Available online at www.sciencedirect.com
ScienceDirect
Information and Software Technology 49 (2007) 493–514
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

Tool support for iterative software process modeling

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

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

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

Abstract

To formalize a process, its important aspects must be extracted and described in a model. This model is often written in a formal language so that the process itself can be automated. Since models are often developed iteratively, this language should support this iterative development cycle. However, many existing languages do not. In this paper, we use an existing high-level process modeling language and present a tool that we have developed for supporting iterative development. We have used our tool to develop and refine a process model of distributed software development for NetBeans.
© 2006 Elsevier B.V. All rights reserved.

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

1. Introduction

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

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

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

The most common approach to designing a process modeling language is to build the language on top of an existing programming language. A typical example of this approach is the language APPL/A [29], which is designed as an extension to the programming language Ada. There are many advantages to using this *bottom-up* approach to language design, most of which pertain to enactability. APPL/A was able to take advantage of features such as concurrency, modularity, and exception handling that are

* Corresponding author.
E-mail address: atkinson@engr.scu.edu (D.C. Atkinson).

0950-5849/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2006.07.006

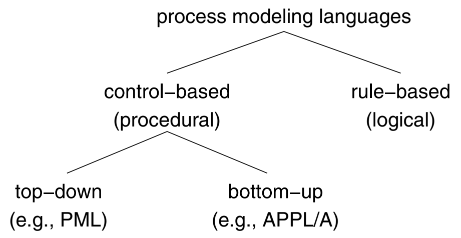

Fig. 1. Different process modeling language paradigms.

part of Ada. In addition, existing compilers provide type checking and error checking capabilities.

However, these same tools 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 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.

Instead of using existing languages to reflect processes, a language can be designed from scratch for process modeling. One such language is the modeling language PML [24,2], which intrinsically supports process-related concepts rather than implementing them in terms of concepts from a programming language.

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 a small set of 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. In brief, a PML program consists of actions that produce and consume resources. Actions are executed sequentially unless otherwise indicated by explicit control-flow constructs: **selection** for executing one of many paths, **branch** for executing many paths in parallel, and **iteration** for executing a sequence of actions repeatedly.

PML is well-suited to iterative 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, consider the iterative development of a process model to describe the traditional waterfall model of software development. Using only actions, it is possible to make a non-trivial model of the waterfall process:

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

Resources are an essential component to creating a process model that does more than just reiterate the steps in a process. The ability to describe the flow of resources allows the modeler to create a variety of dependencies that occur within a process. The only postulate for an action is that the resource is available when the process enters or exits the action. PML allows actions to require and provide resources, which reflects the action's need for or the production of a resource, but gives no indication of its origin or destination. Using these constructs, we can iteratively refine our model to provide more information about the internals of an action:

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

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

```
provides { analysis_documentation }
```

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

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

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

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

```
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