

# Automated Checking of Software Process Models

Darren C. Atkinson, John Noll

*Department of Computer Engineering, Santa Clara University, 500 El Camino Real,  
Santa Clara, CA 95053-0566*

---

## Abstract

Using the process programming concept, processes are modeled as pieces of software. A process programming language is used to formally specify the process. Such a language resembles a conventional programming language, typically providing constructs such as iteration and selection. One advantage of this approach is that a process model once coded can be simulated or enacted easily.

However, this approach also suffers from the same problems that plague traditional programming tasks, such as the question of whether the program itself is meaningful (i.e., semantically correct) or contains errors. Such program errors are the result of a miscoding of the model or of an error in the model itself.

We present an automated approach for detecting errors in such process models. Our approach is based on static code analysis techniques used by optimizing compilers. By slightly adapting these well-understood techniques, errors can be found quickly before simulation. We have developed a tool to perform such semantic checks on processes modeled using the PML process modeling language and have successfully analyzed and subsequently re-designed software process models using our tool.

*Key words:* Software process models, Process programming, Static analysis

---

## 1 Introduction

In 1987, Osterweil asserted that “software processes are software too” (Osterweil, 1987), 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 (Scacchi, 2000). 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 (Sutton, Jr. et al., 1995).

## 1.1 Motivation

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 (e.g., notify a test engineer when a new piece of code is ready for testing). However, process programming (and process program engineering) is also subject to all of the pitfalls of traditional programming and software engineering. Namely, the possibility of errors in the program and, more importantly, errors in the design and in the capturing of the requirements.

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 (e.g., variables that are used without being initialized), and, as in the case of optimizing compilers, automatically redesign the implementation of a program to improve execution performance (e.g., through the movement of loop-invariant code to outside the loop). 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 the 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 data-flow 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?
- Does the flow of resources suggest opportunities for improving the process, for example by increasing parallelism or assigning tasks to different roles?

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 (Noll and Scacchi, 1999). Based on the aforementioned data-flow analysis techniques, the analysis tool allows a pro-

cess designer to quickly and automatically analyze a PML process model to aid in assessing its correctness, and for use in redesign.

We have successfully used our tool to analyze software process models and present an in-depth analysis of the redesign of one model, used by students at Santa Clara University for their senior projects and based loosely on Boehm's Anchoring Milestones (Boehm, 1996). Our tool detected sixty-three errors (of which two were incorrectly reported) in the model, which consisted of only 203 lines of PML code. Through iterative use of the tool, we were able to successfully redesign the model. The redesigned model consisted of 297 lines of PML code and had no legitimate errors, although the tool did report twelve errors incorrectly, indicating a possible direction for tool improvement.

We begin with a brief overview of PML, in order to provide a context for discussing our technique. Then, we present our analysis technique, and the implications for process engineering and re-engineering of the data it provides. Following that, we discuss the implementation of the analysis tool, and the results of its application to actual PML process specifications. We conclude with our assessment of the technique and its implementation, 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 (Noll and Scacchi, 1999). PML was designed specifically for rapid, incremental process capture, to support both process modeling and analysis, and process enactment (Scacchi and Noll, 1997). PML's design emphasizes simplicity, of both the language itself, and the models it specifies. As such, it is a true modeling language intended to capture the essential aspects of a process while abstracting away other details.

Using PML, a process can be specified initially at a very high level that contains only major process steps and control flow. Deployment can be incremental as well: high level specifications can be syntactically correct and thus enactable, yet may be missing significant detail. As understanding of process details emerges, the specification can be elaborated to capture them.

### 2.1 Overview

PML reflects the conceptual model of process enactment developed by Mi and Scacchi (1990). This model views a process as a situation in which agents use tools to perform tasks that require and produce *resources*, as shown in Figure 1. PML

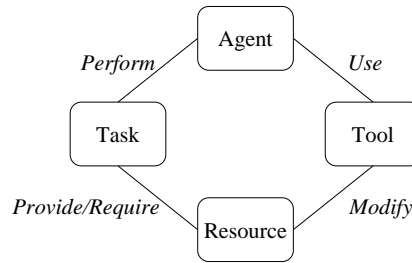


Fig. 1. The PML conceptual model.

models processes as collections of actions that represent atomic process tasks. An action declaration contains an identifier and a (possibly empty) set of additional fields, shown in Table 1. Since only the name of an action is required, PML supports the goal of incremental specification: a process can be quickly described as a set of named actions, validated by process performers, then elaborated over time as the details are understood. 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 {}
  action third {}
}
  
```

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

```

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

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

```

selection {
  action choice_1 {}
  action choice_2 {}
  action choice_3 {}
}
  
```

- **Branch**—A set of tasks that can be performed concurrently:

```

branch {
  action path_1 {}
  action path_2 {}
  action path_3 {}
}
  
```

Table 1  
Action fields (all are optional).

Field	Description
type	Either <i>manual</i> or <i>executable</i> . If an action is type <i>executable</i> , the runtime system will pass the actions script attribute to the underlying platform for execution; these actions, in effect, are performed automatically without human intervention. <i>Manual</i> actions are tasks to be performed by human actors.
agent	The role of the actor that should perform the action.
script	For executable actions, the script is an executable string to be run by the operating system; for manual actions, the script contains documentation about how to perform the task represented by the action.
tool	An application used by the actor to perform the task associated with the action.
requires	Resources required to perform the action. These can be considered the actions input.
provides	Resources produced as a results of performing the action; the actions output.

Figure 2 shows a PML program for the waterfall process model of software development.

## 2.2 Resource predicates and flow

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 (i.e., its preconditions), and what conditions will exist after an action is completed (i.e., its postconditions). 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, and the goal of a process as a whole, in terms of the products it produces. 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 }
```

```

process waterfall {
  action analysis {
    requires { problem }
    provides { analysis }
    agent { analyst }
  }
  branch {
    sequence {
      action design {
        requires { analysis }
        provides { design }
        agent { architect }
      }
      action implementation {
        requires { design }
        provides { code }
        agent { programmer }
      }
    }
    action write_tests {
      requires { analysis }
      provides { tests }
      agent { tester }
    }
  }
  action test {
    requires { code && tests }
    provides { code.status == "tested" }
    agent { tester }
  }
}

```

Fig. 2. Example PML program for the waterfall process model.

---

Here, *op* is any relational operator. Predicates may also be joined using conjunction and disjunction:

```

requires { resource1.attribute1 op value1 &&
          resource2.attribute2 op value2 ||
          resource3.attribute3 op value3 }

```

In short, resource predicates allow 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. At the extreme, an action that does not consume or produce any resources may be redundant and therefore a candidate for elimination.

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 inconsistencies

Resource inconsistencies can be classified into several situations:

- A resource is provided by an action that does not require any resources. This situation (termed a “miracle”) occurs when an action produces something without consuming any resources:

```
action miracle {
    provides { r }
}
```

It 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 }
}
```

- A resource is required by an action that does not provide any resources. This situation (termed a “black hole”) represents a process step that consumes resources, but does not produce any product:

```
action black_hole {
    requires { r }
}
```

This 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, but the actor presumably
       understands the problem now. */
}
```

- A resource is required, but a different resource is provided:

```

action transformation {
    requires { r }
    provides { s }
}

```

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 && design_rationale }
    provides { design_report }
}

```

- **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 }
}

```

- **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. Depending on the context surrounding a particular process, unprovided or unrequired resources represent points where a process consumes some input from the environment, or produces some output to the environment.

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

```

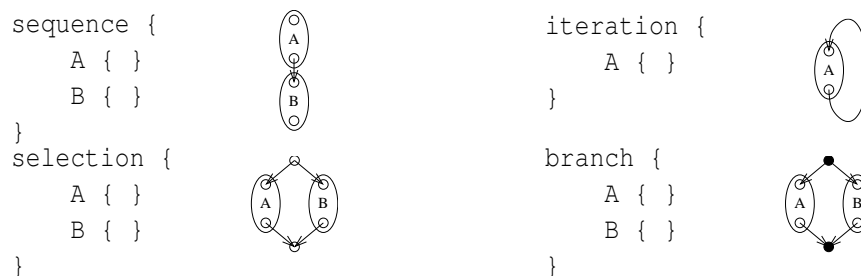
action a {
    provides { r.status == 1 }
}
action b {
    requires { r.status == 2 }
}

```

### 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.

To compute the flow of resources through a PML program, `pmlcheck` constructs a *process graph* similar to a control-flow graph in conventional languages. Each atomic *action* becomes a node in the graph. The graphs for other PML constructs are easily constructed in a syntax-directed, bottom-up 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. Note also that *iteration* is similar to a do-while statement since the body is executed at least once.

Three inconsistencies (miracles, black holes, and transformations) 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 3. The algorithm essentially performs a depth-first search of the process graph looking for a node that provides the required resource. The function  $\phi_N$  is a decision function that updates the status of a node given its current status and the status of a predecessor. Effectively,  $\phi_N$  performs a boolean `and` for a *selection* since a resource must be provided on all paths in order 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.

However,  $\phi_N$  can easily be extended to support operations on four truth values: **true**, **false**, **unknown**, and **maybe**. For example,  $\phi_N$  (**true**, **false**) on a *selection* statement results in **maybe** since the resource is provided along one path, but not along the other. However,  $\phi_N$  (**true**, **false**) on a *branch* statement results in **true** since it is sufficient that the resource is provided along one branch. This allows `pmlcheck` to distinguish between resources that are *possibly* unprovided and those that are *definitely* unprovided.

```

for all  $node \in nodes [graph]$  do
  for all  $resource \in required [node]$  do
    for all  $vertex \in nodes [graph]$  do
       $visited [vertex] := false$ 
       $status [vertex] := unknown$ 
    end for
     $check-if-provided (resource, node, node)$ 
  end for
end for

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

  if  $node \neq start$  and  $resource \in provided [node]$  then
     $status [node] := true$ 
     $necessary [node] [resource] := true$ 
  else
    for all  $predecessor \in predecessors [node]$  do
      if  $visited [N] = false$  then
         $check-if-provided (resource, predecessor, start)$ 
      end if
       $status [node] := \phi_N (status [node], status [predecessor])$ 
    end for
  end if
end function

```

Fig. 3. Basic algorithm used to check if a required resource is provided. The algorithm also records which provided resources are truly necessary.

---

Additional checks (not shown in Figure 3) ensure that the node providing the resource *dominate* (Lengauer and Tarjan, 1979) the node requiring the resource (to ensure that the providing action must always execute before the requiring action) and that the providing node does not execute in parallel with the requiring node. The latter check is not necessary in most traditional static program analysis tools since many programming languages do not inherently support concurrency. The algorithm also records those provided resources that were found during the search. This information is then used to determine which resources are provided, but are 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 language 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 have specifications that are incomplete or inconsistent, but that still provide useful input to downstream tools. 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 associated deliverables roughly based on Boehm's Anchoring Milestones (Boehm, 1996), as shown in Figure 4.

The experiment was performed in two iterations in order to capture, model, analyze, and refine the process specification in PML, as follows:

- (1) Initial capture, in which we simply translated the narrative specification into PML
- (2) Refinement, in which we used the analysis provided by the `pmlcheck` tool 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. This initial model is listed in Appendix A.

The tool reported sixty-three potential inconsistencies in this initial model (see Table 2). 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.

Table 2  
Detailed analysis of the errors reported for all models.

Model	Lines	Actions	Resources	Empty	Unprovided	Unrequired	Miracles	Black holes	Transformations
<b>Senior Design</b>									
senior_design.pml	203	35	69	1	3	16	28	15	36
senior_design2.pml	297	37	122	0	6	6	0	0	42
<b>Graduate S/W Development</b>									
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
<b>TOTAL</b>	<b>866</b>	<b>95</b>	<b>193</b>	<b>8</b>	<b>74</b>	<b>67</b>	<b>2</b>	<b>0</b>	<b>67</b>

Milestone	Deliverables
Conception Phase	
Problem Statement	Motivation Background Business Case User Scenarios Evaluation Criteria
Elaboration Phase	
Design	Project Plan Project Risks Use Cases Conceptual Model List of Requirements System Sequence Diagram / Flow Chart Architectural Diagram / Circuit Diagram Design Rationale Technologies Used Component State Charts Test Plan User Manual Test or Experimental Results
Design Review	System Prototype Collaboration Diagrams Class Diagram or Schematic Risk Resolution or Mitigation
Revised Design	Revised Design Items
Construction Phase	
Operational System	Source Code Build and Install Scripts Deployment Instructions Test Cases Test Results
Delivery Phase	
Final System	Final Source Code Final Test Cases Build and Install Scripts Deployment Instructions User Manual API Documentation Maintenance Guide Suggested Changes Experiences / Lessons Learned Complete Test Results

Fig. 4. Senior Design Process at Santa Clara University

- Spurious error—The tool correctly identified an error, but the error was triggered by another error. For example, if the model misspells the name of a provided resource, an action later in the model that requires the resource (with the correct spelling) will trigger an “unprovided resource” error.
- No error—The tool incorrectly reported an inconsistency.

Of the sixty-three 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; these represented output (the “system prototype” and “problem statement document” resources).

```
senior_design_2.pml:5: problem_statement in action 'create_project_motivation' is unprovided
senior_design_2.pml:9: problem_statement in action 'create_background' is unprovided
senior_design_2.pml:13: problem_statement in action 'create_business_case' is unprovided
senior_design_2.pml:17: problem_statement in action 'create_user_scenarios' is unprovided
senior_design_2.pml:21: problem_statement in action 'create_evaluation_criteria' is unprovided
senior_design_2.pml:26: problem_statement in action 'submit_problem_statement' is unprovided
senior_design_2.pml:87: design_report in action 'submit_design_report' is never required
senior_design_2.pml:113: design_review in action 'perform_design_review' is never required
senior_design_2.pml:210: operational_system in action 'demo_operational_system' is never required
senior_design_2.pml:226: conference_evaluation in action 'perform_presentation' is never required
senior_design_2.pml:254: project_report in action 'submit_project_report' is never required
senior_design_2.pml:284: final_system in action 'demo_final_system' is never required
```

Fig. 5. Output from `pmlcheck` for the revised senior design process.

The remaining fifty-six errors were the result of incorrectly modeling some aspect of the process, such as omitting a required or provided resource from an action (forty-two inconsistencies). These conclusions are summarized in Table 3.

Perhaps most interesting from a process engineering viewpoint, thirteen 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 this analysis of the initial version of the model to correct the modeling errors uncovered by `pmlcheck`. The revised specification is shown in Appendix B; the output from `pmlcheck` is shown in Figure 5. We again analyzed this model with `pmlcheck`; twelve inconsistencies were reported, none of which were errors, as they represented process input or output.

## 4.2 Graduate Software Processes

We also used `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 class-assignment documents and class lectures, augmented by the students’ personal experience in performing these processes. The analysis results are shown in Table 2.

## 4.3 Discussion

It appears from these experiments that the majority of inconsistencies reported by `pmlcheck` are unprovided or unrequired resources. This is the chief limitation of `pmlcheck`: 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 process outputs as unrequired resources.

Table 3

Summary of analysis results for the original and the revised senior design models.

Model	Total	Specification	Modeling	Process	Spurious	No Error
original	63	3	56	0	2	2
revised	12	0	0	0	0	12

Curiously, the Graduate Software Development processes contained only two miracles and no black holes among ninety-five actions; in contrast, the initial Senior Design model had twenty-eight miracles and fifteen 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 sixty-seven transformations in the Graduate Software Development processes; thirty-one 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 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` (Stallman, 1991) and static checkers such as `lint` (Johnson, 1978). Optimizing compilers typically warn the user regarding possibly uninitialized variables (variables whose values are used before being assigned). Our analysis tool informs the user regarding resources that are required without possibly being provided. As another example, register allocation (Chow and Hennessy, 1990), 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. Other common optimizations that follow the flow of variables through a program include common-subexpression elimination and code motion (moving loop-invariant code out of a loop) (Aho et al., 1986).

Algorithms for understanding and analyzing programs described as graphs are well-known (Aho et al., 1986; Cytron et al., 1991; Ferrante et al., 1987) as are algorithms for computing properties of the graphs themselves (Lengauer and Tarjan, 1979). Finally, other tools to aid the programmer in finding errors in programs include program slicing tools (Mock et al., 2002; Tip, 1995; Weiser, 1984) and assertion checkers (Jackson, 1991).

## 5.2 Process validation

Cook and Wolf (1999) 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 (1998) 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.

Woflan (van der Aalst, 1999) is a tool for process specification verification based on a set of process correctness measures derived from properties of Petri-nets. This approach first translates the process description into an equivalent Petri-net, which is then analyzed for properties of Petri-nets that imply process properties such as absence of deadlocks. While these properties ensure that the models are executable, it is not clear how other Petri-net properties relate to real world processes. In contrast, the inconsistencies that `pmlcheck` reports are derived from actual experience with industrial process models (Noll and Scacchi, 2001; Scacchi and Noll, 1997).

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 (Scacchi, 2000). 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* (Scacchi and Mi, 1997). Although PML and the *Articulator* share the same conceptual model of process activity, there are some 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 to process modeling 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 pro-

cess. 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*

The use of data-flow analysis for process re-engineering suggests several opportunities for future work.

### *6.1.1 Automatic re-engineering*

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 be 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 flow of control matches the resource flow.

### 6.1.2 Process decomposition

In certain situations, it is useful to decompose a process into independent *fragments*, that can be assigned to separate actors that may be part of independent, autonomous organizations Noll and Billinger (2002). In order to do this, we need to know where a process transitions from one agent to another, and what resources flow across this transition.

Resource flow analysis identifies resource dependencies among actions. A similar analysis approach that examines the *agent* field of actions could reveal actions that are performed by various actors, producing a set of sub-graphs, each of which is bound to a single actor. Resource flow analysis would then determine how these sub-graphs are connected, through shared resources.

#### A Initial senior design model

```
1  /* COEN Senior Design Process */
2  process senior_design {
3      sequence problem_statement {
4          action understand_problem {
5              provides { problem_statement }
6          }
7          action create_project_motivation {
8              requires { problem_statement }
9              provides { motivation }
10         }
11         action create_background {
12             requires { problem_statement }
13             provides { background }
14         }
15         action create_business_case {
16             requires { problem_statement }
17             provides { business_case }
18         }
19         action create_user_scenarios {
20             requires { problem_statement }
21             provides { scenario }
22         }
23         action create_evaluation_criteria {
24             requires { problem_statement }
25             provides { evaluation_criteria }
26         }
27         action submit_problem_statement {
28             requires { problem_statement && motivation && background &&
29                 business_case && scenario && evaluation_criteria }
```

```

30         provides { problem_statement_document }
31     }
32 }
33 sequence design_report {
34     action create_project_plan {
35         provides { project_plan }
36     }
37     action create_risk_analysis {
38         provides { risk_analysis }
39     }
40     action create_use_cases {
41         provides { use_cases }
42     }
43     action create_conceptual_model {
44         provides { conceptual_model }
45     }
46     action create_requirements_list {
47         provides { requirements }
48     }
49     action create_sequence_diagram {
50         provides { sequence_diagram }
51     }
52     action create_architecture {
53         provides { architecture }
54     }
55     action create_design_rationale {
56         provides { design_rationale }
57     }
58     action create_technologies_used {
59         provides { technologies_used }
60     }
61     action create_component_state_charts {
62         provides { state_chart }
63     }
64     action create_test_plan {
65         provides { test_plan }
66     }
67     action create_user_manual {
68         provides { user_manual }
69     }
70     action capture_test_results {
71         provides { test_results }
72     }
73 }
74 sequence design_review {
75     action create_system_prototype {
76         provides { system_prototype }

```

```

77     }
78     action create_collaboration_diagrams {
79         provides { collaboration_diagram }
80     }
81     action create_class_diagram {
82         provides { class_diagram }
83     }
84     action create_design_presentation {
85         provides { presentation }
86     }
87     action perform_design_review {
88     }
89 }
90 sequence revised_design_report {
91     action revise_risk_analysis {
92         requires { risk_analysis }
93     }
94     action revise_use_cases {
95         requires { use_cases }
96     }
97     action revise_conceptual_model {
98         requires { conceptual_model }
99     }
100    action revise_requirements_list {
101        requires { requirements }
102    }
103    action revise_sequence_diagram {
104        requires { sequence_diagram }
105    }
106    action revise_architecture {
107        requires { architecture }
108    }
109    action revise_design_rationale {
110        requires { design_rationale }
111    }
112    action revise_technologies_used {
113        requires { technologies_used }
114    }
115    action revise_component_state_charts {
116        requires { state_chart }
117    }
118    action revise_test_plan {
119        requires { test_plan }
120    }
121    action revise_user_manual {
122        requires { user_manual }
123    }

```

```

124     action revise_collaboration_diagrams {
125         requires { collaboration_diagrams }
126     }
127     action revise_class_diagram {
128         requires { class_diagram }
129     }
130     action revise_project_plan {
131         requires { project_plan }
132     }
133 }
134 sequence operational_system {
135     action create_implementation {
136         provides { code }
137     }
138     action create_build_scripts {
139         provides { build_script }
140     }
141     action create_deployment_instructions {
142         provides { deployment_instructions }
143     }
144     action create_test_cases {
145         provides { test_cases }
146     }
147     action capture_test_results {
148         provides { test_results }
149     }
150 }
151 sequence design_conference {
152     action revise_presentation {
153         requires { presentation }
154         provides { presentation }
155     }
156     action create_system_demo {
157         provides { system_demo }
158     }
159     action perform_presentation {
160         requires { presentation && system_demo }
161     }
162 }
163 sequence project_report {
164     action create_API_documentation {
165         provides { api_doc }
166     }
167     action create_maintenance_guide {
168         provides { maintenance_guide }
169     }
170     action create_suggested_changes {

```

```

171         provides { suggested_changes }
172     }
173     action create_lessons_learned {
174         provides { lessons_learned }
175     }
176     action capture_test_results {
177         requires { test_cases }
178         provides { test_results }
179     }
180 }
181 sequence final_implementation {
182     action revise_implementation {
183         requires { code }
184         provides { code }
185     }
186     action revise_build_scripts {
187         requires { build_script }
188         provides { build_script }
189     }
190     action revise_deployment_instructions {
191         requires { deployment_instructions }
192         provides { deployment_instructions }
193     }
194     action enhance_test_cases {
195         requires { test_cases }
196         provides { test_cases }
197     }
198     action capture_test_results {
199         requires { test_cases }
200         provides { test_results }
201     }
202 }
203 }

```

## B Revised senior design model

```

1 /* COEN Senior Design Process */
2 process senior_design {
3     sequence problem_statement {
4         branch {
5             action create_project_motivation {
6                 requires { problem_statement }
7                 provides { motivation }
8             }
9             action create_background {

```

```

10         requires { problem_statement }
11         provides { background }
12     }
13     action create_business_case {
14         requires { problem_statement }
15         provides { business_case }
16     }
17     action create_user_scenarios {
18         requires { problem_statement }
19         provides { scenario }
20     }
21     action create_evaluation_criteria {
22         requires { problem_statement }
23         provides { evaluation_criteria }
24     }
25 }
26 action submit_problem_statement {
27     requires { problem_statement && motivation && background &&
28         business_case && scenario && evaluation_criteria }
29     provides { problem_statement_document }
30 }
31 }
32 sequence design_report {
33     branch {
34         action create_project_plan {
35             requires { problem_statement_document }
36             provides { project_plan }
37         }
38         action create_risk_analysis {
39             requires { problem_statement_document }
40             provides { risk_analysis }
41         }
42         action create_use_cases {
43             requires { problem_statement_document }
44             provides { use_cases }
45         }
46         action create_conceptual_model {
47             requires { problem_statement_document }
48             provides { conceptual_model }
49         }
50         action create_requirements_list {
51             requires { problem_statement_document }
52             provides { requirements }
53         }
54         action create_sequence_diagram {
55             requires { problem_statement_document }
56             provides { sequence_diagram }

```

```

57     }
58     action create_architecture {
59         requires { problem_statement_document }
60         provides { architecture }
61     }
62     action create_design_rationale {
63         requires { problem_statement_document }
64         provides { design_rationale }
65     }
66     action create_technologies_used {
67         requires { problem_statement_document }
68         provides { technologies_used }
69     }
70     action create_component_state_charts {
71         requires { problem_statement_document }
72         provides { state_chart }
73     }
74     action create_test_plan {
75         requires { problem_statement_document }
76         provides { test_plan }
77     }
78     action create_user_manual {
79         requires { problem_statement_document }
80         provides { user_manual }
81     }
82     action capture_test_results {
83         requires { test_cases }
84         provides { test_results }
85     }
86 }
87 action submit_design_report {
88     requires { project_plan && risk_analysis && use_cases &&
89         conceptual_model && requirements && sequence_diagram &&
90         architecture && design_rationale && technologies_used &&
91         state_chart && test_plan && user_manual && test_results }
92     provides { design_report }
93 }
94 }
95 sequence design_review {
96     branch {
97         action create_system_prototype {
98             requires { use_cases && requirements }
99             provides { system_prototype }
100        }
101        action create_collaboration_diagrams {
102            requires { architecture }
103            provides { collaboration_diagram }

```

```

104         }
105     action create_class_diagram {
106         requires { requirements && use_cases && scenario }
107         provides { class_diagram }
108     }
109 }
110 action create_design_presentation {
111     requires { system_prototype && collaboration_diagram &&
112         class_diagram }
113     provides { presentation }
114 }
115 action perform_design_review {
116     requires { presentation }
117     provides { design_review }
118 }
119 }
120 sequence revised_design_report {
121     branch {
122         action revise_project_plan {
123             requires { project_plan }
124             provides { project_plan }
125         }
126         action revise_risk_analysis {
127             requires { risk_analysis }
128             provides { risk_analysis }
129         }
130         action revise_use_cases {
131             requires { use_cases }
132             provides { use_cases }
133         }
134         action revise_conceptual_model {
135             requires { conceptual_model }
136             provides { conceptual_model }
137         }
138         action revise_requirements_list {
139             requires { requirements }
140             provides { requirements }
141         }
142         action revise_sequence_diagram {
143             requires { sequence_diagram }
144             provides { sequence_diagram }
145         }
146         action revise_architecture {
147             requires { architecture }
148             provides { architecture }
149         }
150         action revise_design_rationale {

```

```

151         requires { design_rationale }
152         provides { design_rationale }
153     }
154     action revise_technologies_used {
155         requires { technologies_used }
156         provides { technologies_used }
157     }
158     action revise_component_state_charts {
159         requires { state_chart }
160         provides { state_chart }
161     }
162     action revise_test_plan {
163         requires { test_plan }
164         provides { test_plan }
165     }
166     action revise_user_manual {
167         requires { user_manual }
168         provides { user_manual }
169     }
170     action revise_collaboration_diagrams {
171         requires { collaboration_diagram }
172         provides { collaboration_diagram }
173     }
174     action revise_class_diagram {
175         requires { class_diagram }
176         provides { class_diagram }
177     }
178 }
179 action submit_revised_design_report {
180     requires { project_plan && risk_analysis && use_cases &&
181         conceptual_model && requirements && sequence_diagram &&
182         architecture && design_rationale && technologies_used &&
183         state_chart && test_plan && user_manual && test_results &&
184         collaboration_diagram && class_diagram }
185     provides { design_report }
186 }
187 }
188 sequence operational_system {
189     branch {
190         action create_implementation {
191             requires { design_report }
192             provides { code }
193         }
194         action create_build_scripts {
195             requires { code }
196             provides { build_script }
197         }

```

```

198         action create_deployment_instructions {
199             requires { code }
200             provides { deployment_instructions }
201         }
202         action create_test_cases {
203             requires { requirements && use_cases && user_manual }
204             provides { test_cases }
205         }
206         action capture_test_results {
207             requires { test_cases }
208             provides { test_results }
209         }
210     }
211     /* Following added after pmlcheck showed previous
212     provides statements never required. */
213     action demo_operational_system {
214         requires { code && build_script && deployment_instructions &&
215             test_cases && test_results }
216         provides { operational_system }
217     }
218 }
219 sequence design_conference {
220     branch {
221         action revise_presentation {
222             requires { presentation }
223             provides { presentation }
224         }
225         action create_system_demo {
226             requires { code && build_script }
227             provides { system_demo }
228         }
229     }
230     action perform_presentation {
231         requires { presentation && system_demo }
232         provides { conference_evaluation }
233     }
234 }
235 sequence project_report {
236     branch {
237         action create_API_documentation {
238             requires { architecture && code }
239             provides { api_doc }
240         }
241         action create_maintenance_guide {
242             requires { code }
243             provides { maintenance_guide }
244         }

```

```

245         action create_suggested_changes {
246             requires { code }
247             provides { suggested_changes }
248         }
249         action create_lessons_learned {
250             requires { code }
251             provides { lessons_learned }
252         }
253         action capture_test_results {
254             requires { test_cases }
255             provides { test_results }
256         }
257     }
258     action submit_project_report {
259         requires { api_doc && maintenance_guide &&
260             suggested_changes && lessons_learned
261             && test_results }
262         provides { project_report }
263     }
264 }
265 sequence final_implementation {
266     branch {
267         action revise_implementation {
268             requires { code }
269             provides { code }
270         }
271         action revise_build_scripts {
272             requires { build_script }
273             provides { build_script }
274         }
275         action revise_deployment_instructions {
276             requires { deployment_instructions }
277             provides { deployment_instructions }
278         }
279         action enhance_test_cases {
280             requires { test_cases }
281             provides { test_cases }
282         }
283         action capture_test_results {
284             requires { test_cases }
285             provides { test_results }
286         }
287     }
288     /* Following added after pmlcheck showed previous
289     provides statements never required. */
290     action demo_final_system {
291         requires { code && build_script && deployment_instructions &&

```

```

292         test_cases && test_results }
293     provides { final_system }
294 }
295
296 }
297 }

```

## References

- Aho, A. V., Sethi, R., Ullman, J. D., 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.
- Boehm, B. W., Jul. 1996. Anchoring the software process. *IEEE Softw.* 13 (4), 73–82.
- Chow, F. C., Hennessy, J. L., Oct. 1990. A priority-based coloring approach to register allocation. *ACM Trans. Prog. Lang. Syst.* 12 (4), 501–536.
- Cook, J. E., Wolf, A. L., Apr. 1999. Software process validation: Quantitatively measuring the correspondence of a process to a model. *ACM Trans. Softw. Eng. Meth.* 8 (2), 147–176.
- Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., Zadeck, F. K., Oct. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.* 13 (4), 451–490.
- Ferrante, J., Ottenstein, K. J., Warren, J. D., Jul. 1987. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Syst.* 9 (3), 319–349.
- Jackson, D., May 1991. ASPECT: An economical bug-detector. In: *Proceedings of the 13th International Conference on Software Engineering*. Austin, TX, pp. 13–22.
- Johnson, E. W., Brockman, J. B., Jan. 1998. Measurement and analysis of sequential design processes. *ACM Trans. Des. Autom. Electron. Syst.* 3 (1), 1–20.
- Johnson, S. C., Jul. 1978. *Lint: A C program checker*. Computing science technical report, AT&T Bell Laboratories, Murray Hill, NJ.
- Lengauer, T., Tarjan, R. E., Jul. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Prog. Lang. Syst.* 1 (1), 121–141.
- Mi, P., Scacchi, W., Sep. 1990. A knowledge-based environment for modeling and simulating software engineering processes. *ACM Trans. Knowl. Data Eng.* 2 (3), 283–289.
- Mock, M., Atkinson, D. C., Chambers, C., Eggers, S. J., Nov. 2002. Improving program slicing with dynamic points-to data. In: *Proceedings of the 10th ACM International Symposium on the Foundations of Software Engineering*. Charleston, SC, pp. 71–80.
- Noll, J., Billinger, B., Nov. 2002. Modeling coordination as resource flow: An object-based approach. In: *Proceedings of the 6th IASTED International Conference on Software Engineering and Applications*. Cambridge, MA.
- Noll, J., Scacchi, W., Feb. 1999. Supporting software development in virtual enterprises. *J. Digit. Inf.* 1 (4).

- Noll, J., Scacchi, W., Jan. 2001. Specifying process-oriented hypertext for organizational computing. *Journal of Network and Computer Applications* 24 (1), 39–61.
- Osterweil, L. J., Mar. 1987. Software processes are software too. In: *Proceedings of the 9th International Conference on Software Engineering*. Monterey, CA, pp. 2–13.
- Scacchi, W., Jun.–Sep. 2000. Understanding software process redesign using modeling, analysis and simulation. *Softw. Process. Improv. and Pract.* 5 (2–3), 183–195.
- Scacchi, W., Mi, P., Jun. 1997. Process life cycle engineering: A knowledge-based approach and environment. *Int. J. Intell. Syst. Account. Financ. Manage.* 6 (2), 83–107.
- Scacchi, W., Noll, J., Sep.–Oct. 1997. Process-driven intranets: Life-cycle support for process reengineering. *IEEE Internet Computing* 1 (5), 42–49.
- Stallman, R. M., 1991. *GNU Reference Manual*. Free Software Foundation, Cambridge, MA.
- Sutton, Jr., S. M., Heimbigner, D., Osterweil, L. J., Jul. 1995. APPL/A: A language for software process programming. *ACM Trans. Softw. Eng. Meth.* 4 (3), 221–286.
- Tip, F., Sep. 1995. A survey of program slicing techniques. *J. Prog. Lang.* 3 (3), 121–189.
- van der Aalst, W. M. P., 1999. Woflan: A petri-net-based workflow analyzer. *System Analysis - Modelling - Simulation* 34 (3), 345–357.
- Weiser, M., Jul. 1984. Program slicing. *IEEE Trans. Softw. Eng.* SE-10 (4), 352–357.