The Design of Evolutionary Process Modeling Languages

Darren C. Atkinson, Daniel C. Weeks, and John Noll
Department of Computer Engineering
Santa Clara University
Santa Clara, CA 95053-0566 USA
{datkinson,dweeks,jnoll}@scu.edu

Abstract

To formalize a software process, its important aspects must be extracted as a model. Many processes are used repeatedly, and the ability to automate a process is also desired. One approach is to use a notation that already exists, such as a programming language, and extend it. However, the intricacies and restrictions the programming language places on the ability to succinctly and clearly describe a process can be problematic. An alternative approach is to develop a language specifically for describing processes. A significant disadvantage of this approach, however, is the lack of tool support for ensuring model correctness. We discuss a high-level language that encourages evolutionary model development and describe a tool for performing model verification. We have used our language and tool on the NetBeans model for distributed software development.

1. Introduction

Process descriptions [12] characterize the important aspects of processes from which models can be derived. One purpose of a model is to reflect the control-flow of the process without incorporating nonessential properties. 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 its behavior [1].

In addition to disambiguating a complicated process, having a written notation for process description provides the ability both to analyze the process by checking for errors and to automate it. Validating a process before enactment increases quality and ensures correctness. Automation increases efficiency and provides the facility to guide the process through its life-cycle, only stopping for human interaction when necessary.

In order to effectively check models for errors and to automate processes, a *formal* notation (i.e., language) is required to specify the model. The objective of the language is to be as expressive as an unstructured description, but changing the representation so it is unambiguous [4]. 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 be amenable to meaningful analysis or automation.

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 refine. Modeling allows the modeler to easily modify the process and determine if the changes are effective. Furthermore, processes are typically designed starting with abstract concepts and are iteratively refined into detailed descriptions. Therefore, the language used to describe a process needs to reflect this *evolutionary* development cycle, but still provide valuable information about the process at every level of abstraction. Finally, if the conceptual and procedural aspects of a process can be represented in a language, then tools can be designed to automatically check the models before enactment [8].

One common paradigm for modeling processes is rule-based or logical modeling [10, 13]. This method relies on rules to describe the tasks and then generates a model from the dependencies specified in the tasks. The main advantage of this approach is that the modeler need only specify individual tasks, and the associated tools will automatically generate a model with consistent dependencies.

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 an unnecessary layer of complexity. We feel this method is also counterintuitive to how people think about processes. The order in which tasks are performed is a primary concern when defining a process, and the modeler should be able to control it.

The paradigm that we advocate is control-based [3, 19], 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 [11]. At a high level of abstraction, the control is sequential, which allows the modeler to imply the dependencies without actually having to specify them. If it is later decided that the model should be more specific, the actual dependencies can be introduced. This method is more intuitive and reflects the steps that humans normally take.

2. Modeling language design

2.1. Goals and motivation

Although there are many different approaches to process modeling, there is a general consensus about the goals of modeling languages [15]. These goals embody how a language should capture the aspects of a process in order to represent the process properly. The most common goals are:

- *simplicity:* a non-technical person should be able to model a process without being encumbered by the syntactic or semantic requirements of the language
- *flexibility:* the language can be applied to a variety of applications such as business processes, software processes, or any other form of process equally
- *expressiveness:* the ability to accurately reflect a process is essential in order to extract useful information about the process
- *enactability:* the language should enable the model to mimic the actual execution of a process

Though these goals have been repeatedly defined and examined, many language implementations disregard these goals in an effort to achieve additional functionality [6].

The most common approach to designing a process modeling language is to build the language on top of an existing programming language, because of similar concepts and notations in both languages. A typical example of this approach is the language APPL/A [17, 18], which is designed as an extension to the programming language Ada.

There are many advantages to using this *bottom-up* approach to language design, most of which pertain to enactability. APPL/A was able to take advantage of features such as concurrency, iteration, modularity, information hiding, and exception handling that are integrated into Ada. 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. 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 a rich and varied syntax, resulting in a modeling language that is complex, not simple as intended. A person modeling a process may not need to know the meaning of each keyword 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 both languages.

Building on a programming language also limits 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 expressiveness of the modeling language (i.e., APPL/A) 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. Additionally, how the process will be enacted depends on the underlying programming language and how the process will execute once compiled. What may seem intuitive at the modeling level, may not be reflected at the programming language level.

Existing tools that support the language can also be problematic. The error checking capabilities of the Ada compiler 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 languages. 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 is not due to 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, and their target applications are not the same as those of process modeling languages. Though there are many similar concepts between programming and process modeling, there are subtle differences that separate the two. For example, although a program may be evolved, at each step the program must be semantically correct. In process modeling, the modeler may want to experiment with partially correct models to obtain feedback. 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 research domain.

2.2. Our approach

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 [2, 14], which intrinsically supports process-related concepts rather than implementing them in terms of concepts from a programming language.[1]

This *top-down* approach to language development has many advantages that address problems inherent in the bottom-up approach. PML is a simple language with only thirteen keywords. This design decision 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. Another positive aspect of PML is that the syntax is very straightforward and not impacted by that of a programming language. In PML, statements all follow a simple form that 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.

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. For example, the environment can rely on the user in making decisions regarding which action to perform next.

PML is also quite flexible in that it supports evolutionary model development. A high-level process model can be written easily and modified iteratively until the desired level of detail is obtained. To illustrate this ability, we present the syntax for PML in the context of the evolutionary development of a process model to describe the traditional waterfall model of software development.

1 The name PML is an acronym, originally for "Process Markup Language." The language quickly evolved to resemble a programming language, but the PML name was retained for historical reasons.

3. The PML language

3.1. Language fundamentals

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.

3.2. Resources and attributes

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. The only postulate for an action is that the resource is available when the process enters or exits the action. PML allows actions to require and provide resources, which reflects the action's need for or the production of a resource, but gives no indication of its origin or destination. Using these constructs, we can modify our model to provide more information about the internals of an action:

```
action analyze {
  requires { function && behavior && interface }
  provides { requirements && 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, and interface are available and exiting is not possible without requirements and analysis documentation. Using these predicates, a modeler can reconstruct the dependencies that exist within a process by specifying its pre- and postconditions.

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

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

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

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

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

3.3. Control constructs

PML has four mechanisms for describing the control of a process. These control-flow constructs reflect process-related activities and describe the ordering of steps in a process.

3.3.1. 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.

3.3.2. 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, and *repeat* until mixture is thick and consistent, which is clearly subjective. The syntax for an **iteration** follows the same structure as 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.

At first this 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. Also, there are many conditions in processes that are based on human judgment and cannot be evaluated by a machine.

3.3.3. Selection. 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.

This type of decision in process models cannot always be automatically determined and therefore may 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.

3.3.4. Branch. The **branch** construct specifies a set of concurrent 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. Unlike APPL/A, PML does not restrict the way that a rendezvous is handled. Instead, a PML interpreter must decide whether a synchronous or asynchronous rendezvous 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. Of course, artificial constraints in an asynchronous implementation can be introduced to recreate a synchronous rendezvous.

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 { }
}
```

3.4. Advanced language features

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, consider an action in a model that requires both design and funding. This action has two requirements that consist of some tangible resource. 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. Some languages provide keywords associating a resource with being consumed by an action [13]. 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. Furthermore, 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.

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, PML has a construct called a *qualifier*. The qualifier is used to describe characteristics or qualities of a resources that are beyond the scope of the language's regular syntax. 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, otherwise it will be ignored.

Using the language features of PML, we present a detailed, idealized waterfall process model in Figure 1. 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.

We started with a simple model to describe the waterfall process. By adding resources, attributes, expressions, and finally qualifiers, we gradually introduced more and more detail to make the original model more specific. At any point in this evolution, we could have stopped and used the existing model. For example, even the first model presented that consisted solely of actions is enactable. We feel that this type of evolutionary development of models reflects the top-down way in which people reason about and describe most processes, at least at an initial, conceptual level.

4. Model verification

4.1. Tool motivation

Using a process modeling language to recreate an actual process is a complex procedure because the modeler must extract important information about tasks, resources,

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

Figure 1. An elaborated waterfall model.

and control in such a way that the model will properly reflect the process. Consequently, 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 model. With the help of tools that look for these errors, models can be more efficient and accurate.

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 check for 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.

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

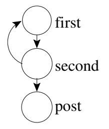
Transitioning to a lower level of abstraction incorporates adding resources to the model which begins the development of dependencies and may result in a considerable number of errors related to modeling. If the name of a resource is misspelled and another step in the model needs that resource, the dependency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

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. This type of error is the result of either not understanding the dependencies of the process or over-specification of concurrency within the process.

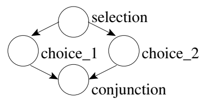
Other control-flow aspects of a model are compromised by common modeling errors. If there are many possible paths in the process, but only one can be taken, then fulfilling dependencies is critical for the modeler. If the modeler notes that a step after a path selection depends on a product that is produced during the path selection, then all possible paths must produce that resource or the modeler has introduced the potential for a stall in the process. As possible paths become more numerous and more complicated, it becomes difficult to track what is produced and where it will be available.

Once a process model has been effectively implemented at a level of resource specification, it is possible to transition to a lower level of abstraction that will illustrate constraints on the state of objects within the process. This level of abstraction is the most detailed and also the most error prone.

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



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



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

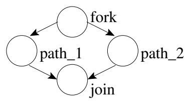


Figure 2. Graph representations of PML control constructs.

When transitioning to a detailed specification, the modeler must keep track of dependencies between the properties of resources as well as the resources themselves. The addition of properties to the model can disrupt the dependencies that were in place at higher levels of abstraction. For example, if the requirements for a step in the model are altered to include the state of a property, but the model fails to specify that the property was introduced by an earlier step, then the dependency between the two steps is broken.

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

- *meaningful feedback:* The tool should attempt to constructively map the errors in the model to conceptual errors in the real process.
- *analysis refinement:* The evolutionary nature of process modeling languages requires that supporting tools operate at each level of refinement in the development of the process model. If the analysis tool is reporting resource and dependency errors when the model is at a higher level of abstraction, then the analyzer has failed to meet the evolutionary requirements of the language.
- *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.

4.2. Tool design and implementation

Our tool is designed to translate a process model into a format that incorporates all aspects of the model and based on the structure of processes, the most intuitive representation is a graph. The procedure for mapping from a PML model to a graph is relatively simple; the nodes of a graph represent actions constructs and the edges represent the flow of control. The language constructs designed for describing control flow are interpreted and constructed into a graph in a syntax-directed, bottom-up manner as shown in Figure 2. 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.

One of the objectives of an automated analysis tool the ability to check a process at many levels of abstraction. Our tool, PMLCHECK, currently provides four conceptual levels of checking: syntax checking only, resource specification, resource dependencies, and expression satisfiability.

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.

We noted that inconsistencies may be introduced into a model because of a failure to specify requirements for a task. In PML, this translates to the failure to require or provide a resource in an action. These types of errors fall into four categories: those requiring and providing no resources ("*empty*"), those only requiring resources ("*black holes*"), those only providing resources ("*miracles*"), and those that provide resources other than those that they require ("*transformations*").

Each of these scenarios is an indicator that something has been left out of the process model and is a projection of problems discussed previously. Since all of these cases are local to an action, PMLCHECK simply examines each action in turn in order to find errors. However, there are legitimate 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 in fact 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.

In contrast, tracing dependencies through a 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 resources required by an action are provided, and provided resources are required by an action.

To implement both of these checks, PMLCHECK uses standard graph propagation algorithms to propagate the availability of resources through the control-flow graph. A resource available along only one path of a **selection** construct is marked as only possibly available. A similar check is performed for resources that are produced or consumed in concurrent actions in a **branch**.

Error Type	Initial	Revised	Final
Empty	2	0	0
Miracle	2	0	0
Black hole	6	0	0
Transformation	32	1	0
Unprovided	24	7	0
Unconsumed	20	12	0

Table 1. Summary of errors reported by PMLCHECK.

Finally, to implement the checks for expression satisfiability, PMLCHECK uses logical equalities to first rewrite the expressions into a canonical form to eliminate negations and many of the relational operators, thereby reducing total number of cases that need to be examined. Since expressions are limited only to logical and relational operations on resources and literals, satisfiability is simple to implement using a straightforward exhaustive algorithm as a unification-based approach is not required. (For full details, see [21].)

5. Experimental results

NetBeans is an IDE for Java, whose requirements and release process is based on a distributed open source development model, and is therefore different than a traditional software process. 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 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. 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.

Using the model from [7], analysis consisted of two levels of refinement in order to capture inconsistencies at different levels of abstraction. On first inspection of the model it is clear that the model is in a very basic state in that it includes control and resources, but no attributes or expressions. Through verification using PMLCHECK, we improved the quality and consistency of the model by removing errors without adversely affecting the underlying process.

The first application of PMLCHECK revealed a significant number of errors in the process and are summarized in Table 1. Empty actions generally indicate that resources are missing from the specification. For example, the empty 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 was 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.

"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. In Figure 3, the action SendMessageToCommunityForFeedback would intuitively imply that feedback is gathered from the community and thus should provide CommunityFeedback as a resource, as Figure 4 shows.[2] These types of oversights are a misrepresentation of the process, and PMLCHECK helped locate the 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. 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, as shown in Figure 3. This action is modifying two resources PotentialRevisionsToDevelopmentProposal and RevisedDevelopmentProposal, but these relate to a single resource: DevelopmentProposal. As shown in Figure 4, by consolidating these resources to a single resource and using attributes, we can

2 Due to space considerations, only extracts from the models are shown here. The complete models are approximately two hundred lines each and can be found in [21].

```
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 }
  }
}
```

Figure 3. Extract from original NetBeans model.

reduce the total number of resources. 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.

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. 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. PMLCHECK provides a simple mechanism for specifying the inputs and outputs of a process to suppress these types of errors.

After applying the types of changes described along with some cosmetic changes of names throughout the process, we arrive at a revised model with the number of errors shown in Table 1. Examining these remaining errors revealed that many were the result of changes made to the process including trivial errors resulting from case-sensitivity and misspellings. Applying the same techniques to our revised model resulted in a final model with no errors.

```
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 }
  }
}
```

Figure 4. Extract from revised NetBeans model.

6. Related work

APPL/A [17] is a process enactment language designed as a superset of Ada to maximize automation. Features specific to modeling that are not implemented in Ada are constructed as extensions to the language. The modeling language JIL [20] 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. While JIL is designed toward complete automation, PML supports user interaction to allow more dynamic models. Also, whereas JIL provides high-level constructs for modeling software processes, PML is not restricted to just the software process domain.

Cook and Wolf [5] 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 [9] use execution histories to validate models for predicting process cycle times. The focus of their work is on estimation rather than validation, and is thus concerned with control flow rather than resource flow.

Scacchi's research uses a knowledge-based approach to analyzing process models. Starting with a set of rules that describe a process setting and models, processes are diagnosed for problems related to consistency, completeness, and traceability. Conceptually, this work is closely related to ours; many of the inconsistencies uncovered by PMLCHECK are also revealed by Scacchi and Mi's *Articulator* [16].

7. Conclusion

We have presented a philosophy of modeling based on the fundamental elements of 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, PML, 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 that allows flexible development and specification. However, the consequence of constructing this new language is lack of tool support and modeling for the purpose of improvement requires verification of the model.

To provide support for PML, we implemented 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. 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. Finally, the model of the NetBeans process that we examined and refined illustrates many benefits of tool-guided analysis. Understanding the resource flow of a process provides useful information to improve the specification of a process and to note 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.

References

[1] P. Armenise, S. Bandinelli, C. Ghezzi, and A. Morzenti. A survey and assessment of software process representation formalisms. *Int. J. Softw. Eng. Knowl. Eng.*, 3(3):401–426, Sept. 1993.

[2] D. C. Atkinson and J. Noll. Automated validation and verification of process models. In *Proc. 7th IASTED Int. Conf. on Softw. Eng. Appl.*, pages 587–592, Cambridge, MA, Nov. 2003.

[3] A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, S. M. Sutton, Jr., and A. Wise. Little-JIL/Juliette: A process definition language and interpreter. In *Proc. 22nd Int. Conf. on Softw. Eng.*, pages 754–757, Limerick, Ireland, June 2000.

[4] R. Conradi and C. Liu. Process modelling languages: One or many? In *Proc. 4th Eur. Work. on Softw. Process Tech.*, pages 98–118, Noordwijkerhout, The Netherlands, Apr. 1995.

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

[6] G. Cugola and C. Ghezzi. Software processes: A retrospective and a path to the future. *Softw. Process Improv. Pract.*, 4(3):101–123, Sept. 1998.

[7] C. Jensen, W. Scacchi, M. Oza, E. Nistor, and S. Hu. A first look at the NetBeans requirements and release process. Technical report, Institute for Software Research, Feb. 2004.

[8] G. Joeris and O. Herzog. Towards flexible and high-level modeling and enacting of processes. In *Proc. 11th Int. Conf. on Adv. Inf. Syst. Eng.*, pages 88–102, Heidelberg, Germany, June 1999.

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

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

[11] G. E. Kaiser, S. Popovich, and I. Z. Ben-Shaul. A bi-level language for software process modeling. In *Proc. 15th Int. Conf. on Softw. Eng.*, pages 132–143, Baltimore, MD, May 1993.

[12] C. D. Klingler. A STARS case study in process definition. Technical Report F19628-88-D-0031, DARPA, 1994.

[13] C. D. Klingler, M. Neviaser, A. Marmor-Squires, C. M. Lott, and H. D. Rombach. A case study in process representation using MVP-L. In *Proc. 7th Annual Conf. on Comp. Assur.*, pages 137–146, Gaithersburg, MD, June 1992.

[14] J. Noll and W. Scacchi. Specifying process-oriented hypertext for organizational computing. *J. Netw. Comput. Appl.*, 24(1):39–61, Jan. 2001.

[15] R. F. Paige, J. S. Ostroff, and P. J. Brooke. Principles for modeling language design. *Inf. Softw. Technol.*, 42(10):665–675, July 2000.

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

[17] S. M. Sutton, Jr. *APPL/A: A Prototype Language for Software-Process Programming*. PhD thesis, University of Colorado, Department of Computer Science, Aug. 1990.

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

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

[20] S. M. Sutton, Jr. and L. J. Osterweil. The design of a next-generation process language. In *Proc. 6th Euro. Softw. Eng. Conf. and 5th ACM Symp. on Found. Softw. Eng.*, pages 142–158, Zurich, Switzerland, Sept. 1997.

[21] D. C. Weeks. Process modeling language design and model verification. Master's thesis, Santa Clara University, Department of Computer Engineering, June 2004.