Environment Support for Improving Software Development Processes: A Vision Influenced by the Work of Barry W. Boehm

Lori A. Clarke

(Department of Computer Science, University of Massachusetts, Amherst, Massachusetts 01003, USA)

Abstract Throughout his career, Barry Boehm has advocated the importance of understanding software development processes, measuring their performance, and using those measurements to guide the development of improved process models. In this paper, we describe PIE, a Process Improvement Environment, which supports that vision. PIE supports the definition of process models that can be analyzed and executed. The analysis is used to detect errors and vulnerabilities in the process models. Validated process models can then be simulated to detect inefficiencies and bottlenecks. Future work includes executing these process models, monitoring their performance, and then using that information to drive further process improvements.

Key words: software development processes; process improvement; process modeling; process analysis; requirement properties; model checking; fault-tree analysis; failure modes and effects analysis; human-intensive systems


1 Introduction

This paper describes environment support for improving software development processes. Much of this work can be traced back to and has built upon contributions made by Barry Boehm over the course of his career. Even in his early papers, Boehm argued for creating a model of the process under which software would be developed (e.g., [2]). He advocated that the process model to be followed in developing a software system was one of the first artifacts that needed to be explicitly articulated and agreed upon by all the stakeholders. Much of Boehm’s career has been devoted to evaluating these process models. From the early waterfall model[25], to the various military standards that encoded this model in doctrine (e.g., [14]), to the more free-wheeling agile methods (e.g., [11, 19, 26]), and to the more thoughtful, risk-based approaches that Boehm has proposed, particularly the Spiral Model[4].

* This material is based upon work supported by the National Science Foundation under Awards CCF-0820198, CCF-0905530 and IIS-0705772. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the NSF.

Paper received on 2011-03-07; Revised paper received on 2011-03-28; Paper accepted on 2011-03-29; Published online 2011-00-00.
and the Incremental Commitment Model\cite{6}, he has undertaken careful assessments of these processes and, based on his experimental findings, reported on their strengths and weaknesses. Working with other pioneers (e.g., \cite{5}), Boehm has argued for a scientific basis for evaluating and improving software development processes, and thus software development.

In this paper, we describe PIE, a Process Improvement Environment, being developed at the University of Massachusetts. This environment provides capabilities for modeling processes, for analyzing these processes for errors and vulnerabilities, and for providing execution and simulation support. We believe that this approach is a step toward supporting the vision that Boehm has consistently advocated throughout his career for explicitly understanding the processes that are being applied, for measuring their actual performance, and for using that information to develop improved process models.

2. Overview of the PIE

The Laboratory of Advanced Software Engineering (LASER) has been developing and evaluating technology to support the continuous improvement of processes. Specifically we have developed a modeling language, called Little-JIL\cite{7}, and a suite of analysis tools for evaluating Little-JIL process definitions. We have applied these technologies to various domains including software development environments\cite{20}, government processes, such as the conduct of elections\cite{28} and on-line dispute resolution\cite{17}, and several life-critical medical processes, such as the administration of chemotherapy\cite{9} and the transfusion of blood\cite{16}.

Creating an accurate and detailed process model is a labor-intensive activity that demands considerable time and effort. Our work has demonstrated, however, that this investment can be leveraged by applying a range of analyzers to the process models to uncover an assortment of errors, vulnerabilities, and inefficiencies. Figure 1 depicts how these analyzers are applied in PIE. The leftmost column in this figure shows the tools used to create a process definition (the Little-JIL editor) and to create properties (the PROPEL property elicitor\cite{10,29}) that specify the requirements that are suppose to be upheld by this process definition. The second column shows the resulting process definition and property specifications and other information needed to support the analysis capabilities, which are shown in the third column. The fourth column shows the outputs from these analysis tools.

![Figure 1. Architecture of the process improvement environment](image-url)
2.1 Little-JIL process definitions

The Little-JIL process language has been shown to be effective in encoding the definition of processes that must handle both normative and exceptional situations, concurrent activities, and the coordination of different types of agents. These agents include human agents with different roles (e.g., developers, testers, managers), different kinds of software systems (e.g., compilers, testing frameworks) and hardware devices (e.g., PDAs). Little-JIL enables specification of a wide range of process details, while still supporting the flexibility and freedom of choice that human agents expect. In addition, it has well-defined semantics so that the resulting process definitions can be rigorously analyzed. Applying Little-JIL to a variety of processes has proved to be challenging, and one of the lessons we have learned is that even so-called simple processes can have considerable complexity. Our experiences have led to improvements to the Little-JIL language, most recently to improvements in the handling of exceptions\(^\text{[18]}\) and to resource modeling\(^\text{[35,36]}\).

Here we show a small example to give a sense of the Little-JIL process definition language. A coordination diagram in Little-JIL specifies the hierarchy of tasks, or steps, in a process. For example, Figure 2 shows a Little-JIL process definition for a Sprint, a subprocess of the Scrum software development process\(^\text{[11,26]}\). In Little-JIL, the steps are shown as a hierarchical decomposition. The steps to be done are each shown as black bars, with each step’s immediate substeps shown underneath. The badge, on the right hand side of the bar, indicates the order in which the substeps are to be done. The Sprint subprocess is the Scrum activity during which actual development work gets done. To be more precise, the Sprint process consists of thirty consecutive performances of the Daily Sprint subprocess (as indicated by the “30” annotation on the edge connecting the Sprint parent step to its Daily Sprint child step and by the right arrow step kind badge in the Sprint step). As indicated by the equal sign step kind badge in the Daily Sprint step, this subprocess is carried out as the parallel performance of its three substeps, Daily Scrum, Checked Work, and Revise Sprint Backlog. Both the Daily Scrum and the Revise Sprint Backlog steps require read and update access to the sprint backlog. The sprint backlog channel provides the concurrency controls needed to coordinates the accesses for these two steps.

The Daily Scrum step is a 15-minute (as indicated by the specification of this deadline by means of the diamond annotation) progress meeting during which the team meets to assess their progress. Note that the sprint backlog artifact is passed in as a parameter, and then also passed out of this step, after which it is written to the sprint backlog channel so that it is made available to the Revise Sprint Backlog step, which may be executing in parallel.

In addition to the execution of the Daily Scrum step, there are multiple performances of the Checked Work step (note the + sign on the edge connecting the Checked Work step to its parent). Each instance of the Checked Work step produces a new version of the product artifact, presumably being comprised of more completed work items after the execution of this step. The agent responsible for performing this step is the team. Concurrently with the performances of the Checked Work step, there may be multiple performances of the Revise Sprint Backlog step (as indicated
Any of the leaf steps in a coordination diagram can be further decomposed as a coordination diagram. Figure 3 shows the coordination diagram for the leaf step Checked Work shown in Fig. 2. Checked Work is comprised of the sequential execution of its two child steps, Work and Integrate, each of which would be further elaborated, but are not shown here. Note that the step Integrate has a post requisite, as shown by the up pointing triangle on the left hand side of the step. If this step’s post requisite is not satisfied, then the exception Build Failed is raised (not actually shown). The closest handler is found by searching up the tree from where the exception is thrown. In this case, the closest exception handler for this type of exception is defined at the immediate parent, Checked Work step. The handler for this exception is defined by the step Rework, the step attached to the edge emanating from the X (for exception) on the right hand side of the step. The step Rework would undoubtedly be further elaborated as a Little-JIL diagram and might throw and handle additional exceptions itself.

Although other process languages could be used in PIE instead of Little-JIL, Little-JIL has several advantages. Because its main abstraction is task decomposition, it is easier than in a data flow representation to specify exception handling with a number of different termination semantics. In addition to rich semantics for exception handling, it provides good concurrency control. Importantly, it also supports hierarchical decomposition and abstraction. Any of the steps, such as Rework, can be invoked from another step, with the appropriate context information being provided through parameter passing. Of particular significance is that the language has well defined semantics and thus processes written in the language can be the subjects of rigorous analysis. We next discuss some of the analysis capabilities that are currently
supported in PIE.

![Figure 3. Elaboration of the checked work step shown in Figure 2](image)

2.2 Process model narrator

The description of the Scrum process given in Section 2.1 is based on the diagrammatic representation shown in Figs. 2 and 3. Alternatively, PIE can take the diagrammatic representation and automatically generate a hyperlinked textual representation. We have found that non-computer scientists often prefer a natural language description. In fact, even computer scientists have found the textual description to be a useful alternative representation that helps them find errors in the encoding of the process in Little-JIL. Figure 4 shows the generated textual description of the Little-JIL process definition shown in Fig. 2. Unlike Electronic Process Guides, which tend to provide a broader range of alternative representations but require considerable manual input, the Little-JIL narrator automatically generates an English description and addresses many of the limitations pointed out by Boehm and his co-authors in their evaluations of EPGs.

The narrative generation is done using templates for each of the Little-JIL constructs. There are a few customization rules that allow for the use of synonyms and control over the level of verbosity. It would be interesting indeed to see if descriptions could be easily generated for other languages in addition to English.

It is also interesting to note that computer scientists went to diagrammatic representations because natural language descriptions are usually imprecise, incomplete, and ambiguous. The generated textual descriptions, however, are as precise, complete and unambiguous as the diagrammatic representation from which they are derived. They are, however, verbose and not very “natural”, although often easier for non-computer scientists to comprehend than the original diagrams.

2.3 Validating process definitions

Our work on developing process definitions has clearly indicated the importance of validating those processes. There are three main thrusts here.

- If a process is important and complex enough to warrant being modeled, then those models need to be scrutinized for errors and weaknesses before using them.
Such analysis might also uncover ways in which the model deviates from the actual desired process. All of these discoveries should lead to improved process models.

- Applying analysis approaches to the carefully validated models will help detect errors or weaknesses in the real processes. This is obviously an important goal that could reduce errors and improve efficiency.
- Analyses can be used to carefully vet proposed process improvements to assure that they do not introduce new errors or inconsistencies. Detecting problems before they are actually introduced could reduce errors and avoid inefficiencies. Again, if processes are complex enough to warrant careful modeling, then the implications of changes to the process may not be obvious in all their manifestations.

As shown in Fig.1, PIE currently supports using model checking to verify safety properties. The properties are represented using PROPEL, a tool to support both domain experts and computer scientists in creating mathematically precise specifications of desired properties. PROPEL allows domain experts to work with natural language, while assisting the specifiers in accurately capturing the often subtle details of such specifications. Specifying the desired properties of complex domains, however, made us aware of the need to extend PROPEL with support for exceptional
After a property is specified, the FLAVERS model checking system[15] determines if there are any possible process executions that could violate the specified property. If so, a path through the process that causes the violation is shown. To mitigate the well-known state explosion problem associated with model checking, we have developed a number of effective optimization techniques[30–32]. Providing process language constructs that give the human agent considerable flexibility tend to increase the size of the model, especially when combined with concurrency and complex exception handling. On the other hand, process models tend to focus on coordination and the flow of control, so many of the problems associated with aliasing and unknown values do not arise. Thus, most of the counter example paths that are found are executable and correspond to real problems, eliminating the abundance of false positives that usually occur with model checking.

To complement property verification, fault tree analysis (FTA) and failure mode and effects analysis (FMEA) techniques are applied. FTA[33] and FMEA[13] are well known safety analysis approaches for identifying process vulnerabilities due to incorrect performance of process activities. In contrast, model checking assumes the steps are done correctly but looks for violations of temporal properties. For a user specified hazard, such as an incorrect product being produced by a Scrum, our FTA tool[8] automatically derives the fault tree from the Little-JIL definition and then determines the minimal cut sets, the minimal combinations of inaccurately applied steps that could cause the hazard to occur. The fault tree for this hazard is shown in Fig.5 for the Sprint process definition given in Figs. 2 and 3. Conversely FMEA can be automatically applied to the Little-JIL process definition[34] to show how a faulty step in the process could propagate inaccurate results to other process steps, providing insight about possible hazards that should be considered for subsequent FTA analysis.

It usually takes considerable human effort to create a single fault tree or FMEA table, but with our approach considerable care is taken to create the process definition that is then used to automatically create FMEA tables or fault trees for a number of potential hazards.

As noted, the safety analysis and model checking approaches complement each other. The safety representations derived from the process model would be of little value if the process model was not carefully verified using model checking or other validation techniques. The minimal cut sets derived from the fault tree, however, describe combinations of events that could lead to a hazard and, thus, could subsequently be the basis for additional properties to be verified using model checking.

2.4 Discrete event simulation

The model checking and the safety analyses described above focus on errors and vulnerabilities that can be detected statically, without the need for execution. Discrete event simulation, on the other hand, is an analysis approach that uses dynamic execution state to study a complementary range of issues. The Little-JIL simulator, JSim, takes as input a Little-JIL process definition and specifications of how to simulate the behaviors of the agents as well as a model of the resources[24,35,36] We have used JSim to study how different resource mixes and allocation strategies affect pro-
cess flow. Our experience, although preliminary, suggests that modeling processes and their required resources in detail seems to improve the accuracy of the simulations.

![Fault tree diagram](image)

Figure 5. Fault tree automatically generated for the hazard in which an incorrect product is provided from the scrum in the sprint defined in Figs. 2 and 3

### 3 Conclusion

We have observed that the very act of trying to model a process or its properties leads to improved understanding of that process, to discovering process defects, and to identifying possible process improvements. The model checking that we applied successfully found important violations. Even when provided with counterexample traces, however, the modifications initially proposed by domain experts sometimes did not fix the problem or introduced new errors. Similarly, our safety analysis tools uncovered previously unknown errors in these processes. Even when the errors were in the introduced artifacts (i.e., the process model or property), finding these errors improved the accuracy of these artifacts and the subsequent analyses based on them. We think it is particularly important to emphasize that this investment enables proposed process modifications to be carefully vetted before being introduced.

In our recent work, we are moving to an environment where the validated process
definitions are executed. We plan to develop and evaluate capabilities that support process execution, monitoring, and guidance. Moreover, context information gathered during monitoring will be used to inform analysis capabilities, while the analysis capabilities can provide valuable insight that will enable the guidance technologies to warn of impending hazards and propose compensating actions. Thus, we plan to move beyond static modeling and analysis techniques, to an environment that synergistically uses static analysis results to provide process execution guidance while gathering operational information that can be used to improve the static analysis capabilities.

In many respects, the work suggested here is consistent with the classical notion of continuous process improvement introduced by Shewhart, effectively applied by Deming, and strongly advocated for software development by Boehm (e.g., [1; 3-6]). The essence of this approach is to capture the process to be improved, compare its characteristics to those that are desired, identify weaknesses and shortcomings, propose and evaluate improvements, and then incorporate those improvements in the process to complete the improvement cycle and form the basis for a subsequent improvement cycle. This cycle relies essentially on the ability to understand the process, understand its desired properties, and analyze the ways in which the process does or does not adhere to those properties. Typically, these analyses have been obtained informally. Recent research, including our own, has shown that processes and properties can be defined with precise notations and evaluated for various kinds of consistency using automated reasoning approaches. It is this rigorous approach for defining properties and properties, along with powerful reasoning techniques, that we now propose to apply to processes. Preliminary work applying this approach to software engineering processes, particularly Scrum and the Spiral Model, have demonstrated some of the insights that can be obtained.

Acknowledgements

The work reported here has been done in close collaboration with Leon J. Osterweil and George S. Avrunin. A number of former and current students have contributed to this project including Aaron Cass, Bin Chen, Stefan Christov, Jamieson Cobleigh, Rachel Cobleigh, Heather Conboy, Matt Dwyer, Eric McCall, Gleb Naumovich, Huong Phan, M.S. Raunak, and Jianbin Tan, as well as staff programmer, Alexander Wise.

References

[6] Boehm BW, Lane JA. Using the incremental commitment model to integrate system acquisition, systems engineering, and software engineering. CROSSTALK The Journal of Defense Software


[29] Smith RL, Avrunin GS, Clarke LA, Osterweil LJ. Propel: An approach supporting property


