

A Knowledge-Based Software Process Library for Process-Driven Software Development

Peiwei Mi[†], Ming-June Lee[†] and Walt Scacchi[‡]

Computer Science Dept.[†] and Information and Operations Management Dept.[‡]

University of Southern California¹

Los Angeles, CA 90089-1421

{scacchi}@gilligan.usc.edu

Tel (213) 740-4782

Fax (213) 740-8494

Abstract

Process-driven software development represents a new technique for software production, in which a conceptual knowledge representation, called a software process, is used to represent and guide development activities. Management and reuse of software processes therefore becomes a requirement for process-driven software development. In this paper, we present a knowledge-based process library (SPLib) that supports the organization, access and reuse of software processes. SPLib consists of a knowledge base of software process representations. It also provides a set of process operations that support browsing, searching composition and abstraction. These operations reason about the content of software processes as well as maintain proper interdependency relationships among the software processes. To demonstrate the use of SPLib in process-driven software development, we provide a usage scenario where SPLib facilitates the access and reuse of software processes in real applications.

¹Acknowledgements: This work has been supported in part by contracts and grants from AT&T Bell Laboratories, Hewlett-Packard, and Northrop B-2 Division. No endorsement implied.

Contents

1	Introduction	1
2	Related Work	2
3	A Usage Scenario: A Customized Software Process	4
4	Requirements for SPLib	5
5	The SPLib Process Representation	6
5.1	A Two-Level Process Representation of SPLib	6
5.2	The SPLib Library Model	7
5.3	The SPLib Process Model	9
6	SPLib Process Operations	12
6.1	A Process Search-and-Query Operation	13
6.2	A Process Composition Operation	14
6.3	A Process Abstraction Operation	15
7	The Usage Scenario Revisited	16
8	Conclusion	20

1 Introduction

Process-driven software development represents a new technique for software production in which a conceptual knowledge representation, called a *software process*, is used to represent and guide development activities [MS92, Ost87]. During process-driven software development, process engineers first specify a software process that is tailored for project goals and other resource constraints, and then enact the process as a guide for developers. Software developers generally follow the process for the roles they play as to what development activities to perform and when to perform them. A process-driven CASE environment is also required to integrate process representation, data management, and tool invocation. Recent progress in software process modeling and process integration has made process-driven software development a very promising, yet realistic engineering technique for the software engineering community [Boe86, HK89, Kai88, MS90, MS91].

When knowledge-based process representations are utilized, management and reuse of the classes and instances of the representations becomes a necessity. To this end, a knowledge-based process library provides a solution the problem of process management and reuse. A knowledge-based process library is able to maintain a large collection of software process descriptions and interdependencies among them, just like the use of knowledge-based component libraries to software reuse. Based on a software process description, the process library can support query and retrieval to make access of software process easier and more convenient. More important and unique to process reuse, the process representation forms a foundation for advanced operations that generate new software processes out of existing ones. In sum, such a knowledge-based process library enhances the accessibility and reusability of existing process knowledge.

In this paper, we present a knowledge-based approach to organize, access and reuse software processes. We describe the initial design and prototype implementation for a knowledge-based software process library called SPLib. SPLib supports an extended version of the Articulator meta-model of software processes [MS90] and provides knowledge-based operations to access and reuse of software processes. As such, we first discuss two types of related work that lead to the SPLib: one that relates a knowledge-based approach to our study of software process reuse; the other that describes how software process modeling provides formalisms to describe and reason about software processes. Next, we provide a scenario for the process library in process-driven software development. Based on this scenario, we

identify and specify requirements for SPLib. Then, we describe the initial implementation of SPLib which consists of a process representation and process operations respectively. After this, we revisit the usage scenario to see how SPLib could be used. Finally, we conclude with a brief discussion of our ongoing work and future plans.

2 Related Work

Reuse of software components is an area where the ideas for component libraries and knowledge-based techniques are being investigated. One approach to software reuse is to construct and use source code libraries for software. For example, a standard software distribution package, such as X-windows, normally includes a large number of reusable functions and a library directory. This kind of function library consists of a list of the functions, textual description of their functionality, and a calling convention to invoke them. In this case, locating and determining which functions to use relies upon directory information and keyword/string search which is often cumbersome and problematic.

Subsequently, knowledge-based techniques are being investigated for more sophisticated support mechanisms. Bauhaus [AL89] is a knowledge-based software parts composition system shell. It has a knowledge base of reusable software component description, a catalog for browsing and editing the knowledge base, a composition editor for component specification, and a code generator for composed or tailored components. LaSSIE [DBe90] is a knowledge-based software information system. It has a frame-based knowledge representation for software objects and relations, and it provides functions to query and browse software objects. Software Components Catalogue [WS88] is another knowledge-based system for software reuse. It is an integrated component classification and retrieval system. It utilizes a conceptual dependency database describing software components and their relations, then matches users requests for software components with descriptions of components which satisfy these requests. Finally, it provides a natural language interface to specify user requests. In sum, the core of these knowledge-based software reuse systems includes a knowledge representation of software components, and a reasoning mechanism to search and match user requests for software components. The advantages of using a knowledge-base approach include: aggregation of information about individual components, semantic retrieval, use of classification and inheritance to support updates, and use of a knowledge base as an index.

The success of the knowledge-based approach to software reuse naturally leads to its use in software process reuse. However, reuse of software processes is different from reuse of software source code components. Software processes are typically more complex and related to more classes of development entities, such as developers in different engineering roles, multiple tools, multiple tasks, schedules, organizational policies and procedures, etc. Software processes are therefore require a rich and formal knowledge representation.

Modeling of software processes is a new research area that has emerged in recent years. Software process modeling originally started with informal and narrative descriptions, such as natural language, which is aimed at recording experiential knowledge about development processes. Frailey and Bate [FBe91] describe Texas Instruments' effort to define a corporate-wide software process for the last three years, which is documented by English following DOD or IEEE documentation standards. Ramesh and Dhar [RD91] describe an effort to record process knowledge through a semi-formal representation. The weakness of an informal or semi-formal representation lies in its inability to reason about process details, be symbolically executed, or downloaded into process-driven software engineering environments [MS92]. Therefore, informal process descriptions are limited to serve primarily as recording media, which often fall out of date.

Alternatively, knowledge representations of software processes have been introduced. For example, Grapple [HL88] uses a set of goal operators and a planning mechanism to represent software processes. These are used to demonstrated goal-directed reasoning about software processes. Marvel [Kai88] uses inference rules to model software processes. The condition part of a process rule identifies preconditions for a process to start and the action part of a process rule then describes the effect of the process and its outcome. The Articulator [MS90] describes software processes in terms of object classes and relations, such as task decomposition hierarchies. The defined process classes and relations form formal models of software processes, organizations, and resources, which are used to store process knowledge and simulate process enactment. The results of these efforts provide formalisms that describe basic characteristics of software process components, which in turn can be used to specify a formal representation of process components.

Subsequently, reuse of software processes requires not only knowledge-based retrieval, but also more advanced operations that can compose and tailor software processes to meet user requests. We now turn to discuss a scenario for using a process library where these

operations are needed.

3 A Usage Scenario: A Customized Software Process

To illustrate the use of SPLib in process-driven software development, we provide a usage scenario based on our experiences in modeling large-scale software processes for our industrial sponsors.

A major aerospace contractor is awarded a contract to build an aircraft and its flight control software. Before starting the software development, the contractor decides to specify a formal process model which describes development of the flight control software.

Development of the flight control system is constrained by several factors: First, it has to follow several national standards as indicated by the original contract-awarding agency. Second, the company has its own policies and procedures for how to do certain types of development. Third, the development has to address technological challenges that are unique to this flight control software. For the first two types of constraints, formal process descriptions at either the national level or the organizational level can be created that characterize the required development processes. Unfortunately, no pre-existing process models can meet the unique features. The task, therefore, is to construct a process model with following characteristics:

- it describes the production of the flight control software;
- it complies with the necessary national and organizational development standards;
- it specifies innovative approaches to implement the unique features.

SPLib should help the construction of the process model as follows: First, SPLib can represent and store development standards, methodologies, and the contractor's organizational policies in form of process descriptions. Second, knowledge-based search functions can be used to retrieve needed process description upon user requests. Third and more important, knowledge-based operations can be used to compose and tailor the involved processes to construct a desired process model.

This usage scenario suggests the ways SPLib enhances the organization, accessibility and reusability of software processes. This can lead to high quality software processes for process-driven software development. Later, we will revisit the scenario to investigate how

SPLib supports its construction tasks in terms of software processes and process operations. As such, we now use the scenario to identify requirements that the SPLib should satisfy.

4 Requirements for SPLib

As seen in the previous section, a knowledge-based process library should enhance the accessibility and reusability of software processes. Such knowledge is otherwise difficult to collect and organize for easy access and reuse. Based our analysis, the requirements for SPLib must specify access and reuse requirements.

SPLib should provide *readily available* software processes to a broad community of users while protecting proprietary information. SPLib should manage and store *prescriptive* development plans and *descriptive* development histories. Physically, SPLib could be *distributed* across a wide-area network that may span multiple organizational units and industrial firms, yet be accessible to specific projects or individuals where appropriate. To access software processes, SPLib should provide access operations such as upload, download, retrieval, and query.

SPLib should also facilitate construction of formally represented software processes, from either informal process descriptions or tailorable formal process descriptions. It should support a single formal process representation. All software processes in SPLib will be either described in or translated into the single process representation. Such a process representation should at the same time form an object-oriented hierarchy of software processes with multiple levels of details. For the moment, three levels of details are defined: national, organizational and project. More levels can be added when necessary. To reuse software processes, SPLib should provide operations, such as specialization, composition, abstraction, and tailoring.

Overall, SPLib can be structured as an object-oriented hierarchical collection of different types of formal process descriptions with a set of process operations to facilitate the use of software processes. While the access requirements provide similar capabilities to those required for software source code reuse, the reuse requirements for SPLib must facilitate more advanced forms of reuse, such as abstraction and composition. However, support for accessibility and reusability can be implemented separately. In other words, the access requirements can be implemented first, while the reuse requirements implemented later. Accordingly, we

now turn to discuss the SPLib process representation and operations respectively.

5 The SPLib Process Representation

The SPLib software process representation is an extended version of the Articulator meta-model we previously developed [MS90]. It consists of two parts: the *library model* is the extended part that describes interactions and interdependencies of software processes, the *process model* is the original Articulator meta-model that describes software processes themselves. These two parts represent two levels of descriptions. One represents interdependencies among software processes, the other represents the contents of software processes. SPLib has been prototyped using the Articulator process modeling environment [MS90] and Knowledge-Craft [Car86], which specifies objects and relations as schemata with attributes. In this section, we first present the reasons to have these two parts separate. Then we discuss the SPLib process representation, i.e. the library and the process model in detail. Process operations that manipulate both models appear in the next section.

5.1 A Two-Level Process Representation of SPLib

As we said before, the SPLib process representation has two parts: the library model describes interactions and interdependencies of software processes and the process model describes a formalism of software processes. Such a structure is determined by issues of software process modeling and accessing.

On the one hand, SPLib is a library of software processes. Users of SPLib are interested in knowing about and accessing different classes and levels of software processes through their relationships. This requires that software processes be queried, accessed and referenced as a whole without necessarily specifying their contents. To this end, the library model describes classes and instances of software processes, and their interdependencies to support this kind of usage.

On the other hand, searching and reasoning about software processes are not limited to the class level. Sometimes, search for processes and related resources is based on the content of software processes, such as their input and output resources. To this end, the process model provides a language specifying the attributes, values, and methods of software processes. The process model is also an abstract, fine-grained, and editable view of

the underlying software processes, which operations, such as search and composition, can manipulate.

As a result, process operations navigate between the library model and the process model to maximize their functionality. For instance, when a search request is specified, it can be implemented as a combination of library search to identify candidate software processes, and model search to infer properties of the candidate software processes. We will later illustrate the power of this two-level process representation in terms of process operations.

5.2 The SPLib Library Model

The SPLib library model describes classes of software processes and their relationships in terms of access and usage. It is different from the SPLib process model in that it deals only with software processes as classes and instances, while the process model identifies objects and relations that constitute a software process. In the SPLib library model, SPLib consists of an object-oriented hierarchy of interrelated process descriptions. There are *classes* of different software process *descriptions* at different *levels* that are linked through *relations* (Figure 1). In the figure, circles represent software process descriptions and lines are relations between software processes.

There are different classes of software processes in SPLib:

- A *process model* is a class of software development processes. It represents a type of software development approach or methodology, which can be instantiated for development, or specialized for different situations.
- A *planned instance* is a particular development plan created before software development. It represents a particular binding of actual agents, tools, and resources to the project plan, and serves as a developer role-specific guide during process-driven software development.
- A *history instance* is a particular process record that describes the trajectory of process-driven software development. It represents historical events that led to the completion or failure of the development activities and serves as a record of the process' enactment.

Figure 2 lists its schematic definition for classes of software processes. Attributes in the processes are either value attributes or relation attributes, which will be explained next.

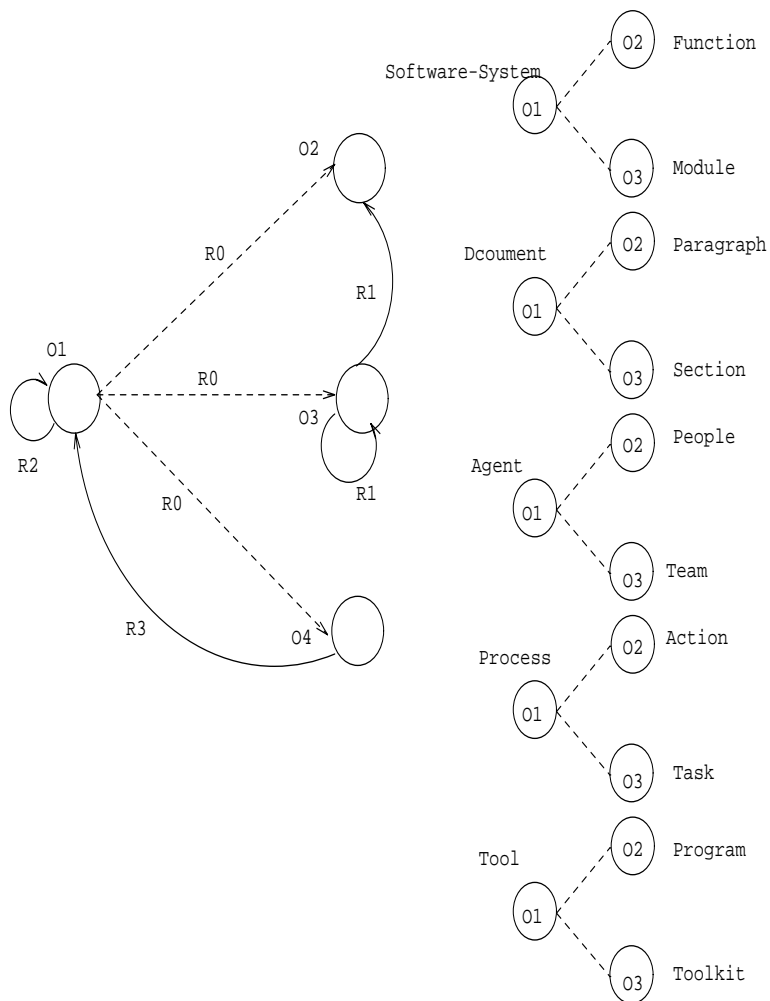


Figure 1: SPLib Architecture

Processes in SPLib are stored at different levels. These levels of processes indicate level of details and form a hierarchy that identifies access to process descriptions. At present, three levels are supported, but other customized levels can be added: The *national* level is the highest of representation in SPLib where processes are available to an open community of users. It is a broad and generic level of process description. The *organizational* level is for different organizations. It contains organization-specific information. The *project* level is for a particular project within an organization.

Relations among processes in SPLib describe conceptual causal relationships between processes and are defined as a pair of invertible relations from opposite directions:

- *has-detailed-process* / *has-abstracted-process* describes a relationship of specialization and generalization. A pair of processes related through the relations is said that one process is a specialization (generalization) of the other.

```

{{ SOFTWARE-PROCESS
  IS-A: SOFTWARE-OBJECT
  IS-A+INV: PROCESS-INSTANCE    PROCESS-MODEL
  LEVEL:
  PROCESS-CONTENT: <pointer-to-process-description>}}

{{ PROCESS-MODEL                {{ PROCESS-INSTANCE
  IS-A: SOFTWARE-PROCESS        IS-A: SOFTWARE-PROCESS}}
  HAS-DERIVATION:
  DERIVATION-OF:
  HAS-INSTANTIATION:
  HAS-HISTORY:
  HAS-DETAILED-PROCESS:
  HAS-ABSTRACTED-PROCESS:}}

{{ PLANNED-INSTANCE            {{ HISTORY-INSTANCE
  IS-A: PROCESS-INSTANCE        IS-A: PROCESS-INSTANCE
  INSTANTIATION-OF:            HISTORY-OF:
  HAS-HISTORY:}}              HAS-DETAILED-PROCESS:
                              HAS-ABSTRACTED-PROCESS:
                              ENACTMENT-TYPE:}}

```

Figure 2: Schematic Definitions of SPLib Processes

- *has-derivation / derivation-of* describes a derivation relationship among processes. A pair of processes related through them is said that one process derives, or is derived from the other.
- *has-instantiation / instantiation-of* links a process model to its planned instances, which are ready for execution in process-driven software development.
- *has-history / history-of* links a process model to its histories that have been enacted or simulated during process-driven software development.

5.3 The SPLib Process Model

The SPLib process model describes a formalism for software processes. It actually reuses the Articulator meta-model of software processes explained in [MS90]. Here we only give a brief discussion about the Articulator meta-model in order to help understand the process operations. In the Articulator, a software process is specified as an interrelated collection of objects which represent development activities, artifacts, tools, and developers. Each object

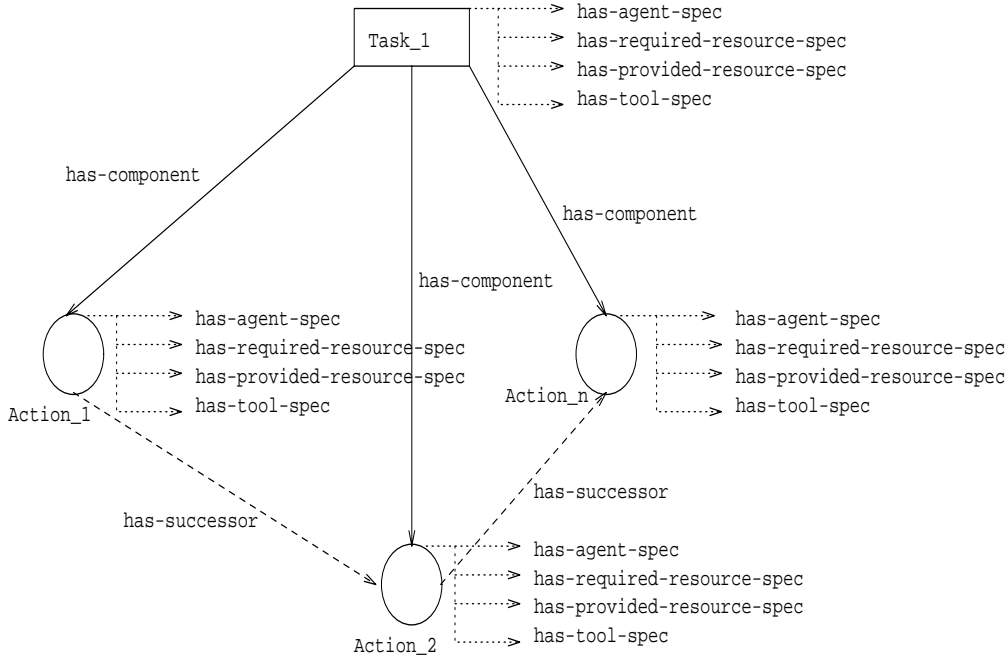


Figure 3: A Sample Software Process Fragment

describes a kind of information that is involved in software development. Further, these objects are linked through many kinds of relations. Altogether, software process models serve as a repository of information on the status of development processes and activities that get manipulated throughout a software development project [MS90].

A software process model includes an activity hierarchy that describes a decomposition of development activities and resource requirements including software artifacts, tools, developer roles, and other critical resources. Figure 3 shows the partial schematic activity hierarchy and resource specification of a sample process fragment.

An *activity hierarchy* represents the decomposition of a software process into a hierarchy of smaller activities called *subtasks*. Levels of decomposition can be arbitrary depending on the complexity of the process. The top-level description is a *task*, which is recursively decomposed into a set of interrelated subtasks and actions. *Actions*, at the bottom of this hierarchy, represent single tool/function invocations and simple resource transformation. Within a level of decomposition, a partial order for subtask execution is specified by several types of precedence relationships among the subtasks, such as sequential, parallel, iterative, and conditional.

Four types of *resource requirements* specify descriptions of resources needed for a subtask

and the expected products that result. First, a binding of users to the various developer and organizational roles taken during subtask performance. Second, software artifacts that are needed, created or enhanced during a subtask, called required and provided resources. Third, tools that are used. Last, information about subtask scheduling and their expected duration. These resources are represented as independent object classes and have relations that link them to process models. For example, a product model of a software system could be defined to have a module decomposition structure, whose modules are linked to their producer and consumer subtasks [CS89].

In sum, the object classes used in software processes include:

- A *task* and an *action* is a representation of development work. Tasks are decomposable, actions are not.
- *Agents* are developers that perform role-specific development activities during software development. Agents are divided into individuals, teams, or organizations.
- A *resource* or *product* is an entity consumed and produced by development tasks and actions.
- A *tool* is a resource utilized during development actions that can affect a product transformation.

In the Articulator, all object classes have a structural definition that describes their organization or configuration.

Relations among the object classes in software processes describe their conceptual relationships and form two structures called activity hierarchy and resource requirements. These are also defined as a pair of invertible relations from opposite directions:

- *has-component* / *component-of* describes a relationship of task decomposition. A pair of subtasks related through the relations is said that one subtask is a component of the other. Multiple levels of decomposition are allowed.
- *has-predecessor* / *has-successor* describes a precedence relationship among subtasks. A pair of subtasks so related means one subtask precedes the other during development. There can be linear, parallel, iterative, or conditional precedences among subtasks.

- *has-agent-spec / agent-spec-of* links a subtask to the necessary roles of the agent who performs it. It specifies both classes of agents, their availability status, and the needed quantity.
- *has-required-resource-spec / required-resource-spec-of* links a subtask to the input resources to be consumed. It specifies both classes of required resources, the needed quantity, and their status.
- *has-provided-resource-spec / provided-resource-spec-of* links a subtask to its intended output product specifications. It specifies both classes of products, the produced quantity, and their status.
- *has-tool-spec / tool-spec-of* links a subtask to the development tools to be used in it. It specifies both classes of tools, the needed quantity, and their status.

6 SPLib Process Operations

Process operations perform two types of manipulation: one is to maintain the proper relations for a process within the library model, such as *has-derivation* and *derivation-of* relations. The other is to reason about the content of a process description within the process model. For instance, *composition* takes some process models as its input, modifies their specification, creates a new process model that meets an input requirement, then inherits certain properties from all of its input process models.

There are several kinds of process operations in SPLib. Some simple operations provide basic services to access SPLib and maintain proper relations among the processes. Additionally, other process operations provide basic reasoning capability in order to access software processes upon user request. Finally, advanced operations directly modify the contents of process descriptions.

Some of the advanced operations are sufficiently complicated that each is really a research topic of its own. In recent years, we have studied some of these operations in great detail. Here without reiterating these details, we simply list these process operations and their references when available. The simple process operations include *upload*, *download*, *create-process-views*, *create-process-measurements*, *get-historical-process*, and *process search-and-query*. The advanced process operations include *process definition*, *process composition*,

process specialization, process abstraction, process instantiation, process simulation [MS90], *process enactment* [MS92], and *process articulation* [MS91]. Due to space limitations, we discuss three of these process operations in detail next. These are process search-and-query, process composition, and process abstraction.

6.1 A Process Search-and-Query Operation

Process search-and-query based on a user request is a very important operation in SPLib. The main benefit of this kind of knowledge-based or semantic search is that it allows users who can not specify their requests exactly to traverse through SPLib in an 'intelligent' manner, and to help them navigate through a large collection of software processes. Since processes in the SPLib are organized through a number of relations at the library level and specified by a group of characteristics at the process level, users are able to move around along the relations and characteristics in order to identify and browse potentially interesting software processes. A very good approach to do this search is by a classification algorithm described in [DBe90]. In SPLib, we extend this algorithm to incorporate the additional process relations as defined earlier.

The search-and-query operation first allows users to specify a level of detail to search, but it also allows them to switch levels as needed during search. Users then are asked to specify relations to traverse in SPLib. These relations can be *has-detailed-process / has-abstracted-process*, *has-derivation / derivation-of*, *has-instantiation / instantiation-of*, *has-history / history-of*, or some combination of these. Finally, the operation provides a template for users to specify their desired properties for the process. They include object about agents, required-resources, provided-resources, and tools.

When a search starts, it first limits the scope according to the specified process level. Then it identifies a possible set of candidates through the specified search relations. For each of the candidates, the operation browses its definitions of the activity hierarchy and resource requirements to find a possible match. This may entail examining the process models' functional description, agents, required-resources, provided-resources, and tools. When it identifies a match, the operation will stop and prompt the user with the match. Otherwise, it iterates the preceding steps, provides the user with existing choices among different relations and process properties, then asks the user to provide more information in order to continue the search.

6.2 A Process Composition Operation

Process composition realizes high-level construction goals specified by a user, and creates a new derived process model from a set of existing composable process models in SPLib. During this operation, the search-and-query operation is frequently invoked to find the appropriate process model components for composition.

A simple form of composition is refinement, where a simple subtask or an action is replaced by a multi-level process model. Before replacement, the process model to be inserted is retrieved when it matches the resource requirements, i.e. inputs and outputs, of the subtask or action. When we revisit the usage scenario later, we will show an example of refinement where `formulate-design` in `preliminary-design` is replaced by the `00-design` process. `00-design` produces an upward-compatible set of provided-resources to those of `formulate-design`, and it also specifies a particular design methodology, e.g. Booch's object-oriented design method [Boo91].

In more complicated cases, the composition operation must identify both the candidate process models and their precedence order. This is accomplished as follows: First, a user specifies construction goals for the derived process model in terms of its provided-resources, required-resources, agents, and tools. Among these goals, provided-resources, (i.e. products) are most important since they determine which processes are possible candidates for composition. After that, the composition operation helps a user to expand these goals into complete semantic models of products, inputs, agents, and tools in terms of their decomposition and state information. For instance, a semantic product model can include product decomposition, development status, and assembly sequence relations among product components.

The next step is to search-and-query SPLib to find process models that match the specified model of provided-resource, required-resource, agent, and tool. This is done incrementally as more and more component process models are identified and merged. A difficulty in this step is determining the precedence relation among the component process models. Possible solutions can come from either the given product model, or from additional user input.

Finally, the complete process model will be evaluated by the user to determine whether the composition is successful. Partial re-work is possible given the fact that user-specified goals are often ambiguous or non-deterministic.

Sometimes there is more than one way that process components can be put together without violating the goals. Therefore, the input from users and other outside knowledge is required. For example, when we build a software development process, we know from prior knowledge that design tasks usually precede implementation tasks. Common knowledge such as this can be summarized as rules to guide process composition.

6.3 A Process Abstraction Operation

Process abstraction is another operation that creates a higher-level generalized process from a set of lower-level processes. The abstraction operation is needed when process details are considered redundant or proprietary. For example, in a large-scaled software design, more than one group is involved in different design tasks. Sometimes, a group would like to disseminate its process model in a more abstract way without disclosing project-specific details or proprietary information. Abstraction can also generalize case-based process knowledge into a more useful form to other users.

The abstraction operation starts from a user specification. It gathers information about the level of details to be exported and the goals to unify comparable subtasks into a generic representation of the subtasks.

The abstraction operation proceeds iteratively from one subtask to another subtask, trying to unify five objects for each pair of subtasks in different process models, i.e. activity hierarchy, provided-resources, required-resources, agents, and tools until all subtasks are merged into a new process. Unifying objects requires selection of the common class representation from all the values of each object in the merged subtasks. We unify a pair of values of the same object, such as agent, by reasoning about whether and how process' agents are related in terms of their classification and location in the object's class hierarchy. First, the operation determines whether there exists a match of the classification between the two values to be unified. Then it categorizes the two object values in the subtasks into their corresponding object classes such as individuals, teams, or organizations for agents. If there is an intersection among all the values, the intersection class is determined to be a common class for that object and will be further matched to subclasses of this class. The kind of matching continues along the subclasses until no match can be identified. If the unified object set is empty, unification fails; otherwise the final matched object class is the value to be export to the generic process model in the position of the original object such as agents.

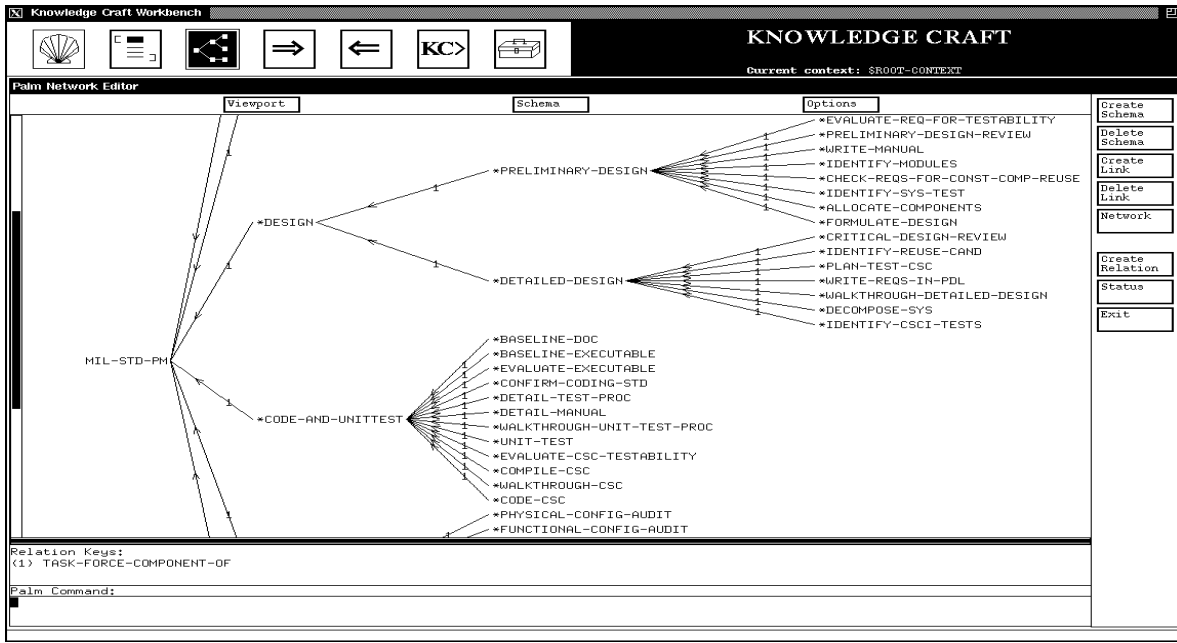


Figure 4: The Activity Hierarchy of MIL-STD-2167A Process Model

7 The Usage Scenario Revisited

Given the SPLib architecture and its operations, let us revisit the previous usage scenario to see how SPLib is able to accomplish the intended tasks.

First, all necessary processes have been represented and stored in SPLib. At each level, we assume there is only one process to be composed. With the support of SPLib operations, process construction can be accomplished through following steps:

- Step 1: Search for necessary process models based on the contract-awarding agency's requirements at the national level. For instance, a formal process model based on MIL-STD-2167A can be retrieved and downloaded to the organizational level, as shown in its activity hierarchy displayed in Figure 4.
- Step 2: Search for necessary process models based on the contractor's own policies at the organizational level. For instance, a process model based on Booch's object-oriented design method [Boo91] can be retrieved as a method for software design (Figure 5).
- Step 3: Compose these two process models into a tailored development plan for the project. Figure 6 shows one part of composition, in which the design stage in MIL-STD-2167A process model is expanded to incorporate Booch's object-oriented design.

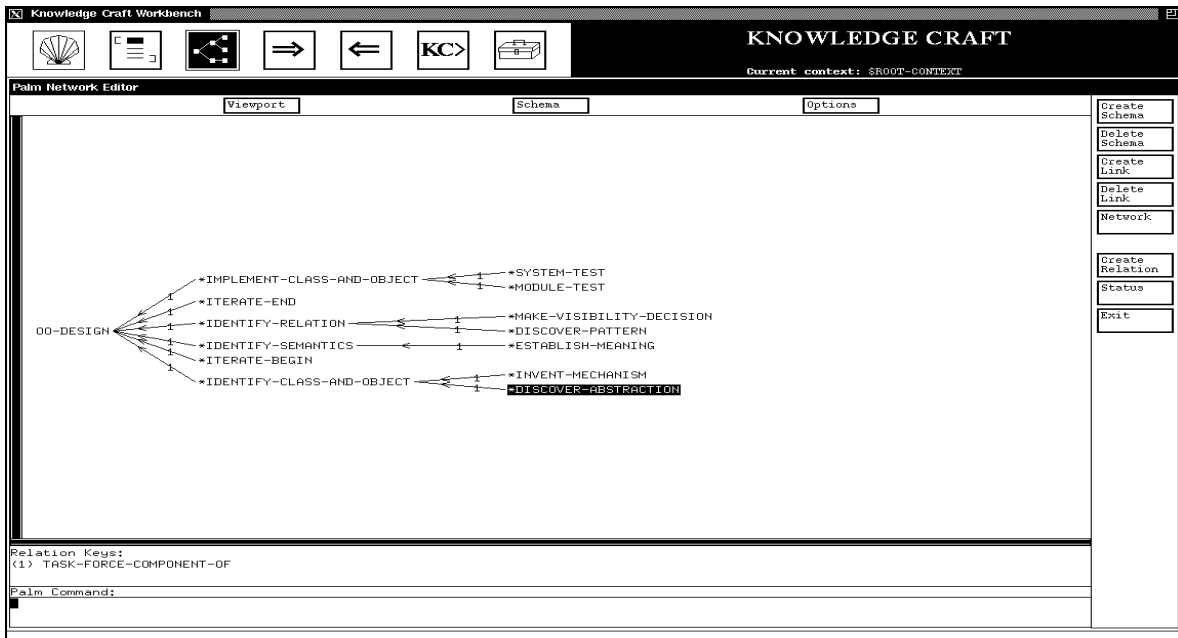


Figure 5: The Activity Hierarchy of Booch's OO-Design Process Model

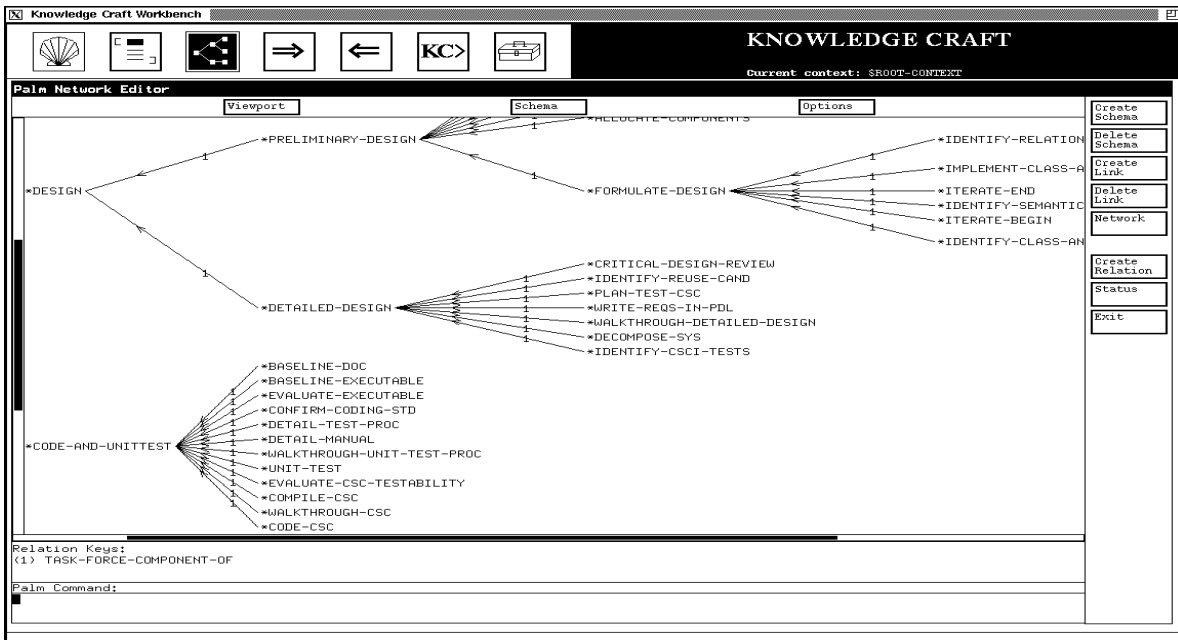


Figure 6: The Activity Hierarchy of the Tailored Development Plan

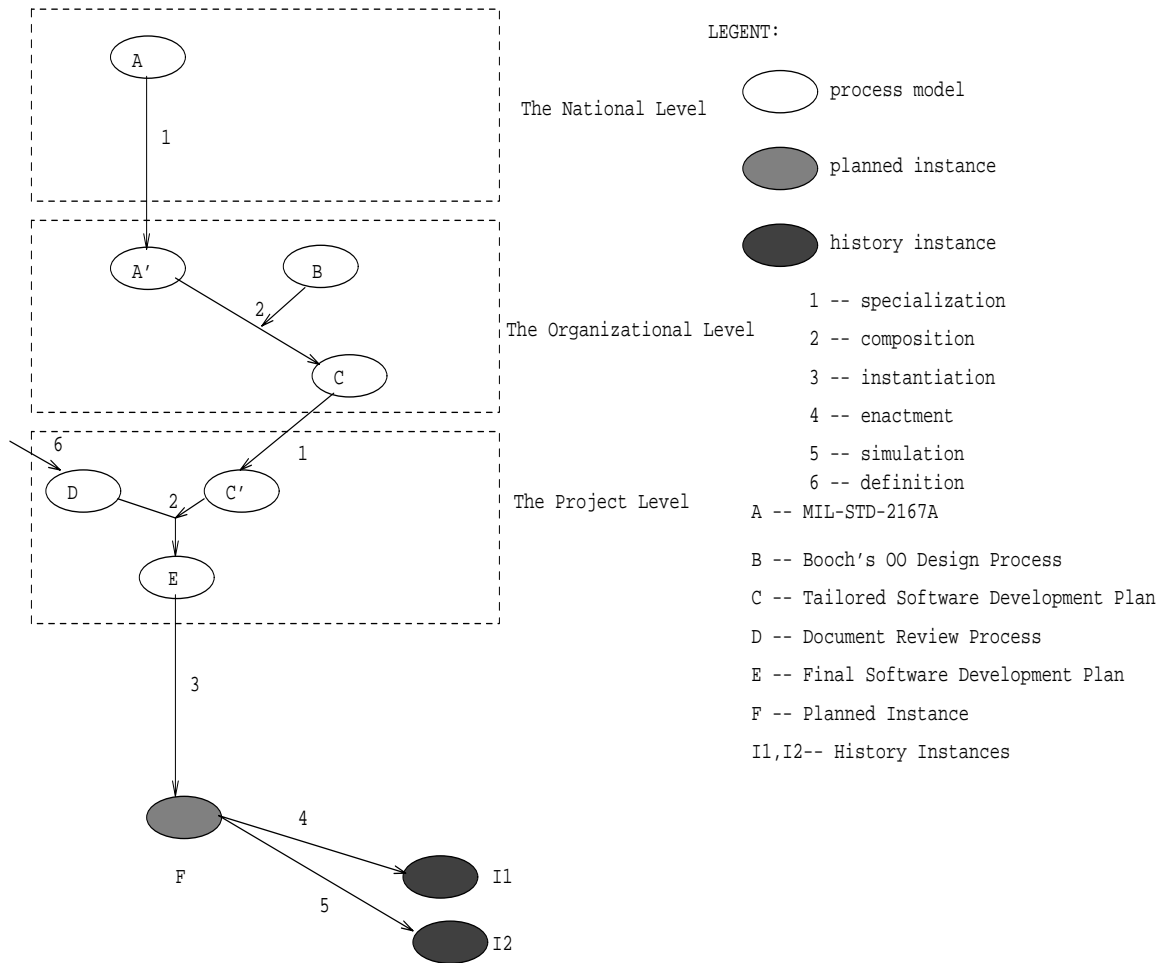


Figure 7: Operations in the Scenario

- Step 4: Create needed processes for the unique features at the project level. For instance, a process model describing the project-specific document review process can be specified.
- Step 5: Compose the above process models into a final software development plan for the project.
- Later Steps include: Instantiate the final process model to a planned instance (i.e., bind specific resource values to the resource classes included in the plan); Simulate the planned instance in order to analyze and adjust resource allocations; Enact the planned instance within a process-driven software engineering environment [MS92].

Figure 7 lists the processes and operations for this scenario. Figure 8 shows a partial view of the final customized process model after all the process operations.

```

;; Process models at the national level
  {{ MIL-STD-2167A
    INSTANCE: PROCESS-MODEL
    HAS-DETAILED-PROCESS: 2167-A
    LEVEL: NATIONAL}}

;; Process models at the organizational level
  {{ OODP-B
    INSTANCE: PROCESS-MODEL
    HAS-DERIVATION: DVP-PLAN
    LEVEL: ORGANIZATIONAL}}
  {{ 2167-A
    INSTANCE: PROCESS-MODEL
    HAS-ABSTRACTED-PROCESS: MIL-STD-2167A
    HAS-DERIVATION: DVP-PLAN
    LEVEL: ORGANIZATIONAL}}

  {{ DVP-PLAN
    INSTANCE: PROCESS-MODEL
    DERIVATION-OF: OODP-B 2167-A
    HAS-DETAILED-PROCESS: DVP-PLAN-C
    LEVEL: ORGANIZATIONAL}}

;; Process models at the project level
  {{ DVP-PLAN-C
    INSTANCE: PROCESS-MODEL
    HAS-ABSTRACTED-PROCESS: DVP-PLAN
    HAS-DERIVATION: DVP-PLAN-E
    LEVEL: PROJECT}}
  {{ DOC-REVIEW-PROCESS
    INSTANCE: PROCESS-MODEL
    HAS-DERIVATION: DVP-PLAN-E
    LEVEL: PROJECT}}

  {{ DVP-PLAN-E
    INSTANCE: PROCESS-MODEL
    HAS-ABSTRACTED-PROCESS: DVP-PLAN-C
    HAS-INSTANTIATION: DVP-INSTANCE-F
    LEVEL: PROJECT}}

;; A planned instance
  {{ DVP-INSTANCE-F
    INSTANCE: PLANNED-INSTANCE
    INSTANTIATION-OF: DVP-PLAN-C
    HAS-HISTORY: DVP-SIMU-1 DVP-ENACT-1}}

;; two history instances
  {{ DVP-ENACT-1
    INSTANCE: HISTORY-INSTANCE
    HISTORY-OF: DVP-INSTANCE-F
    ENACTMENT-TYPE: ENACTED}}
  {{ DVP-SIMU-1
    INSTANCE: HISTORY-INSTANCE
    HISTORY-OF: DVP-INSTANCE-F
    ENACTMENT-TYPE: SIMULATED}}

```

Figure 8: Final Object Schemata for the Scenario in SPLib

8 Conclusion

In this paper, we presented the design and initial implementation of a knowledge-based process library (SPLib) that supports process-driven software development and process reuse. SPLib stores and organizes a collection of well-defined software processes in form of a multi-level knowledge base. It also provides a set of process operations that make meaningful access and reuse possible and more convenient.

By utilizing such a knowledge-based process library, software process descriptions can be readily managed, reasoned about, accessed, and reused. Through the operations, users can readily query and retrieve a collection of software processes that can be shared across a diverse user community. Users are also able to perform operations to compose, abstract and manipulate the process descriptions.

Development is underway to host the SPLib on top of a distributed hypertext repository called DHT [NS91] to accommodate heterogeneous storage servers and remote access. SPLib is also being expanded to support different forms of process descriptions, such as textual [FBe91] and process programming [Ost87]. We therefore believe that the future use and reliance upon process-driven software development environments [MS92] will require and benefit from a knowledge-based process library such as we have presented here.

References

- [AL89] B.P. Allen and S.D. Lee. A Knowledge-based Environment for the Development of Software Parts Composition Systems. In *Proc. of the 11th International Conference on Software Engineering*, pages 104–112, Pittsburgh, PA, May 1989.
- [Boe86] B. Boehm. A Spiral Model of Software Development and Enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4):22–42, Aug 1986.
- [Boo91] G. Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 1991.
- [Car86] Carnegie Group Inc. *Knowledge Craft User's Guide (Vol.1, Vol.2, and Vol.3)*, 1986.
- [CS89] S.C. Choi and W. Scacchi. Assuring the Correctness of Configured Software Descriptions. *ACM Software Engineering Notes*, 17(7):67–76, 1989.

- [DBe90] P. Devanbu, R.J. Brachman, and etc. LaSSIE: A Knowledge-based Software Information System. In *Proc. of the 12th International Conference on Software Engineering*, pages 249–261, Nice, France, March 1990.
- [FBe91] D. Frailey, R. Bate, and etc. Modeling Information in a Software Process. In *Proc. of the 1st International Conference on the Software Process*, pages 60–67, Redondo Beach, CA, Oct 1991.
- [HK89] W.S. Humphrey and M.I. Kellner. Software Process Modeling: Principles of Entity Process Models. In *Proc. of the 11th International Conference on Software Engineering*, pages 331–342, Pittsburgh, PA, May 1989.
- [HL88] K.E. Huff and V.R. Lesser. A Plan-Based Intelligent Assistant That Supports the Process of Programming. *ACM SIGSOFT Software Engineering Notes*, 13:97–106, Nov 1988.
- [Kai88] G.E. Kaiser. Rule-Based Modeling of the Software Development Process. In *Proc. of the 4th International Software Process Workshop*, pages 84–86, New York, NY, 1988.
- [MS90] P. Mi and W. Scacchi. A Knowledge-based Environment for Modeling and Simulating Software Engineering Processes. *IEEE Trans. on Knowledge and Data Engineering*, 2(3):283–294, Sept 1990.
- [MS91] P. Mi and W. Scacchi. Modeling Articulation Work in Software Engineering Processes. *Proc. of the 1st International Conference on the Software Process*, pages 188–201, Oct 1991.
- [MS92] P. Mi and W. Scacchi. Process Integration in CASE Environments. *IEEE Software*, 9(2):45–53, March 1992.
- [NS91] J. Noll and W. Scacchi. Integrating Diverse Information Repositories: A Distributed Hypertext Approach. *Computer*, 24(12):38–45, Dec. 1991.
- [Ost87] L. Osterweil. Software Processes are Software Too. In *Proc. of the 9th International Conference on Software Engineering*, pages 2–13, Monterey, CA, Apr 1987.

- [RD91] B. Ramesh and V. Dhar. Representation and Maintenance of Process Knowledge for Large Scale Systems Development. In *Proc. of 6th Knowledge-based Software Engineering Conference*, pages 223–231, Sept 1991.
- [WS88] M. Wood and I. Sommerville. A Knowledge-based Software Components Catalogue. In P. Brereton, editor, *Software Engineering Environments*, pages 116–133. Ellis Horwood Limited, 1988.