From the Proc. of the 3rd Inter. Conf. on the Software Process, Reston, Virginia, USA, October 10–11, 1994

Toward Metrics for Process Validation

Jonathan E. Cook and Alexander L. Wolf

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

{jcook,alw}@cs.colorado.edu

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 are developing techniques for uncovering discrepancies between models and executions under the rubric of process validation*. Further, we are developing metrics for process validation that give engineers a feel for the severity of the discrepancy. We view the metrics presented here as a first step toward a suite of useful metrics for process validation.*

1 Introduction

Over the past several years, software process researchers have demonstrated the utility of formal process models. Their advantages are numerous; a formal model of a process can serve as documentation, as a basis for automating portions of the process, and as a framework in which to perform deductive analyses.

Yet, as useful as these models may be, a common complaint raised by practitioners is that a formal model of a process often does not correspond to what is "really going on" in the process. This could be true because it is difficult to construct an accurate model of a complex process or because the process can change after the model is constructed. In either case, without confidence in its accuracy, practitioners are loath to invest effort in defining the model or to make critical decisions based on the model.

This work was supported in part by the National Science Foundation under grant CCR-93-02739.

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 [23, 25, 26]. 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.

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 [2, 17]. 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.

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

The next section of the paper describes the framework within which the techniques have been developed. Section 3 introduces the process validation metrics. A portion of the ISPW6/7 example [19] is used in Section 4 to demonstrate the application of the metrics to a process. We conclude in Section 5 with a discussion of related work and future directions.

2 Validation Framework

The framework in which the process validation techniques presented here were developed is based on a view of processes as a sequence of actions performed by agents, whether human or automaton, possibly working concurrently. Thus, we are taking a decidedly behavioral view of processes, primarily because we are interested in the dynamic activity displayed by the processes, rather than the static roles and responsibilities of the agents or the static relationships among components of the products. Of course, this does not mean that the static aspects of a process are irrelevant to validation. It is simply that the metrics we have chosen to first investigate are those having to do with behavior rather than structure.

Following Wolf and Rosenblum [25], 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. For purposes of maintaining information about an action, events are typed and can have attributes; one attribute is the time the event occurred. Because events are instantaneous, 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". The overlapping activities of a process, then, are represented by a sequence of events, which we refer to as an *event stream*.

Separate from the process itself are the *(formal) model* of the process and the *execution* of the process. The model is a specification of predicted or intended behavior of the process, while the execution is the actual behavior of the process. In a sense, the process is what the agents think they are doing, the formal model is a description of what the agents should (or could) be doing, and the execution is what the agents are really doing.

We make no assumptions about the formalisms used to model processes, other than that they must have a 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 in-

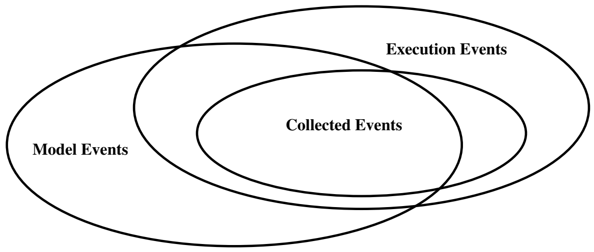

Figure 1: Venn Diagram of Event Types.

clude models based on state machines (e.g., Statemate [16]), Petri nets (e.g., Slang [3] and FUNSOFT Nets [15]), and procedural languages (e.g., APPL/A [24]).

Techniques for process validation clearly depend on an ability to collect data about an executing process, but we do not address this topic here. Fortunately, a variety of methods for collecting process execution data have been devised. Basili and Weiss describe a method for manual, forms-based collection of data for use in evaluating and comparing software development methods [4]. Amadeus is a system for automated collection and analysis of process metrics [22]. Wolf and Rosenblum use a hybrid of manual and automated collection methods [25]. We also do not address the issue of data integrity; we assume that the data are correct (i.e., the events that are collected have actually occurred) and consistent (e.g., all "begin" events for an activity have a corresponding "end" event).

From the previous discussion it should be evident that 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. This is depicted in Figure 1. For example, consider a project that executes a process and performs occasional code inspections as part of that process. A formal model of that process might not account for such an activity.

Figure 1 also illustrates another facet of the problem. The set of events about which data are collected is in general a subset of the set of events generated by the process execution. This means that data about some events, including some events called for in the model, may not get 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.

2.1 Event Visibility

Cost is probably the overriding consideration when deciding which types of events to collect. Thus, one would like to have some sort of justification for collecting or ignoring particular types of events. We can think of the collected event types as acting like a window onto the execution event types, limiting what can be seen of the process execution. The question then becomes, what portion of the model event type set overlaps the window? The answer gives us a characterization of the *visibility* of the execution from the perspective of the model. A low visibility might serve as justification for collecting additional types of events, while too high a visibility might indicate wasted effort collecting irrelevant data.

Let C be the set of collected event types and M be the set of event types associated with the model. Then a simple visibility metric is given by the following ratio.

$$\frac{|C \cap M|}{|M|}$$

Event visibility is thus described by the fraction of model event types that are collected. A value of 1.0 indicates maximal visibility, while a value of 0 indicates no visibility. A somewhat more refined visibility metric normalizes against the number of event sites in a model. It is given by the following ratio

$$\frac{\sum_{i=1}^{|C|} S(c_i)}{\sum_{j=1}^{|M|} S(m_j)}$$

where $c \in C$, $m \in M$, and $S(t)$ is a function that returns the number of event sites in a model for an event type t. The numerator is the total number of event sites for all collected event types that are modeled, while the denominator is the total number of all event sites in the model. This second metric effectively weights the calculation of visibility toward the types of events that appear most often in the model.

Wasted collection effort—that is, effort spent collecting events that are not visible in the model—is also an important thing to measure. Using the style of the first visibility metric we can define the following metric.

$$\frac{|C - C \cap M|}{|C|}$$

This metric is a measure of the fraction of collected events that are only found in the set of execution events and not in the set of model events (see Figure 1). A value of 1.0 indicates that all of the event types being collected are invisible with respect to the model, while a value of 0 indicates that all collected event types are represented in the model.

Notice that these metrics take a static perspective on event visibility. One could go further and devise metrics that normalize against the number of actual events produced dynamically during a process execution. However, it is not clear that rather complicated, dynamic visibility metrics would be any more useful than the simpler, static ones. Of course, more experience is needed with both static and dynamic metrics before definitive statements about their relative usefulness can be made.

3 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 3.4. We defer detailed examples of applying the metrics to Section 4.

3.1 Recognition Metric

The first metric is a very straightforward one that has just two values, true and 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 2, where the lettered boxes indicate events. The recognition metric applied to the two streams would result in the value false because they diverge after the third event.

3.2 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 [20] 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 [21].

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 2 shows two event streams, represented as strings, and one possible correspondence between their events. To transform stream 1 into stream 2, 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. For our purposes in computing a distance between execution and model event streams, we arbitrarily choose the execution event stream as the one to which the operations are applied.

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 [20].

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 paper.

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

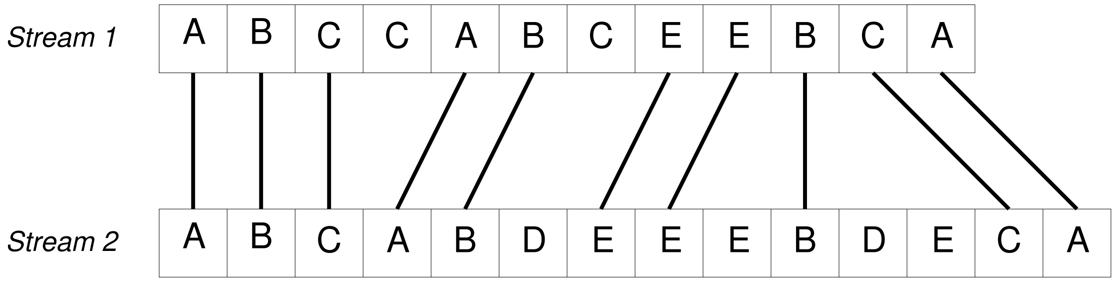

Figure 2: Two Event Streams and One Possible Correspondence of Events.

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. Of course, it would also be useful to highlight the relative importance of different types of events. This can be easily done by varying the weights according to the type of event involved in an operation, although we do not demonstrate that in this paper.

The values of the metric are, for all intents and purposes, bounded between 0 and 1.0; although technically a value greater than 1.0 could appear (e.g., if all events are deleted and some others are inserted), this is highly unlikely. Thus, one might pick the standard statistical correlation rules of thumb [10] 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.)

3.3 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 2, for example, the consecutive D and E insertions required at the end of the streams is 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.

3.4 Producing a Process Model Event Stream

A potential difficulty with our validation metrics is that they assume the existence of two event streams, whereas we really only 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. Of course, this is known to be a hard problem. It requires a search of the state space of a model, and the state space is in general an exponential explosion of the model size. Therefore, we may need to use techniques that can reduce this search.

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. In particular, we can traverse the execution event stream and incrementally derive events for the model event stream by consulting the model. There are elegant methods 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?*". One such method is Constrained Expressions [1]. If the answer is "*yes*", then, in the case of Constrained Expressions, heuristics are used to produce a plausible behavior that leads to the event. This behavior constitutes the next sequence of model events. Constrained Expressions is just one example of the kinds of techniques available to generate a complete model event stream.

4 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 [19]. This is a very 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. 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 3 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.[1]

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 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.[2] 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.[3] 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 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

[1] 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.

[2] Recall that we chose to use the execution event stream as the one to which the transformation operations are applied (see Section 3.2).

[3] The cost ratio given by $W_I = 4W_D$ was chosen arbitrarily for this example. In practice, a process engineer would choose a ratio that best reflects the actual situation that they are analyzing.

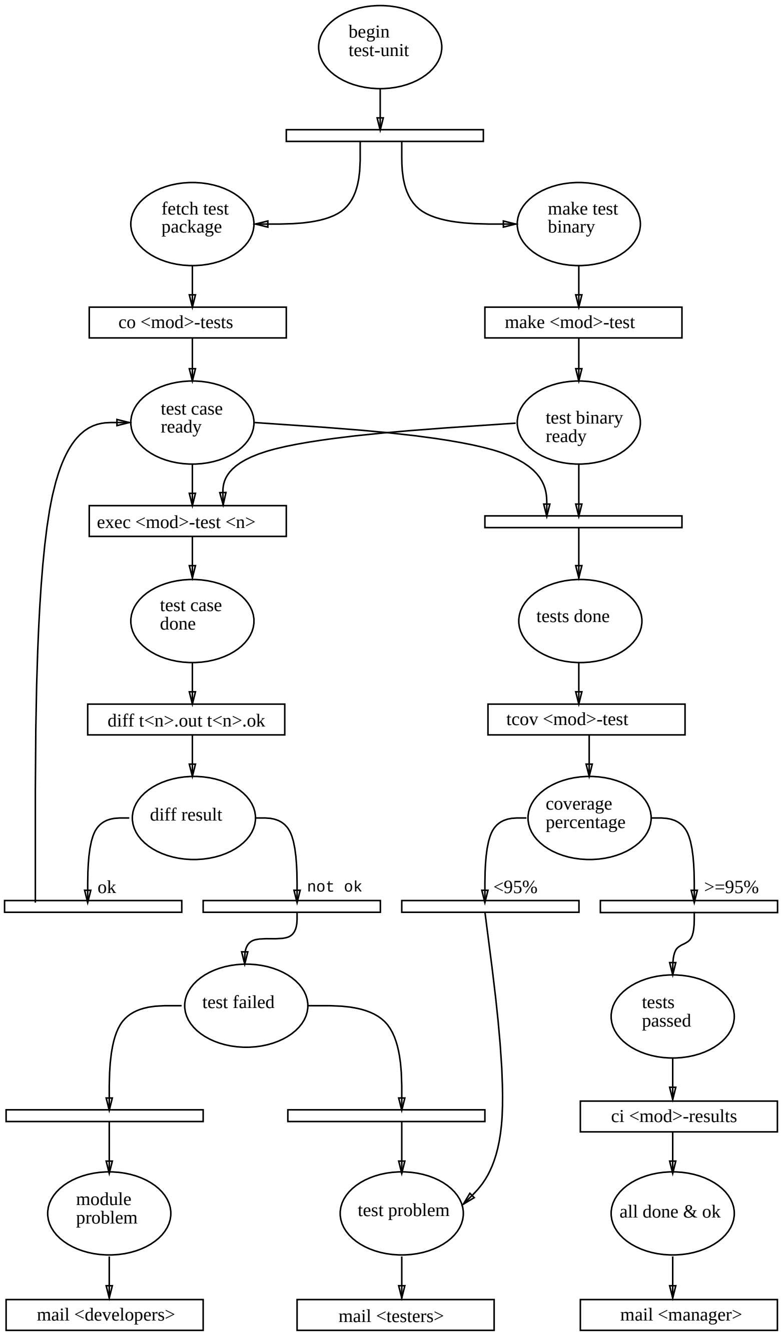

Figure 3: Petri Net Model of the ISPW 6/7 Test Module Task.

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

Table 1: Example Pairs of Execution and Model Event Streams.

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.

Collected Events	Visibility	SSD $W_I = 4$ $W_D = 1$	NSD $W_I = 4$ $W_D = 1$ $k = 1.5$
All	100%	0.15	0.21
make, exec, diff, tcov, ci, mail	88.9%	0.17	0.24
co, make, diff, ci, mail	77.8%	0.19	0.27
make, exec, ci, mail	66.7%	0.38	0.53

Table 3: Process Visibility Effects for the Third Event Stream Pair of Table 1.

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.

Table 3 demonstrates how event visibility (Section 2.1) can affect the validation measures. The table shows various visibilities for the third event stream pair of Table 2. As visibility decreases, the validation measurements deviate more and more from the measurements at the ideal, 100% visibility level. At 66.7% visibility, which at first would seem to be relatively high, the measurements are already quite different from the ideal. Of course, the very small size of our example prevents us from deducing the appropriate visibility percentage cutoff. Yet this example still shows that visibility is a critical issue. Providing guidelines for using visibility measurements will only come with extensive experimentation on real processes.

The example presented here illustrates how the metrics we have defined in this paper 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.

5 Conclusion

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.

- Chmura et al. [8] and Bhandari et al. [6] try to deduce problems in the process by looking at defect data in the products.

- Selby et al. [22] 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.

- Basili and Weiss [4] 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.

- Kellner [18] 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 [13] 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. [14] 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. [7] 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 [25] 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.

We feel that the work described in this paper 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, the most closely related work to ours is in the areas of distributed debugging and history checking.

- Bates [5] 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. [9] 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. [11, 12] 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 separate issue.

In summary, we have proposed and demonstrated several metrics for process validation. These metrics range from simple exact matching, to a linear distance measure in terms of event insertions and deletions, and finally to a non-linear distance measure that takes a broader view of event comparison. We have also developed a notion of event visibility that is intended to inform decisions about the event data to collect or to ignore, and demonstrated how visibility can affect process validation metrics.

Our current work is looking at techniques other than string distance for calculating correspondence metrics. We are also exploring the use of event streams for other measurement purposes, such as efficiency metrics. Besides validation, efficiency metrics are important for determining what areas of the process might need to be optimized and what areas are not performing up to expected efficiency levels.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] V.R. Basili and D.M. Weiss. A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering*, SE-10(6):728–738, November 1984.

[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. 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.

[7] M.G. Bradac, D.E. Perry, and L.G. Votta. Prototyping a Process Monitoring Experiment. In *Proceedings of the 15th International Conference on Software Engineering*, pages 155–165. IEEE Computer Society, May 1993.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.