Thus, in general, these techniques are not applicable to our model event stream derivation problem. Nevertheless, they demonstrate two important points: optimal solutions are cost-prohibitive and heuristics can be effectively employed in practice.

6.1.2 Behavior Searching

Model checking is a technique to efficiently explore a finite state space for inherent behavioral properties, including whether a particular behavior is allowed by a model. In one example, Burch et al. [8] 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 being analyzed were of a pipelined arithmetic logic unit, which has many self-similar states resulting from the width of bits that make up 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 of a technique to search for a behavior is found in the Constrained Expressions framework [2, 17]. This is a 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. 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 solver produces a binary answer indicating success or failure of the search and, if it finds a solution, several parameters.

If the solver answer is "yes", heuristics are used along with the parameters from the solver to produce a plausible behavior that leads to the event. This behavior constitutes the next sequence of model events. Unfortunately, if the answer is "no", the Constrained Expressions framework cannot 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 process validation needs techniques that analyze the model in the continual presence of deviations from the model. Thus, it appears that techniques like those of the Constrained Expressions framework are not applicable. The issue is that while they leverage system transformations to gain speed and scalability, the transformations make the system inherently uninspectable.

6.1.3 Regular Expression Matching

Myers, Miller, and Knight [34, 37] describe algorithms for approximately matching a string to a regular expression, using insertion, deletion, and substitution operations. These methods build on the dynamic programming techniques of string-to-string comparison algorithms, and extend this to regular expressions. 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 [21] and string-to-regular-expression comparisons [37], arbitrary cost functions for blocks require at least $O(MNmax(M, N))$, or cubic time.[7]

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 studied. For example, context-free languages are thought to have high-order polynomial time algorithms for solving approximate matching [34].

In general, these super-quadratic to cubic techniques, while providing optimal answers, are impractical. They also generally have large constants in the actual running times.

6.2 Incremental, Data-Driven Matching

The survey of related approaches presented above leads one to the conclusion that the general model event stream derivation 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. First, we observe that the execution event stream likely corresponds, at least in places, to any reasonable model of the process. If this

[7]Better results can be obtained if one assumes concave block costs, where $F(B_i) - F(B_{i-1}) \geq F(B_{i+1}) - F(B_i)$, that is, where the difference between the cost of a block of length $i+1$ and i is non-increasing as i increases. With this assumption, regular expression matching takes $O(MN(logM + log^2N))$ time, but also takes $O(MN + Nlog^2N)$ space. Our NSD metric, in general, does not have concave block costs. In fact, in our formulation, we use convex costs because a longer block represents a more serious deviation from the process model. 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 then the cost of breaking can be amortized over the length of the block. This results in a concave cost function.

were not true, then the model would be so contrary as to be immediately and obviously useless. Second, we can use the execution event stream to help guide our search of the model, thus significantly cutting down on the required search space. In particular, we traverse the execution event stream and incrementally derive events for the model event stream by consulting the model. Where the model and the execution stream match, the model stream will simply mirror the execution stream. Where they do not match, some method that searches the model will need to be employed to find the minimum cost set of inserted and deleted events so that the execution stream can be changed to continue to match the model. The matched, inserted, and deleted events exactly describe the model stream that is induced from the model and that the metric then uses.

The resulting approach implies a state-space search, one that uses heuristics to control the state explosion. The states in the space are not just the states in the process model state space, but also includes the position of the event in the execution stream that is currently being examined, and the operation (match, insertion, or deletion) that led to the state. Since the search only moves forward in the execution stream, the whole state space is always a tree of states, even if the process model's state space is not.

Figure 5 shows a partial view of the search space matching a string to an FSM. The search states are labeled with the FSM state, the position in the given string, and the operation (match, insertion, or deletion) with token type that created this search state. The bold search states represent the lowest-cost path, assuming that deletion is weighted less than insertion. The lowest-cost path deletes one token, and the resulting model string ABA is the closest one to the given string, $ABBA$. If insertion is weighted less than deletion, the model string $ABABA$ would be the closest one, since inserting the single A would be cheaper than deleting a single B.

An obvious candidate for this approach is best-first search. 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 paths in the state space. For each state S with a parent S_p, a cost is estimated by

$$\mathit{EstimatedCost}(S) = \mathit{Cost}(S_s, S_p) + \mathit{Cost}(S_p, S) + \mathit{Estimate}(S, S_g)$$

where S_s is the start state and S_g is the goal state. In other words, the total estimated cost is the known cost of getting from the start state S_s 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 state S_g. The heuristic is in estimating the cost of going from S to a goal state S_g.

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 goal found 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 [39]. 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 referred to as *uniform cost*, since it also always finds a lowest-cost goal. Unfortunately, always underestimating the true cost (which includes uniform cost) also guarantees that every state that is lower cost than the goal itself will be inspected.

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. The goal state in validation can be defined as the state matching (or deleting) the last event in the execution event stream. One might want to define

a goal state as a state that contains a termination state of the process model, but this is not as flexible, since it does not allow validation of incomplete process executions.

The SSD and NSD metrics are 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—that is, the cost of the current state. This assumes that the event stream processed so far is representative of the total stream, and that the metric calculation will not greatly change. This heuristic estimator (and cost metric) does not always underestimate the goal state cost, so it is not guaranteed to find the minimum cost goal. In particular, from a given state there may be continual matches (zero cost) to the end of the stream, thus allowing the distance metric to diminish towards zero as the length increases. Trying to construct a complex estimator that is tailored toward indentifying these anomolous cases is counterproductive. However, we have found it to be a reliable estimator and in practice we have never seen a case where the minimum cost goal was not reached.

6.3 Pruning

The technique of *pruning* a state space has proven to be useful in reducing the cost of finding a low-cost goal [39]. Pruning discards portions of the state space that look unpromising. By pruning, one cannot guarantee a lowest-cost goal. But in some domains, such as game playing, "smart" pruning has negligible effects on the outcome of the search while dramatically reducing search costs. Pruning takes many forms and can use vastly different methods and heuristics.

One heuristic that we employ is to discard any newly generated state that has an estimated cost higher than some threshold relative to the current best-looking state. We refer to this as *cost pruning*. The hypotheses behind cost pruning 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, cost pruning assumes that one can set a single threshold for the whole state space, and that this threshold will work consistently.

Our initial observations show that the variability in costs assigned to states should change over time; in the beginning, especially, there is much larger variability. Thus, for our pruning method, 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.

Another pruning method we employ is to discard any state that is some specified distance behind in the execution event stream from the current farthest state. We refer to this as *position pruning*. The position pruning heuristic assumes that the most likely paths to the lowest-cost goal state will be examined in an interleaved fashion—that is, 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 (i.e., more than the specified position pruning parameter) in the event stream is ignored as unlikely to be on a path to a good goal state. For example, if a state that was at the 36th event in the event stream was the furthest along in the stream, and the position pruning parameter was set at 5, all unexamined states that were at events previous to the 31st event would be discarded. Figure 6 shows the effect of position pruning on a small search tree. All of the open, not-yet-searched states (the leaves) that were behind the furthest state by more than 20 were discarded, along with the parent states whose children have all been discarded. This narrows the search space, resulting in a direct, linear state path to the area of the state space still being searched.

Our experience with pruning has shown that position pruning consistently performs well for the

model event stream derivation problem, while cost pruning is highly variable and often poor in its performance. We have studied our pruning methods extensively, but detailed discussion is beyond the scope of this paper. We refer the reader elsewhere for more information [10].

7 Using Validation in an Industrial Case Study

We recently performed a case study of an industrial software process [11]. The goal of the study was to statistically identify process behaviors that correlated with successful and unsuccessful executions of the process. One component of that study involved the use of our validation techniques. In this section we review the study as an example application of process validation in a real-world setting.

The study focused on a change request process for a large telecommunications software system. In this process, a customer reports a problem, the problem is assigned to a developer, the developer completes and tests a fix, and the fix is sent out to the customer. A successful execution of the process results in the fix being accepted by the customer, while an unsuccessful execution results in the fix being rejected. There was no existing formal model of the process, but enough documentation and informal knowledge existed to be able to create one. In addition, data analysis techniques were employed to discover possible model fragments from the data themselves [10]. Figure 7 depicts a state machine model of the process.

We were able to obtain data for 159 executions of this process by extracting events from several historical archives and merging them to form complete execution streams. The archives included a source code control system, a modification request tracking system, a customer response database, and loose-leaf binders of code inspection reports. These 159 streams were divided into two populations based on their success—one where the fix was accepted (141 streams) and one where the fix was rejected (18 streams).

Table 3 shows the validation metrics calculated for the case study, where the statistical test for significant difference between the two populations was the Wilcoxon Rank Sum test, shown in column 3. The other columns are the p-values (column 2) and the means and standard deviations of each metric for the two populations. In general, the process execution was highly variable, with somewhat less than 65% of the behavior matching the model. With equal insertion and deletion weights, the NSD metric, using $k = 1$, showed statistically significant differences between the successful (accepted fix) and unsuccessful (rejected fix) process executions, while the SSD metric did not. Looking at the components of the metrics, we see that only deletions were significant, but then only weakly so.

By using the detailed information provided by the validation techniques—especially the event type that was matched, inserted, or deleted, and the model state in which each such operation was applied—we were able to localize the statistical differences between the two populations in terms of where in the model the differences appeared and on what specific type of events. The shaded states in Figure 7 are those locations in the model. That is, the difference in the relative amount of events matched from a shaded state (instead of inserted or deleted) is statistically significant between the accepted-fix and rejected-fix populations. In this study, the accepted-fix population had more matches, and thus followed the process model more closely.

The multi-leveled analysis of this study shows the power of the information that the validation methods provide: once a gross metric (such as NSD) indicates a problem, the detailed information available lets one perform an in-depth analysis, directly seeing where the process might be breaking down. Indeed, our results led to suggestions about where in the process some adjustments might be useful.

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

Table 3: Validation Metrics Calculated in the Case Study.

8 Related Work

There is related work in the area of process improvement that uses data to characterize processes, but none that uses data in a process validation activity.

- Chmura et al. [9] and Bhandari et al. [7] try to deduce problems in the process by looking at defect data in the products. Specifically, they statistically analyze change data and effort data to determine the behavior of the process. For example, they can see ripple effects from interface changes and high percentages of fix-on-fix changes.

- Garg et al. [26] employ a manual process history analysis in the context of a meta-process for creating and validating domain-specific process models and software toolkits.

- Pérez et al. [38] propose methods for evaluating the congruence of process models. Congruence is a measure of how well an environment can accommodate a given process model, based on the tools and activities already in that environment. The effort here is to predict how well a specific process will fit into an environment, rather than whether or not the model is followed once it is deployed.

- Cugola et al. [14] define a formal framework for reasoning about inconsistencies and deviations in a process. Their approach is directed towards processes that are controlled by a process support system using an enacted model. Their goal is to enable these systems to allow, coordinate, and resolve deviations from the model. In this respect, their work is similar to ours, but they do not use data derived from the process execution to measure the deviations.

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.

In using event-based data to compare an execution with a formal model, our work also relates to that of distributed debugging and history checking.

- Bates [5] uses "event-based behavioral abstraction" to characterize the behavior of programs. He 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 do not match, rather than attempting to provide aggregate measures of disparity.

- Cuny et al. [15] build 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. [22, 23] 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 goal is to quantify discrepancies, and therefore we take a more pragmatic approach to "correctness".

9 Conclusion

We have developed two metric-oriented techniques for process validation, from a linear distance measure in terms of event insertions and deletions, to a non-linear distance measure that takes into account the size of discrepancies. The metrics are independent of any specific behavioral process modeling paradigm, and thus have wide applicability.

The process validation techniques have been implemented as part of the Balboa process data analysis framework [13]. The current implementation works with finite state machine models of processes. The user interface for selecting execution streams and process models, and for viewing the results of a process validation, is shown in Figure 8.

The upper portion of the window provides a quantitative view of the validation results at three levels of detail: the counts of individual insertions and deletions; the counts of the blocks of insertions and deletions; and the calculated distance measurements. To the left of these results are a summary of the parameters used to control the metrics calculations.

The lower portion of the window provides a scrollable, visual summary of the detailed differences between the process as executed and the process as predicted. Extra events in the execution stream are highlighted in one color (shown here as light grey bands), while missing events in the execution stream are highlighted in another color (shown here as dark grey bands).

There are many directions that future work in process validation can take, including the following.

- *Identifying additional properties of process models that can be exploited in performing validation.* For example, points in a model where one can fix the execution stream and ignore previous behavior could help reduce the search cost in a large model. This is similar to the concept of trace change points [19].

- *Developing improved techniques for visualizing the results of validation.* For example, overlaying the differences onto the process model rather than onto the model event stream may help a process engineer better understand the problems in the process.

- *Investigating other analyses for process executions and process models.* For example, time-oriented metrics, perhaps derived from the area of real-time analysis [23, 42], would be a useful extension to execution stream analysis. Methods for measuring the efficiency of a process would be another useful analysis method. Both would help in the optimization of a process that has already been behaviorally validated.

We intend to explore these and other directions, to continue improving the practicality and usefulness of process validation techniques, and to further experiment with their application in industrial settings.

Acknowledgments We appreciate the many helpful comments on this work provided by Clarence (Skip) Ellis, Dennis Heimbigner, David Rosenblum, Lawrence Votta, and Benjamin Zorn.

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] 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.

[3] 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.

[4] 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.

[5] 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, 1989.

[6] I.S. Ben-Shaul and G.E. Kaiser. A Paradigm for Decentralized Process Modeling and its Realization in the Oz Environment. In *Proceedings of the 16th International Conference on Software Engineering*, pages 179–188. IEEE Computer Society, May 1994.

[7] 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.

[8] 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.

[9] 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.

[10] J.E. Cook. Process Discovery and Validation through Event-Data Analysis. Technical Report CU-CS-817-96, University of Colorado, University of Colorado, Boulder, Colorado, November 1996.

[11] J.E. Cook, L.G. Votta, and A.L. Wolf. A Methodology for Cost-Effective Analysis of In-Place Software Processes. Technical Report CU-CS-825-97, University of Colorado, University of Colorado, Boulder, Colorado, January 1997.

[12] J.E. Cook and A.L. Wolf. Toward Metrics for Process Validation. In *Proceedings of the Third International Conference on the Software Process*, pages 33–44. IEEE Computer Society, October 1994.

[13] J.E. Cook and A.L. Wolf. Balboa: A Framework for Event-Based Process Data Analysis. June 1998. To appear.

[14] G. Cugola, E. Di Nitto, A. Fuggetta, and C. Ghezzi. A Framework for Formalizing Inconsistencies and Deviations in Human-Centered Systems. *ACM Transactions on Software Engineering and Methodology*, 5(3):191–230, July 1996.

[15] 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, 1993.

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

[17] 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.

[18] M.W. Du and S.C. Chang. A Model and a Fast Algorithm for Multiple Errors Spelling Correction. *Acta Informatica*, 29:281–302, 1992.

[19] 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.

[20] C.A. Ellis, K. Keddara, and G. Rosenberg. Dynamic Change within Workflow Systems. In *Proceedings of the Conference on Organizational Computing Systems*, pages 10–21. ACM SIGOIS, August 1995.

[21] D. Eppstein. Sequence Comparison with Mixed Convex and Concave Costs. *Journal of Algorithms*, 11:85–101, 1990.

[22] 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.

[23] 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.

[24] C.N. Fischer and J. Mauney. A Simple, Fast, and Effective LL(1) Error Repair Algorithm. *Acta Informatica*, 29:109–120, 1992.

[25] P.K. Garg and M. Jazayeri. Process-Centered Software Engineering Environments: A Grand Tour. In A. Fuggetta and A.L. Wolf, editors, *Software Process*, number 4 in Trends in Software, pages 25–52. Wiley, London, 1996.

[26] 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.

[27] J. Grudin. Groupware and Cooperative Work: Problems and Prospects. In B. Laurel, editor, *The Art of Human Computer Interface Design*. Addison-Wesley, Reading, Massachusetts, 1990.

[28] 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.

[29] 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.

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

[31] K. Jensen. Coloured Petri nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299. Springer, 1986.

[32] R.L. Kashyap and B.J. Oommen. The Noisy Substring Matching Problem. *IEEE Transactions on Software Engineering*, 9(3):365–370, 1983.

[33] 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.

[34] J.R. Knight and E.W. Myers. Approximate Regular Expression Pattern Matching with Concave Gap Penalties. *Algorithmica*, 14:85–121, 1995.

[35] 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.

[36] 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.

[37] E.W. Myers and W. Miller. Approximate Matching of Regular Expressions. *Bulletin of Mathematical Biology*, 51(1):5–37, 1989.

[38] G. Pérez, K. El Emam, and N.H. Madhavji. A System for Evaluating the Congruence of Software Process Models. Technical Report SE-94-7, McGill University, Montreal, Canada, March 1994.

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

[40] J. Röhrich. Methods for the Automatic Construction of Error Correcting Parsers. *Acta Informatica*, 13:115–139, 1980.

[41] 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.

[42] R.L. Schwartz, P.M. Melliar-Smith, and F.H. Vogt. An 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.

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

[44] S.M. Sutton, Jr., D. Heimbigner, and L.J. Osterweil. APPL/A: A Language for Software Process Programming. *ACM Transactions on Software Engineering and Methodology*, 4(3):221–286, July 1995.

[45] M.S. Waterman. General Methods of Sequence Comparison. *Bulletin of Mathematical Biology*, 46:473–501, 1984.

[46] 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.

[47] 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. IEEE Computer Society Press, March 1993.

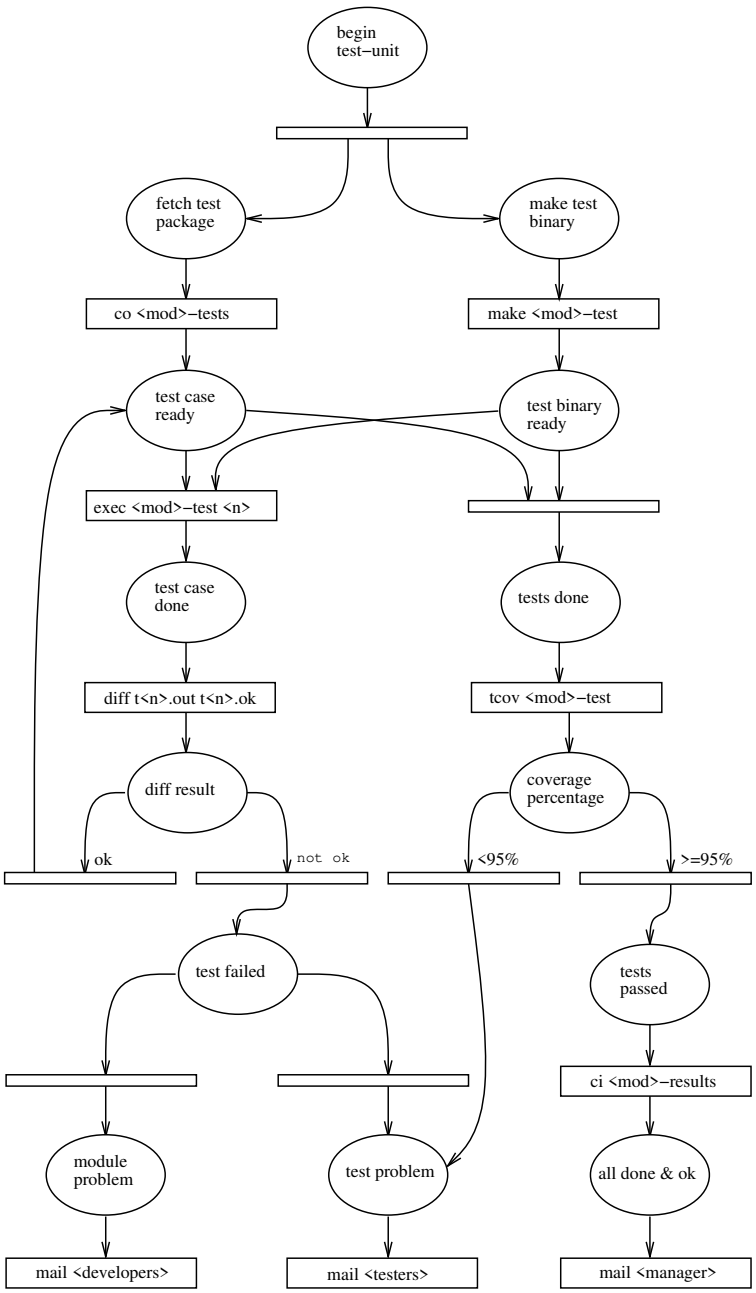

Figure 4: Petri Net Model of the ISPW 6/7 Test Module Task.

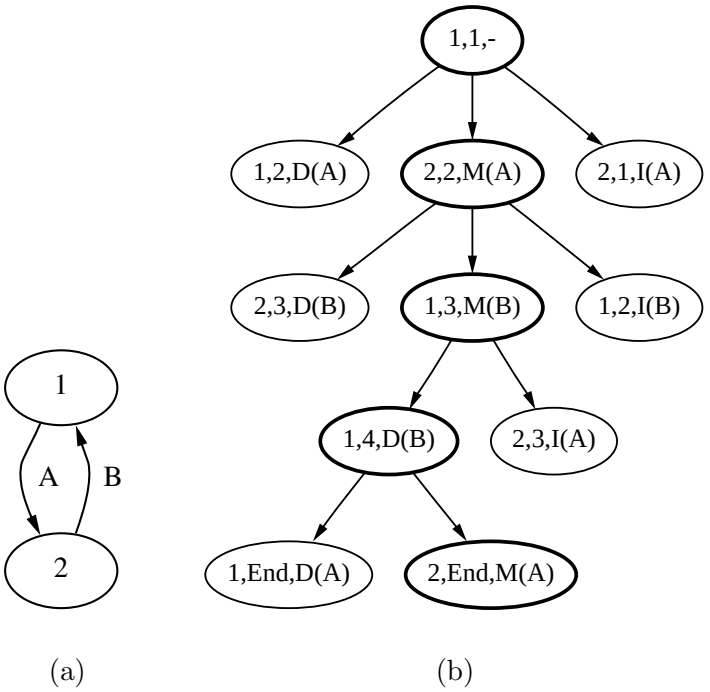

Figure 5: A Two-State FSM Model (a) and a Partial Search Tree (b) Attempting to Match the String $ABBA$.

Figure 6: Search Tree Before and After Position Pruning.

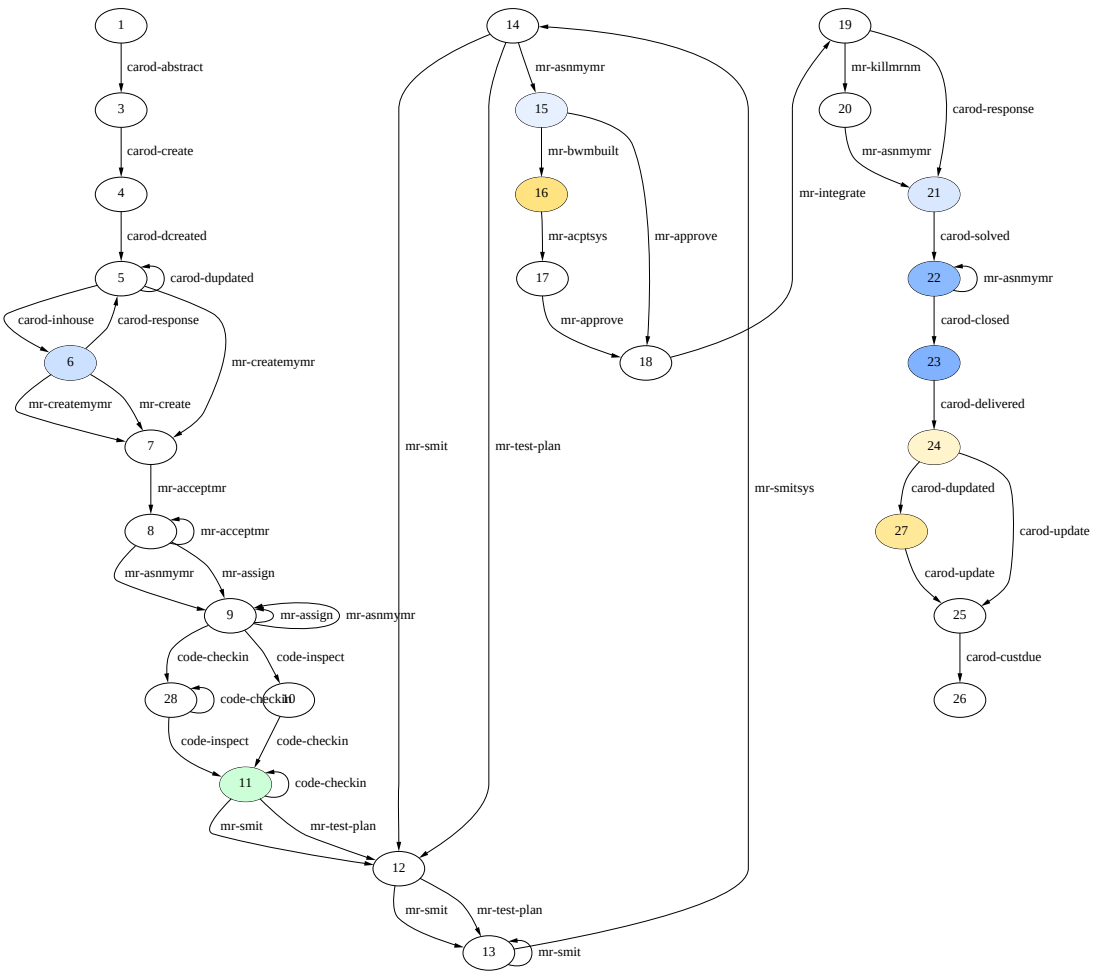

Figure 7: Process Model Used in the Case Study.

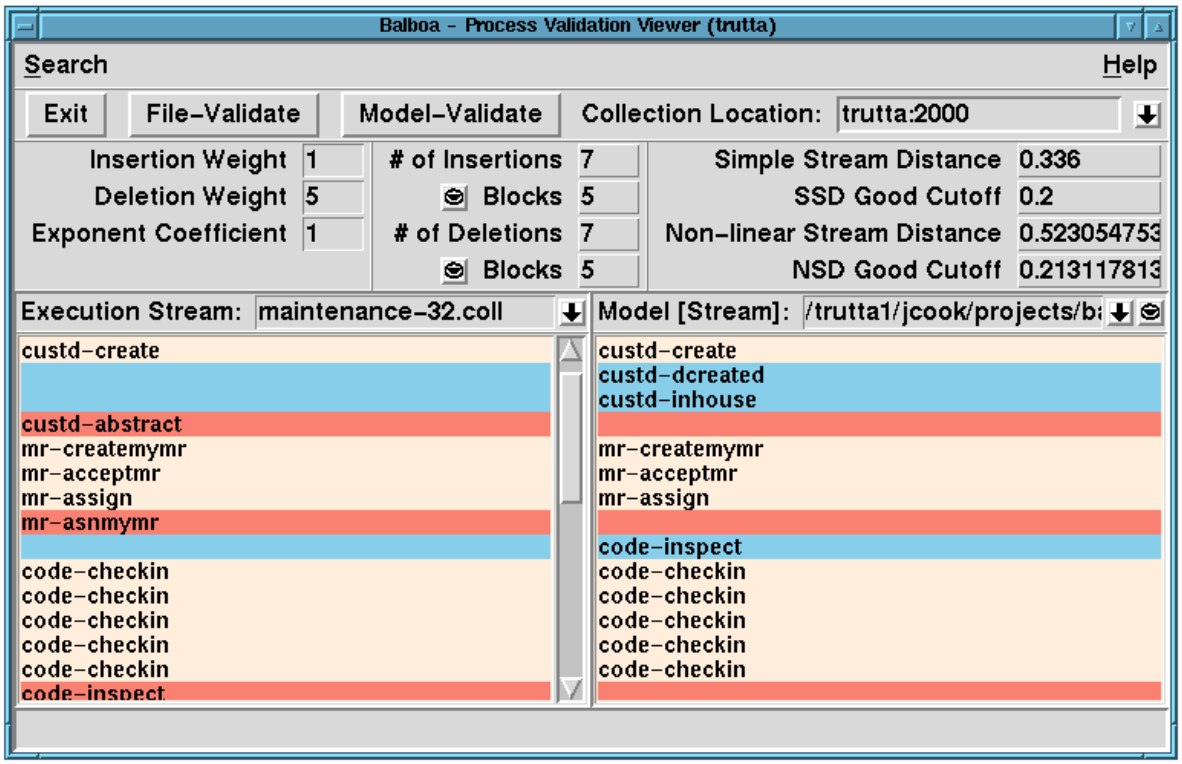

Figure 8: User Interface of the Validation Tool.