Automated Validation and Verification of Process Models

Darren C. Atkinson
Department of Computer Engineering
Santa Clara University
Santa Clara, CA 95053-0566
atkinson@engr.scu.edu

John Noll
Department of Computer Engineering
Santa Clara University
Santa Clara, CA 95053-0566
jnoll@engr.scu.edu

ABSTRACT
In process programming, processes are modeled as pieces of software, and a process programming language is used to specify the process. Such a language resembles a conventional programming language, providing constructs such as iteration and selection. This approach allows models to be simulated and enacted easily. However, it also suffers from the same problems that plague traditional programming, such as the question of whether the program itself is semantically correct or contains errors. We present an automated approach for detecting errors in such process models. Our approach is based on static code analysis techniques. We have developed a tool to analyze processes modeled using PML and have subsequently successfully redesigned models using our tool.

KEY WORDS
Process Programming, Modelling Languages, Modelling and Simulation, Static Analysis

1 Introduction

1.1 Motivation

In 1987, Osterweil asserted that "software processes are software too" [1], and thus could (and should) be developed, analyzed, and managed using the same software engineering methods and techniques that are applied to software. This idea implies there is a software process life-cycle that resembles the software life-cycle, involving analysis, design, implementation, and maintenance of software processes [2]. One of the outgrowths of this line of research is the notion of *process programming*: the specification of process models using process programming languages that resemble, and in some cases are derived from, conventional programming languages [3].

One advantage of process programming is that a process model can be coded and simulated or enacted easily. An enactment engine can, for example, automatically notify actors when they should begin execution of a particular task. However, process programming is also subject to all of the pitfalls of traditional programming and software engineering. In particular, there is the possibility of errors in the program and, more importantly, errors in the design and in the capturing of the requirements.

Copyright 2003 IASTED. Published in the Proceedings of the 7th IASTED International Conference on Software Engineering and Applications (SEA'03), November 3–5, 2003, Marina Del Rey, CA. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from IASTED.

There is a large body of knowledge comprising techniques for analyzing programs written in conventional programming languages. These techniques enable programmers to assess the correctness of their programs, identify potential faults, and, as in the case of optimizing compilers, automatically redesign the implementation of a program to improve execution performance. We would therefore like to apply these techniques to the analysis of process programs in order to help the process engineer find errors before simulation or enactment of a model. Specifically, we would like to use these techniques to validate the correctness of process programs as models of real-world processes and aid in process redesign.

1.2 Approach

In this paper we present a technique for analyzing the flow of *resources* through a process, as specified by a process program. This technique, derived from research into dataflow analysis of conventional programming languages, enables a process designer to answer important questions about a process model, including:

- Does a process actually produce the product that it is supposed to produce?
- Are intermediate products consumed by later steps in a process actually produced by earlier steps?
- Does the flow of resources through a process match the flow of control?

The answers to these questions can result from errors in the specification, indicating a need for further capture and modeling activities; or, they may highlight flaws in the underlying process, indicating a potential for process improvement. To validate our hypotheses, we have developed a tool to analyze specifications written in the PML process programming language [4].

We have used our tool to analyze software process models and present an in-depth analysis of the redesign of one model, used by students for their senior projects. Our tool detected 63 errors in the model, which consisted of only 204 lines of PML code. Through iterative use of the tool, we were able to successfully redesign the model.

We begin with a brief overview of PML, to provide a context for discussing our technique. Then, we present our

analysis technique and discuss the implementation of our tool. We discuss our results of applying the tool to actual PML specifications. We conclude with our assessment of the technique and potential directions for future work.

2 The PML Language

PML is a simple process programming language that is intended to model organizational processes at varying levels of detail [4]. PML was designed specifically for rapid, incremental process capture, to support both process modeling and analysis, and process enactment [5]. Using PML, a process can be specified initially at a very high level that contains only major process steps and control flow.

PML reflects the conceptual model of process enactment developed by Mi and Scacchi [6]. This model views a process as a situation in which agents use tools to perform tasks that require and produce *resources*. PML models processes as collections of actions that represent atomic process tasks. PML specifies the order in which actions should be performed using conventional programming language control flow constructs such as sequencing, iteration, and selection, as well as concurrent branching of process flows:

- Sequence—A series of tasks to be performed in order:

```
sequence {
    action first {}
    action second {}
}
```

- Iteration—A series of tasks to be performed repeatedly:

```
iteration {
    action first {}
    action second {}
}
action go_on {}
```

- Selection—A set of tasks from which the actor should choose *one* to perform:

```
selection {
    action choice_1 {}
    action choice_2 {}
}
```

- Branch—A set of tasks that can be performed concurrently (all tasks in a branch must be performed before the process can continue):

```
branch {
    action path_1 {}
    action path_2 {}
}
```

The *provides* and *requires* fields of an action specify how resources are transformed as they flow through a process. As such, they capture several important facts about a process, namely what conditions must exist before an action can begin, and what conditions will exist after an action is completed. As a result, the *requires* and *provides* predicates specify the purpose of an action, in terms of how the action affects the products under development. The simplest form of a resource predicate simply names the resource:

```
provides { resourceName }
```

This predicate states that the output of an action is a resource bound to the variable *resourceName*. Resource specifications may also be predicates that constrain the state of the resource:

```
requires { resourceName.attributeName op value }
```

Here, *op* is any relational operator. Predicates may also be joined using conjunction and disjunction. In short, resource predicates allow process designers to specify in some detail how a product evolves as a process progresses, as well as what resources are required to produce a product, and the state those resources must have before the process can proceed.

3 Analysis of Resource Flow

What can we learn from analysis of syntactically correct process programs? Analysis helps in two phases of the process life-cycle. First, by analyzing the flow of resources through a process specification, we can identify situations where provided and required resources do not match. This information is useful for validating process specifications against reality; such inconsistencies may indicate gaps in process capture and understanding.

Second, resource analysis can also point out potential areas of improvement in the process being modeled. Inconsistencies between provided and required resources signal a potential for re-engineering to make the process more effective. For example, if a sequence of actions does not have a resource flowing from one action to the next, it may be possible to perform those actions concurrently.

In the following sections, we examine in detail the kinds of inconsistencies that can exist in a specification and their potential impact on a process. Then, we discuss the design of a tool for detecting these inconsistencies in PML specifications.

3.1 Categories of Resource Inconsistencies

Inconsistencies can be classified into several situations:

1. A resource is provided by an action that does not require any resources. This situation (termed a "miracle") could represent a modeling error where the modeler failed to capture an action's inputs; or, it could represent a real situation where the actor generates something like a document from (intangible) ideas:

```
action describe_problem {
    /* requires inspiration */
    provides { problem_description }
}
```

2. A resource is required by an action that does not provide any resources. This situation (termed a "black hole") could represent a legitimate activity, such as a task that requires the actor to read certain documents and develop an "understanding" of their contents; the action produces no tangible results, but is worthwhile nevertheless:

```
action understand_problem {
    requires { problem_description }
    /* Provides nothing tangible */
}
```

3. A resource is required, but a different resource is provided. Occasionally, this situation (termed a "transformation") represents a modeling error, but is more often the desired result: an action consumes some resources in the production of another. A simple example happens when a document is assembled from different sections: the action requires each section, and provides the completed document:

```
action submit_design_report {
    requires { use_cases && architecture }
    provides { design_report }
}
```

4. Required resource not provided. In this situation, an action requires a resource that is not provided by any preceding action:

```
action a { provides { r } }
action b { requires { s } }
```

5. A provided resource is never used. An action might provide a resource that is never required by a subsequent action:

```
action a { provides { r && s } }
action b { requires { r } }
```

Inconsistencies due to unprovided or unrequired resources are not necessarily errors: an unrequired resource could indicate an action that represents an output of a process; an unprovided resource could indicate a point where the process receives input from another process.

6. A provided resource does not match a subsequent resource requirement. Here, the resource is not missing, but rather in the wrong state:

```
action a { provides { r.status == 1 } }
action b { requires { r.status == 2 } }
```

3.2 Analysis Tool Design

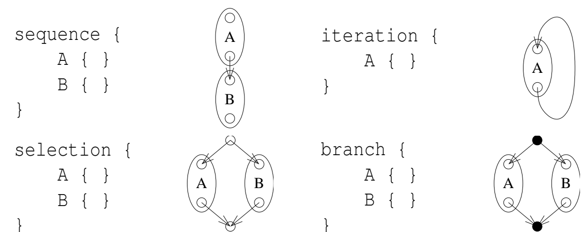
Our analysis tool, called `pmlcheck`, is designed to complement the PML compiler. The compiler generates executable models, useful for simulation and enactment, and `pmlcheck` can tell the process engineer interesting things about these models.

```
function check-if-provided (node,resource,start)
  visited [node] := true
  status [node] := unknown

  if node ≠ start and resource ∈ provided [node] then
    status [node] := true
    necessary [node] [resource] := true
  else
    for pred in predecessors [node] do
      if visited [node] = false then
        check-if-provided (pred,resource,start)
      end if
      status [node] := φ_node (status [node], status [pred])
    end for
  end if
end function
```

Figure 1. Basic algorithm used by `pmlcheck`.

To compute the flow of resources though a PML program, `pmlcheck` constructs a *process graph* similar to a control-flow graph in conventional languages. Each atomic *action* becomes a graph node. The graphs for other constructs are easily constructed in a syntax-directed manner:



The colored nodes in the *branch* graph distinguish it from the *selection* graph, since in the former all paths are always executed and in the latter only one path is executed.

The first three inconsistencies described in the Section 3.1 are *local* to an action node and are easily checked without traversal of the graph. However, the latter three inconsistencies require *global* knowledge of resources and therefore require a graph traversal.

The basic algorithm used to check if a required resource is provided is given in Figure 1. The algorithm performs a depth-first search of the process graph looking for a node that provides the required resource. The function ϕ_{node} is a decision function that updates the status of a node given its current status and the status of a predecessor. Effectively, ϕ_{node} performs a boolean `and` for a *selection* since a resource must be provided on all paths to be definitely provided, and performs a boolean `or` for a *branch* since it is enough that the resource be provided on any path since all paths are guaranteed to be executed. The algorithm also records those provided resources that were found during the search. This information is used to determine which resources are provided, but never required.

3.3 Further Design Considerations

Rather than using a separate analyzer, we could require that global consistency be enforced at compile time, as many

modern programming languages do. However, such a policy is generally not desirable. First, it is not necessary: useful process analysis and enactment are possible without global consistency. Second, it is not always possible. Process capture is an iterative process that uncovers hidden activities over time, as process understanding emerges. Thus it is desirable to allow specifications that are incomplete or inconsistent. Finally, valid models can be inconsistent, because the underlying process being modeled is inconsistent. An organization's processes may contain useless steps, missing steps, or sequences of activities that do not produce desired results. Nevertheless, it is important to document these processes accurately, to establish a baseline for process redesign. Therefore, the process engineering environment must be tolerant of inconsistencies that exist in the real world.

4 Examples and Results

To assess the effectiveness of `pmlcheck`, we analyzed two software development processes: the development process used to conduct Computer Engineering Senior Design projects at Santa Clara University, and a graduate Software Engineering course software development process.

4.1 SCU Senior Design Process

Our first experiment employed `pmlcheck` to aid in the creation of a model of the Santa Clara University Computer Engineering department's senior design project process. The process spells out a set of milestones and deliverables roughly based on Boehm's Anchoring Milestones [7].

We first did an initial capturing of the process in which we simply translated the narrative specification into PML. Then, we used the analysis provided by `pmlcheck` to improve the accuracy of the model by correcting specification errors and elaborating resource specifications.

The first version of the model was a simple translation of the narrative specification into a PML specification. We modeled each milestone as a sequence of actions, each action producing a single deliverable.

The tool reported 63 potential inconsistencies in this initial model (see Table 1). How many of these were actual errors? To determine the answer, we analyzed the reported inconsistencies in detail, categorizing them as follows:

- Specification error—The modeler made a mistake in the program specification such as misspelling a resource name.
- Modeling error—The model did not match the underlying process. For example, an action was out of order or was missing.
- Process error—The model was correct, but the underlying process contained an inconsistency.
- Spurious error—The tool correctly identified an error, but the error was triggered by a previous error.
- No error—The tool incorrectly reported an inconsistency.

Of the 63 reported inconsistencies, three were specification errors where a resource name was misspelled, and two were spurious errors, caused by the specification errors. An additional two were not errors as they represented process output.

The remaining 56 errors were the result of incorrectly modeling some aspect of the process, such as omitting a required or provided resource from an action (42 inconsistencies). These conclusions are summarized in Table 2.

Perhaps most interesting from a process engineering viewpoint, 13 errors were the result of omitting actions to capture and deliver document components as a single document; for example, one sequence was missing a "submit design report" action to assemble the document parts and deliver them as a completed "design report" resource. We used our analysis of the initial version of the model to correct the errors uncovered by `pmlcheck`. In the new model, 12 inconsistencies were reported, none of which were errors, as they represented process input or output.

4.2 Graduate Software Processes

We also used to `pmlcheck` to analyze twenty-four process models developed by graduate software engineering students to describe the class project development process. The intent was to develop formal models of the processes specified by the instructor as narrative text in assignments and lectures, augmented by the students' personal experience. The analysis results are shown in Table 1.

4.3 Discussion

It appears from these experiments that the majority of inconsistencies reported by `pmlcheck` are unprovided or unrequired resources. This is its chief limitation: since PML does not distinguish between provided and required resources and process inputs and outputs, `pmlcheck` takes a conservative approach and reports process inputs as unprovided resources, and outputs as unrequired resources.

Curiously, the Graduate Software Development processes contained only two miracles and no black holes among 95 actions; in contrast, the initial Senior Design model had 28 miracles and 15 black holes. This appears to be the due to careful attention to detail on the part of the three modelers who wrote these specifications. Also, `pmlcheck` reported 67 transformations in the Graduate Software Development processes; 31 of these proved to be specification errors that caused the provided resource to appear to be a new resource rather than a modification of the required resource. This was a surprise: we had anticipated that most actions identified as transformations would

Model	Lines	Actions	Resources	Empty	Unprovided	Unrequired	Miracles	Black Holes	Trans.
Senior Design									
senior_design.pml	204	35	69	1	3	16	28	15	36
senior_design2.pml	290	37	122	0	6	6	0	0	42
Graduate S/W Development									
Architecture.pml	130	16	26	3	5	5	0	0	13
Checkout.pml	11	1	2	0	1	1	0	0	1
Commit.pml	12	1	2	0	1	1	0	0	1
Edit.pml	27	3	6	0	2	2	0	0	3
PostMortem.pml	44	7	4	4	0	2	2	0	3
Update.pml	18	2	4	0	2	2	0	0	2
checkin.pml	13	1	2	0	1	1	0	0	1
checkout.pml	16	1	2	0	1	1	0	0	1
make.pml	13	1	2	0	1	1	0	0	1
milestone1.pml	172	18	36	0	13	13	0	0	8
milestone5.pml	141	15	30	0	10	10	0	0	6
updateANDresolve.pml	22	2	4	0	2	2	0	0	1
Analysis.pml	10	1	3	0	2	1	0	0	1
FunctionalRequirements.pml	10	1	3	0	2	1	0	0	1
Milestone2.pml	59	7	21	0	7	7	0	0	7
Milestone3.pml	45	5	15	0	5	5	0	0	5
NonFunctionalRequirements.pml	10	1	3	0	2	1	0	0	1
OperationalConcept.pml	10	1	3	0	2	1	0	0	1
ProjectLog.pml	10	1	3	0	2	1	0	0	1
RepositoryCheckIn.pml	10	1	3	0	2	1	0	0	1
RepositoryCheckOut.pml	20	2	5	0	3	2	0	0	2
RepositorySynchronize.pml	20	2	5	0	3	2	0	0	2
RiskIdentification.pml	10	1	3	0	2	1	0	0	1
SourceCodeEdit.pml	33	4	6	1	3	3	0	0	3
TOTAL	866	95	193	8	74	67	2	0	67

Table 1. Detailed analysis of the errors reported for all models.

Model	Total	Spec.	Model.	Proc.	Spurious	No Error
original	63	3	56	0	2	2
revised	12	0	0	0	0	12

Table 2. Classification of analysis results for the original and the revised senior design models.

actually transform resources into new resources. Thus, it appears to be useful to optionally flag actions that transform resources for closer examination.

5 Related Work

5.1 Program Analysis

Many of the checks performed by our tool are analogous to those checks performed by optimizing compilers such as `gcc` and static checkers such as `lint`. Optimizing compilers typically warn the user regarding possibly uninitialized variables. Our analysis tool informs the user regarding resources that are required without possibly being provided. As another example, register allocation [8], the process of effectively assigning registers to variables to increase execution speed, requires knowledge of the lifetimes of variables in a program. Such knowledge is obtained by computing when a variable is first and last possibly referenced, which is analogous to determining when a resource is first provided and last required.

Algorithms for analyzing programs described as graphs are well-known [9, 10] as are algorithms for computing properties of the graphs [11]. Finally, other tools to aid the programmer in finding errors in programs include assertion checkers [12] and program slicing tools [13, 14].

5.2 Process Validation

Cook and Wolf [15] discuss a method for validating software process models by comparing specifications to actual enactment histories. This technique is applicable to downstream phases of the software life-cycle, as it depends on the capture of actual enactment traces for validation. As such, it complements our technique, which is an upstream approach.

Similarly, Johnson and Brockman [16] use execution histories to validate models for predicting process cycle times. The focus of their work is on estimation rather than validation, and is thus concerned with control flow rather than resource flow.

Scacchi's research employs a knowledge-based approach to analyzing process models. Starting with a set of rules that describe a process setting and models, processes are diagnosed for problems related to consistency, completeness, and traceability [2]. Conceptually, this work is most closely related to ours; many of the inconsistencies uncovered by `pmlcheck` are also revealed by Scacchi and Mi's *Articulator* [17]. Although PML and the *Articu-*

lator share the same conceptual model of process activity, there are important differences. Their approach is based on knowledge-based techniques, with rule-based process representations and strong use of heuristics. This is a different approach than PML's, which closely resembles conventional programming. Thus, our analysis technique is derived from programming language research.

6 Conclusion

What can we conclude about data-flow analysis of process programs? Data-flow analysis can uncover specification errors, such as misspelled resource names, that can exist in otherwise syntactically correct process specifications. Without analysis, these errors would not be detectable until the process is executed. Also, resource flow analysis can identify inconsistencies between a specification and the process it models. This was shown in Section 4, where our initial Senior Design process model was missing several resource dependencies that were important to the process. Further, data-flow analysis can validate that a process produces the products it was intended to produce. By verifying that the resource flow specified by the process program proceeds correctly from beginning to end, the process designer can validate that the process does in fact transform its inputs into the desired outputs. Finally, in addition to identifying potential errors in a process specification, resource flow analysis can suggest opportunities for redesign of a valid process.

For example, our revised Senior Design model contains six actions that require the "problem statement" resource. Where does this resource come from? At present, the process assumes that the problem statement exists prior to the beginning of the process. But the intent of the process is for professors to provide problems for student teams to solve; so the process should include a phase where students and professors negotiate the problem statement, which then serves as the input to the Conception phase.

6.1 Future Work

A sequence specifies a temporal dependency between actions: a predecessor must be completed before the successor can begin. This implies that the predecessor does something that the successor needs; in other words, the predecessor provides something that the successor requires. If the resources analysis shows that no resource flows between sequential actions, however, it may indicate an opportunity for concurrency. In this case, the process specification indicates a dependency among actions that does not exist.

The opposite situation occurs when the control-flow specification indicates that actions can be performed concurrently, but the resource flow among them requires that they performed in a certain order. This situation may indicate either an error in process capture, or a problem with the process itself.

This suggests a tool for automatically transforming a specification into an equivalent specification based on the resource flow graph. Such a tool would analyze the resource dependencies among actions, then re-arrange their ordering so that the control flow matches the resource flow.

Finally, the analysis of actual PML programs discussed in Section 4 revealed certain deficiencies in PML. Specifically, since PML makes no distinction between resources provided by or required from actions within the process and resources provided by or to the external environment, `pmlcheck` cannot distinguish between an unprovided resource and a process input, and likewise between an unrequired resource and a process output. This suggests the need for an enhancement to PML to allow the process modeler to specify the process inputs and outputs.

References

[1] L. J. Osterweil, Software processes are software too, *Proc. 9th Intl. Conf. Soft. Eng.*, Monterey, CA, 1987, 2–13.

[2] W. Scacchi, Understanding software process redesign using modeling, analysis and simulation, *Softw. Process. Improv. and Pract.*, 5(2–3), 2000, 183–195.

[3] S. M. Sutton, Jr., D. Heimbigner, and L. J. Osterweil, APPL/A: A language for software process programming, *ACM Trans. Softw. Eng. Meth.*, 4(3), 1995, 221–286.

[4] J. Noll and W. Scacchi, Supporting software development in virtual enterprises, *J. Digit. Inf.*, 1(4), 1999.

[5] W. Scacchi and J. Noll, Process-driven intranets: Life-cycle support for process reengineering, *IEEE Inter. Comput.*, 1(5), 1997, 42–49.

[6] P. Mi and W. Scacchi, A knowledge-based environment for modeling and simulating software engineering processes, *ACM Trans. Knowl. Data Eng.*, 2(3), 1990, 283–289.

[7] B. W. Boehm, Anchoring the software process, *IEEE Softw.*, 13(4), 1996, 73–82.

[8] F. C. Chow and J. L. Hennessy, A priority-based coloring approach to register allocation, *ACM Trans. Prog. Lang. Syst.*, 12(4), 1990, 501–536.

[9] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools* (Reading, MA: Addison-Wesley, 1986).

[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren, The program dependence graph and its use in optimization, *ACM Trans. Prog. Lang. Syst.*, 9(3), 1987, 319–349.

[11] T. Lengauer and R. E. Tarjan, A fast algorithm for finding dominators in a flowgraph, *ACM Trans. Prog. Lang. Syst.*, 1(1), 1979, 121–141.

[12] D. Jackson, ASPECT: An economical bug-detector, *Proc. 13th Intl. Conf. Soft. Eng.*, Austin, TX, 1991, 13–22.

[13] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers, Improving program slicing with dynamic points-to data, *Proc. 10th ACM Sym. Foun. Soft. Eng.*, Charleston, SC, 2002, 71–80.

[14] M. Weiser, Program slicing, *IEEE Trans. Softw. Eng.*, SE-10(4), 1984, 352–357.

[15] J. E. Cook and A. L. Wolf, Software process validation: Quantitatively measuring the correspondence of a process to a model, *ACM Trans. Softw. Eng. Meth.*, 8(2), 1999, 147–176.

[16] E. W. Johnson and J. B. Brockman, Measurement and analysis of sequential design processes, *ACM Trans. Des. Autom. Electron. Syst.*, 3(1), 1998, 1–20.

[17] W. Scacchi and P. Mi, Process life cycle engineering: A knowlege-based approach and environment, *Int. J. Intell. Syst. Account. Financ. Manage.*, 6(2), 1997, 83–107.