SANTA CLARA UNIVERSITY
Department of Computer Engineering

Date: _____

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

Daniel Charles Weeks

ENTITLED

Process Modeling Language Design and Model Verification

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

OF

MASTER OF SCIENCE IN COMPUTER ENGINEERING

_____
Thesis Adviser

_____
Thesis Reader

_____
Chairman of Department

PROCESS MODELING LANGUAGE DESIGN AND MODEL VERIFICATION

BY

DANIEL CHARLES WEEKS

Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Engineering
in the School of Engineering of
Santa Clara University, 2004

Santa Clara, California

PROCESS MODELING LANGUAGE DESIGN AND MODEL VERIFICATION

Daniel Charles Weeks

Department of Computer Engineering
Santa Clara University, 2004

ABSTRACT

Process oriented activities are present in many daily activities in that everything from getting prepared for work in the morning to baking a cake has a series of steps that are completed to accomplish some goal. While these processes are trivial, many that are involved in daily life are far more complicated. For example, the steps needed to complete a loan application are not readily known or understood to many because of the complexity of the process. Banks and agencies must collaborate and use a specific criteria to determine who is approved. As processes become more complex and people are added to the equation, the exact workings of the process become difficult to understand. At some point, the complexity of the process requires formalization to eliminate confusion.

To formalize a process, the important aspects must be extracted and described in an unambiguous manner. In order to achieve this level of specification, there must be some form of notation to which the description adheres. In addition, many of these processes, such as applying for a loan, are used repeatedly and the ability to automate the process would be advantageous. A notation for clearly describing processes that can also be used for both a better understanding and automation of the process would be an ideal solution.

There are two common approaches to conjoin notation and automation. The first method is to start with notations that already exist, such as programming languages, and extend them to describe processes. This method introduces a number of problems, the first being the complexity of the notation. For someone to model a process, they must first understand the intricacies of the language on which the modeling language is built. Another concern is that the strict interpretation of most programming languages can make some process-related concepts difficult to express. We propose an alternative solution that addresses these issues.

By looking at the conceptual aspects of modeling a process and deriving a language based on these concepts, we have developed a language that is specifically designed for describing processes. There are a number of advantages to this approach in that complexities introduced by combining programming languages with modeling languages are alleviated which results in a more concise language specification. Finally, there is significantly less of a learning curve. Despite the many benefits, there are some disadvantages to this method such as the lack of tool support. In order to provide this functionality, we have developed a static analyzer that examines process models in order to identify possible errors based on concepts that are specifically attuned to processes.

This thesis explores the concept of process modeling in conjunction with the design, development, and implementation of the process modeling language, PML, and the static analyzer, pmlcheck. Our examples demonstrate how these tools can be applied to describe and minimize errors in a process model. The objective of this thesis is to illustrate the advantages of process modeling and how it can be achieved through the use of PML and pmlcheck.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Processes that occur regularly in daily life range in complexity from trivial activities, such as baking a cake, to complex activities, such as getting approval for a loan. Tasks that are required in a simple process are intuitively understood or easily derived by the person performing the process. However, more complex processes are not intuitive and require careful consideration and detailed analysis of each task. In complex processes, a significant amount of time is spent discerning what the next task is and how to transition from the current task to the next. Being able to unambiguously describe a process removes the uncertainty involved in performing the process.

A process description [Klingler, 1994] characterizes the important aspects of a process from which a model of the process can be derived. This model is called a *process model*. The purpose of a model is to reflect the control-flow of the process without incorporating nonessential properties. Though it is possible to have many different descriptions that satisfy the requirements of a process, a concise representation of a complex process is generally preferred. An unclear representation may be misleading while a concise and complete representation can disambiguate unclear aspects of the process.

In addition to disambiguating a complicated process, having a written notation for process description provides the ability to analyze the process by checking for errors. Having the ability to validate a process before enactment increases quality and ensures correctness. One must take into consideration that processes are designed starting with abstract concepts and are iteratively refined into detailed process descriptions. Therefore, the notation used to describe processes needs to reflect this evolutionary development cycle, but still provide valuable information about the process at every level of abstraction.

A concise and clear description is not the only advantage of process modeling. Processes are generally repeatable and having the ability to automate the process increases efficiency. Full automation may not be possible because of the high level of the process description and the necessary human interaction, but some task-related responsibilities can be removed, such as determining when a loan application has been approved and sending confirmation. Automation provides the facility to guide the process through its life-cycle, only stopping for human interaction when necessary.

In order to check models for errors and to automate processes, a formal notation is required to specify the model. Necessary syntactic and semantic rules qualify this form of formal notation as a language. Without these rules, the language is ambiguous, which introduces a level of uncertainty into the process model. The objective of the language is to be as expressive as an unstructured description, but changing the representation into a more useful format [Conradi and Liu, 1995].

The design of the language constrains how and where the process model can be applied. If the language is too complicated or strict, it may not be expressive or flexible enough to be useful in a broad range of applications. If the language definition is too loose, it may not have the ability to provide meaningful analysis or automation. Between these extremes exists the potential for a language that is expressive, flexible, and simple enough to provide meaningful feedback and assist in process development.

This thesis is motivated by the need to define process modeling techniques that will increase the efficiency of process design through model specification, language design, and methods for analyzing

process models.

1.2 Approach

There are currently many efforts to successfully describe, validate, and automate processes [Agostini and Demichelis, 2000; Cass et al., 2000; Pinheiro da Silva, 2001; J. M. Rib, 2000; Dami et al., 1998], but these approaches do not provide efficient solutions. Through research into process description, validation, and automation, we have developed a philosophy based on process design. This thesis illustrates the essential aspects required for successful process modeling.

We define and describe the objectives and requirements of process modeling based on process related activities. Then we examine different paradigms of process modeling and the advantages and disadvantages of each approach. These paradigms are divided by the techniques employed in finding efficient process models. The rule-based or logical paradigm [Klingler et al., 1992] describes processes through dependencies and goals. The actual control of the process is then derived from these specifications. Another common approach is a control-based or imperative approach [Sutton, 1990], which defines the control of the process and then validates the dependences within the process. We implement the latter approach because it is more intuitive to process description in that it gives the modeler control of task organization.

In addition to these modeling paradigms, there are a variety of approaches to how process modeling languages are constructed. A popular method is to design the modeling notation as an extension of an existing programming language. Though this design inherits many properties of its base language, we argue that using an underlying language is counterproductive to the objectives of simplicity and clarity. Instead, we construct an entirely new language based on the fundamentals of process modeling. However, this approach introduces new challenges in automating and analyzing process models for errors. We describe methods for solving these challenges through techniques of identifying and eliminating process related errors.

To illustrate how this philosophy of process design can be implemented, we designed PML, a stereotypical process modeling language, which fulfills specified objectives and requirements that we derive from examining processes. Based on this language notation, we describe an approach to understanding process related errors and analyzing process models. Using the software development process for Netbeans, an Integrated Development Environment for Java programmers, we present techniques for describing, refining, and analyzing process models.

Chapter 2

Processes and Modeling

2.1 Process Definition

People are continuously involved with process related activities on both a conscious and unconscious level. Some processes are so trivial that they are completed without even thinking about the events that take place, such as getting ready for work in the morning. However, even this process is more involved than it originally appears. Some common tasks that a person might perform include the following:

- Making bed
- Taking a shower
- Getting dressed
- Eating breakfast
- Brushing teeth

These tasks are performed so often that they are second-nature, but this list has already made some important assumptions about this process. There are dependencies between these steps that determine the order in which they are performed. It makes little sense to get dressed before taking

a shower. There is a similar dependency between eating and brushing teeth. These dependencies are significant factors in determining the order in which the tasks can be completed.

Despite the limitations that dependencies put on task ordering, there are still many ways in which these tasks can be performed. Some of the tasks have interdependencies, but others remain independent, which allows for some variation in how the process is described. Instead of the order given previously, the tasks can be arranged as:

- Eating breakfast
- Brushing teeth
- Making bed
- Taking a shower
- Getting dressed

This ordering achieves the same result as the first, but sets of dependent tasks are rearranged without breaking their interdependencies. In fact, there are many possible arrangements of tasks for this process that will all achieve the same result without violating any dependencies. However, this is a very simple example of a process. More complex processes have tens to hundreds of tasks that need to be performed and each task has specific dependencies that must be fulfilled. As the number of tasks increases the dependencies become more intricate and difficult to track.

2.2 Control Flow

The order that tasks in a process are performed is referred to as the control, or control-flow of the process. The example of getting prepared for work in the morning is a very simple process, but there are more complex aspects of this process that must be taken into consideration. One such consideration is that the person performing this process has more options than originally indicated. For example, if the person was short on time, they may have to choose between eating and showering in order to get to work on time. This represents a choice that has to be made during the performance

of the process. People make these types of decisions all the time while performing tasks. A common activity in processes is selecting between various tasks that can be performed and is an essential component in control flow.

Many tasks or series of tasks in a process need to be repeated, such as tying a tie. When tying a tie, if the length of the tie is not correct when the knot is complete, then the tie must be undone and retied. This task can be done just once or many times based on the need to fulfill some requirement. This sort of situation occurs frequently within processes.

Another control related activity is being able to perform tasks concurrently. If there is more than one person performing a process, then it is possible for them to perform independent tasks at the same time. Concurrency provides the ability to do two things simultaneously, but to satisfy this, the order of the concurrent tasks must be independent. If there exists a dependency, then one task must be finished before the other can begin, which effectively eliminates any efficiency gained by concurrency. With complex processes, concurrency is important for efficiency because many people can be involved in a process and if they each have to wait for the others to complete tasks, then efficiency is lost.

Each of these control properties are integral to performing processes. Without these concepts, a process would only allow tasks to be completed sequentially in a predefined order. However, processes are rarely a strict sequence of tasks, and are generally a complex mixture of decisions, concurrent actions, and iterations.

2.3 Task Properties

Tasks are simply some action or activity that is performed during the execution of a process. These are the building blocks for a process, but even they are complex and have a collection of properties associated with them. At a high enough level of abstraction, tasks are simple and their actual meaning is inferred. However, to gain more insight into the process, tasks have to be inspected more

carefully.

Tasks have three basic aspects:

Actors For a task to be performed, someone or something must perform the task. This is the actor of the task. In large processes, there may be many actors who have to work on a single task.

Requirements Before a task can begin, certain conditions must be met. This usually means that something has to be available for the task to be performed. If the task is to brush your teeth, then a toothbrush is required. Without this, the task cannot be performed. Requirements impose dependencies upon the order in which tasks can be executed [Sutton, 1995a; Sutton, 1995b]. If some required item is not ready, then the task must wait until it is provided.

Products When a task is complete, some activity has been performed and the result is some form of product. For example, when you brush your teeth, the resulting product is clean teeth. These products have a direct relation with requirements because what one task produces, some other task may require. This results in a complex set of relations between tasks within a process.

These three elements constitute the internals of most tasks. Understanding the relationships between these tasks is a fundamental goal of examining processes. In simple cases, these relations are well understood, but complex cases pose a more difficult problem. When the number of tasks in a process grows to the point that it is difficult to keep track of the interrelationships between them, an automated method of tracking dependencies is needed.

2.4 Modeling Goals

Process modeling provides a means to control the growing complexity of processes. Understanding and checking all of the dependencies that occur within complex models is a difficult task. If the conceptual and procedural aspects of a process can be captured and represented in a model, then tools can be designed to automatically check the models [Joeris and Herzog, 1999]. The objective of

modeling is not to recreate every minute aspect of the process, but instead, to extract the meaningful properties of the process and imitate the behavior of the actual process [Armenise et al., 1993]. The ability to model processes before performing them allows errors to be caught before they are manifested in the performance of the process, which might result in a complete redesign of the process.

In addition to finding problems in a process, modeling allows the process designers to explore many different designs before enactment. Complex processes may be too costly to actually implement and perform in order to find the best design. Modeling allows the modeler to easily modify the process and determine if the changes are effective.

Once a proper model has been created, the model can serve as a guide through the actual process [Cugola and Ghezzi, 1998]. This automation of error checking ensures that the process described is the same process that is being performed. In addition to ensuring the process is carried out correctly, the process is repeatable with a high degree of accuracy.

2.5 Modeling Paradigms

There are many methods to approach the problem of modeling processes and different paradigms provide numerous views about what properties of processes are important. One common paradigm is rule-based or logical modeling [Dami et al., 1998; Klingler et al., 1992; Junkermann et al., 1994]. This method relies on rules to describe the tasks and then generates a model from the dependencies specified in the tasks. This method does have advantages. One advantage is that the modeler only has to worry about the individual tasks because the control of the process is the result of tools used to generate the model. Another advantage is that when the model is created, the dependencies can be fulfilled. However, there are a significant number of drawbacks that make this method less than ideal.

The most obvious problem with this method is that the modeler has difficulty controlling the

order of tasks in the process. If two steps are independent, but the modeler wants them to be performed in a sequence, then a false dependency must be introduced in order to achieve the desired results, which adds a layer of complexity that should be unnecessary. We feel this method also is counterintuitive to how people think about processes. The order in which tasks are performed is a primary concern when defining a process. Therefore, it should be within the power of the modeler to control the flow of the process.

Another disadvantage of logical modeling is that it can produce undesirable results especially at high levels of abstraction. The rules required to provide control to a process are too low-level to adequately control an abstract model. If no rules are specified, then it is difficult to generate a model that accurately depicts the process at a high-level. For example, the high level description presented for waking up in the morning could be generated in any possible order, which means that each task could be set to run concurrently. However, this is not an accurate model because there are dependencies that this type of modeling cannot illustrate without detailed specification.

The paradigm that we use is control-based [Cass et al., 2000; Sutton et al., 1997], which resolves many of the problems inherent in the rule-based paradigm. 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 abstract processes, detailed processes, and every layer of abstraction between the two [Kaiser et al., 1993]. At a high level of abstraction, such as the process of waking up in the morning, 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. This method is more intuitive and reflects the steps that humans take when describing a process.

Chapter 3

Process Modeling Languages

3.1 Language Goals

Although there are many different approaches to process modeling, there is a general consensus about the goals of modeling languages. These goals embody how a language should capture the aspects of a process in order to represent the processes properly. The most common goals are:

Simplicity: The complexity of the language should not prevent a person without strong technical background from using the language. A non-technical person should be able to model a process, such as the one described in Section 2.1, without being encumbered by the syntactic or semantic requirements of the language.

Flexibility: The language can be applied in a variety of applications. The concept of processes covers a wide variety of domains and a language should not be limited to a single area of application. This language should be able to describe business processes, software processes, or any other form of process equally.

Expressiveness: Being able to accurately reflect a process is essential in order to get any useful information about the process.

Comprehensibility: The language should represent a process in such a way that the model can be easily understood.

Enactability: The language should enable the model to mimic the actual execution of a process. This automation of the process should not affect other goals of the language by introducing control constructs that have no relevance to the actual process.

Though these goals have been repeatedly defined and examined, many language implementations disregard these goals in an effort to achieve some additional functionality [Cugola and Ghezzi, 1998].

3.2 Graphical Language Approach

Though formal notations generally refer to written languages, there are approaches that use a graphical representation to model a process [Dami et al., 1998; Cass et al., 2000; Minas and Hoffmann, 2001]. The advantage to a graphical approach is that the process is displayed as a graph or flow chart that can be easily followed. However, the attractiveness of the graphical display eventually erodes as the detail of the model increases. In order to represent properties that are essential to a process, the graphical approach must introduce some visual symbols to represent these properties. As the model becomes more detailed, these symbols compound until they clutter the display and distract from the actual model. With the introduction of symbols, a key is required to understand their meaning, which is an added level of complexity that is not necessary in written languages. In addition, the graph of a large and complex model can span a considerable area, which introduces a considerable physical distance between sections of the process. These problems deal with the actual layout of the model, but there are more inherent problems to this approach.

Graphical language approaches are generally built on top of a written modeling language, and the addition of a graphical layer may introduce problems when translating the graphical model of the process to a written model. This extra step adds an extra level of complexity, which allows for the possibility of introducing errors into the process. Though the graphical language is designed

to have the same expressiveness that the written language has, there is always the possibility of misinterpretation or misrepresentation of some section of the process. Therefore, the graphical language approach has a dependency on both the process modeling language it is built upon, and the method of translation to the lower-level language. This compounds the problems of both the written and graphical languages, which can result in an complicated and error prone modeling technique.

3.3 Programming Language Approach

The most common approach to designing a process modeling language is to build the language on top of an existing programming language. There are strong ties between concepts in process modeling and programming which makes a programming language seem like the ideal notation for process modeling. A stereotypical example of this language design approach is the language APPL/A [Sutton, 1990; Sutton et al., 1995]. This language is designed as an extension to the programming language Ada [Barnes, 1998] and was developed as a supplement to Ada because it has well established support for many of the constructs needed for a modeling language such as concurrency and iteration. Building upon Ada allowed for the reuse of a significant amount of research already applied to solve similar problems.

There many advantages to using this *bottom-up* approach to language design. APPL/A was able to take advantage of features such as concurrency, iteration, modularity, information hiding, and exception handling that are integrated into Ada. Another important feature is the ability to do mathematical operations. Expressions are native to the language, which provides a wide variety of numeric operations and comparisons. In addition, the existing compilers provide type checking and error checking capabilities. Other modeling constructs such as relations and tasks were effectively implemented using Ada constructs of packages and tasks, though tasks have a different meaning between the two languages. Using Ada as a framework for a process modeling language allowed the

language designers to base their efforts on well established constructs and implementations. Despite numerous advantages, there are some fundamental concerns raised by this language design approach.

Though the use of an underlying programming language has obvious benefits, it compromises many of the goals of modeling languages. Ada 95 has ninety-eight keywords specified in its grammar and building a language upon this means introducing even more keywords. The result is that from the very first stage in the design of a modeling language, the language is complex, not simple as intended. A person who is modeling a process may not need to know the meaning of all of the keywords in Ada, but they must recognize them in order to avoid using them. Using a keyword unintentionally could result in checking errors caused by the lower level language that would confuse the modeler because they have no relevance to problems in the actual model. This places an additional burden on the modeler to understand aspects of two languages, the modeling language and the programming language.

Building on a programming language places limitations on the expressiveness of the modeling language. Process related activities may not be expressible in the underlying programming language and therefore cannot be expressed in the modeling language. Ada has a requirement that concurrent tasks that need to communicate must synchronously meet, which is called a *rendezvous*. This constrains the modeling language because asynchronous activities exist in processes. The primary concern with this limitation is that it is not a problem with the modeling language. Instead, it is a limitation imposed by the language on which it was developed.

The representation of the process model in this style of modeling language can also be confusing. To understand how the process will be enacted depends on the underlying language and how the process will execute once it is compiled. What may seem intuitive at the modeling level, may not be reflected at the programming language level, which introduces the potential for misrepresentation of the process within the inter-operation of the two languages.

Existing tools that support the language can also be problematic. The error checking capabilities of the Ada complier are designed for checking errors in computer programs. However, the errors that

can occur in a process are based on a different criteria than those of programming language. While compilers are designed to examine programs for static errors, processes are dynamic in nature and many of the useful features of static checking, such as type checking, are not essential for process models. This limitation of the tool is not because of an oversight in the modeling language design, but a conceptual difference between processes and programs.

The primary concern of this style of language design is that it relies on a language paradigm that is not explicitly designed for process modeling. Programming languages are designed for computation. Considerable research has gone into developing programming languages, but their target applications are not the same as process modeling languages. Though there are many similar concepts between programming and process modeling, there are subtle differences that separate the two fields. These subtle differences prevent the two from interacting in a mutually beneficial manner. Therefore, a new language design is needed that focuses on and represents the concepts of process models rather than relying on programming languages, which are a product of a different domain research.

3.4 Process Language Approach

The language design approach that we propose is based on a philosophy derived from the fundamental concepts in process modeling described in Chapter 2. Instead of using existing languages to reflect processes, we examine the requirements of process modeling languages and design a language based on those principles. The result of our approach is the modeling language PML [Atkinson and Noll, 2003; Noll and Scacchi, 2001]. This language is a process modeling language that we developed without any external support by existing languages and though it does not have the support of a well established programming language, the benefits of this approach outweigh the losses.

This *top-down* approach to language development has many advantages that address problems inherent in the bottom-up approach. One advantage is that the initial level of complexity of the design is considerably less because there is no reliance on a lower-level programming language. The

PML grammar incorporates only thirteen keywords, which is a significant simplification compared to the hundred or more keywords in similar languages. This simplification of the language has many implications: the language is much easier to learn, which makes it more attractive to those who do not have a background in programming, and modelers do not have to worry about understanding restrictions of an underlying language that could result in modeling errors.

Another positive aspect of this language is that the syntax is very straightforward. Using a programming language imposes the syntax of the programming language on the modeling language. However, PML is not subject to this restriction, which allows for a very simple syntax. In PML, all statements follow the form: *keyword* [*identifier*] { . . . }. This syntax is very simple and helps to eliminate the confusion of complicated grammars. The only consideration is about what statements can be nested inside other statements, but nesting is generally shallow.

Another problem that this language design addresses is the difficulty in achieving the right level of expressiveness. Modeling languages built on programming languages are restricted by the underlying language, but not having that restriction allows for a much more adaptable grammar. PML incorporates a language construct called a *qualifier*, which is not a keyword in the language, but is a user-defined specification that enumerates the characteristics or qualities of a resource, which allows the modeler to emphasize, constrain, or modify resources in the process model without complicating the grammar. This also increases the expressiveness of the language by allowing the modeler to specify how the resource should be treated.

The simplicity of the grammar has a direct correlation to the understandability of language. Because the language has a simple syntax, the resulting models are easier to understand. The uniform nature of statements allows anyone to follow a process model even if they do not fully understand the intricacies of the language.

This design approach also makes the process model easier to enact. Instead of relying on a compiler to generate code that is later executed to enact the process, the actual process model can be enacted. PML employs an enactment environment to interpret and enact the process model and

is designed specifically for execution of process models. Therefore, it understands how to handle process related activities as opposed to a program that is designed for computation.

Though this approach does address many of the conceptual concerns raised by the programming language approach, there are many implementation concerns that need to be addressed. The advantage of using a programming language is all of the support that exists for the language, but by removing the programming language, the advantage is lost. PML does not inherit the tool and language support that benefit other modeling languages. Therefore, these tools must be reconstructed, but this allows for the tools to be redesigned in a way that reflects processes rather than programming.

One feature that PML does lose by not extending a programming language is mathematical support. Mathematical concepts are essential to programming languages and have considerable support because they are designed for computational purposes. Complicated mathematical expressions are trivial for a programming languages, but adding support for this in a process modeling language like PML would require extensive work for a feature that would rarely be used. Therefore, PML supports only a small subset of math, which allows representation of mathematical quantities, but limits the computational ability of the language. This restriction forces the modeler to work at a higher level of abstraction than a programmer would, which prevents the model from becoming too detailed.

Another feature that programming languages provide and must be recreated is the ability to check for errors, but this loss is not as detrimental as it may appear. The type of checking that a programming language performs does not reflect the type of errors that are common in a process model. Redesigning the way that checking is done requires significant effort, but the result is a tool that implements the concepts of a process model rather than a program. Even in the context of a process modeling language with programming language support, this tool could be a useful addition.

Finally, the actual means of enacting the process model is lost. A programming language can compile a process model and execute it on a computer. However, PML does not natively support this functionality. The specification of the language does not translate directly to code, and the

constructs that were added to make the language more expressive and flexible, like the qualifier, make translation to a programming language infeasible. Though creating an interpreter requires additional effort, the interpreter would also be designed under the philosophy of process modeling rather than programming.

A process modeling language can inherit many features by extending a programming language. Though these features seem to complement the modeling language, they actually complicate the language while providing a false sense of achievement. Programming languages are well researched, but they are designed for problems that are conceptually different than those found in process modeling. Though the two language paradigms share many of the same concepts, they are enough fundamental differences that they should be treated separately.

Chapter 4

Language Design

4.1 Language Fundamentals

The language goals specified in Section 3.1 define the framework on which PML was designed. Using the software engineering waterfall process described in [Pressman, 1992], we will illustrate how these design decisions were arrived at and implemented. The syntax for defining a process in PML is simply:

```
process identifier { . . . }
```

The evolutionary nature of this language allows for a very trivial definition of a process model:

```
process waterfall_model { }
```

While this is a syntactically legal PML model, little information can be derived from it other than the fact that a process exists. Any other information drawn from the model at this point is based on intuition from the name of the process. However, this is the basis for a refineable model, which will eventually result in a detailed recreation of the process.

The most fundamental component of a process is a task or action, which are terms that can be used interchangeably. The PML syntax for an action is:

```
action identifier { . . . }
```

With just the process and action statements it is possible to make a non-trivial model of the waterfall process:

```
process waterfall {
   action analyze { }
   action design  { }
   action code    { }
   action test    { }
}
```

This high-level description provides information about what steps need to be completed and the order in which they should be performed. Though there is little detail about any of these steps, the model has enough information for a basic understanding of the waterfall development process.

4.1.1 Resources

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 recreate a variety of dependencies that occur within a process. In early development of the language, PML incorporated a specific set of constructs for specifying the relationship between actions and their resources, such as inputting, outputting, requiring, and providing resources. Inputting and outputting resources illustrated that the resource's origin is from an environment external to the process, while requiring and providing resources concerned resources within the process. However, after careful analysis of task structure, it was clear that defining the source of a resource is not necessary to an action. The only postulate for an action is that the resource is available when the process enters or exits the action. Though constructs for defining inputs and outputs to the process may be utilized by external tools, they have no meaning within the language and were removed for simplicity. The new specification allows actions in PML 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 modify the current example to provide more information about the actual internals of an action:

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

This statement illustrates the conditions that must be met for this action to be performed and to terminate. Entrance to the action is not possible unless the function, behavior, performance, and interface are available and exiting is not possible without analysis and analysis documentation. Using these predicates, a modeler can reconstruct the dependencies that exist within a process.

4.1.2 Attributes

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 previously stated that the action `analysis`:

```
provides { analysis_documentation }
```

However, introducing a new resource to describe the fact that the analysis portion of the documentation is 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. Therefore, we provide attributes to solve this problem by describing the state of a resource and thus it would be more clear to state:

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

While `analysis_documentation` is a 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.

4.1.3 Expressions

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 becomes more detailed, constraining the state of resources and attributes provides more explicit control over the process. By adding expressions the model transitions to another level of detail and can represent state:

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

This statement clearly emphasizes that the attribute has a specific state, but it is important to understand the actual meaning of the expression. Declaring that an attribute has a specific state does not bind the attribute to that state or change the value of the property. In fact, nothing in the model actually changes any resource or attribute. Expressions assert that some relation is true at a specific point in the enactment of the model. Any form of binding or changes to state happen during the execution of an action and are not illustrated in the model. The details of how the internals of the action are being performed has no relevance to the predicates of the model.

4.2 Control

PML has four mechanisms, based on the control related activities described in Section 2.2, to describe the control of a process. These control flow constructs reflect process related activities and describe the ordering of steps in a process.

4.2.1 Sequence

A `sequence` is the most basic form of control and is the default control mechanism when nothing else is specified. The actions in a `sequence` construct are performed in the order that they are specified:

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

```
    action third  { }
}
```

This construct is the most natural and intuitive form of control for a process. When one thinks about performing any process, a simple sequence of steps to accomplish the final goal is often the easiest representation.

4.2.2 Iteration

A condition that occurs quite frequently within processes is the need to repeat certain steps. While iterating over these steps, there are two concerns that must be addressed: when to go back and repeat the steps, and when to stop repeating and continue the process. Generally, this decision is handled by an expression that is evaluated to determine if the steps need to be repeated. This method works well if the number of repetitions is known when the loop begins, but the dynamic nature of a process often results in this information being unavailable. An example of this non-deterministic nature processes is making a cake where the instructions state: add flour, stir mixture, test for consistency, repeat until mixture is thick and consistent. There is no indication of how many times the steps in the process need to be repeated and the judgment of when the mixture is ready relies entirely on the person making the cake. Based on this dynamic decision procedure, we constructed PML to model iterative constructs as they exist in processes and focus on the concerns of when to exit and when to continue. The syntax for an iteration follows the same structure that a sequence:

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

When determining which path to take in PML, the predicates of the first action in the loop, `first`, and the first action following the loop, `post`, are the points of interest. When the last action in a loop is complete, the loop determines how to proceed based on whether or not the requirements

in the first action of the loop and the first action following the loop are satisfied. There are four possible scenarios:

- If the requirements for the first action in the loop are met, but not for the first action following the loop, then the loop is reentered.
- If the requirements for the first action following the loop are met, but not for the first action in the loop, then the loop is exited.
- If the requirements for the first action in the loop and the first action following the loop are met, then it up to the person performing the process to choose which action is taken.
- If the requirements for the first action in the loop and the first action following the loop are not met, then the process waits until one of the other conditions is true.

At first this dynamic decision procedure may appear to be inconvenient because processes may need to wait for a human to choose the proper path, but it actually allows the process to be more dynamic by providing multiple options when they exist and suppressing them when only one path is available. Additionally, there are many conditions in processes that are based on human judgment and cannot be evaluated by a machine.

With this control construct we can describe additional information about the waterfall process. In theory, the waterfall process should enter each task and proceed directly to the next task in sequence until the final task is completed and the product is ready. However, in reality, there are many factors that can cause problems to occur within the process, such as errors found in the code, which results in redesign and additional coding. To illustrate this behavior we can state:

```
iteration {
   action design  { }
   action code    { }
   action testing { }
}
```

This high-level example describes how testing can result in redesign, recoding, and retesting until the product is actually complete.

4.2.3 Selection

Another activity that occurs regularly in processes is choosing between possible paths. Selecting one of many paths requires that a decision be made about which direction to take. The selection construct in PML defines possible paths of execution with only one being performed:

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

The decision procedure for determining which path to take is handled in a similar manner to iterations. In this case, the predicates of the first actions in each possible path are the focus. The requirements for the first action in each path are evaluated and the result is one of three possible conditions:

- If the requirements for only one of the actions in the selection are satisfied, then that path is taken.
- If the requirements for two or more actions in the selection are satisfied, then the person performing the process must choose which path to take.
- If none of the requirements of the action in the selection are met, then the process waits until one of the other conditions is true.

This construct is derived directly from how a person would consider possible paths by examining what paths are available and choosing the path that best serves their needs. This type of decision in process models cannot always be automatically determined and therefore must rely on human interaction to choose which path to take. Though it is possible to simply choose the first available path, therefore avoiding human interaction, there might be external considerations about which path should be taken that an automatic procedure cannot foresee.

4.2.4 Branch

The branch construct specifies concurrency of actions within a process:

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

Concurrency is usually employed as an optimization, which is generally performed implicitly and does not have a decision procedure associated with it. Each path must be performed, which removes any need for human interaction related to control. However, there are problems that arise from using concurrency in a process model. In Section 3.3 we noted that APPL/A was limited by its inability to express asynchronous rendezvous because of the constraints of Ada. PML does not restrict the way that a rendezvous is handled. Instead, the tools that interpret PML must implement which method is used. We realize that this introduces an ambiguity as to what will actually happen at a rendezvous, but processes do not adhere to the strict nature of programming languages and the dynamic nature of processes requires that the decision be left to the modeler.

Relying on synchronous rendezvous has a significant disadvantage in that many processes require asynchronicity. However, artificial constraints in an asynchronous implementation can be introduced to recreate a synchronous rendezvous:

```
branch {
   action path_1 { provides { r_1 }  }
   action path_2 { provides { r_2 }  }
}
action post { requires { r_1 && r_2 } }
```

In this example action `post` requires resources from both paths, so they must both complete before the process can continue. Though this does allow both branching methods to be represented, introducing artificial dependencies into the model complicates the model with artifacts that only relate to solving control related problems.

The waterfall model states that testing should be done after the code is written, but writing tests is often started at the same time as coding, so that tests can be prepared as the code is written

rather than having to wait until the code is complete. To represent this we can change our model to:

```
branch {
   action code { }
   action write_tests { }
}
```

4.3 Advanced Language Features

With the language features of PML mentioned in the previous section, a model can be constructed that will portray the control and flow of resources within the process. For many models, this information is enough to extract meaningful feedback from the process. However, more complicated and detailed models require additional features. The evolutionary nature of PML allows for more precise constructs to be used to refine the model to embody the specifics of a process and allow processes to interact.

4.3.1 Qualifiers

Though attributes and expressions provide methods for describing properties and states of resources, not every quality of a resource can be expressed in this manner. There are aspects of a resource that are extrinsic to the resource and apply to how the resource is handled, modified, and restricted. For example:

```
action code {
   requires { design && funding }
}
```

This action has two requirements that consist of some tangible resource. However, the two resources are profoundly different and must be treated in different ways. The design is an inexhaustible resource in that it can conceivably be used over and over again without losing any of its substance or quality. However, funding is exhaustible and can only be used until the funding is gone. Representing this difference in a model is a difficult task. Some languages provide keywords associating

a resource with being consumed by an action [Klingler et al., 1992]. Though adding keywords will make modeling a specific situation, such as this one, much easier, there are many possible situations that cannot be conceived of while designing the language.

Attempting to enumerate all of the possible factors that affect resources by introducing terms into the language to describe how a resource should be treated is infeasible because it is not possible to characterize every quality a resource could possess. There are a number of situations that are difficult to describe and adding a language construct to clarify how each situation should be handled explicitly violates our goal of simplicity and the expressiveness of the language would rely on how many situations we could envision. One such situation was related to the iteration construct. In Section 4.2.2 we describe an iteration over the design, code, and test stages of the waterfall, but it is difficult to describe the role of resources within this construct and an interesting question was posed: If the coding phase produces code and an executable during each iteration, are they the same resources or are they new ones? Intuitively, the code is still relatively the same because only small modifications were made, but the executable could be entirely new and not the same as the executable from the previous iteration. Therefore, we needed a way to describe that a resource can change or be replaced.

A similar problem occurs when creating a new resource. Providing more information about how a resource was created is not possible with the basic language constructs of PML. For example, code does not spontaneously appear in the coding stage, but is derived from the design, but it is not possible to illustrate this quality of the code without additional levels of specification.

To alleviate these problems, we introduced a construct called a *qualifier* into PML. The qualifier is used to describe characteristics or qualities of a resources that are beyond the scope of the regular syntax of the language. With this construct we can state:

```
(partially_consumed) funding
```

In this example `partially_consumed` is a user-defined quality of the resource `funding`. This language feature also supports multiple layers of qualifiers, such as:

```
(new) (generated) executable
```

With this construct, the model can better represent the process, but there are some difficulties associated with using a qualifier. For the process to be enacted, the environment must understand how to handle the qualifier if it has a direct impact on the execution of the process. This means that additional functionality must be provided to interpret the meaning of a qualifier. Using the language features of PML, we now present a detailed waterfall process model:

```
process waterfall {
   action analyze {
      requires { function && behavior }
      requires { performance && interface }

      provides { requirements }
      provides { documentation.analysis == ``complete'' }
   }
   action design {
      requires { requirements }
      requires { documentation.analysis == ``complete'' }

      provides { design }
      provides { documentation.design == ``complete'' }
   }
   branch {
      action code {
         requires { design }
         requires { documentation.design == ``complete'' }

         provides { documentation.code == ``complete'' }
         provides { (derived) code && (new) executable }
      }
      action write_tests {
         requires { requirements && design }
         requires { documentation.design == ``complete'' }

         provides { test_cases }
      }
   }
   action test {
      requires { code && test_cases && executable }

      provides { code.tested }
   }
}
```

This example is one of many possible models of the waterfall development process. Even this model can be refined to include more detail to meet the needs of the person performing the process, such as adding scheduling, funding, and project specific information. However, this model can be applied to any waterfall development process without modification because it is at a high enough level to describe the general process, but low enough to capture the essential control and resources

of the process.

4.3.2 Process Linking

In Section 4.1.1 we mentioned that PML had previously implemented an input and output construct to illustrate where resources were located. Though these features were removed from the language because they did not address language specific issues, external tools rely on this information to have a better understanding of the process. For example, an enactment environment may want to know which resources are inputs to the process and which resources are products of the process so that they can be linked to other related processes. After considering the impact of including constructs within the language to specify these inputs and outputs, we decided that it would complicate and detract from the cohesiveness of the language to include constructs that were not language specific. Therefore, we constructed an external utility to handle process inputs and outputs while maintaining the structure of the PML grammar.

The syntax for inputs and outputs is:

```
input  { . . . }
output { . . . }
```

This additional utility of the language can be used with the waterfall model to describe some inputs to the process such as:

```
input { time_line && funding }
```

Using inputs and outputs makes the environment aware of resources that can be linked to other processes. In this example the `time_line` and `funding` for the project are external to the process, which means that these resources were created by another process or were defined prior to the entering the process.

In order to tie two separate processes together, information must be passed between the two processes. Separate processes do not have access to the internals of other processes, so they must

communicate through resources passed between them. Once declared, the environment can determine how the processes relate to each other.

The problem with linking processes in this fashion is that the environment must construct a path of execution based on the inputs and outputs of several processes. This introduces a logical paradigm on top of a control based paradigm. There are advantages and disadvantages to constructing process execution logically. One advantage is that a second language is not needed to describe how the processes interact. The dependences at this level are arguably simpler than those contained within a process. However, mixing the two paradigms complicates the enactment process.

4.4 Advanced PML Example

So far we have examined how processes can be modeled using PML, but we have only provided trivial examples. In this section, we will explore a complicated process and illustrate the evolutionary nature of the language along with the facilities provided to deal with unforeseen problems. While processes in business and engineering are commonly used for modeling, we aim to expand the scope of process modeling beyond traditional modeling topics.

The model that we present is related to chemistry and was developed with the aid of a chemistry student performing research using peptides. Peptides are chains of two or more amino acids and have a variety of applications within the field of chemistry relating to metabolic functions, but the application of peptides are not relevant to this example. The process we will model is the actual building of a peptide, which implements a variety of language features from PML.

4.4.1 Abstract Peptide Model

Initially, we can say very little about the actual process for building a peptide. Without an explanation of what the process entails, all we can describe is that it is actually a process. This is a trivial aspect of modeling, but can be described by PML as:

```
process peptide_synthesis { }
```

Providing useful detail to this model requires some understanding of the process of synthesizing a peptide.

Building a peptide requires two components: a resin, and amino acids. The resin acts as an anchor to which the first amino acid attaches after which additional amino acids are added to form a chain of two or more acids. However, the resin must be prepared before an amino acid can be added, which is referred to as swelling the resin. The amino acid must also be prepared before it can be added to the resin or to another acid, which is called activating the amino acid. Once these steps have been performed the amino acid can be added to the chain. Adding an acid to the chain can introduce impurities in the peptide, so the chain must be purified by a step that is called flow-washing. Once purified, the amino acid must be deprotected by removing the tail section of the chain (this section is known as the BOC group) so that another amino acid can be added, which is accomplished by a process called cleaving the BOC group. This step can also introduce impurities, so the chain must be flow-washed again. These steps, with the exception of swelling the resin, must be performed for each acid that is added to the chain. When the amino acid chain is complete, the resin needs to be removed to free the peptide, but before the resin can be removed, it must be deswelled to prepare it for removal. Once deswelled, the resin can be cleaved from the amino acids. The final step is to lyophilize, or dry, the peptide so that it can be used in experiments.

By looking at the steps in the description, we were able to make an abstract, but informative model of the process:

```
process peptide_synthesis {
   action swell_resin { }

   iteration {
      action activate_amino_acid { }
      action add_amino_acid { }
      action flow_wash { }
      action cleave_boc_group { }
      action flow_wash { }
   }

   action deswell_resin { }
   action vacuum_pump { }
   action cleave_peptide { }
```

```
    action lyophilize_peptide { }
}
```

At this level, the model articulates the tasks and the control of the process. Though there is not enough information for a complete understanding of the process, the order of the tasks is clear and the visual presentation is much more precise than a standard written explanation. While this is enough information to achieve an understanding of the process, further description will lend to a much more precise understanding of synthesizing a peptide.

4.4.2 Adding Resources and Dependencies

The next step in designing a process model is building the dependencies by describing the resources that are needed by each task. At this point, we are confronted with determining what level of detail is needed for resources. This poses an interesting problem because the resources that are used in making a peptide can be described at many different levels. Chemistry experiments, such as synthesizing a peptide, often rely on very hazardous materials that require the use of protective gear. Having to specify every piece of protective equipment, such as goggles and gloves, is tedious and not necessarily related to the actual process of building a peptide. However, depending on the target audience of the model, this information could be helpful. The same applies for tools used in building the peptide. If a step requires that a solution be stirred, it is up to the modeler to decide if the step should require a stirring stick. While a stirring stick is obviously needed to perform a task, the use of such a tool is so intuitive that it may not be material to the task. Therefore, a primary concern in developing the model is deciding how much detail is actually necessary.

In this example, we assume that noting equipment that is standard for a laboratory is non-essential to the process of building the peptide unless it has an important role throughout the process. Many of the resources that are needed by steps in this process are chemicals that are used for performing actions such as flow-washing or activating amino acids. In addition, there are tools that are necessary, such as vacuums that are used to remove extraneous chemicals. Understanding

what these chemicals are (such as TFA, DMF, etc.) or the exact operation of a certain piece of equipment is not necessary to modeling the process. However, they are essential resources and by adding this information, the resulting model is:

```
process peptide_synthesis {
   action swell_resin {
      requires { TFA && DMF && resin }
      requires { scintillation_vile && vacuum }

      provides { swelled_resin }
   }

   iteration {
      action activate_amino_acid {
         requires { DIEA && HBTU && amino_acid }

         provides { activated_amino_acid }
      }
      action add_amino_acid {
         requires { activated_amino_acid }
         requires { scintillation_vile && swelled_resin }

         provides { resin_amino_acid_complex }
      }
      action flow_wash {
         requires { DMP && vacuum }
         requires { resin_amino_acid_complex }

         provides { purified_resin_amino_acid_complex }
      }
      action cleave_boc_group {
         requires { TFA && purified_resin_amino_acid_complex }
         requires { vacuum }

         provides { deprotected_resin_amino_acid_complex }
      }
      action flow_wash {
         requires { DMP && vacuum }
         requires { deprotected_resin_amino_acid_complex }

         provides { purified_deprotected_resin_amino_acid_complex }
      }
   }

   action deswell_resin {
      requires { purified_deprotected_resin_amino_acid_complex }
      requires { MEOH && CH2CL2 && vacuum }

      provides { deswelled_resin_amino_acid_complex }
   }
   action vacuum_pump {
      requires { deswelled_resin_amino_acid_complex }
      requires { vacuum_pump }

      provides { dried_deswelled_resin_amino_acid_complex }
   }
   action cleave_peptide {
      requires { dried_deswelled_resin_amino_acid_complex }
      requires { HF && HF_cleavage_device }

      provides { crude_peptide }
```

```
    }
    action lyophilize_peptide {
       requires { crude_peptide && MEOH && dry_ice && lyophilizer }

       provides { lyophilized_peptide }
    }
}
```

Clearly this revision of the model is more complicated than the first, but it provides essential information about what is needed and produced at each step in the process. However, this model exhibits problems discussed in Section 4.1.2 where new resources are being created at every action to describe changes that take place. Though the model is comprehensible, it does not reflect the changes to resources in the process. The majority of actions in this model are modifying the `resin_amino_acid_complex` rather than creating a new resource, but with just resources, it is not possible to represent these types of changes.

Choosing what objects are represented as resources can be challenging because of how they relate to other resources and how they change throughout the process. The scintillation vile, resin, and amino acids have a complex relationship because each resource begins as a separate entity, but as the process progresses, they are combined. The scintillation vile holds the resin and amino acids as they are being fused together, so the modeler is faced with the problem of describing this relationship. In this situation, it makes sense to create a new resource to describe the combination of these components to simplify the model and to describe how the resources are being collectively modified. Many actions later in the model do not contribute additional objects to this combined resource, but modify some property, so it does not make sense to create a new resource each time the resource is altered.

The control specification of the model also appears to have problems. Amino acid chains can have two or more amino acids linked together, but there is no specification of how long the chain is or how many times the iteration should be repeated and in what order the amino acids should be added to the compound. In addition, the amino acid resource is ambiguous in that it could mean that the same amino acid is added each time or it could mean a new acid is added. There are various

methods for dealing with these problems. One method is to eliminate the iteration construct and explicitly specify the steps to add each amino acid, but this is an inelegant solution. If each addition of an amino acid means that the same steps must be rewritten then the resulting model will be much longer than necessary. If this is were the only possible solution, then PML would not have the level of expressiveness needed to model this process. Another possibility is that it is improper to treat this situation as one would a loop in programming.

One of the objectives of the process model is to represent the actual process and guide the person performing the process. However, this does not mean that the model should be so detailed that it can foresee every course of action. If this were the case, then it would be a programming or work-flow problem. Though it is possible that a machine could be designed to replicate each step in the process of making a peptide, it would only be able to make the peptides that it was designed to make. The general process for making a peptide is much more abstract, which is why human interaction is necessary. Therefore, the process should not necessarily be concerned with how many and in what order amino acids should be added because that is the responsibility of the person synthesizing the peptide. PML is designed to model the process, not force the enactor to strictly follow a defined number of instructions. Enacting a process is a collaborative effort between the model and the person performing the process and both have certain responsibilities that must be fulfilled for the process to be completed.

4.4.3 Detailed Specification Model

In Section 4.4.2 we noted that the first revision of the model did not provide adequate representation of the process, though it did illustrate the flow of resources and dependencies. Using language constructs from PML, such as attributes, expressions, and qualifiers, we can eliminate spurious resources and reduce the ambiguity of the model.

Initially, we attempted to use attributes to replace statements, such as:

```
provides { swelled_resin }
```

with a more precise notation:

```
provides { resin.status == ``swelled'' }
```

Though this clearly depicts that the resin was only modified rather than transformed into a new resource, it introduces a number of conceptual problems. If the status attribute of the resin is describing that resin is swelled, then how does changing the attribute affect the resource? If we wanted to change the state from swelled to deswelled, then it is obvious that we want to replace the attributes value. However, if we wanted to change the state from swelled to purified, then there could be two desired results: either the status is no longer swelled or the state is both swelled and purified. Trying to consolidate these two possible interpretations produces a confusing model, so we developed a more intuitive solution that eliminates this confusion. Instead of relying on one attribute to describe the state of a resource, a variety of attributes should be used. Therefore, if we want to specify that the resource has two properties simultaneously we can state:

```
provides { resin.swelled == ``true'' && resin.purified == ``true'' }
```

While using attributes in this manner solves the problem of spurious resources, describing how the scintillation vile, resin, and amino acids are combined and how the amino acids should be treated within the iteration requires the use of qualifiers. Qualifiers allow additional specification of qualities of resources and how they should be handled in the process. In this example, we want to illustrate that the scintillation vile, resin, and amino acids are combined to form a new resource. One solution is to declare that the new resource is *derived* from other resources:

```
action add_amino_acid {
   requires { amino_acid.activated == ``true'' }
   requires { scintillation_vile && resin.swelled == ``true'' }

   provides { (derived) resin_amino_acid_complex }
}
```

Applying the same construct to the amino acid problem we can state:

```
action activate_amino_acid {
   requires { DIEA && HBTU && (new) amino_acid }

   provides { amino_acid.activated == ``true'' }
}
```

This extra level of specification does not actually modify the resource, but describes a quality of the resource, such as an amino acid being new to the chain.

Combining all of these enhancements results in a very detailed model of the process, but this does not mean that more detail is not possible. This level of detail was chosen because it describes the process by including resources that are necessary for the process and abstracts less important aspects. Much more detail could be provided about resources and the properties they have, but there is a point where the detail can overshadow the process, which reduces the comprehensibility of the model. After applying all of these enhancements, the resulting model is:

```
process peptide_synthesis {
   action swell_resin {
      requires { TFA && DMF && resin }
      requires { scintillation_vile && vacuum }

      provides { resin.swelled == ''true'' && scintillation_vile }
   }

   iteration {
      action activate_amino_acid {
         requires { DIEA && HBTU && (new) amino_acid }

         provides { amino_acid.activated == ''true'' }
      }
      action add_amino_acid {
         requires { amino_acid.activated == ''true'' }
         requires { scintillation_vile && resin.swelled == ''true'' }

         provides { (derived) resin_amino_acid_complex }
      }
      action flow_wash {
         requires { DMP && vacuum }
         requires { resin_amino_acid_complex }

         provides { resin_amino_acid_complex.purified == ''true'' }
      }
      action cleave_boc_group {
         requires { TFA && purified_resin_amino_acid_complex }
         requires { vacuum }

         provides { resin_amino_acid_complex.deprotected == ''true'' &&
               resin_amino_acid_complex.purified == ''false'' }
      }
      action flow_wash {
         requires { DMP && vacuum }
         requires { resin_amino_acid_complex.purified == ''false'' }

         provides { resin_amino_acid_complex.purified == ''true'' }
      }
   }

   action deswell_resin {
      requires { resin_amino_acid_complex.purified == ''true'' &&
            resin_amino_acid_complex.deprotected == ''true'' }
```

```
        requires { MEOH && CH2CL2 && vacuum }

        provides { resin_amino_acid_complex.deswelled == ''true'' }
    }
    action vacuum_pump {
        requires { resin_amino_acid_complex.deswelled == ''true'' }
        requires { vacuum_pump }

        provides { resin_amino_acid_complex.dried == ''true'' }
    }
    action cleave_peptide {
        requires { resin_amino_acid_complex.dried == ''true'' }
        requires { HF && HF_cleavage_device }

        provides { (derived) peptide.crude == ''true'' }
    }
    action lyophilize_peptide {
        requires { peptide.crude == ''true'' && MEOH && dry_ice && lyophilizer }

        provides { peptide.lyophilized == ''true'' }
    }
}
```

Chapter 5

Tool Motivation

5.1 Why Tools Are Needed

Using a process modeling language to recreate an actual process is a complex procedure because the modeler must extract important information about tasks, resources, and control in such a way that the model will properly reflect the process. However, the resulting model often contains errors that can be attributed to two sources: the process and the modeler. Errors that are contained within the process are problematic in that they represent some inefficiency or mistake in the process that could result in any number of problems including slow performance or even preventing the process from continuing after it reaches a certain point. Problems introduced by the modeler represent human error by either improperly representing the process, or making a typographical error that has repercussions throughout the process. With the help of tools that look for these errors, models can be more efficient and accurate.

Previously we noted that modeling languages implemented using programming languages have inherited tool support for checking errors in models, but these tools are not specifically designed for process related errors. Compilers perform type-checking, look for undeclared variables, and other syntactic errors. The problem with using these methods is that they do not represent the kind of

errors that occur in a process model. Therefore, we need to explore the types of errors that might occur in a process and how they would be reflected in a model.

5.2 Modeler Errors

When a model for a process is developed there are a number of errors that can be introduced by the modeler that result in an improper representation of the process. Understanding how these errors are introduced and what implications they have on the model will provide the means to detect the errors and provide meaningful feedback. Without eliminating these errors, it is difficult to discern between errors created by the modeler and errors in the process. In order to improve the quality of the process, the model first must correctly represent the process at which point the actual process can be examined by means of the model. Therefore, we must examine what common errors are made in modeling a process and how they can be extracted from the model.

5.2.1 Modeling Errors at High Levels of Abstraction

The evolutionary nature of process modeling may result in a series of problems 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. At this level, the errors that can be introduced by a modeler are simple and include problems such as syntax errors caused by misunderstanding of the language structure or typographical mistakes such as misspellings. These errors have a very limited impact on the model and are easily recognizable because of the lack of information present at such a high level of abstraction.

5.2.2 Effects of Modeling Errors on Dependencies

Transitioning to a lower level of abstraction incorporates adding resources to the model which begins the development of dependencies and makes a considerable number of errors related to modeling

possible. Errors at this stage of development already have a significant impact on the model. While syntactic errors are still problematic, misspellings and dependency errors cause more difficult problems. 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 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. In this situation, the model represents a step in the process that needs resources but does not produce anything useful to the process and is also a common oversight when transitioning from a high level of abstraction to a more detailed level.

5.2.3 Effects of Modeling Errors on Control Flow

Dependencies at low levels of abstraction have a direct impact on the control of the process, which can lead to difficulties in trying to satisfy both control flow and dependencies put in place by the modeler. If the modeler wants to specify that two steps in the process are concurrent, but unintentionally creates a dependency that would prevent concurrency, such as having the first concurrent thread rely on a product of the second concurrent thread, then the model would not represent the real process. In the worst case scenario it is possible to introduce a circular dependency between two concurrent steps which would result in preventing the process from continuing from that point. This type of error is the result of either not understanding the dependencies of the process or over-specification of concurrency within the process.

Other control flow aspects of a model are compromised by common modeling errors. If there

are many possible paths in the process, but only one can be taken, then fulfilling dependencies is critical for the modeler. If the modeler notes that a step after a path selection ends depends on a product that is produced during the path selection, then all possible paths must produce that resource or the modeler has introduce the potential for a stall in the process. As possible paths become more numerous and more complicated, it is difficult to track what is produced and where it will be available.

5.2.4 Modeling Errors in Detailed Specifications

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. All of the modeling errors previously mentioned still apply at this level, but there are additional problems that can be introduced. 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.

There is also the potential to introduce errors by unintentionally creating inconsistent states within the model. If the property of a resource is specified to be greater than five, but later requires it to be less than ten, then there is the possibility that the actual value of the property will not fulfill the requirement of a step. This type of mistake is subtle and easy to overlook, but can have catastrophic effects and possibly stall the process. Identifying these types of errors can help to avoid misrepresenting the process, which could lead to confusion because the process itself can contain similar errors.

5.3 Process Errors

One of the goals of process modeling is to use the model to find errors in the actual process and eliminate them to make the process more efficient, but process related errors are closely related to modeling errors except their source is the from inefficiencies in the process rather than mistakes made in the representation. Understanding how process related errors are manifested in the model and how they differ from modeling errors will create a foundation for recognizing and extracting the errors from both the process and the model.

5.3.1 Resources Within Processes

Modeling a process at high levels of abstraction is generally simple because there are few constraints about what is correct and we are only bound by intuitive interpretation of the process. The highest level of abstraction usually incorporates naming the tasks that must be performed and providing a basic control for the process. The only process related errors that can occur at this level is the misspecification of the control flow in the process. However, there are no constructs at this level that would show that a problem has been introduced, but as the model is refined, the errors will become apparent.

Once resources and dependencies are introduced to the model, errors in the process are easier to find and result from a variety of sources. If something is created in the process but is never used, there is extra work being done that can be eliminated. For example, if a company is producing documents that are never used, then there is no reason to expend the effort in creating them. Another problem is that some step in the process may be waiting for a resource that will never be available, such as a loan application that requires a credit approval form that does not exist. These errors are intrinsic to the process and represent the type of problems that process modeling can eliminate.

5.3.2 Process Errors in Control Flow

Much like the modeling errors that impact the process model, actual errors in the process being modeled can have control errors. These fall into the same categories as the errors that a modeler can introduce. The process can be trying to perform concurrent steps that have interdependencies that prevent them from being fully concurrent. If there are multiple paths that can be taken, the process may not be producing all of the necessary resources that are used in later stages of the process.

5.4 Tool Goals

The primary objective of a tool designed to analyze a process model is to examine the model for the types of errors illustrated in Section 5.2 and Section 5.3. In order to fulfill this objective, there are a number of requirements that a tool must fulfill and failing to meet these requirements is detrimental to the tool's usefulness.

Meaningful Feedback: Meaningful feedback is necessary for a modeler to understand what is wrong with the model and determine if the root cause of the error the model or process. The tool should attempt to constructively map the errors in the model to conceptual errors in the real process. Having a tool that simply reports that there are unfulfilled dependencies provides only limited help in diagnosing the problem with the model. If the tool can identify what steps and resources are affected in the process, then the modeler has much more information about the root cause and should be able to extrapolate the errors in the model to errors in the process.

Analysis Refinement: The evolutionary nature of process modeling languages requires that tools that support the language should 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 by giving errors that are not related to context of the model. Therefore, supplementary tools need to be adjustable in order to compensate for the various levels of specification in a process model.

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. The tools associated with the language should reflect principles of the language such as simplicity and flexibility. Failing to meet these requirements can result in a tool that analyzes correctly, but is never used because of poor design and inability to meet the demands of the language.

Chapter 6

Tool Design and Implementation

6.1 Analyzing and Representing a Process Model

In Chapter 5, we described errors that can be introduced and how these types of errors are manifested within a model. To develop a means to check for these types of errors automatically requires understanding of these errors affect the model and developing a proper structure for representing the model that will allow for effective analysis.

6.1.1 Checking for Errors Local to a Task

In the previous chapter, we uncovered errors caused by a failure to express what resources a task produces or requires. As a model specification becomes more detailed, failure to note the needed resources is one of the first problems that can occur. These errors are local to a task in the model, which means that information from other tasks are not necessary to determine if something is missing. The only consideration for these types of errors is to determine that each task both produces and requires at least one resource. If a task fails to accomplish this, then the modeler should be notified to inform him that there is a possible inconsistency in the model. However, as mentioned in Section 5.2.2, it is possible that this condition is intentionally introduced by the modeler to represent

a step that does not rely on resources. This type of condition is relatively rare and it is not possible for an automatic tool to determine the modeler's intentions in this type of situation and therefore should be considered an error.

6.1.2 Analyzing Resource Dependencies

Another problem described in Section 5.2.2 is requiring resources that are not available or producing a resource that is never used. As resource dependencies are integrated into the model, errors of this nature become a concern and tracking these types of errors requires information related to the dependencies created in the model. To ensure that a resource is available to a specific task, previous tasks must be examined to determine that the resource exists. If a resource is being provided, then the tasks following the current task must be examined to determine if the produced resource is ever used by another task. This type of analysis requires that the representation the model takes will provide information about the order in which the tasks occur.

The complexity of this problem is compounded by the various control flow constructs that exist within modeling languages. If there are paths that are performed concurrently, then it is important to check other paths for resources that are being used or produced. Selections between multiple paths can create a situation where a resource is only available if the proper path is chosen. These problems must be taken into consideration when analyzing the model to prevent the tool from falsely assuming the availability of resources.

6.1.3 Evaluating Expressions

The addition of expressions to a modeling language complicates the level of checking that must be performed. In Section 4.1.3 we explained that expressions merely assert the value of some property, which makes evaluating expressions very difficult. The actual value of a property is never defined by the language, but are determined by the actual tasks being performed in the process. This means that the model will only represent as much information about the value of properties as the modeler

specifies. The result is that a tool will have difficulty tracking the value of a property if the model does not provide specific information about how the property is changing. For example, if a task asserts some property of a resource to be five, and a later task asserts that the same property is greater than four, it is not possible for the tool to determine if the actual value of the property is still five, or if it was changed. The advantage of this type of checking is that a model that passes all of these types of checks is very well described, but the disadvantage is that the modeler may consider the level of description too detailed to incorporate expressions into the model.

6.1.4 Model Representation

Based on the type of error checking involved in analyzing a process model, we can determine the information that is necessary to perform these checks. A process model can be mapped to a variety of data structures such as trees, hash tables, sets, or graphs. Independent of what structure is used to describe the model, there are certain aspects that must be preserved. The checks that are local to a task require information about each task and what resources are used and produced. Dependency checks rely on control based information to track dependencies throughout the process. Therefore, components that are essential to this type of checking include: being able to associate resources with the tasks that use and produce them, knowledge of the structure of the model including what nodes come before or after other nodes, and control based information describing concurrency, iteration, and path selection. Using this information as a framework, there are many approaches to check for these types of errors.

6.2 Process Model Representation with PML

6.2.1 Model Translation

PML 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

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. Figures 6.1 – 6.4 illustrate the graph translation of PML control flow constructs.

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

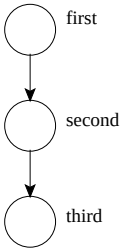

Figure 6.1: Graph representation of a sequence

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

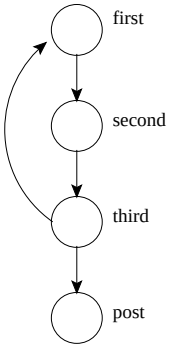

Figure 6.2: Graph representation of an iteration

Figure 6.5 illustrates how the waterfall process described in Section 4.3.1 would be mapped from a PML model to a graph that can be used to analyze the process. The graph is directed and contains additional links between nodes designed to assist in traversing the graph.

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. In addition, conjunctions and disjunctions of expressions and resources are described using the same structure. Figure 6.6 illustrates how a qualified resource with an attribute is constructed from the PML gram-

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

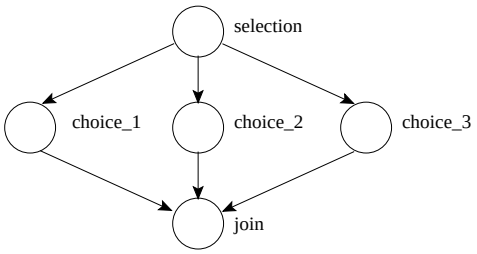

Figure 6.3: Graph representation of a selection

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

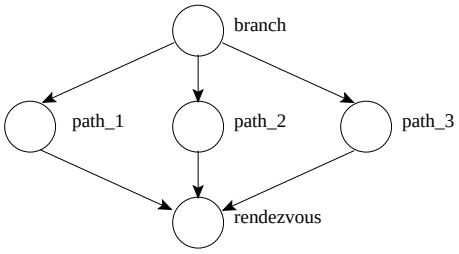

Figure 6.4: Graph representation of a branch

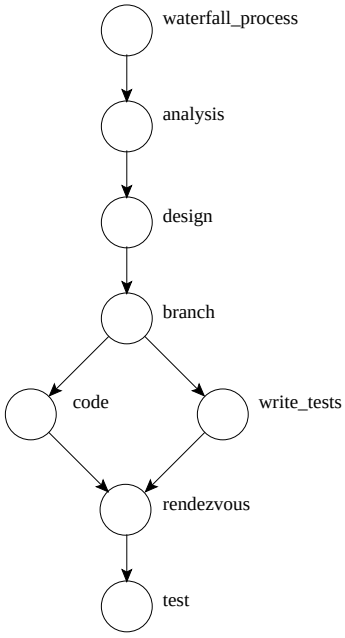

Figure 6.5: Software process model from Section 4.3.1

mar. Expressions, conjunctions, and disjunctions extend the tree to join resources together through common parents.

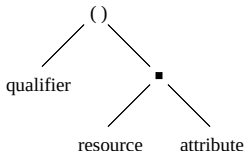

Figure 6.6: Tree representation of a resource

There are a number of concerns addressed by this method of representation. One consideration was that more than one tool would need a representation of the process model. Having a structure that facilitates only one tool is inefficient and possibly detrimental in that tools that choose to map the model to some other structure may improperly represent the model or fail to capture all the necessary information. Having a common and complete representation available ensures that any tool support will have a common view of the model.

6.3 Analysis and Verification of PML Process Models

In order to automate these types of checks on PML models, we have developed *pmlcheck*; a tool for analyzing the dependencies and structure of the model in an effort to uncover inconsistencies. This tool implements the graph mapping of a process model described in the previous section to traverse the model and look for common modeling and process errors.

6.3.1 Implementing Refinable Error Checking

One of the objectives of an automated analysis tool described in Section 5.4 is the ability to check a process at many levels of abstraction. Pmlcheck currently provides four conceptual levels of checking:

Syntax: This level of checking is provided as a means to ensure that the process model is well-formed, but it does not provide any meaningful feedback about the process itself.

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.

Dependency: Introducing resources into the model creates dependencies between actions. At this level, the dependences can be examined to determine if any were omitted or misrepresented.

Expression: 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 pmlcheck can focus analysis on a particular point of interest. This flexibility was intentionally designed to reflect the evolutionary nature of process specification and the PML language while providing the modeler with control over information gathered by the tool.

6.3.2 Reducing and Simplifying the Process Specification

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 expressions in expression trees. A complex expression that incorporates negation might look something like:

```
!( (q) a.b != c.d || !(a.b >= (q) e.f) )
```

The tree constructed from this expression is represented in Figure 6.7

Through applying DeMorgan's Law and manipulating the signs in the expression, it is possible to effectively eliminate negations. The objective of this operation is to push the negation as far

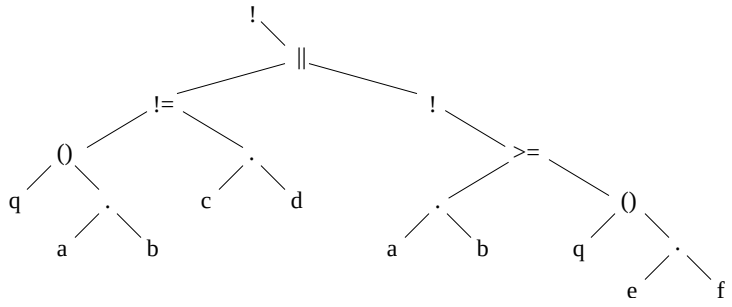

Figure 6.7: Tree representation of a complex expression

down in the expression tree as possible, thus alleviating the need to handle them at various levels in the tree structure. Reducing Figure 6.7 results in the tree illustrated in Figure 6.8.

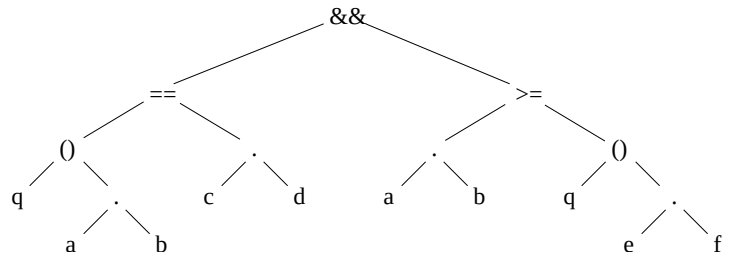

Figure 6.8: Tree after reducing negation

The second method of reduction is canonicalizing the expressions to simplify analysis. When designing pmlcheck, we considered two forms of canonicalization. The first option was to restructure the trees so that resources would always appear on the left hand side of an expression and numbers and strings would appear on the right hand side. Because these expressions are merely assertions and not performing assignments, the expressions are symmetric, which allows for manipulation of the trees without fear of introducing inconsistencies. However, performing this type of canonicalization is ineffective in this language because the semantics allow for resources to be present on both sides of an expression. In addition, the weak semantics of PML allow for strings and numbers to be present on both sides of an expression. Reducing expression trees with this approach does not significantly reduce the complexity of evaluating these expressions.

The other canonicalization method is to reduce the number of operators by reforming the trees and altering the existing operators, but without altering the meaning of the expression. Pmlcheck reduces the number of operators from six to four, which significantly reduces the number of comparison cases. The operators $\geq$ and $>$ are replaced with $<$ and $\leq$ respectively after swapping the children of the expression tree node. By performing this method of of canonicalization on Figure 6.8, the resulting tree is provided in Figure 6.9

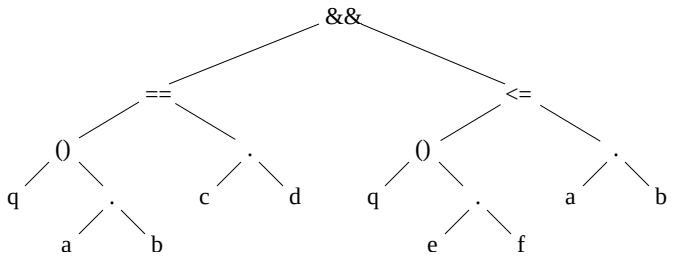

Figure 6.9: Canonicalized tree from Figure 6.8

6.3.3 Checking for Local Resource Specification Errors

In Section 6.1.1, 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:

Miracle: This situation is when an action requires no resources, but provides a resource.

Black Hole: When an action requires resources, but does not provide any.

Transformation: The most common operation of an action is to modify a resource. Having an action that provides a resource that it did not require means that the action created a new resource rather than modifying an existing one and is considered a transformation.

Empty: When an action does not specify any resources to be required or provided.

Each of these scenarios is an indicator that something has been left out of the process model and is a projection of problems in the process. 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.

In Section 4.3.1 we noted that 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. Recall that 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 additional checking is a more complicated feature of pmlcheck, it allows the analysis tool to match the flexibility of the language.

6.3.4 Dependency Checking

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. These two checks encompass model inconsistencies noted in Section 5.2.2 and Section 5.3.1. Another key component to understanding how pmlcheck performs these operations is how resources are treated within the model.

Resources in PML can be specified in two ways: as an individual resource, or a resource accompanied by an attribute. If a resource is provided or required without an attribute, then the existence of the resource is the only concern. However, if an attribute is provided with a resource,

then the resource must exist in order to have an attribute associated with it. Pmlcheck describes the availability of a resource to exist as one of three possible states: specified, partially specified, and unspecified. If a resource is specified, then the resource exists as does any attribute associated with it. Being partially specified means that the resource component exists, but the attribute does not. If neither the resource nor the attribute exist, the resource is unspecified. These states are the foundation for dependency-based checks in pmlcheck.

Initially, we tracked the availability of resources using a single value to express the state at any point in the process, but after examining various models, it became clear that more information needed to be gathered. Not only does a resource have a level of specification, it has 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 three states: true, maybe, and false.

The first check performed by pmlcheck is to ensure that resources that are required by an action are provided in an earlier action. Algorithm 1 describes the basic procedure for determining if a required resource exists. The function *is-provided()* examines the structure of the resource tree provided by a node and determines whether the required resource has been specified. This algorithm assesses the availability of a resource by looking at the current node and its predecessor to determine if a dependency has been fulfilled. By using the ϕ_S function, the current node can determine the status of a resource's specification at a specific point in the process and, similarly, the ϕ_A function determines the availability of the resource. ϕ_S and ϕ_A use a decision procedure based on the context of the current node to determine what the resulting status of the resource is. For example, $\phi_S(unspecified, specified)$ in the proper context will result in *partially specified*. A similar decision procedure is used for control flow constructs such as branches and selections, where all predecessors of the branch or selection are examined. This check is accomplished by performing a breadth first search of the graph in order to achieve a reversed traversal of the control flow of the process graph.

Algorithm 1 Algorithm for checking that required resources are provided

```
for N ∈ nodes[G] do
  for R ∈ requires[N] do
    for V ∈ nodes[G] do
      visited[V] := false
      available[V] := unknown
      specification[V] := unknown
    end for
    check-if-provided (R, N, N)
  end for
end for


function check-if-provided (R, N, S) do
  visited[N] := true
  available[N] := unknown
  specification[N] := unknown

  if N ≠ S then
    specification[N] = is-specified (R, provides[N], N)
    if specification[N] ≠ unspecified then
      available[N] := true
    end if
  end if

  if specification[N] ≠ specified and available[N] ≠ true then
    for P in predecessors[N] do
      if visited[P] ≠ true then
        check-if-provided (R, P, S)
        availale[N] := φ_A(available[N], available[P])
        specification[N] := φ_S(specification[N], specification[P])
      end if
    end for
  end if
end function
```

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 to Algorithm 1 and is described in Algorithm 2. This algorithm performs a breadth first 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 check using ϕ_S.

Algorithm 2 Checking that provided resources are required

```
for N ∈ nodes[G] do
  for R ∈ provides[N] do
    for V ∈ nodes[G] do
      visited[V] := false
      available[V] := unknown
      specification[V] := unknown
    end for
    check-if-required (R, N, N)
  end for
end for


function check-if-required (R, N, S) do
  visited[N] := true
  available[N] := unknown
  specification[N] := unknown

  if N ≠ S then
    specification[N] = is-specified (R, provides[N], N)
    if specification[N] ≠ unspecified then
      available[N] := true
    end if
  end if

  if specification[N] ≠ specified and available[N] ≠ true then
    for P in predecessors[N] do
      if visited[P] ≠ true then
        check-if-required (R, P, S)
        availale[N] := φ_A(available[N], available[P])
        specification[N] := φ_S(specification[N], specification[P])
      end if
    end for
  end if
end function
```

In Section 4.3.2, we described the need to specify the inputs and outputs to a process. In many cases the initial and final nodes in a process either make use of a resource that does not preexist 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 the linking file. This allows the modeler to specify inputs and outputs of a process which 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 requirement being checked.

6.3.5 Evaluating Expressions

Evaluating expressions follows a similar procedure for checking dependencies but is directed toward the satisfiability of the expressions specified in the model. Earlier we stated 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.

In order to evaluate these types of expressions a function, *is-satisfied()*, was designed to compare two expressions and determine if one expression satisfies another. There are three possible results of this function: never satisfied, sometimes satisfied, and always satisfied. An expression that is never satisfied is a contradictory expression, such as comparing $r.a == 0$ and $r.a == 1$. Expressions were some values satisfy the expression, but others do not, are sometimes satisfied, such as $r.a \geq 0$ and $r.a > 0$. When two expressions are equivalent, the expression is always satisfied.

Algorithm 3 describes the a method for checking expressions, which is very similar to Algorithm 1 in information tracking and traversal of the graph. The function ϕ_E has a similar operation to ϕ_A and ϕ_S in that the current state of an expressions satisfiability is determined using the value from the current and previous nodes.

Algorithm 3 Algorithm for checking that expressions are satisfied

```
for N ∈ nodes[G] do
  for E ∈ requires[N] do
    for V ∈ nodes[G] do
      visited[V] := false
      available[V] := unknown
      satisfied[V] := unknown
    end for
    check-if-satisfied (E, N, N)
  end for
end for


function check-if-satisfied (E, N, S) do
  visited[N] := true
  available[N] := unknown
  satisfied[N] := unknown

  if N ≠ S then
    satisfied[N] = is-satisfied (E, provides[N], N)
    if satisfed[N] ≠ always then
      available[N] := true
    end if
  end if

  if satisfied[N] ≠ always and available[N] ≠ true then
    for P in predecessors[N] do
      if visited[P] ≠ true then
        check-if-provided (E, P, S)
        available[N] := φ_A(available[N], available[P])
        satisfied[N] := φ_E(satisfied[N], satisfied[P])
      end if
    end for
  end if
end function
```

Chapter 7

Analysis and Results

One of the primary goals of modeling a process is to acquire more information about inconsistencies in resources and performance. In Chapter 5 we examined the motivation for analyzing models and in Chapter 6 we determined the type of problems that exist in model and how they can be detected. The objective of this chapter is to illustrate how a model checker can be utilized to guide refinement of a process model through verification.

7.1 The Netbeans Requirements and Release Process

Netbeans is an IDE for Java developers based on an open source development model. The process used 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 stage 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 Netbeans process was developed in conjunction with the Netbeans team and is given in Appendix A.1. For clarity and to facilitate the analysis of the process, we have removed any additional information from the process that is not necessary for the verification tool to operate. Table 7.1 provides a summary of the actions involved in the Netbeans process.

7.2 Refining the Netbeans process

Analysis of this model 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 demonstrate how to improve the quality and consistency of the model by removing errors without adversely affecting the underlying process.

7.2.1 Analysis Local to Actions

The first application of pmlcheck reveals a significant number of errors in the process and are summarized in Table 7.2. Each of these errors is representative of a error in the model, the process, or the analysis tool. Empty, miracles, and black hole errors generally indicate that resources are

Netbeans Requirements and Release Process	
Requirements Stage	ReviewRoadmap SetReleaseDate ReviewNetBeansVisionStatmenet ReviewUncompletedMilestonesFromPreviousRelease ReviewIssuzillaFeatureRequests CompileListOfPossibleFeaturesToInclude CategorizeFeaturesProposedFeatureSet SendMessageToCommunityForFeedback ReviewFeedbackFromCommunity ReviseProposalBasedOnFeedback PostFinalDevelopmentProposalToNetBeansWebsite AssignDevelopersToCompleteProjectMilestones SetFeatureFreezeDate SetMilestoneCompletionDates
Release Stage	EmailSolicitationForReleaseManager WaitForVolunteer CollectCandidacyNominations EstablishReleaseManagerConsensus AnnounceNewReleaseManager SolicitModuleMaintainersForInclusionInUpcomingRelease ChangeBuildBranchName MakeInstallTar BuildBinaryReleases UploadInstallTarFilesToWebRepository UpdateWebPage MakeReadmeInstallationNotesAndChangelog SendReleaseNotificationToCommunity ExecuteAutomaticTestScripts ExecuteManualTestScripts ReportIssuesToIssuezilla UpdateStandingIssueStatus PostBugStats ExamineTestReport WriteBugFix VerifyBugFix CommitCodeToCvsCodeRepository UpdateIssuezillaToReflectChanges CompleteStabilization

Table 7.1: Summary of actions in the Netbeans process model

Initial Inconsistencies	
Empty	2
Miracle	2
Black Hole	6
Transformation	32
Unprovided	24
Not Consumed	20

Table 7.2: Summary of errors indicated by pmlcheck

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 `NetBeansRoadmap.Reviewed`. 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 situation where the tool is improperly reporting an inconsistency because the release date is derived from the road-map. In Section 4.3.1 we describe a method for qualifying resources in order to notify tools about how they should be treated. 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.

7.2.2 Verification of Resource Dependencies

Pmlcheck is reporting 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 over looked the creation of a resource. In this model, the action `ReviewFeedbackFromCommunity` requires `FeebackMessagesOnMail`, 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. In Section 4.3.2, we noted that there is a utility to indicate that a resource is an input to the process and will be considered available even though it is not specified in the process itself. Identifying the resources that should be considered inputs to the process and specifying them using a linking file will allow the modeler to concentrate on resource dependency problems within the process.

Pmlcheck also reports resources that are provided by an action but are not used later in the process. 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 analysis tool from improperly reporting errors.

7.2.3 Consolidating Resources

In Section 4.1.2, 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
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.

7.3 Revised Netbeans Model

After applying the types of changes described in the previous section along with some cosmetic changes of names throughout the process, we arrive at the model given in Appendix B.1 with Appendix B.2 as a linking description. Applying pmlcheck a second time reveals that the number of reported errors is much lower than in the original model and the results are surmised in Table 7.3. Examining these remaining 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 actually solve the problem. Restructuring the resource to be handled as an attribute to the Issuzilla Issue Repository would eliminate the error without creating extraneous outputs to the process.

Remaining Inconsistencies	
Empty	0
Miracle	0
Black Hole	0
Transformation	1
Unprovided	7
Not Consumed	12

Table 7.3: Summary of errors in revised model

7.4 Finalized Netbeans Model

Correcting the mistakes found in the previous revision results in the model given in Appendix C.1 with Appendix C.2 as the linking file. This model produces no errors from the analysis tool, which ensures that the tool is satisfied with the way the dependencies are built. Though this does not indicated that there are no problems in the process, the problems that have indicators have been effectively removed.

Chapter 8

Related Work

Process modeling language development, analysis, and enactment is an active field in software engineering, but despite numerous approaches to process improvement, no single solution has emerged as a standard.

8.1 Existing Modeling Languages

There are many existing modeling languages designed to model software processes and each approach addresses different aspects of modeling based on the designers particular approach. Some languages emphasize the design aspect of modeling a process while others closely resemble work-flow systems by focusing on automation.

APPL/A [Sutton, 1990] 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 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.

The modeling language JIL [Sutton and Osterweil, 1997] aims to recreate many of the functionalities of languages such as APPL/A, but without the underlying programming language. JIL is designed with a combination of proactive and reactive control constructs allowing the modeler to define the control flow, or have it determined by the interpreter. High-level constructs designed to reflect software development products and relations are integrated in the language. Exception handling, inspired by APPL/A, is integrated in the language to provide reactive control of the process. A graphical extension, Little-JIL [Cass et al., 2000], attempts to simplify the process of designing by providing a clear graph based description of the model.

Merlin [Junkermann et al., 1994] is a rule-based language that constructs the process by assessing the availability of resources and documents associated with process activities. This language is process-centered, as opposed to enactment-centered, in that it is focused on specifying the elements of the process and not the control. Using decision procedures similar to PROLOG, the process is constructed to maximize concurrency. This design was based on the need for flexible process enactment and the nondeterministic nature of processes.

8.2 Process Model Analysis

Process modeling is conceptually very different from programming but there are many similarities between analysis of a process model and a software program. Tools such as gcc [Stallman, 1991] and lint [Johnson, 1978] 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. Additionally, optimizations can be applied, such as moving loop-invariant code outside of a loop.

Analysis of programs represented as graphs is a well-established field including algorithms for understanding and computing properties of graphs. There are many tools designed to assist programmers in finding errors in programs and include program slicing tools and assertion checkers.

8.3 Process Validation

Cook and Wolf [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 [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].

Cobleigh, Clarke, and Osterweil [Cobleigh et al., 2000] utilize verification tools such as FLAVORS [Cobleigh et al., 2002], a finite state verification system, to determine whether the behavior of a process is consistent with specific properties. This tool operates on both sequential and concurrent control-flow graphs to exhaustively check properties during process execution.

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 language research.

Chapter 9

Conclusion

In this work we have presented a philosophy of modeling based on the fundamental elements of domain-independent processes with the intention of highlighting the essential components of processes in order to create informative models for analysis and enactment. We utilized this philosophy as a framework for designing a high-level language that has the expressive capability to model processes at abstract and concrete levels of specification. This language has a number of features such as qualifiers and process linking that allows flexible development and specification. However, the consequence of constructing this new language is lack of tool support and modeling processes for the purpose of improvement requires verification of the model.

In order to provide support for PML, we chose to implement a new method of process checking based on our research into process structure. The resulting 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 Netbeans process that we examined and refined in Chapter 7 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. We assert that tools such as pmlcheck provide a necessary function in the design and development of processes models regardless of the domain.

9.1 Open Issues

Checking for inconsistencies in a model provides some level of assurance that the model is properly specified, but much more information can be gathered from the process and provides several opportunities for future work.

Examining the interdependencies between resources and control flow often result in optimizations that will increase the efficiency of the process. One possible optimization is automatically determining if two steps in a process can be performed concurrently. If there are no dependences between two successive tasks (meaning that neither task requires something that the other is providing), then it is possible to automatically restructure the graph so that the tasks are performed in parallel.

The same method can be applied for detecting and restructuring processes to conform to the limitations enforced by the dependencies. This means detecting dependencies within concurrent actions and restructuring the process to explicitly constrain and redesign the control flow to a sequential operation.

Bibliography

Agostini, A. and Demichelis, G. (2000). A light workflow management system using simple process models. *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 9(3–4):335–363.

Armenise, P., Bandinelli, S., Ghezzi, C., and Morzenti, A. (1993). A survey and assessment of software process representation formalisms. *International Journal of Software Engineering and Knowledge Engineering*, 3(3):401–426.

Atkinson, D. C. and Noll, J. (2003). Automated validation and verification of process models. In *Proceedings of the 7th IASTED International Conference on Software Engineering and Applications*.

Barnes, J. (1998). *Programming in Ada 95*. Addison-Wesley Pub Co.

Cass, A. G., Lerner, B. S., McCall, E. K., Osterweil, L. J., Sutton, Jr., L. J., and Wise, A. (2000). Little-JIL/Juliette: A process definition language and interpreter. In *Proceedings of International Conference on Software Engineering*, pages 754–757.

Cobleigh, J. M., Clarke, L. A., and Osterweil, L. J. (2000). Verifying properties of process definitions. In *Proceedings of the ACM SIGSOFT 2000 International Symposium on Software Testing and Analysis*, pages 96–101.

Cobleigh, J. M., Clarke, L. A., and Osterweil, L. J. (2002). FLAVERS: A finite state verification technique for software systems. *IBM Systems Journal*, 41(1):140–165.

Conradi, R. and Liu, C. (1995). Process modelling languages: One or many? In Schäfer, W., editor, *Proceedings of 4th European Workshop on Software Process Technology*, pages 98–118. Springer–Verlag.

Cook, J. E. and Wolf, A. L. (1999). Software process validation: Quantitatively measuring the correspondence of a process to a model. *ACM Transactions on Software Engineering and Methodology*, 8(2):147–176.

Cugola, G. and Ghezzi, C. (1998). Software processes: A retrospective and a path to the future. *Software Process Improvement and Practice*, 4(3):101–23.

Dami, S., Estublier, J., and Amiour, M. (1998). APEL: A graphical yet executable formalism for process modeling. *Automated Software Engineering*, 5(1):61–96.

J. M. Rib, X. F. (2000). PROMENADE: A PML intended to enhance standardization, expressiveness and modularity in software process modelling. Technical report, Universitat De Lleida.

Joeris, G. and Herzog, O. (1999). Towards flexible and high-level modeling and enacting of processes. In *Proceedings of 11th International Conference on Advanced Information Systems Engineering*, volume 1626, pages 88–102.

Johnson, E. W. and Brockman, J. B. (1998). Measurement and analysis of sequential design processes. *ACM Transactions on Design Automation of Electronic Systems*, 3(1):1–20.

Johnson, S. C. (1978). Lint, a C program checker. In *Unix Programmer's Manual*. AT&T Bell Laboratories.

Junkermann, G., Peuschel, B., Schäfer, W., and Wolf, S. (1994). Merlin: Supporting cooperation in software development through a knowledge-based environment. In *Software Process Modelling and Technology*, pages 103–129. Research Studies Press Ltd.

Kaiser, G. E., Popovich, S., and Ben-Shaul, I. Z. (1993). A bi-level language for software process modeling. In *Proceedings of the 15th International Conference on Software Engineering*, pages 132–143. IEEE Computer Society Press.

Klingler, C. D. (1994). A STARS case study in process definition. Technical Report F19628-88-D-0031.

Klingler, C. D., Neviaser, M., Marmor-Squires, A., Lott, C. M., and Rombach, H. D. (1992). A case study in process representation using MVP-L. In *Proceedings of the 7th Annual Conference on Computer Assurance*, pages 137–146.

Minas, M. and Hoffmann, B. (2001). Specifying and implementing visual process modeling languages with diagen. In Ehrig, H., Ermel, C., and Padberg, J., editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier.

Noll, J. and Scacchi, W. (2001). Specifying process-oriented hypertext for organizational computing. *Journal of Network and Computer Applications*, 24(1):39–61.

Pinheiro da Silva, P. (2001). A proposal for a LOTOS-based semantics for UML. Technical Report UMCS-01-06-1, Department of Computer Science, University of Manchester, Manchester, UK.

Pressman, R. S. (1992). *Software Engineering: A Practitioner's Approach*. McGraw-Hill.

Scacchi, W. (2000). Understanding software process redesign using modeling, analysis and simulation. *Software Process Improvement and Practice*, 5(2–3):183–195.

Scacchi, W. and Mi, P. (1997). Process life cycle engineering: A knowlege-based approach and environment. *International Journal of Intelligent Systems in Accounting, Finance, and Management*, 6(2):83–107.

Scacchi, W. and Noll, J. (1997). Process-driven intranets: life-cycle support for process reengineering. *IEEE Internet Computing*, 1(5):42–49.

Stallman, R. M. (1991). *GCC Reference Manual*. Free Software Foundation, Cambridge, MA.

Sutton, Jr., S. M. (1990). *APPL/A: A Prototype Language for Software-Process Programming*. PhD thesis, University of Colorado.

Sutton, Jr., S. M. (1995a). Accounting for purpose in specifying requirements for process programs. Technical Report UM-CS-1995-076.

Sutton, Jr, S. M. (1995b). Preconditions, postconditions, and provisional execution in software processes. Technical Report UM-CS-1995-077.

Sutton, Jr., S. M., Heimbinger, D., and Osterweil, L. J. (1995). APPL/A: A language for software-process programming. *ACM Transactions on Software Engineering and Methodology*, 4(3):221–286.

Sutton, Jr., S. M., Lerner, B. S., and Osterweil, L. J. (1997). Experience using the JIL process programming language to specify design processes. Technical Report UM-CS-1997-068, University of Massachusetts.

Sutton, Jr., S. M. and Osterweil, L. J. (1997). The design of a next-generation process language. In *Proceedings of the 6th European Software Engineering Conference*, volume 22, pages 142–158.

van der Aalst, W. M. P. (1999). Woflan: A petri-net-based workflow analyzer. *System Analysis - Modelling - Simulation*, 43(3):345–357.

Appendix A

Initial Netbeans Requirements and Release Model

A.1 Model Specification

```
process RequirementsAndRelease {
  sequence Requirements {
    sequence SetProjectTimeline {
      action ReviewNetBeans {
        requires { NetBeansRoadmap }
        /*provides { }*/
      }
      action SetReleaseDate {
        requires { NetBeansRoadmap }
        provides { ReleaseDate }
      }

      sequence DetermineProject {
        branch SunONEStudioDevelopmentMeeting {
          action ReviewNetBeansVisionStatmenet {
            requires { NetBeansVisionStatement }
            /*provides { }*/
          }
          action ReviewUncompletedMilestonesFromPreviousRelease {
            requires { PreviousVersionReleaseDocuments }
            provides { ProspectiveFeaturesForUpcomingRelease }
          }
          action ReviewIssuzillaFeatureRequests {
            requires { IssuzillaIssueRepository }
            provides { ProspectiveFeaturesForUpcomingRelease }
          }
        }

        iteration EstablishFeatureSet {
          action CompileListOfPossibleFeaturesToInclude {
          requires { ProspectiveFeaturesGatheredFromIssuezilla &&
            ProspectiveFeaturesFromPreviousReleases }
          provides { FeatureSetForUpcomingRelease }
          }
          action CategorizeFeaturesProposedFeatureSet {
            requires { FeatureSetForUpcomingRelease }
            provides { WeightedListOfFeaturesToImplement }
          }
          action SendMessageToCommunityForFeedback {
            requires { WeightedListOfFeaturesToImplement }
            /*provides { }*/
          }
          action ReviewFeedbackFromCommunity {
            requires { FeebackMessagesOnMail }
            provides { PotentialRevisionsToDevelopmentProposal }
          }
          action ReviseProposalBasedOnFeedback {
            requires { PotentialRevisionsToDevelopmentProposal }
            provides { RevisedDevelopmentProposal }
          }
        }

        action PostFinalDevelopmentProposalToNetBeansWebsite {
          requires { RevisedDevelopmentProposal }
```

```
        provides { FinalDevelopmentProposal }
      }
      action AssignDevelopersToCompleteProjectMilestones {
        requires { RevisedDevelopmentProposal }
        /*provides { }*/
      }
    }

    sequence SetReleaseStageCompletionDates {
      action SetFeatureFreezeDate {
        requires { ReleaseDate }
        provides { FeatureFreezeDate }
      }
      action SetMilestoneCompletionDates {
        requires { FeatureFreezeDate && ReleaseDate }
        provides { MilestoneCompletionDates }
      }
    }
  }

  sequence EstablishReleaseManager {
    action EmailSolicitationForReleaseManager {
      /*requires { }*/
      provides { ReleaseManagerRequest }
    }
    action WaitForVolunteer {
      /*requires { }*/
      /*provides { }*/
    }

    iteration CollectCandidacyNominations {
      action SendCandidacyAnnouncement {
        requires { ReleaseManagerRequest }
        provides { ReleaseManagerCandidacyAnnouncement }
      }
    }

    action EstablishReleaseManagerConsensus {
      requires { ReleaseManagerCandidacyAnnouncements }
      provides { ReleaseManagerDecision }
    }
    action AnnounceNewReleaseManager {
      requires { ReleaseManagerDecision }
      provides { ReleaseManagerAnnoucementToNbdevMailingList }
    }
  }

  action SolicitModuleMaintainersForInclusionInUpcomingRelease {
    requires { FeatureFreezeDate }
    provides { ModuleInclusionNoticeToNbdevMailingList}
  }
}

sequence Release {
  iteration Stabilization {
    sequence Build {
      action ChangeBuildBranchName {
        requires { CvsCodeRepository }
        provides { NewBranchForCurrentBuild }
      }

      iteration MakeInstallTar {
        action MakeInstallTarForEachPlatform {
          requires { DevelopmentSourceForEachPlatform }
          provides { InstallExecutableTar }
        }
      }
    }

    sequence Deploy {
      action UploadInstallTarFilesToWebRepository {
      requires { BinaryReleaseDownloads && WebRepository }
      /*provides { }*/
      }
      action UpdateWebPage {
        requires { ProjectWeb }
        provides { UpdatedWeb }
```

```
        }
        action MakeReadmeInstallationNotesAndChangelog {
          requires { ChangesFromIndividualModules}
          provides { README && InstallationNotes && Changelog}
        }
        action SendReleaseNotificationToCommunityInvitingTheCommunityToDownloadAndTestIt {
          /* requires { }*/
          provides { ReleaseNotice }
        }
      }

      sequence Test {
        action ExecuteAutomaticTestScripts{
          requires { TestScripts && ReleaseBinaries}
          provides { TestResults}
        }

        action ExecuteManualTestScripts{
          requires { ReleaseBinaries}
          provides { TestResults}
        }

        iteration UpdateIssuezilla{
          action ReportIssuesToIssuezilla{
            requires { TestResults}
            provides { IssuezillaEntry}
          }
          action UpdateStandingIssueStatus{
            requires { StandingIssueFromIssuezilla && TestResults}
            provides { UpdatedIssuezillaIssueRepository}
          }
        }

        action PostBugStats{
          requires { TestResults}
          provides { BugStatusReport && TestResultReport}
        }
      }

      sequence Debug {
        action ExamineTestReport {
          requires { TestReport && BugStats }
          /* provides { }*/
        }
        action WriteBugFix {
          requires { ErroneousSource }
          provides { PotentialBugFix }
        }
        action VerifyBugFix {
          requires { PotentialBugFix }
          provides { WorkingBugFix }
        }
        action CommitCodeToCvsCodeRepository {
          requires { WorkingBugFix && CVSCodeRepsository}
          provides { UpdatedSource}
        }
        action UpdateIssuezillaToReflectChanges{
          requires { IssuezillaIssueRepository}
          provides { UpdateIssueStatus}
        }
      }
    }

    action CompleteStabilization {}
  }
}
```

Appendix B

First Revision of Netbeans Model

B.1 Model Specification

```
process RequirementsAndRelease {
  sequence Requirements {
    sequence SetProjectTimeline {
      action ReviewRoadmap {
        requires { Roadmap }
        provides { Roadmap.Reviewd }
      }
      action SetReleaseDate {
        requires { Roadmap }
        provides { (derived) ReleaseDate }
      }

      sequence DetermineProject {
        branch SunONEStudioDevelopmentMeeting {
          action ReviewNetBeansVisionStatmenet {
            requires { VisionStatement }
            provides { VisionStatement.reviewed }
          }
          action ReviewUncompletedMilestonesFromPreviousRelease {
            requires { PreviousVersionReleaseDocuments && ProspectiveFeatures }
            provides { (derived) ProspectiveFeatures.PreviousVersions == "complete" }
          }
          action ReviewIssuzillaFeatureRequests {
            requires { IssuzillaIssueRepository && ProspectiveFeatures }
            provides { (derived) ProspectiveFeatures.Issuzilla == "complete" }
          }
        }

        iteration EstablishFeatureSet {
          action CompileListOfPossibleFeaturesToInclude {
            requires { ProspectiveFeatures.Issuezilla && ProspectiveFeatures.PreviousVersions }
            provides { (derived) ReleaseFeatureSet }
          }
          action CategorizeFeaturesProposedFeatureSet {
            requires { ReleaseFeatureSet }
            provides { ReleaseFeatureSet.weighted }
          }
          action CreateDevelopmentProposal {
            requires { ReleaseFeatureSet.weighted }
            provides { (derived) DevelopmentProposal }
          }
          action SendMessageToCommunityForFeedback {
            requires { ReleaseFeatureSet.weighted && DevelopmentProposal && CommunityMailingList }
            provides { (derived) CommunityFeedback }
          }
          action ReviewFeedbackFromCommunity {
            requires { CommunityFeedback && DevelopmentProposal }
            provides { DevelopmentProposal.PotentialRevisions }
          }
          action ReviseProposalBasedOnFeedback {
            requires { DevelopmentProposal.PotentialRevisions }
            provides { DevelopmentProposal.Revised }
          }
        }
```

```
      action PostFinalDevelopmentProposalToNetBeansWebsite {
        requires { DevelopmentProposal.Revised }
        provides { DevelopmentProposal.Finalized }
      }
      action AssignDevelopersToCompleteProjectMilestones {
        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 { CommunityMailingList }
      provides { (derived) ReleaseManagerRequest }
    }

    iteration CollectCandidacyNominations {
      action SendCandidacyAnnouncement {
        requires { ReleaseManagerRequest }
        provides { (derived) ReleaseManagerCandidacyAnnouncement }
      }
    }

    action EstablishReleaseManagerConsensus {
      requires { ReleaseManagerCandidacyAnnouncements }
      provides { (derived) ReleaseManagerDecision }
    }
    action AnnounceNewReleaseManager {
      requires { ReleaseManagerDecision }
      provides { (derived) ReleaseManagerAnnoucementToNbdevMailingList }
    }
  }

  action SolicitModuleMaintainersForInclusionInUpcomingRelease {
    requires { FeatureFreezeDate }
    provides { (derived) ModuleInclusionNoticeToNbdevMailingList}
  }
}

sequence Release {
  iteration Stabilization {
    sequence Build {
      action ChangeBuildBranchName {
        requires { CvsCodeRepository }
        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 { ReleaseBinaries.Uploaded }
      }
```

```
        action UpdateWebPage {
          requires { Webpage }
          provides { Webpage.Updated }
        }
        action MakeReadmeInstallationNotesAndChangelog {
          requires { Modules.Changes }
          provides { (derived) README && (derived) InstallationNotes && (derived) Changelog }
        }
        action SendReleaseNotificationToCommunityInvitingTheCommunityToDownloadAndTestIt {
          requires { CommunityMailingList }
          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 ReportIssuesToIssuezilla {
            requires { TestResults.ScriptResults && TestResults.ManualResults }
            provides { (derived) IssuezillaEntry }
          }
          action UpdateStandingIssueStatus {
            requires { IssuezillaIssueRepository.StandingIssues && TestResults }
            provides { IssuezillaIssueRepository.Updated }
          }
        }

        action PostBugStats {
          requires { TestResults }
          provides { (derived) BugStatusReport && (derived) TestReport }
        }
      }

      sequence Debug {
        action ExamineTestReport {
          requires { TestReport && BugStatsReport }
          provides { TestReport.Examined }
        }
        action WriteBugFix {
          requires { BuildBranch }
          provides { (derived) PotentialBugFix }
        }
        action VerifyBugFix {
          requires { PotentialBugFix }
          provides { (derived) WorkingBugFix }
        }
        action CommitCodeToCvsCodeRepository {
          requires { WorkingBugFix && CVSCodeRepsository }
          provides { CVSCodeRepository.Updated }
        }
        action UpdateIssuezillaToReflectChanges {
          requires { IssuezillaIssueRepository }
          provides { IssuezillaIssueRepository.Updated }
        }
      }
    }

    action CompleteStabilization {
      requires { ReleaseBinaries && TestReport && Webpage}
      provides { (derived) FinalRelease && Webpage.Updated}
    }
  }
}
```

B.2 Linking Specification

```
input { Roadmap }
input { VisionStatement }
input { PreviousVersionReleaseDocuments }
input { IssuzillaIssueRepository }
input { CommunityMailingList }
input { CvsCodeRepository }
input { WebRepository }
input { Webpage }
input { Modules }


output { VisionStatement }
output { Webpage.Updated }
output { IssuezillaIssueRepository.Updated }
output { CVSCodeRepository.Updated }
output { FinalRelease }
output { Webpage.Updated }
```

Appendix C

Final Revision of Netbeans Model

C.1 Model Specification

```
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 ReviewNetBeansVisionStatmenet {
            requires { VisionStatement }
            provides { VisionStatement.Reviewed }
          }
          action ReviewUncompletedMilestonesFromPreviousRelease {
            requires { Documentation.PreviousVersionRelease }
            provides { (derived) ProspectiveFeatures.PreviousVersions == "complete" }
          }
          action ReviewIssuzillaFeatureRequests {
            requires { IssuzillaIssueRepository }
            provides { (derived) ProspectiveFeatures.Issuezilla == "complete" }
          }
        }

        iteration EstablishFeatureSet {
          action CompileListOfPossibleFeaturesToInclude {
            requires { ProspectiveFeatures.Issuezilla &&
                 ProspectiveFeatures.PreviousVersions }
            provides { (derived) ReleaseFeatureSet }
          }
          action CategorizeFeaturesProposedFeatureSet {
            requires { ReleaseFeatureSet }
            provides { ReleaseFeatureSet.weighted }
          }
          action CreateDevelopmentProposal {
            requires { ReleaseFeatureSet.weighted }
            provides { (derived) DevelopmentProposal }
          }
          action SendMessageToCommunityForFeedback {
            requires { ReleaseFeatureSet.weighted &&
                 DevelopmentProposal &&
                 CommunityMailingList }
            provides { (derived) CommunityFeedback }
          }
          action ReviewFeedbackFromCommunity {
            requires { CommunityFeedback && DevelopmentProposal }
            provides { DevelopmentProposal.PotentialRevisions }
          }
          action ReviseProposalBasedOnFeedback {
            requires { DevelopmentProposal.PotentialRevisions }
```

```
            provides { DevelopmentProposal.Revised }
          }
        }

        action PostFinalDevelopmentProposalToNetBeansWebsite {
          requires { DevelopmentProposal.Revised }
          provides { DevelopmentProposal.Finalized }
        }
        action AssignDevelopersToCompleteProjectMilestones {
          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 { CommunityMailingList }
        provides { (derived) ReleaseManagerRequest }
      }

      iteration CollectCandidacyNominations {
        action SendCandidacyAnnouncement {
          requires { ReleaseManagerRequest }
          provides { (derived) ReleaseManagerCandidacyAnnouncement }
        }
      }

      action EstablishReleaseManagerConsensus {
        requires { ReleaseManagerCandidacyAnnouncement }
        provides { (derived) ReleaseManagerDecision }
      }
      action AnnounceNewReleaseManager {
        requires { ReleaseManagerDecision }
        provides { (derived) ReleaseManagerAnnoucementToNbdevMailingList }
      }
    }

    action SolicitModuleMaintainersForInclusionInUpcomingRelease {
      requires { FeatureFreezeDate }
      provides { (derived) ModuleInclusionNoticeToNbdevMailingList}
    }
  }

  sequence Release {
    iteration Stabilization {
      sequence Build {
        action ChangeBuildBranchName {
          requires { CvsCodeRepository }
          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 { ReleaseBinaries.Uploaded }
      }
      action UpdateWebPage {
        requires { Webpage }
        provides { Webpage.Updated }
      }
      action MakeReadmeInstallationNotesAndChangelog {
        requires { Modules.Changes }
        provides { (derived) README && (derived) InstallationNotes && (derived) ChangeLog }
      }
      action SendReleaseNotificationToCommunityInvitingTheCommunityToDownloadAndTestIt {
        requires { CommunityMailingList }
        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 ReportIssuesToIssuezilla {
          requires { TestResults.ScriptResults && TestResults.ManualResults }
          provides { (derived) IssuezillaIssueRepository.StandingIssues }
        }
        action UpdateStandingIssueStatus {
          requires { IssuezillaIssueRepository.StandingIssues && TestResults }
          provides { IssuezillaIssueRepository.Updated }
        }
      }

      action PostBugStats {
        requires { TestResults }
        provides { (derived) BugStatusReport && (derived) TestReport }
      }
    }

    sequence Debug {
      action ExamineTestReport {
        requires { TestReport && BugStatusReport }
        provides { TestReport.Examined }
      }
      action WriteBugFix {
        requires { BuildBranch }
        provides { (derived) PotentialBugFix }
      }
      action VerifyBugFix {
        requires { PotentialBugFix }
        provides { (derived) WorkingBugFix }
      }
      action CommitCodeToCvsCodeRepository {
        requires { WorkingBugFix && CvsCodeRepository }
        provides { CvsCodeRepository.Updated }
      }
      action UpdateIssuezillaToReflectChanges {
        requires { IssuezillaIssueRepository }
        provides { IssuezillaIssueRepository.Updated }
      }
    }
  }

  action CompleteStabilization {
    requires { ReleaseBinaries && TestReport && Webpage}
    provides { (derived) FinalRelease && Webpage.Updated}
  }
}
```

```
}
```

C.2 Linking Specification

```
input { Roadmap }
input { VisionStatement }
input { Documentation.PreviousVersionRelease }
input { IssuzillaIssueRepository }
input { CommunityMailingList }
input { CvsCodeRepository }
input { WebRepository }
input { Webpage }
input { Modules.Changes }


output { VisionStatement.Reviewed }
output { Webpage.Updated }
output { IssuezillaIssueRepository.Updated }
output { CvsCodeRepository.Updated }
output { FinalRelease }
output { Webpage.Updated }
output { README }
output { InstallationNotes }
output { ChangeLog }
output { ReleaseManagerAnnoucementToNbdevMailingList }
output { ModuleInclusionNoticeToNbdevMailingList }
output { DeveloperAssignments }
output { MilestoneCompletionDates }
output { ReleaseNotice }
```