Tool support for iterative software process modeling

Darren C. Atkinson *, Daniel C. Weeks, John Noll

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

Received 13 January 2006; received in revised form 14 July 2006; accepted 24 July 2006
Available online 7 September 2006

Abstract

To formalize a process, its important aspects must be extracted and described in a model. This model is often written in a formal language so that the process itself can be automated. Since models are often developed iteratively, this language should support this iterative development cycle. However, many existing languages do not. In this paper, we use an existing high-level process modeling language and present a tool that we have developed for supporting iterative development. We have used our tool to develop and refine a process model of distributed software development for NetBeans.

2006 Elsevier B.V. All rights reserved.

Keywords: Process modeling; Model verification; Static analysis; PML

1. Introduction

Process descriptions [20] characterize the important aspects of processes from which models can be derived. One benefit of having a written notation for process description is the ability to analyze the process to check for errors. Validating a process before enactment increases quality and ensures correctness. In addition to finding problems in a process, modeling allows process designers to explore many different designs before enactment. Complex processes may be too costly to actually implement and refine. Modeling allows the modeler to easily modify the process and determine if the changes are effective. Finally, if the conceptual and procedural aspects of a process can be represented in a language, then tools can be designed to automatically check the models before enactment [16]. The ability to check processes before performing them allows errors to be caught before they are manifested in the performance of the process.

Since models are difficult to derive correctly from a real-world process in a single modeling step, process models are typically designed starting with abstract concepts and are iteratively refined into detailed descriptions. Therefore, the modeling language used to describe a process needs to reflect this iterative development cycle, but still provide valuable information about the process at every level of abstraction.

Many approaches to modeling processes exist, as shown in Fig. 1. The paradigm that we advocate for iterative model development is control-based [4]. In this approach, the control is specified by the modeler, which allows her to describe the flow of control in the process. This method can be used to model processes at various levels of abstraction [19]. At a high level of abstraction, the control is sequential, which allows the modeler to imply the dependencies without actually having to specify them. If it is later decided that the model should be more specific, the actual dependencies can be introduced.

The most common approach to designing a process modeling language is to build the language on top of an existing programming language. A typical example of this approach is the language APPL/A [29], which is designed as an extension to the programming language Ada. There are many advantages to using this bottom-up approach to language design, most of which pertain to enactability. APPL/A was able to take advantage of features such as concurrency, modularity, and exception handling that are
Resources are an essential component to creating a process model that does more than just reiterate the steps in a process. The ability to describe the flow of resources allows the modeler to create a variety of dependencies that occur within a process. The only postulate for an action is that the resource is available when the process enters or exits the action. PML allows actions to require and provide resources, which reflects the action’s need for or the production of a resource, but gives no indication of its origin or destination. Using these constructs, we can iteratively refine our model to provide more information about the internals of an action:

```plaintext
action analyze {
  requires { function & behavior & performance && interface }
  provides { analysis & analysis_documentation }
}
```

In most cases, resources alone are not enough to provide the detail needed for an accurate model. While many actions in a process may require a resource, there are specific qualities or characteristics of the resource that are essential and cannot be described by the resource’s name. We stated that the action `analyze`:

```plaintext
provides { documentation.analyze }
```

However, introducing a new resource to describe the fact that the analysis portion of the documentation is now complete complicates the process. Without being able to modify the properties of a resource, a new resource needs to be created to describe any change in the process. We can use attributes to solve this problem by describing the state of a resource and thus it would be more clear to state:

```plaintext
provides { documentation.analysis }
```

Unlike `documentation.analyze` which is an abstract resource created to describe the result of an action, documentation is a concrete resource that will persist throughout the process as new sections of the documentation are added. Attributes provide a means to describe changes to resources without having to create spurious resources.

Finally, attributes alone cannot always adequately describe specific qualities and states of resources or their properties. Actions often rely on attributes having specific values and as the model evolves and detail is added, constraining the state of resources and attributes provides more explicit control. By adding expressions the model transitions to another level of detail and can represent state:

```plaintext
provides { documentation.analysis == "complete" }
```

This statement is an assertion regarding the state of the attribute of a resource, and does not affect the value of the...
attribute. The enactment environment simply ensures that the attribute has the correct state when the action terminates. Such level of detail can be gradually added to further specify or constrain the model.

Although well-suited to iterative model development, PML does have disadvantages, the principal of which is the inability to ensure that a model is correct. For example, simply spelling the name of a resource incorrectly causes the model to be incorrect. Because of the dynamic nature of PML and its reliance on the enactment environment, such an error would not be caught until enactment where it could result in starvation or deadlock. In the following sections, we address the verification of process models written in PML.

2. Model verification

In iterative process model development, any errors are usually related to the many levels of abstraction that the model must pass through before arriving at a detailed representation of the process. The first level of abstraction in a model is a list of tasks that must be performed. However, at this level the errors that can be introduced by a modeler are simple and include problems such as syntax errors or typographical mistakes.

Transitioning to a lower level of abstraction incorporates adding resources to the model which begins the development of dependencies and may result in a considerable number of errors related to modeling. If the name of a resource is misspelled and another step in the model needs that resource, the dependency will be broken because the task was expecting the resource to have a different name. A modeler might also forget to state that a step has requirements or that it provides something. These types of errors manifest themselves as broken dependencies and extraneous steps in the model. Similarly, if a modeler fails to note what a step requires, but does note what it produces, then it appears that the step is creating some resource out of nothing. Though some steps in a process may only rely on abstract concepts or ideas that would not be properly represented by a requirement, this type of mistake is generally a problem that is introduced as an oversight. The same type of concern is raised when a step requires resources but a product for the task is not specified. to track what is produced and where it will be available.

Once a process model has been effectively implemented at a level of resource specification, it is possible to transition to a lower level of abstraction that will illustrate constraints on the state of objects within the process. This level of abstraction is the most detailed and also the most error prone. When transitioning to a detailed specification, the modeler must keep track of dependencies between the properties of resources as well as the resources themselves. The addition of properties to the model can disrupt the dependencies that were in place at higher levels of abstraction. For example, if the requirements for a step in the model are altered to include the state of a property, but the model fails to specify that the property was introduced by an earlier step, then the dependency between the two steps is broken.

The primary objective of a tool designed to analyze a process model is to examine the model for the types of errors mentioned. In this manner, the use of our tool, pmlcheck, is analogous to that of a compiler, as shown in Fig. 2. Compiler errors are most useful to the programmer, but may in fact be useful to a program designer as well. For example, if an error indicates that part of a required interface is unavailable, then that may be a signal to the designer that the component-level design is incorrect. Similarly, errors from pmlcheck are most useful to the modeler, but may also be useful to the domain expert who derived the model. Thus, we see that two feedback loops exist: a modeler-tool feedback loop that helps the modeler, similar to the edit-compile-debug loop, and the domain expert-tool feedback loop. One significant difference between a compiler and pmlcheck is that the errors from a compiler are fatal errors, but those from pmlcheck are merely informative. A process model that contains some errors can still be enacted because the actor can often provide the missing details. This ability does not exist for a compiled language.

We propose a number of requirements that a tool must meet and failing to meet these requirements is detrimental to the tool’s usefulness:

Fig. 2. The use of pmlcheck is analogous to that of a compiler.
• meaningful feedback: The tool should attempt to constructively map the errors in the model to conceptual errors in the real process.
• analysis refinement: The evolutionary nature of process modeling languages requires that supporting tools operate at each level of refinement in the development of the process model. If the analysis tool is reporting resource and dependency errors when the model is at a higher level of abstraction, then the analyzer has failed to meet the evolutionary requirements of the language.
• proper level of detail: An analysis tool that provides too much or too little detail about errors in the model is difficult to apply to development. If there is not enough detail, then it is difficult to discern what the problem is and how it affects the model. However, if the level of detail is too high, then users will be discouraged by the number of extraneous errors and warnings that appear when the tool is used. Therefore, it is important to provide the capability to target some aspect of the analysis while ignoring others in order to get the proper level of detail.
• ease of use: If the analysis tool is cryptic, slow, or difficult to use, then it will deter users from utilizing it to aid their model development.

3. Tool design and implementation

3.1. Tool design

One of the objectives of an automated analysis tool is the ability to check a model at many levels of abstraction, and therefore our tool, pmldcheck, currently provides four conceptual levels of checking:

1. syntax: This level of checking is provided as a means to ensure that the process model is well-formed, but does not provide any meaningful feedback.
2. resource specification: As a process model is refined from an abstract specification to include resources, the first check that should be performed is that the resources for each action in the process model have been specified.
3. resource dependencies: Introducing resources into the model creates dependencies between actions. At this level, the dependencies can be examined to determine if any were omitted or misrepresented.
4. expression satisfiability: Once resource dependencies have been established, the next step is to evaluate the properties of resources throughout the process model to determine if they are consistent.

Pmlcheck is not strictly limited to providing information at these levels of refinement and within each conceptual level there are a variety of checks that are performed and pmldcheck can focus analysis on a particular point of interest. This flexibility was intentionally designed to reflect the evolutionary nature of model development and the PML language while providing the modeler with control over information gathered by the tool.¹

3.2. Model representation

Our tool is designed to translate a process model into a format that incorporates all aspects of the model and based on the structure of processes, the most intuitive representation is a graph. The procedure that we use for mapping from a PML model to a graph is relatively simple; the nodes of a graph represent actions constructs and the edges represent the flow of control. The language constructs designed for describing control flow are interpreted and constructed into a graph in a syntax-directed, bottom-up manner as shown in Fig. 3. Each action node describes the resources that are used and produced through the provides and requires properties. A tree structure is used to describe resources and expressions.

When the PML model is translated to a graph, the resource trees are constructed directly from the model without any form of reduction. By doing this, resource trees match the exact representation given in the model, but this is not the most efficient format for performing analysis. Reducing the complexity of expressions decreases the possible cases for determining if expressions match. Pmlcheck performs two methods of reduction on every expression tree in the model to simplify analysis.

The first method of reduction is designed to reduce negations in expression trees. A complex expression that incorporates negation might look something like:

\[ !( (q) \ a.b != c.d ) \]

The tree constructed from this expression is represented in Fig. 4a. By applying DeMorgan’s Law and manipulating the operators in the expression, it is possible to effectively eliminate negations. The objective of this operation is to push the negation as far down in the expression tree as possible, thus alleviating the need to handle them at various levels in the tree structure. Reducing the original tree in Fig. 4a using this algorithm (shown in Fig. 5) results in the tree illustrated in Fig. 4b.

A second method of reduction is canonicalizing the expressions to simplify analysis. This reduction reduces the number of operators by reforming the trees and altering the existing operators, but without altering the meaning of the expression. The operators \( \geq \) and \( \geq \) are replaced with \( \leq \) and \( \leq \), respectively, after swapping the children of the expression tree node. This reduces the number of operators from six to four, which significantly reduces the number of comparison cases. By performing this method of canonicalization (shown in Fig. 6) on Fig. 4b, the tree provided in Fig. 4c results.

¹ The tool and the example models from Section 4 are available from http://www.cse.scu.edu/~atkinson/software.html.
Algorithm 1: \texttt{REDUCE-TREE}(node, negate)

\begin{algorithmic}
\If {operator[node] = \texttt{NOT} \Then}
\State \Return \texttt{REDUCE-TREE}(right[node], \neg negate)
\EndIf
\If {negate \Then}
\If {operator[node] = \texttt{OR} \Then}
\State operator[node] \leftarrow \texttt{AND}
\State left[node] \leftarrow \texttt{REDUCE-TREE}(left[node], \texttt{TRUE})
\State right[node] \leftarrow \texttt{REDUCE-TREE}(right[node], \texttt{TRUE})
\ElsIf {operator[node] = \texttt{AND} \Then}
\State operator[node] \leftarrow \texttt{OR}
\State left[node] \leftarrow \texttt{REDUCE-TREE}(left[node], \texttt{TRUE})
\State right[node] \leftarrow \texttt{REDUCE-TREE}(right[node], \texttt{TRUE})
\Else
\If {operator[node] = \texttt{EQ} \Then}
\State operator[node] \leftarrow \texttt{NE}
\ElsIf {operator[node] = \texttt{NE} \Then}
\State operator[node] \leftarrow \texttt{EQ}
\ElseIf {operator[node] = \texttt{LT} \Then}
\State operator[node] \leftarrow \texttt{GE}
\ElsIf {operator[node] = \texttt{GE} \Then}
\State operator[node] \leftarrow \texttt{LT}
\ElsIf {operator[node] = \texttt{GT} \Then}
\State operator[node] \leftarrow \texttt{LE}
\ElsIf {operator[node] = \texttt{LE} \Then}
\State operator[node] \leftarrow \texttt{GT}
\EndIf
\State left[node] \leftarrow \texttt{REDUCE-TREE}(left[node], \texttt{FALSE})
\State right[node] \leftarrow \texttt{REDUCE-TREE}(right[node], \texttt{FALSE})
\EndIf
\EndIf
\EndIf
\State \Return node
\end{algorithmic}
We noted that inconsistencies are introduced into a model because of a failure to specify requirements for a task. In PML, this translates to the failure to require or provide a resource in an action. These types of errors fall into four categories: those requiring and providing no resources (empty), those only requiring resources (black holes), those only providing resources (miracles), and those that provide resources other than those that they require (transformations).

Each of these scenarios is an indicator that something has been left out of the process model and is a projection of problems discussed previously. Because of the design and structure of the PML graph, these types of checks are almost trivial to implement. Pmlcheck accomplishes this by traversing the graph of the process examining each action node and based on the presence of resources, issues notifications to the modeler. The only exception to looking strictly at the availability of resources is when a resource has a qualifier. There are cases where a new resource is created and we want to explicitly state that it is not an error. Using qualifiers provides the ability to state that a transformation should occur. We provide a predefined qualifier, derived, that will suppress a warning in the case of a transformation, but this is only one of many uses for a qualifier. It is not possible to enumerate all possible qualifiers and how they should be handled, which is why we provide the person analyzing the process model the capability to implement their own qualifier checking through an abstraction. Though adding this type of additional checking is a more complicated feature of pmlcheck, it allows the analysis tool to match the flexibility of the language.

Let us construct two sets of resources for each action: those resources that are required, required[a], and those that are provided, provided[a]. Each resource has three pieces of information associated with it: name[r] refers to the name of a resource (i.e., the text to the left of any period), attribute[r] refers to the optional attribute of a resource (i.e., the text to the right of any period), and qualifiers[r] denotes the set of qualifiers of the resource. The four error types described previously can be easily checked using simple set logic as shown in Fig. 7. For example, the is-transformation algorithm checks if any provided resource is not required by the action and such a resource has not been explicitly qualified as being a derived resource (by checking if derived is a member of the resource’s qualifiers). The is-specified algorithm expects a set and a resource and returns a ternary value (one of full, partial, or none), rather than a simple binary value, in order to reflect partial specification of resources. Being partially specified indicates that a resource with the same name exists, but the attribute does not. For example, if a resource x.y is required but x or x.z is provided, then the resource is only partially specified.

3.4. Errors across actions

Tracing dependencies through a process model is much more complicated than simple specification checks. Control flow constructs and the level of specification of a resource play an important role in determining whether or not resources are available. Pmlcheck implements two types of resource-based dependency checks: assuring that resources required by an action are provided, and provided resources are required by an action. Another key component to understanding how pmlcheck performs these operations is how resources are treated within the model.

A resource has not only a level of specification, but also an availability within the process graph. If a resource is provided in one path of a selection statement, the resource is only possibly available later in the process. Therefore, pmlcheck reflects the status of a resource through two values: the level of specification and the availability of the resource. The availability of a resource can also be expressed as one of always, sometimes, and never. To simplify our algorithms, let us assume that the set of provided...
resources, provides\([a]\) has been partitioned into two sets, provides\(_a\) and provides\(_b\), that specify those resources that are always and conditionally provided, respectively.

In the absence of disjunctions, a resource is always available if it is partially or fully specified. Fig. 8 shows the algorithm for determining if a resource is produced by an action. For example, if a resource \(x\) is sought and an expression \(x \& \& y\) is encountered, then the resource is always available. However, if an expression \(x \| y\) is encountered, then the resource is only sometimes available, since an action can satisfy the assertion provided by the expression by providing \(y\) but not \(x\).

The first check performed is to ensure that resources that are required by an action are provided in an earlier action. Fig. 9 describes the basic procedure for determining if a resource is produced by an action. For example, if a resource \(x\) is sought and an expression \(x \& \& y\) is encountered, then the resource is always available. However, if an expression \(x \| y\) is encountered, then the resource is only sometimes available, since an action can satisfy the assertion provided by the expression by providing \(y\) but not \(x\).

The first check performed is to ensure that resources that are required by an action are provided in an earlier action. Fig. 9 describes the basic procedure for determining if a resource is produced by an action. For example, if a resource \(x\) is sought and an expression \(x \& \& y\) is encountered, then the resource is always available. However, if an expression \(x \| y\) is encountered, then the resource is only sometimes available, since an action can satisfy the assertion provided by the expression by providing \(y\) but not \(x\).

The second check is to ensure that a resource provided by a node is required by a node in the process follows a similar algorithm. This algorithm performs a search of the process graph following the control flow of the process looking for a node that requires the provided resources. The availability of the resource is computed in the same manner as the previous checks.

In many cases, the initial and final nodes in a process either make use of a resource that does not pre-exist in the process or produces an object that is really an output of the process and is therefore never used by another action. pmlcheck allows the suppression of warnings created by these inconsistencies by making use of a linking file. This allows the modeler to specify inputs and outputs of a process that are converted into resource trees and appended to a node representing the beginning of the process for inputs and a similar node for outputs at the end of the process. These nodes are available for the checking algorithm.
to determine if the resources they contain satisfy a require-
ment being checked.

Evaluating expressions follows a similar procedure for checking dependencies but is directed toward the satisfiabil-
ity of the expressions specified. It is rare to have all the information needed to properly assure that all expressions are fulfilled. Therefore, we provide expression checking as a best-effort check that may result in spurious warnings. However, that does not mean that a model cannot contain enough detail to fully express the state changes of a resource using expressions. A model with inconsistent expressions may perform correctly, but having the expressions properly defined increases the chances of proper completion. There are three possible results when checking for satisfiability: never satisfied, sometimes satisfied, and always satisfied. An expression that is never satisfied is a contradiction, such as comparing \( r.a = 0 \) and \( r.a = 1 \). Expressions where some values satisfy the expression, but others do not, are sometimes satisfied. For example, if \( r.a \geq 0 \) is required and \( r.a > 0 \) is provided by a previous action, then the required expression will always be satisfied. However, if \( r.a > 0 \) is required and \( r.a \geq 0 \) is provided by a previous action, then the required expression is only satisfied some-
times, since there exists a value (i.e., zero) that could be pro-
vided that would not satisfy the requirement.

4. Analysis and results

One of the primary goals of modeling a process is to acquire more information about inconsistencies in resource-
and performance. Previously, we examined the motivation for analyzing models and we determined the types of problems that exist in model and how they can be detected. The objective of this section is to illustrate how a model checker can be utilized to guide evolutionary refinement of a process model through verification.

Algorithm 9 FIND-UNPROVIDED-RESOURCES(\( g \))

```plaintext
for all \( n \in \text{nodes}(g) \) do
  for all \( r \in \text{requires}(n) \) do
    for all \( v \in \text{nodes}(g) \) do
      visited[v] ← FALSE
      availability[v] ← UNKNOWN
      specification[v] ← UNKNOWN
    end for
    CHECK-IF-PROVIDED\( (r, n, n) \)
  end for
end for
```

Algorithm 10 CHECK-IF-PROVIDED(\( rsrcc, node, start \))

```plaintext
visited[node] ← TRUE
if node ≠ start then
  availability[node] ← IS-PRODUCED(\( rsrcc, node \))
  specification[node] ← IS-SPECIFIED(\( rsrcc, requires[node] \))
end if
if availability[node] ≠ ALWAYS ∨ specification[node] ≠ FULL then
  for all pred ∈ predecessors[node] do
    if ¬visited(pred) then
      CHECK-IF-PROVIDED(\( rsrcc, pred, start \))
      availability[node] ← σ_node[availability[node], availability[pred]]
      specification[node] ← σ_node[specification[node], specification[pred]]
    end if
  end for
end if
```

Fig. 9. Algorithms for checking that required resources are provided.

500 D.C. Atkinson et al. / Information and Software Technology 49 (2007) 493–514

Fig. 10. Decision tables used when merging information for (a) the specification of a resource in a branch statement, and (b) a selection statement. The decision tables for computing the availability of a resource are similar.
4.1. The NetBeans requirements and release process

NetBeans is an IDE for Java developers based on an open source development model. The process used for requirements and release by the NetBeans team is different than a traditional software process because it is centered on distributed development. In open source projects such as this, the actual coding of the system is external to the requirements and release of the product and the software development process is not concerned with how the code is written because the authors develop in a variety of environments. The decentralized nature of open source development relies on the guidance of a defined process to provide cohesion to releases of the software. Therefore, the NetBeans team uses their process to define the requirements and perform tasks that relate to the release without prescribing how the actual coding should be performed.

The development process for NetBeans has two components: eliciting requirements and releasing the next version of the software. The first entails detailing what features should be included in the next version of the software and the second is based on establishing that the code is ready for release and generating a deliverable. Each stage is comprised of a series of tasks related to fulfilling the next stage in development. The NetBeans development process is not self-contained because it relies on the previous revision of the process to continue. Though many software projects are terminated when the product is finalized, a release for NetBeans signifies a specific level of achievement of the software, but development continues to proceed. The initial description of the requirements and release process for NetBeans is given in Appendix A. Fig. 11 provides a summary of the actions involved.

4.2. Refining the process model

Using the model from Jensen et al. [15], analysis consisted of two levels of refinement in order to capture inconsistencies at different levels of abstraction. On first inspection of the model it is clear that the model is in a very basic state in that it includes control and resources, but no attributes or expressions. Through verification using pmlcheck, we improved the quality and consistency of the model by removing errors without adversely affecting the underlying process.

4.2.1. Analysis local to actions

The first application of pmlcheck reveals a significant number of errors in the process and are shown in Table 1. Each of these errors is representative of a error in the model, the process, or the analysis tool. Empty actions,
miracles, and black holes generally indicate that resources are missing from the specification. For example, the action CompleteStabilization is the final action in the model, but it does not require anything and does not produce anything. However, this action is clearly included to finalize the product and make it available, but any information about what resources are required has been omitted. The action WaitForVolunteer also does not contain resources, but for a different reason. This action is an artificial action created to represent what the process is doing in preparation for the next action to take place. It is not essential for the process because the next action must be ready before the process can continue, so it can be removed without adversely affecting the rest of the model.

Miracles and black holes pose a problem similar to empty actions. Though actions such as ReviewNetBeans and SendMessageToCommunityForFeedback were initially specified as not providing anything, they do contribute to the process. ReviewNetBeans may not provide anything new, but it does affect a property of the road-map and should reflect those changes by providing a resource NetBeansRoadmap.Reviewed. In Appendix A, the action SendMessageToCommunityForFeedback would intuitively imply that feedback is gathered from the community and thus should provide CommunityFeedback as a resource. These type of oversights are a misrepresentation of the process, and the errors that pmlcheck provides helps to locate the root cause of these inconsistencies.

Pmlcheck reports that there are a significant number of transformations being performed in the process, but this report has two possibilities: the transformation is correct and the tool should not consider the created resource as an error, or the transformation is indicative of a change to a resource that was not specified as a requirement to the action. The only possible way to determine the actual meaning is to carefully inspect the process model. Action SetReleaseDate is an obvious example where the tool is improperly reporting an inconsistency because the release date is derived from the road-map. By qualifying the created resource as (derived) ReleaseDate, pmlcheck will understand that the resource is intended to be available at this point in the process. Action ReviseProposalBasedOnFeedback is an example of where a transformation is improper. This action is modifying two resources PotentialRevisionsToDevelopmentProposal and RevisedDevelopmentProposal, but these relate to a single resource: DevelopmentProposal. By consolidating these resources to a single resource and using attributes, we can reconstruct the action and describe it more accurately as:

```
action ReviseProposalBasedOnFeedback {
  requires { DevelopmentProposal.PotentialRevisions }
  provides { DevelopmentProposal.Revised }
}
```

Representing the task in this form removes unnecessary resources and clearly depicts how the resource is being affected.

Typically, a pmlcheck user will want to check errors local to an action first looking for errors across actions. In fact, a user may wish to look for the simplest types of errors first, namely, miracles, black holes, and empty actions. This type of pinpoint analysis is easily accomplished with pmlcheck. For example, a command to report only black holes would be simply pmlcheck -b netbeans.pml, which results in:

```
netbeans.pml:4: action ‘ReviewNetBeans’ provides no resources
netbeans.pml:15: action ‘ReviewNetBeans-VisionStatement’ provides no resources
netbeans.pml:38: action ‘SendMessageToCommunityForFeedback’ provides no resources
netbeans.pml:56: action ‘AssignDeveloperToCompleteProjectMilestones’ provides no resources
netbeans.pml:124: action ‘UploadInstallTarFilesToWebRepository’ provides no resources
netbeans.pml:171: action ‘ExamineTestReport’ provides no resources
```

The full set of options to pmlcheck is given in Table 2. Allowing users to concentrate on the simple types of errors first gives us the proper level of detail we desire and allows the tool user to refine the analysis. Once the simple errors are fixed, then the user will likely enable the reports the check for the consistent usage of dependencies across actions.

<table>
<thead>
<tr>
<th>Option</th>
<th>Report</th>
</tr>
</thead>
<tbody>
<tr>
<td>-s</td>
<td>syntax only</td>
</tr>
<tr>
<td>-p</td>
<td>provided resources that are never required</td>
</tr>
<tr>
<td>-r</td>
<td>required resources that are never provided</td>
</tr>
<tr>
<td>-e</td>
<td>actions that neither require nor provide resources (empty actions)</td>
</tr>
<tr>
<td>-m</td>
<td>actions that provide but do not require resources (miracles)</td>
</tr>
<tr>
<td>-b</td>
<td>actions that require but do not provide resources (black holes)</td>
</tr>
<tr>
<td>-t</td>
<td>actions that provide different resources than those required (transformations)</td>
</tr>
<tr>
<td>-x</td>
<td>unsatisfiable expressions</td>
</tr>
</tbody>
</table>
4.2.2. Verification of resource dependencies

Pmlcheck reports that there are a number of resources that do not exist before they are used or are created but never used. This type of error can be caused by a number of problems with the model specification. One possibility is that the modeler overlooked the creation of a resource. In this model, the action ReviewFeedbackFromCommunity requires FeedbackMessagesOnMail, but this resource does not exist prior to this point in the process. Action SendMessageToCommunityForFeedback seems to indicate the presence of feedback, so it would stand to reason that feedback would be created, but it has not been specified. By redefining this action to provide feedback when the messages are sent to the community, the model will more accurately represent the process and eliminate the dependency error.

Though a report of an unprovided resource can mean a misrepresentation of process, it can also be indicative of a resource that should preexist the process. Action ReviewNetBeans requires the NetBeansRoadmap, but this is the first action in the process which means the resource cannot be specified prior to its use. To identify resources that should be considered inputs to the process, pmlcheck allows the user to specify such resources in a linking file. Outputs to the process can be specified in the linking file as well.

Pmlcheck also reports resources that are provided by an action but are not used later in the process, as shown in Fig. 12. One possible cause for this error is that a task later in the process has been misspecified and does not note that it requires a certain resource. For example, action ReportIssuesToIssuezilla provides IssuezillaEntry, but this resource is never used in the process. The following action looks at standing issues, but does not explicitly require this resource. As with unprovided resources, it is also possible that the analysis tool is catching errors for resources that are intended to be outputs to the process, such as a release notice. In the same way that an input can be specified, the linking file will notify pmlcheck of resources that are intended to be outputs of the process. Identifying these types of resources will prevent the tool from improperly reporting errors.

4.2.3. Consolidating resources

We discussed how changing a resource is often accompanied by the creation of new, but only slightly different resources. By identifying the common resource and applying attributes to indicate the changes, it is possible to construct a more cohesive model of the process. For example, actions ReviewFeedbackFromCommunity, ReviseProposalBasedOnFeedback, PostFinalDevelopmentProposalToNetBeansWebsite, and AssignDevelopersToCompleteProjectMilestones, all rely on some variation of the development proposal and by extracting small changes and combining them into a single resource, the model becomes more intuitive. In addition to clarifying the model, this change brings forth a more critical problem: nowhere in the specification of the process is the development proposal created. The first indication of a development proposal is in action ReviewFeedbackFromCommunity which provides PotentialRevisionsToDevelopmentProposal, but prior to this action there is no development proposal, so it is difficult to discuss potential revisions to a nonexistent proposal.

4.3. Revised models

After applying the types of changes described in the previous section along with some cosmetic changes of names throughout the process, we arrive at a revised model (not shown for brevity). Applying pmlcheck a second time reveals that the number of errors is much lower than in the original model and the results are shown in Table 1. Examining these errors reveals that many were the result of changes made to the process including trivial errors resulting from case-sensitivity and misspellings.

Other errors consist of overlooking inputs and outputs to the process. Resources such as the Changelog and InstallationNotes were not included in the outputs and are still indicated as errors. Identifying what is considered an input...
or output can be difficult because the resource may be required in a later step but was simply overlooked. Though marking it as an output will suppress the warning message from pmlcheck, it is important to ensure that the error is not indicative of another type of problem. For example, resource IssuezillaEntry in ReportIssuesToIssuezilla might be considered an output, but it is being used by the following task. Marking it as an output would prevent the error message, but does not solve the problem. Restructuring the resource to be handled as an attribute to the Issuezilla Issue Repository would eliminate the error without creating extraneous outputs to the process.

Correcting the mistakes found in the previous revision results in a model that produces no errors from the tool, which ensures that the tool is satisfied with the way dependencies are built. Though this does not indicate that there are no problems in the process, the problems that have indicators have been effectively removed. The final model is shown in Appendix B.

4.4. An example from another domain

PML is capable of expressing process models for non-software processes as well. We used an existing model [25] based on the Dutch Motor Insurance Liability Act for settling motor claims and translated it into PML. Since the original model is described graphically, we first captured the control flow as a PML model with empty actions. We then used pmlcheck to verify that the model had no syntax errors.

We then proceeded to add resources to the actions. In the original model, resources could be marked as mandatory, created, changed, or deleted. In PML, all resources required by an action are mandatory, so we used an qualifier (optional) to indicate optional resources. Resources that were considered to be provided by an action in the resulting PML model (Appendix C). Interestingly, in the model we considered no resources were marked changed. The overall results are shown in Table 3.

We first concentrated on the empty actions to determine if we missed recording any resources, but that was not the case. However, actions such as File should require some resource on which to operate, even if they do not provide any tangible output to the process. We also wanted to eliminate the qualifier that we had introduced. Upon closer examination, we discovered that most resources were mandatory and only a few were (somewhat) consistently optional. We determined that the reason for many of the optional resources was a limitation in the original modeling language: the inability to express disjunctions. For example, action Assign_Handler requires two optional resources CNF (Claim Notification Form) and Telephone_Notification_Form, when in fact it should require one or the other, depending upon whether the claim was received by mail or telephone. We also discovered that the action Determine_Mail required an optional resource SMC, which is the name of the process and therefore most likely an error.

When examining unprovided and unrequired resources (Fig. 13), we noticed that the model discusses the claim at two levels of detail. Earlier actions refer to or create the various forms (Statement_of_Liability, CNF, and Telephone_Notification_Form). Later actions typically refer to individual items on those forms such as Accident_Date or NAC_of_Third_Party. It is this separation that results in many of the errors. One solution to this problem would be make the items be attributes of the various forms. However, this solution would require that either later actions simply require the forms, which would result in a loss of information in the model, or require the individual attributes from each of the possible forms, which seems redundant. We decided instead to have the earlier actions provide the individual items as derived resources.

After determining the inputs and outputs to the process, the final results are shown in Table 3. Unlike the earlier model for NetBeans, this model has many selection statements, and therefore many of the errors involve resources that are possibly unrequired or unprovided. Ignoring the possibly unrequired or unprovided resources reduces the error counts to four and two, respectively. In the final model, given in Appendix D, we have added some new resources to represent states such as Damage_Known that were listed in the original model. In doing so, we have also introduced one blocked action that pmlcheck correctly detected:

\[
\text{smc.pml:146: Damage_Known in action 'Specify_Payment' is blocked}
\]

We cannot pay the claim until the damage is known; however, if no blame was found, then the process would reject the third party’s claim and the process would not need to wait to determine the damage. This situation was explicitly described in the original model, so we left our model as is.

4.5. Experience

We specified several criteria at the conclusion of Section 2 that a modeling analysis tool must meet in order to be effective. We now summarize our experience of how well pmlcheck met those criteria:

<table>
<thead>
<tr>
<th>Model</th>
<th>Empty Miracle Black Transformation Unrequired Unprovided</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>7 5 12 9 5 30</td>
</tr>
<tr>
<td>Revised</td>
<td>4 0 9 0 9 7</td>
</tr>
</tbody>
</table>

2 All errors should first be interpreted as errors in our translation and understanding of the original model, rather than as errors in the original model itself.
• **meaningful feedback:** For this goal, the tool was less successful than we hoped. We were unable to map many errors in the model to errors in the real process. We attribute this in large part to the fact that we were unfamiliar with actual process. We anticipate that if a domain expert who was familiar with the process were to use the tool, then the tool would be more useful.

• **analysis refinement:** We believe the tool was quite successful in meeting this goal. We were able to iteratively develop two models and enabled certain checks as necessary. For example, it was beneficial in the process model for claims settlement that we could check the syntax first and then check for simple errors such as empty errors before moving on to the global checks.

• **proper level of detail:** While the tool was able to focus on some aspect of the analysis while ignoring others, such as looking for transformations only, the level of detail provided by the tool was too detailed in some areas and not detailed enough in others. In particular, we found that in many cases we would have liked the tool output to be orthogonal with regard to actions and resources. Rather than reporting for each action or line if a resource caused an error, it would have been more useful if the tool had reported for each resource the actions involved. Since the models were evolved often by changing the name of a resource or using an attribute instead of a resource, finding all actions involving that resource would have been quite beneficial. Finally, although possibly unprovided resources should be normally flagged as such, it would have been helpful if possibly unrequired resources could be ignored. If a provided resource is used anywhere downstream then it is required. As an analogy, a compiler indicating all declared variables in a function that are only possibly used in that function would be very annoying to use.

• **ease of use:** We found the tool quite easy to use. However, during the course of using the tool, we decided to add an option for checking syntax only, since the default action if no checks were specified was to enable all checks. Finally, all of the models were analyzed within about a second.

5. Related work

5.1. Modeling languages

PML is first discussed in Noll and Scacchi [24] and is based on the conceptual model from Mi and Scacchi [23]. Atkinson et al. [3] discuss using the language as an evolutionary process modeling language and presents a limited case study using pmld. Whereas the previous work motivates using PML for evolutionary or iterative model development, this work discusses the tool infrastructure, its algorithms, and presents a more detailed analysis of the requirement and release model for NetBeans.

APPL/A [29] is a process enactment language designed as a superset of the Ada programming language to maximize automation. Many features of Ada including concurrency, abstraction, and encapsulation are inherited by the language to facilitate related concepts in process modeling. Features specific to modeling that are not implemented in the underlying programming language, such as...
as triggers and relations, are constructed as extensions to the language. In addition, existing tool support provides the language with error checking and the ability to be compiled and executed. APPL/A is designed for automation and execution as opposed to other languages that focus on analysis and design.

5.2. Program analysis

Process modeling is conceptually very different from programming but there are many similarities between analysis of a process model and of a software program. Tools such as gcc [28] and lint [18] warn users about inconsistencies within programs such as uninitialized variables (variables used without being assigned a value). Pmlcheck performs similar analysis such as issuing warnings if resources are required before being provided. As another example, gcc and lint provide warnings for variables that are declared but never used. Our tool provides the same conceptual level of checking applied to process modeling by ensuring that resources that are provided are used at some point in the process. Register allocation [5], 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) [1].

Algorithms for understanding and analyzing programs described as graphs are well-known [1,9,11] as are algorithms for computing properties of the graphs themselves [21]. Finally, other tools to aid the programmer in finding errors in programs include program slicing tools [30,32] and assertion checkers [14].

McAdam [22] mentions how too many low-level error messages negatively impact the adoption of high-level functional programming languages. Hristova et al. [13] discuss a tool to help students understand cryptic compiler error messages. The former focuses on type errors and latter focuses on syntax errors, neither of which is especially relevant to PML given its lack of typing and simple syntax. Rather, the errors that pmlcheck reports are semantic errors, many of which require data-flow analysis to detect.

5.3. Process validation

Cook and Wolf [8] 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 lifecycle 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 [17] 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 [31] 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 [24]. Similarly, Gruhn [12] introduces a modeling language called FUNSOFT that has its semantics defined in terms of Petri-nets and discusses an analysis tool, ANAMEL, that also uses well-known Petri-net algorithms.

Cobleigh et al. [6] also apply program analysis techniques to process programs. They describe how the FLAVERS [10,7] finite state analysis tool could be applied to analyze processes described in the Little-JIL [33] process programming language. By transforming a Little-JIL process specification into a finite state machine describing the process control flow graph, FLAVERS verifies properties of process execution that can be expressed as finite state automata. However, while FLAVERS can verify a variety of user-specified properties, it does not address resource production/consumption; thus this technique could be viewed as complementary to pmlcheck. It is also interesting to note that finite state analysis is resource intensive, requiring tens or hundreds of seconds on common PC-class platforms to analyze fairly simple models. In contrast, pmlcheck analyzes our most complex models in less than one second on similar platforms.

Scacchi’s [26] 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. 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 [27]. 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 language research.

6. Conclusion

We have presented a use of PML for iterative process model development. Starting with a high-level abstract description of a process, details are gradually added. To provide support for PML, we chose to implement a new method of process checking based on our research into process structure. Our tool, pmlcheck, examines process
models looking for common errors that result from process development and design. By noting inconsistencies in the process, it is possible for modelers to refine a process model until it properly represents the process. The flexibility of the language and the tool allow for specification and verification at many levels of abstraction. Using a general approach to process modeling and analysis allows for the concepts presented in this paper to be applied to a variety of modeling languages and analysis tools. Though our language and analysis tool provide an implementation of our design philosophy, the process-related concepts discussed in this paper supersede the implementation.

The model of the requirement and release process for NetBeans that we examined and refined illustrates many benefits of tool guided analysis. Understanding the resource flow of a process provides useful information to improve the specification of a process and to detail areas of ambiguity. Examining the interaction of resources in the process can also improve the enactability of a model by ensuring that resource flow is consistent throughout the process. Tools such as pmlcheck provide a necessary function in the design and development of processes models regardless of the domain.

Appendix A. Initial requirements and release model for NetBeans

A Initial requirements and release model for NetBeans

```java
process RequirementsAndRelease {
  sequence Requirements {
    sequence SelfProjectTasks {
      action ReviewTasks {
        requires (NotReadyToBegin) 
        /provides ( )
      }
      action SelfReleaseDate {
        requires (NotReadyToBegin) 
        provides (ReleaseDate)
      }
      sequence DetermineProject {
        branch SunONEStudioDevelopmentMeeting {
          action ReviewReleaseDate {
            requires (NotReadyToBegin) 
            /provides ( )
          }
          action ReviewUncompletedMilestonesFromPreviousRelease {
            requires (PreviousVersionCompletedMilestones) 
            /provides (ProspectsForUpcomingRelease)
          }
          action ReviewRelease {
            requires (IssuedReleaseToRepository) 
            /provides (ProspectsForUpcomingRelease)
          }
          iteration EstablishReleaseSet {
            action ComputersinOFPClientsFeaturesToInclude {
              requires (ProspectsForUpcomingRelease) 
              /provides (FeaturesForUpcomingRelease)
            }
            action CategoryOfFeaturesProposal {
              requires (FeaturesForUpcomingRelease) 
              /provides (WeightedListOfFeaturesToImplement)
            }
            action SendMessageToCommunity {
              requires (WeightedListOfFeaturesToImplement) 
              /provides ( )
            }
            sequence ReviewFeedbackFromCommunity {
              requires (FeedbackMessagesOnMail) 
              /provides (PotentialRevisionsToDevelopmentProposal)
            }
            sequence ReviewProposalBasedOnFeedback {
              requires (PotentialRevisionsToDevelopmentProposal) 
              /provides (RevisedDevelopmentProposal)
            }
          }
          iteration CollectCandidateNominations {
            action EstablishReleaseManager {
              requires (ReleaseManagerRequest) 
              provides (ReleaseManagerCandidateAnnouncement)
            }
            action EstablishReleaseManagerConsensus {
              requires (ReleaseManagerCandidateAnnouncements) 
              provides (NewBranchForCurrentBuild)
            }
            action AnnounceRelease {
              requires (ReleaseManagerAnnouncement) 
              provides (ReleaseManagerAnnouncementToUsers)
            }
            iteration MakeInstallTar {
              action AddFileToWebRepository {
                requires (BinaryReleaseDownloads) 
                /provides ( )
              }
              sequence Deploy {
                action UpdateWebPage {
                  requires (UpdatedWebPage) 
                  /provides ( )
                }
                action MakeInstallTar {
                  requires (ChangeFromIndividualModules) 
                  /provides ( )
                }
              }
            }
          }
        }
      }
    }
  }
}
```
process RequirementsAndRelease {
    sequence Requirements {
        sequence SetProjectTimeline {
            action ReviewRoadmap {
                requires { Roadmap }
                provides { Roadmap, Reviewed }
            }
        }
        action SetReleaseDate {
            requires { Roadmap }
            provides { (derived) ReleaseDate }
        }
        sequence DetermineProject {
            branch SunONEStudioDevelopmentMeeting {
                action ReviewNetBeansVisionStatement {
                    requires { VisionStatement }
                    provides { VisionStatement, Reviewed }
                }
            }
        }
    }
}
Appendix B. Final revision of the model for NetBeans

```java
action ReviewUncompletedMilestonesFromPreviousRelease {
    requires { Documentation, PreviousVersionRelease }
    provides { (derived) ProspectiveFeatures, PreviousVersions = "complete" }
}

action ReviewIssuesAndFeatureRequests {
    requires { IssueListRepository, IssueZilla = "complete" }
    provides { (derived) ProspectiveFeatures, IssueZilla = "complete" }
}

Iteration EstablishFeatureSet {
    action CompleteListOfPossibleFeaturesToInclude {
        requires { ProspectiveFeatures, PreviousVersion }
        provides { (derived) ReleaseFeaturesSet }
    }
    action CategorizeFeaturesProposedForFeatureSet {
        requires { ReleaseFeaturesSet }
        provides { ReleaseFeaturesSet, weighted }
    }
    action CreateDevelopmentProposal {
        requires { ReleaseFeaturesSet, weighted }
        provides { (derived) DevelopmentProposal }
    }
    action SendMessageToCommunityForFeedback {
        requires { ReleaseFeaturesSet, DevelopmentProposal, CommunitySite }
        provides { (derived) CommunityFeedback }
    }
    action ReviewFeedbackFromCommunity {
        requires { CommunityFeedback, DevelopmentProposal }
        provides { DevelopmentProposal, PotentialRevisions }
    }
    action RefineProposalBasedOnFeedback {
        requires { DevelopmentProposal, PotentialRevisions }
        provides { DevelopmentProposal, Revised }
    }
    action PostFinalDevelopmentProposalInNetBeansWebsite {
        requires { DevelopmentProposal, Revised }
        provides { DevelopmentProposal, Finalized }
    }
    action AssignDeveloperToCompleteMilestone {
        requires { DevelopmentProposal, Finalized }
        provides { (derived) DeveloperAssignments }
    }
}

sequence SetReleaseStageCompletionDates {
    action SetFeatureFreezeDate {
        requires { ReleaseDate }
        provides { (derived) FeatureFreezeDate }
    }
    action SetMilestoneCompletionDates {
        requires { FeatureFreezeDate, ReleaseDate }
        provides { (derived) MilestoneCompletionDates }
    }
}

sequence EstablishReleaseManager {
    action EmailSolicitationForReleaseManager {
        requires { CommunitySiteList }
        provides { (derived) ReleaseManagerRequest }
    }
}

Iteration CollectCandidateNominees {
    action SendCandidateNomination {
        requires { ReleaseManagerRequest }
        provides { (derived) ReleaseManagerCandidateNomination }
    }
    action EstablishReleaseManagerConsensus {
        requires { ReleaseManagerCandidateNomination }
        provides { (derived) ReleaseManagerDecision }
    }
    action AnnounceNewReleaseManager {
        requires { ReleaseManagerDecision }
        provides { (derived) ReleaseManagerAnnouncementToNetBeansMailingList }
    }
}

action SolicitModuleMaintainersForInclusionInUpcomingRelease {
    requires { FeatureFreezeDate }
    provides { (derived) ModuleInclusionNoticeToNetBeansMailingList }
}
```
sequence Release {  
iteration Stabilization {  
sequence Build {  
action ChangeBuildBranchName {  
requires { CurrentCodeRepository }  
provides { (derived) BuildBranch }  
}  
}  
iteration MakeInstallTar {  
action MakeInstallTarForEachPlatform {  
requires { BuildBranch }  
provides { (derived) InstallExecutableTar }  
}  
}  
}  
}  
sequence Deploy {  
action BuildBinaryReleases {  
requires { BuildBranch & InstallExecutableTar }  
provides { (derived) ReleaseBinaries }  
}  
action UploadInstallTarFilesToWebRepository {  
requires { ReleaseBinaries & WebRepository }  
provides { ReleaseBinariesUploaded }  
}  
action UpdateWebPage {  
requires { Webpage }  
provides { WebpageUpdated }  
}  
}  
action MakeReadmesInstallNotesAndChangelog {  
requires { Modules, Changes }  
provides { (derived) README & (derived) InstallationNotes & (derived) Changelog }  
}  
action SendReleaseNotificationToCommunityExtendingTheCommunityToDownloadAndTest {  
requires { CommunityMail\ingList }  
provides { (derived) ReleaseNotice }  
}  
}  
sequence Test {  
action CreateTestScripts {  
requires { BuildBranch & ReleaseFeatureSet }  
provides { (derived) TestScripts }  
}  
}  
action ExecuteAutomaticTestScripts {  
requires { TestScripts & ReleaseBinaries }  
provides { (derived) TestResults, ScriptResults }  
}  
}  
action ExecuteManualTestScripts {  
requires { TestScripts & ReleaseBinaries }  
provides { (derived) TestResults, ManualResults }  
}  
iteration UpdateIssuezilla {  
action ReportIssueToIssuezilla {  
requires { TestResults, ScrollResults & TestResults, ManuscriptResults }  
provides { (derived) IssuezillaIssueRepository, Manuscript }  
}  
action UpdateManuscriptIssuezilla {  
requires { IssuezillaIssueRepository, Manuscript & TestResults }  
provides { IssuezillaIssueRepositoryUpdated }  
}  
}  
action PostBugStatus {  
requires { TestResults }  
provides { (derived) BugStatusReport & (derived) TestReport }  
}  
}  
sequence Debug {  
action ExamineTestReport {  
requires { TestReport & BugStatusReport }  
provides { TestReportExamined }  
}  
action WriteBugFix {  
requires { BuildBranch }  
provides { (derived) PotentialBugFix }  
}  
action VerifyBugFix {  
requires { PotentialBugFix }  
provides { (derived) WorkingBugFix }  
}  
action CommitCodeToSourceRepository {  
requires { WorkingBugFix & SourceCodeRepository }  
provides { SourceCodeRepositoryUpdated }  
}  
action UpdateIssuezillaToTestChanges {  
requires { IssuezillaIssueRepository }  
provides { IssuezillaIssueRepositoryUpdated }  
}  
}  
iteration CompleteStabilization {  
requires { ReleaseBinaries & TestReport & Webpage }  
provides { (derived) FinalRelease & WebpageUpdated }  
}  
}  

Appendix C. Initial process model for claims settlement

C Initial process model for claims settlement

```plaintext
process SNC {
  selection {
    action Record Mail {
      requires { (optional) OCM }
      requires { (optional) Statement of Liability }
      requires { Notification Date }
    }
    action Record Telephone {
      requires { Accident Date }
      requires { NAC of Policyholder }
      requires { Car Details }
      provides { Telephone Notification Form }
    }
  }
  sequence Process Notification {
    action Assign Header {
      requires { (optional) Telephone Notification Form }
      requires { (optional) OCM }
    }
    action Create Folder {
      requires { Claim Number }
      provides { Claim Number }
    }
    action Check Cover {
      requires { OCM }
      requires { Telephone Notification Form }
    }
    selection {
      sequence Enter In Claims System {
        requires { Claim Number }
        requires { Notification Date }
        requires { Accident Date }
        requires { NAC of Third Party }
        requires { NAC of Policyholder }
        requires { Claim Amount }
        requires { Confirmation }
      }
      action Check Entered Data {
        requires { Claim Number }
        requires { Notification Date }
        requires { Accident Date }
        requires { NAC of Third Party }
        requires { NAC of Policyholder }
        requires { Claim Amount }
        requires { Confirmation }
      }
      sequence Reject Cover {
        iteration {
          action Prepare Rejection Letter {
            provides { Rejection }
          }
          action Assess and Sign Letter {
            requires { Rejection }
          }
          action Send {
            requires { Rejection }
            provides { }
          }
          action Files {
            provides { }
          }
        }
      }
      branch Process Third Party {
        sequence {
          action Record Data {
            requires { Claim Amount }
            requires { (optional) NAC of Third Party }
            requires { (optional) NAC of Policyholder }
            requires { Third Party Confirmation Letter }
          }
          selection {
            sequence Perform Expert Assessment {
              iteration {
                action Call In Expert {
                  provides { Expert Assessment }
                }
                iteration {
                  action Remind Expert {
                    requires { }
                    provides { }
                  }
                }
Appendix D. Final process model for claims settlement

26 sequence Process Notification {
    action Assign Handler {
        requires { Order | Telephone Notification Form } 
        /\ provides { Handler } */
    }
    29 action Create Folder {
        /\ - Presumably, this requires one of the other. */
        requires { Order | Telephone Notification Form } 
    }
    action Assign Claim Number {
        requires { Order | Telephone Notification Form } 
        provides { (derived) Claim Number } 
    }
    action Check Cover {
        requires { Order | Telephone Notification Form } 
        provides { (derived) Cover.value = "true" | (derived) Cover.value = "false" } 
    }
    selection {
        sequence {
            action Enter in Claims System {
                requires { Cover.value = "true" } 
                requires { Claim Number } 
                requires { Notification Date } 
                requires { Accident Date } 
                requires { NAC of Third Party } 
                requires { NAC of Policyholder } 
                requires { Claim Amount } 
                /\ provides { } */
            }
            action Check Entered Data {
                requires { Claim Number } 
                requires { Notification Date } 
                requires { Accident Date } 
                requires { NAC of Third Party } 
                requires { NAC of Policyholder } 
                requires { Claim Amount } 
                provides { (derived) Confirmation } 
            }
        }
        sequence Reject Cover {
            iteration {
                action Prepare Rejection Letter {
                    requires { Cover.value = "false" } 
                    requires { Order | Telephone Notification Form } 
                    provides { (derived) Rejection } 
                }
            }
            action Assess and Sign Letter {
                requires { Rejection } 
                provides { Rejection.signed } 
            }
            action Send {
                requires { Rejection.signed } 
                /\ provides { } */
            }
            action File {
                requires { Rejection.signed } 
                /\ provides { } */
            }
        }
    }
}
branch Process Third Party {
    sequence {
            action Notify Date {
                requires { Claim Amount } 
                requires { (optional) NAC of Third Party } 
                provides { (derived) Third Party Confirmation Letter } 
            }
            selection {
                sequence Perform Expert Assessment {
                    iteration {
                        action Visit in Expert {
                            requires { Claim Amount, value >= 200 } 
                            requires { Order | Statement of Liability } 
                            provides { (derived) Expert Assessment } 
                            /\ provides { (optional) Expert Assessment Invoice } */
                        }
                    }
                    action Request Expert {
                        /\ requires { } */
                        /\ provides { } */
                    }
                    action Receive Report {
                        requires { (optional) Expert Assessment Invoice } 
                        requires { Expert Assessment } 
                        provides { (derived) Damage Known } 
                    }
                    action Pay Expert Assessment {
                        requires { (optional) Expert Assessment Invoice } 
                    }
                }
            }
        }
    }
}
References


[12] V. Gruhn, Validation and verification of software process models, in: Proceedings of the 1991 European Symposium on Software Devel-

```c
sequence Process Invoice 
    action Request Invoice 
        requires (Claim.Amount <= 250 ) 
        provides ( (derived) Requesting of Invoice ) 
        provides ( (derived) Claim Invoice ) 
    iteration 
        action Refund Invoice 
            requires (Claim Amount < 250 ) 
        action Receive Invoice 
            requires (Claim Invoice ) 
            provides ( (derived) Damage Known ) 
    
sequence 
    action Determine Liability 
        requires (Claim || Statement of Liability ) 
        provides ( (derived) Blame.value = "true" || (derived) Blame.value = "false" ) 
    iteration 
        require (claim) 
        requires (Claim.value = "true" && Damage Known ) 
        requires ( (optional) Claim Payment ) 
    action Authorizes 
    
sequence Reject Third Party 
    action Prepare Rejection Letter 
        requires (Blame.value = "false" ) 
        requires (Claim || Statement of Liability ) 
        provides ( (derived) Draft Rejection ) 
    action Assess and Sign Letter 
        requires (Draft Rejection ) 
        provides ( (derived) Rejection ) 
    action Filing 
        requires (Rejection ) 
        // provides ( ) // 
    
action Advise Policyholder 
    requires ( (optional) Result Third Party ) 

action No Claims Bonus 
    // provides ( ) // 
}
```


