Software Configuration Management in PROTEUS

In Proc. 4th Software Configuration Management Workshop (SCM4)
ICSE-15, Baltimore, Maryland, May 1993

Eirik Tryggeseth* Reidar Conradi Bjørn Gulla

Abstract

We give an overview of the ESPRIT PROTEUS project, its basic tool-set architecture, and the PROTEUS Configuration Language (PCL). It identifies several contributions made to the SCM field during the past decade, and argues for the need to integrate these contributions into one coherent environment. The objectives of the PCL is then presented, and the different concepts in the language are presented. Several examples of using PCL is shown.

Keywords: SCM, Configuration Management, MIL, Configuration Languages

1 Introduction

It is generally agreed that understanding the software development life-cycle is a complex undertaking. Several formalisms for developing software products exist, both in research and in industry, but most of these systems are either very naive in what support they give to the software developers, or impose a very strict flow of work organization. Systems of the first type may not support the complete software development process and may not be able to model the relations among all artifacts produced during development. Systems of this type includes basic version handling systems such as RCS and SCCS. Strict configuration management (CM) systems such as PCMS may not be flexible enough for integrating its CM abilities with third-party development tools. This makes the transition of developing software with ad-hoc or basic CM support to an integrated, full-fledged software configuration management system difficult and many times impossible for software engineering companies.

1.1 The PROTEUS Project

PROTEUS1 is an ESPRIT-3 joint European project for providing a methodology and a set of tools to support for software systems evolution. The partners in the project include Cap Gemini Innovation, Cap Sessa Telecom, Cap Debis GEI, Matra Marconi Space, Intecs, Lancaster University and SINTEF/NTH. The project started in May 1992, and will run for three years, with a total effort of about 60 man-years.

PROTEUS is an application-driven project, meaning that requirements for such support are given from the application developer partners to the different working groups in the project.

Based on these requirements, the technical working groups will define, assess and develop methods and tools for:

- Support for integrated hardware and software configuration, so that a complete description of a versioned product (possibly distributed in a heterogeneous environment) can be maintained. This is important both for the comprehension of the complete product to new developers, and for the system building process.

- Support for system composition, to be able to reuse large granularity components from other projects in the same (or possibly other) domain.

- Support for process change, in case working procedures are not static during a long-lived product’s life.

- Support for domain analysis, to be able to predict which parts of a software product will remain stable throughout the product’s life, and which will be subject to changes.

The application partners will transfer these individual results to the industrial world to enhance and combine them into a a consistent whole. Two prototypes of this integration will be provided within the PROTEUS project (one midway and one at the end of the project).

1.2 Organization of the paper

The rest of this paper is organized as follows – Section 2 relates the SCM work to be done in PROTEUS to other relevant work in this field. Section 3 describes the basic architecture of the PROTEUS environment, and identifies the techniques used to integrate this multi-partner project. Section 4 describes the current status of the
2 Related Work

2.1 Definition of CM

[Dar91] identifies four different concepts that a selected range of current software configuration management systems cover:

1. Repository
2. System modeling and construction
3. Process
4. Transaction

The PROTEUS environment’s SCM abilities will cover concepts 1-3 above. Transactions are not integrated into our current design. However, as we will later see, the environment provides some basic locking mechanisms on individual system components, and on selected parts of a system model.

From these concepts, RCS and SCCS belong to concept 1; module interconnection languages such as MIL75, Intercol, Jasmine and DSEE fits into group 2, while advanced SCM systems such as PCMS and Changeman support the concept of process.

We will describe the basic capabilities of some of these systems below. The Proteus environment will use and extend these systems, especially focusing on easy integration with existing software development systems, and extend the functionality with respect to support for systems under constant evolution.

2.2 CM tools

Fundamental to the CM field is the notion of a versioned, immutable software object ([ANS87], [Tie88], [Con90]). There exist several tools for managing versions of software objects and automatically building the software product that these objects comprise, both commercially and in research: The functionality of these can be summarized as follows:

Repository tools

Of most general interest, and thus surveyed in the project are the Source Code Control System (SCCS) [Roc75], the Revision Control System (RCS) [Tie82], and the Concurrent Versions System (CVS)2. These systems provide mechanisms to support the checkout/checkin model as discussed by Feiler in [Fei91].

Build tools

Typically for the system build phase is to run a makefile using make [Fel79]. Make offer the following advantages:

- Avoid repetitive typing when building programs.
- Save processing by reusing derived objects.
- Provide compact description of derivation rules by using file-suffix based rules.
- Automatic system generation based on this description of derivation rules (makefile).

There exists several extensions to make to enhance its capabilities and ease its use. Examples of such are nmake [Fow90], imake [Ful89], makedepend, and mkmf.

Tightly integrating make with repository tools have caused some problems. This has led to some systems incorporating build mechanisms similar to make, and extended the derivation model to reduce the compile-and-link time. (E.g. DSEE [LC84].)

2.3 Configuration languages

We can term a language used for controlling the product and version structure of a large program system a configuration language. Configuration languages mainly fall into two areas: Those that arise from early work into the difficulties found in programming-in-the-large and those whose main concern is the field of dynamic reconfiguration. Those that fall into the first area are known as Module Interconnection Languages (MILs) (MIL75 [DK76], Coopider’s MIL [Coo79], INTERCOL [Tie80], Jasmine [MW86], SySL [TS89]), they allow the description of large software projects in terms of interconnected modules and the relationships that exist between these modules. Languages concerned with dynamic reconfiguration are known as configuration programming languages (Conic [JMS89]) and have closer links to traditional programming languages than found in MILs. Traditionally, MILs have described software systems in terms of their static, logical structure. The configuration programming languages have expressed their descriptions in a more detailed, procedural form.

During a survey of configuration languages [Dea92], we have identified six features of configuration languages

2 Brian Berliner: CVS II: Parallelling Software Development. Documentation included with the CVS package. CVS is a front end to RCS.

2
which highlight the type of support a configuration language can give: system modelling (1), interface definition (2), version selection (3), consistency checking (4), system building (5) and configuration programming (6).

We have identified several areas where the traditional configuration language approaches can be extended and integrated to better support for system evolution: (numbers correspond to those in the above list)

1. Composition, hierarchical structuring, variability in structure, representation of families, generic modelling, refinement of generics to realizations,

2. Abstract interfaces, concrete interfaces, provides and requires relationships, name scoping, accessibility of resources, mandatory or optional interface specification,

3. Mechanisms for version selection support such as intentional (conditional), default, and instantiation selection,

4. checking descriptions for completeness and consistency, validation of family properties,

5. How to specify derivation rules – inside or outside the language?: intelligent rebuilding, specification of installation of software modules onto possible distributed hardware.

While system traditionally are developed for one platform, new systems typically exist in several variants on a myriad of platforms. This lead us to believe that integrating techniques from traditional MILs (e.g. abstraction techniques, version selection, etc.) with features found in configuration programming languages (e.g. software - hardware mapping). Section 4 describes our approach to a configuration language; the PROTEUS Configuration Language – PCL.

2.4 Advanced SCM systems

In addition to the mechanisms presented in the two subsections above, advanced SCM systems incorporates process and control support into the SCM framework. Examples of such systems include PCMS [MH88], Adele [BE86], DSEE [LC84], Aide-De-Camp (ADC) [Har89].

Process support includes the notion of implicit or explicit process management (PM), where the goal is to develop a generic PM model and associated environment, where project- or organization-specific processes can be defined, reasoned about, executed and monitored [Con90].

Control support includes incorporating change-request authorization into the SCM framework. This means that a configuration control board (CCB) must authorize changes proposed in a change-request, after considering their impacts.

PROTEUS will support these issues. A deep coverage is not within the scope of this paper, and will be left out.

3 Basic architecture of PROTEUS

Figure 1 presents the status of the overall architecture of the common PROTEUS tool-set as of the end of 1992. This basic architecture is expected to extend during the project.

Based on Wasserman’s faceted definition of systems integration [Was90], we have based the integration of the first prototype of the PROTEUS tool-set on the following decisions:

Platform integration: Only one platform is supported – SUN 4 machines, running SUN OS 4.1, with X Windows. Programming languages used are object-oriented, Smalltalk for the monitoring and PPIS tools, while the rest will be implemented using C++.

Presentation integration: The X/Motif interface will be used, following the MotifStyle Guide.

Data integration: The individual components of the Proteus tool-set can be viewed as specialists working on different parts of the total systems. These tools will have their own representation of their special information, being able to respond to messages issued through the Message Server. A more specified data model may be specified in the second prototype. For now, the component called the Repository Interface will provide the necessary data integration.

The objective of the Repository Interface (RI) is to provide a standard interface to the underlying storage systems. The RI defines a platform independent interface for storing, retrieving and querying for elements of the products being produced with the PROTEUS tools.

The RI might be realized on top of different version management systems such as RCS, SCCS, Adele, or even just the UNIX file system.

Control integration: To achieve active cooperation among the different tools, we will structure the PROTEUS environment around a basic integration service called the Message Server. Its goal is to provide mechanisms to broadcast messages from a tool-component to a set of tool-components. The messages will be dispatched only to the components which have specified their interest for these messages. Messages will be used to invoke external services, to exchange information about the products, and to notify events.
4 The PROTEUS Configuration Language

In this section we describe the PROTEUS Configuration Language (PCL), and how it communicates with other parts of the PROTEUS environment in order to control the evolving software product. Figure 2 gives an overview of the latter. The arch labeled ‘PCL-import/export’ indicates how descriptions of software artifacts and their contents flows through the PCL-edit tool from and to the loosely integrated software development system (SDS). We intend to support semi-automatic maintenance of the PCL description using knowledge automatically extracted from the SDS.

The rest of this section covers the system modelling and system build capabilities of PCL in greater depth.

PCL will be both programming language and system independent and will provide a complete system description, i.e. both software and hardware, mappings from software to hardware, and rules for how to generate an instance of a described system. (By software, we mean all artifacts produced throughout the software development life-cycle, not only code.)

4.1 PCL overview

PCL supports modelling of structured software and hardware systems at three different levels of abstraction:

The generic level: This level is concerned with the modelling of application domain concepts without reference to their use in specific systems. The structure of a generic system or component is described. Structural variability must be expressible. It must also be possible to associate attributes (names + allowed values) to provide non-functional information.

The family level: This level is concerned with modelling a group or family of related software systems. This level is also recognized by INTERCOL [Tie80] and NuMIL [Nar85]. This reflects the fact that system evolution results in several software systems which have evolved from a common base. A generic system structure is instantiated by giving it a type and mapping the identified structural entities onto a set of logical system components. Logical system components are component names which later can be mapped onto physical entities (e.g. hardware, design documents, code modules, etc.).

The instance level: A specific instance of a system is modelled by specifying how the logical system components are realized by physical components and by defining the attributes of each individual instance. At the instance level, all bindings are completed. (E.g. attributes to their actual values, logical to physical components, and tools to their instantiations.)

A system that is partly bound, i.e. some variability is yet unbound, still resides at the family level of abstraction.

Following, we will describe the capabilities of PCL in terms of system modelling and system building. Below is an example of a PCL description exemplifying the division into three abstraction levels. In Section 4.4 further examples will be given.
// Generic level

generic DESIGN-ENVIRONMENT is
attributes
  GRAPHIC-ENVIRONMENT := {TRUE, FALSE}
  // Default is TRUE
structure
  TEXT-EDITOR
  CM-SYSTEM
  if GRAPHIC-ENVIRONMENT = TRUE then
    DRAWING-TOOL
  endif
end DESIGN-ENVIRONMENT

// Family level

system SDL-DESIGN-ENVIRONMENT refines
  DESIGN-ENVIRONMENT is
attributes
  MACHINE-TYPE := {SUN4, IBMRS/6000, HP6000}
properties
  GRAPHIC-ENVIRONMENT := TRUE
structure
  TEXT-EDITOR
  CM-SYSTEM => SDL-CM-SYSTEM
  DRAWING-TOOL => SDL-TOOL
end SDL-DESIGN-ENVIRONMENT

node SDL-CM-SYSTEM is
structure
  if MACHINE-TYPE = SUN4 then
    ...
  elseif
    ...
  endif
end SDL-CM-SYSTEM

node SDL-TOOL is
structure
  ANALYSIS-TOOL
  DOCUMENTATION-TOOL
  DESIGN-TOOL
documentation

end SDL-TOOL

// Instance level

system instance MyDesignEnvironment refines
  SDL-DESIGN-ENVIRONMENT
properties
  MACHINE-TYPE := SUN4
structure
  // All slots/logical component names at
  // generic/family level must be bound to
  // a concrete file, either implicit or
  // explicit. See Section 4.4
physical
  workspace is "/home/trygge/Editor/SDL/"
  use version "Release 4.0"
end MyDesignEnvironment

4.2 System modelling capabilities of PCL

As of being in the first stages of the design of PCL, we have identified the following broad range of issues that the PCL language must provide on the system modelling side:

Structural description : Untyped slots at the generic level, refined by association to logical and physical components. A 'N:M' relation between slots and a set of logical and physical components is permitted for abstraction, and reuse.

Attributes : These can be used to specify non-functional parameters for the structural description, for describing variability in the structure, and other information designers wish to associate with components.

Structural variability : It is possible to model alternative structures in PCL. This is important if only parts of a software system varies across platforms.
Such variability is usually specified with attributes, and the structure is decided by the value of attributes or based on other structural instantiation.

Constraints: Forbidden and minimal constraints can be associated with an unbound system description, e.g., specifying that certain bindings of logical to physical components are not permitted, and that a software module cannot be bound to a hardware unit with less than 2Mb memory, respectively.

Shared declarations: Attributes, values and relations can be dynamically created, and shared by several product descriptions, if appropriate.

Libraries: To support reuse of earlier system models, PCL support a library mechanism from which existing generic and family level descriptions can be referenced. The name scope mechanism supported by PCL is a combination of near name overloading with a differencing attribute, and a package construct resembling the Ada package. These packages are stored in libraries.

Types: Nodes in a family level descriptions are typed. These types are e.g. documentation, source code, hardware-component, etc.

Relations: PCL provides a broad set of pre-defined relations among component types. Some examples are refinement, composition, inheritance and dependency. PCL provides support for user-defined relations.

Together, these building blocks provides a comprehensive set of primitives for modelling any system, not only computer related.

4.3 System building capabilities

The system building process is outlined in Figure 3.

Requirements to the system building process followed by PCL is summarized below:

Binding logical to physical components/version selection: Both selecting the right physical (i.e. software object) component corresponding to a logical component in a PCL by means of explicit mapping, and by more advanced intensional selection is supported by PCL. The latter is very important for large-scale configurations, where listing all bindings explicit will be very tedious. Intensional selection of the right software objects based on a logical configuration description relies on a mapping based on rules, defaults, and properties of the software objects. Rules and defaults can be explicitly expressed from the configuration description, while software object properties are attributes associated with the versions in the repository.

Primary elements: Primary elements in any format is supported. Primary elements are the input to the transformation/building process (simplistically, they are source code). All elements that are not currently reproducible form other elements are treated as primary elements in the repository.

During generation, the build process will check whether the same primary element has previously been transformed to a derived element. If so, and the same transformation parameters, and tool and primary versions were used, the retransformation of this particular primary will be omitted (i.e. smart recompilation)

Tool selection: The build process can be visualized as a set of transformations and aggregation of these aggregated derived elements. Translation tools are selected based on the type of the primary elements. This could be determined either by explicitly stating, in the configuration description, which tools to use, or implicitly based on the file suffix of the primary components. As the latter will suffice in most cases, it is not general enough. PCL will support both mechanisms.

Tool parameters: Tools can be modelled separately from the configuration description. The model of the tool will include the type of input, type of output, particular parameters (or compilation flags) that can be used when invoking the tool. PCL will use explicit parameter binding for passing actual values to the tool.

Tool location: Versions of tools (executables) with the same name, may be located in different directories (with perhaps some association between directory names and version identification). PCL will provide a mechanism for selecting the correct tool. This can either interactively by browsing and selecting a specific tool version. Or it may be more dynamic, by specifying certain rules in the PCL description for selecting the correct tool version.

External derivation: In some cases, executable systems or components derived using the PCL and PCL tools must be linked with other components which have been derived using some other approach. PCL must include facilities which allow the PCL writer to specify that a component has been externally derived and that the build process should not be applied to that component.

Installation: For distributed and customized systems, the installation of a system is not a straightforward process. PCL will provide language extensions to specify how and where software should be installed on the target system.

Consistency and completeness checking: During the configuration binding process, the consistency of the resulting configuration is checked. Contradictions and unsatisfied constraints will be reported.
4.4 An example

A small example to present the modeling part of PCL is given below:

```
generic WINDOW is
  attributes
    Scrollable-window := "No"

interface
  MOVE-UP
  MOVE-LEFT
  POSITION-XY
  if Scrollable-window = "Yes" then
    interface
      SCROLL-UP
      SCROLL-DOWN
  endif
structure
  WINDOW-OPERATIONS
  WINDOW-CONTENT
end WINDOW

node TEXT-WINDOW refines WINDOW is
  attributes
    ...
  interface
    POSITION-XY => Prompt-XY
  structure
    ...
end TEXT-WINDOW
```

This example shows several aspects of PCL. We give a generic description of a window. This contains a simple interface provided by the window object. The structure of the logical window component is also given. We define an attribute 'Scrollable-window' with default value "No". If, in a refinement or instantiation of this window description is set to "YES", the interface of the WIN- Dow is extended with the two operations 'SCROLL-UP' and 'SCROLL-DOWN'.

The node TEXT-WINDOW refines WINDOW, and specifies that the POSITION-XY operation should be bound to the more specific Prompt-XY operation.

To illustrate how specifications for the building process can be expressed in PCL, consider the following example:

```
system SUN-TEXT-WINDOW refines
  TEXT-WINDOW is
  properties
    // Binds attributes expressing variability
    Scrollable-window := "Yes"
  interface
    // This is inherited from the generic and family level
    structure
      // Structure is not further refined, but bound to
      // physical objects in the physical part
      physical
        for WINDOWS-OPERATIONS use "text-window.h"
        version "1.3"
        for WINDOWS-CONTENT use "text-window.c"
        version "1.3.2"
  derivation
    foreach ".c" do "/usr/bin/CC -c $1"
    do "/usr/bin/CC -o my_text_win $@"
end SUN-TEXT-WINDOW
```

This example shows how binding the logical component names to realizing software objects can be done explicitly. Also the derivation rules are explicitly defined, us-
ing no previous defined tool model.

Below is the above example modified to use intensional version selection with default mapping to physical files. The example also shows how a predefined model of the derivation process can be reused.

```
system SUN-TEXT-WINDOW refines TEXT-WINDOW is
   properties
      // Binds attributes expressing variability
      Scrollable-window := "Yes"
   interface
      // This is inherited from the
      // generic and family level
   structure
      // We must now refine the logical names to
      // associate them with real software objects.
      // This is not necessary if we use the same
      // names both in the logical model and in the
      // file structure
   WINDOWS-OPERATIONS =>
      text-window is "header"
   WINDOWS-CONTENT =>
      text-window is "body"
physical
   system is "C"
   use version "TESTED = TRUE && SHADOWED = TRUE"
```

derivation
   C-COMPILE(files="all") LINK()

We specify that the structural part consist of two different primary types, and that the system to be generated is written in C, to distinguish it from its Ada variant.

5 Discussion and Conclusion

This paper has given an overview of the PROTEUS project, its basic tool-set architecture, and the PROTEUS Configuration Language (PCL). It identified several contributions made to the SCM field during the past decade, and argued for the need to integrate these contributions into a coherent environment. The objectives of the Proteus Configuration Language were then presented, and finally a few examples of PCL was shown.

The presentation in this paper is based on achievements from the requirements and early design phase of PCL. Several issues concerning PCL, besides from those described here have been discussed in the project.

Among these, we can mention that we will provide automatic extraction of structure and related information from the different partners’ design tools. This will provide an automatic facility for inserting a project under SCM control. It will also allow semi-automatic maintenance of the system description with respect to evolution in the controlled product.

The examples shown in this paper are small, and rather easy to follow on paper. Large-scale software projects contain too much information to be sequentially described with text. A graphical structured editor is under implementation to support the creation and maintenance of PCL descriptions.

PCL mechanisms to describe distributional and installation aspects have not been covered in the examples above. Although the concrete syntax used for these is not yet properly defined, we have constructs in the language for grouping different components in the configuration description, and mapping this aggregate to particular machines/platforms in a heterogeneous network.

It should be noted that the design of PCL is still in its early development phase, but we think the concepts to be covered are well understood. The first prototype of the PCL integrated with the rest of the PROTEUS tools, as indicated in Figure 1 will be in the 4th quarter of 1993.

Relating to the discussion on page 2, PCL will provide all system modelling and construction capabilities of PROTEUS, and be integrated with the repository (probably CVS). The process aspects will be supported by the Process and Products Information System (PPIS) and Process Weaver, which also will be integral parts of the PROTEUS environment.

Acknowledgments

We would like to thank the whole PCL working group within the PROTEUS project for valuable discussions through our plenary meetings and document exchange. Special thanks goes to Ian Sommerville and Graham Dean from Lancaster University, England, and Björn Grounslet from Cap Gemini Innovation, Grenoble, France.

References


