Modelling Systems with Variability using the PROTEUS Configuration Language

Eirik Tryggeseth, Bjørn Gulla and Reidar Conradi

Department of Computer Systems and Telematics
Norwegian Institute of Technology (NTH)
N-7034 Trondheim, Norway
{eirik|bjorngu|conradi}@idt.unit.no

Abstract. To respond to environmental changes and customer specific requirements, industrial software systems must often incorporate many sources of variability. Developers use a diverse range of representations and techniques to achieve this, including structural variability, component version selection, conditional inclusion, and varying derivation processes.
This paper advocates specifying *all* potential variability within a system using a single formalism. PCL, the configuration language defined in the PROTEUS[1] project, provides uniform facilities for expressing and controlling variability in all aspects of a system and its manufacturing process. PCL is supported by a comprehensive tool set and is integrated with several design methods. The paper uses a simple example throughout to illustrate the facilities of PCL and how these are supported by the tool set.

Keywords: Configuration language, software evolution, software configuration management.

1 Introduction

The objective of the PROTEUS project is to provide support for system evolution. The project name is inspired by the mythological Greek sea-god who was capable of changing his shape at will to adapt to prevailing circumstances. The project has developed methods and tools for (1) domain analysis, (2) adapting existing design methods (SDL, HOOD, MD) to support evolving systems, and (3) modelling system structure and manufacture. This paper deals with the last issue. PROTEUS is an application driven project - four industrial companies operate as “users”, i.e. state requirements, validate specifications and evaluate developed methods and tools in operational divisions. PROTEUS is nearing completion, and we are now in the process of evaluating tool and methodological support.

PCL, the PROTEUS Configuration Language, is a formalism for system mod-

1. PROTEUS is project no. 8086 in the European research programme ESPRIT III. PROTEUS started in May 1992 for a period of three years and has a budget of 9,6 MECUs. Participants are CAP Gemini Innovation (F), Matra Marconi Space (F), CAP debis SSP (D), SINTEF (N), Lancaster University (UK), Intecs (I), CAP Sesa Telecom(F), and Hewlett Packard (F). NTH is a subcontractor to SINTEF.

elling, configuration definition and system manufacture. As systems evolve, large numbers of system and component versions with slightly different properties are created. The objective of PCL is to support product management in a broad sense throughout the complete system lifetime: manage components and sub-systems, their interconnections, their variability, their evolution and their potential derivation processes. PCL covers both software, hardware and documentation parts of products. We will in this paper focus on aspects of software management.

The paper is organized as follows. Section 2 gives a compressed state of the art review of work on which PCL is partly based. Section 3 presents the PCL language constructs for system modelling with emphasis on how to express variability. A working example is used throughout the section. Section 4 provides an overview of tool support for the PCL language and the current status of the implementation. Section 5 reports some experiences gained so far in the project, while Section 6 offers some conclusions. Finally, the full PCL source for the example used in this article is listed in an Appendix.

2 State of the Art

A system model is a description of the items of a system and the relationships between them. For such a model to support configuration management, it must uniquely identify the comprising components, their static structure and derivation processes. It is a principle in configuration management that the system model must be explicit, unambiguous, and be managed as the system evolves [17].

Module Interconnection Languages, MILs, is a common approach for expressing system models. Sommerville and Dean [13] give an overview of existing module interconnection languages and compare these with the capabilities of PCL. System models are also employed by current SCM systems, although the model is usually embedded in a tool or in a database. We have extended the comparison in [13] with more fine-grained criteria and replaced the description of some MILs with characterization of three SCM systems. Table I presents a summary of the comparison, which is to some degree influenced by the concrete requirements expressed by the application partners in PROTEUS. The requirements assessed in the table can be summarized as

- **Integrated system modelling:** Modelling all aspects of the product in one formalism, i.e. incorporate descriptions of and interrelationships between software, hardware and documentation elements.
- **Multiple structural viewpoints:** Be able to express and show several viewpoints of the same system, e.g. its interface, its logical composition and its run-time structure.
- **Structural variability**: The ability to define variability in the logical composition of a system, in interfaces and in relationships in which an entity participates.
- **Component variability**: The ability to represent variability in the concrete system (e.g. revisions and variants of source files), and to allow intensional version selection. Versions should be logically characterized, related to the system model.

TABLE I Support offered by MILs and SCM systems for PROTEUS requirements

	MIL75 [3]	Cooprider's MIL [2] INTERCOL [15]	Jasmine [10]	SySL [14]	ClearCase [8]	Adele [5]	PCL [12]
Integrated system modelling	None	None	None	Good	None	None	Good
Multiple structural viewpoints	Limited	None	None	None	None	Limited	Good
Structural variability	None	None	None	Limited	Limited	Good	Good
Component variability	None	Limited	Limited	None	Good	Good	Good
Flexible manufacture support	None	None	Limited	Limited	Good	Good	Good
Object-oriented modelling	None	Limited	Limited	Good	None	Good	Good
User tailorability	None	None	None	Limited	Good	Good	Good

- **Flexible manufacture support**: The details of the system manufacture process must be controlled from the system model. Definition of generic yet instrumentable manufacture tasks should be supported. The aggregation of such tasks into a manufacture process should be computed from the system model.
- **Object-oriented modelling:** The extent to which the language uses the concepts provided in object-oriented formalisms, such as classification, inheritance and encapsulation.
- **User tailorability:** Ability to provide an extensible, multi-dimensional classification scheme and offer integration with a range of different design methods. User-defined relations to tailor the modelling capabilities should be supported.

The only two formalisms offering direct support for integrated system modelling are SySL and PCL. In large-scale system evolution it is essential to capture the dependency relationships to enable successful change management. Incorporating non-software items is also necessary for proper modelling of distributed applications and embedded systems.

Although MIL75, the original module interconnection language, offered virtually no support for most of our requirements, it did offer limited support for multiple structural viewpoints. Future work largely ignored this early insight and still only provided limited support. PCL is the first language to provide good facilities to model a range of these different structural viewpoints.

Narayanaswamy identified the need for structural variability [11], although the proposed NuMIL does not contain constructs for expressing it. In SySL some

variability may be expressed using cardinality on the composition relation. ClearCase supports limited structural variability by allowing directories to be versioned. PCL allows structural variability to be explicitly declared, i.e. stating which parts of the system are stable and which parts vary by using conditional expressions in the system model. It also recognizes that variance can occur within any of the structural viewpoints, and supports reconciliation across a complete model.

Cooprider was the first to incorporate component variability into the MIL framework. INTERCOL allows structuring this information within the notion of a family, and supports version selection, i.e. allowing the system to determine which version to use in a configuration. More advanced SCM systems offer intensional configuration descriptions, consisting of a product part and a version part. Such descriptions serve as partially bound system descriptions, and must be expanded into fully bound configurations (extensional lists) by exploiting stored product and versioning information. The sequence of product and version binding varies. MILs usually first perform product elaboration into relevant product families, and then version binding for each atomic family. In Adele, an intertwined binding process over the product is used, exploiting preferences and constraint rules. Yet other systems, such as ClearCase and EPOS [9], first perform version binding, allowing transparent access to a uni-version view.

Automated support for system manufacture was introduced by Feldman [6] with the Make system. ClearCase provides more accurate and optimized re-generation by managing ‘configuration records’ for derived objects. In Adele manufacture support may be implemented by triggers. PCL advocates user control of re-compilation, using automatically generated makefiles tailored to the selected product configuration.

Object-oriented modelling has recently gained popularity in the software engineering community. Some of the principles behind the object-oriented paradigm, such as information hiding and grouping have been supported in previous languages. Only SySL, Adele and PCL offer extensive object-oriented facilities in their modelling languages.

User tailorability is an important requirement for enabling seamless integration with a diverse set of design methods. Different design methods often need specific types and relations for expressing their architectures, and rather than trying to include all possible ones in one language, an extensible framework should be offered. PCL does just that. Adele allows user-defined object and relationship types, and roles of these.

3 System Modelling and Variability

The aim of PCL is to provide a notation in which all aspects of a system family may be modelled. This includes software, hardware, documentation, possible configurations, how these configurations are instantiated into a system, and finally how the software parts of an instantiated system are processed into executable

programs.

PCL [12] defines six distinct entity types for modelling families of systems, as defined in Table II. An entity description is organized in sections, each consisting of a sequence of named slots. These entity types are related to each other by a set of language-defined relations as shown in Figure 1. The remainder of this section explains how these concepts and relations are used to support comprehensive modelling of system families.

TABLE II PCL entity types and sections

Entity type	Sections
family	classification, attributes, interface, parts, physical, relationships
version description	attributes, parts
tool	inputs, outputs, attributes, scripts
relation	domain, range
class	physical, tool
attribute type	enumeration

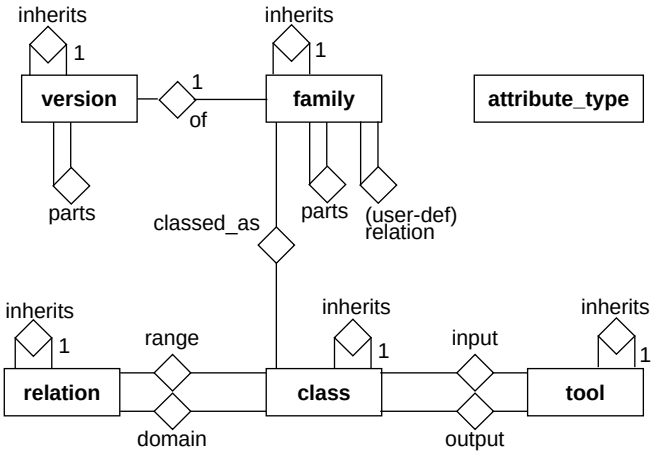

Fig. 1. Language-defined relations

3.1 Composition Structure

The basic assumption of PCL is that a system can be organized as a layered composition structure at the logical level. This means that one component may be a part of another component, and may itself have sub-components.

The family entity is the core entity in PCL. All logical components and their

structure are defined by a set of family entities.

Logical Structure

In the remainder of the paper we will use a calculator program as a small, but yet complete example for exemplifying the constructs in PCL. The complete PCL source for the example is given in an Appendix. The basic composition structure in the calculator program can graphically be illustrated as in Figure 2. In PCL this logical system structure is expressed as:

(i)

```
family CalcProg
  parts
    calc => Calculator;
    math => mathlib;
  end
end

family Calculator
end

family mathlib
end
```

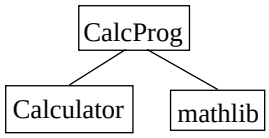

Fig. 2. Logical structure of the calculator program

The logical composition structure of a system is specified in the parts section. Note the use of *slots* (named 'calc' and 'math' in the parts section) in which the actual references to the sub-families are declared. Since PCL supports entity refinement through inheritance, slots are used to distinguish between items in a section, allowing selective addition, redefinition or removal of information.

Physical Structure

Parallel to the logical structure is the physical structure of the system, i.e. which tangible objects and computer files constitute the system and how these are organized. A logical component may be represented by none, one or several physical components. This information is given in the physical section. The calculator example can be extended with (omitting the parts section):

(ii)

```
family CalcProg
  attributes
    HOME : string default "/home/ask/proteus/test";
    workspace := HOME ++ "/calc/src/"; // string concatenation
    repository := "calc/";
  end
  physical
    main => "main.C";
    defs => "defs.h";
    exe => "calc.x" attributes workspace := HOME ++ "/calc/bin"; end
                    classifications status := standard.derived; end; // This is not a primary object
  end
end
```

```
family Calculator
  attributes
    workspace := ‘workspace ++ “Calculator/”;
    repository := ‘repository ++ “Calculator/”;
  end
  physical
    calc => (“Calculator.C”, “Calculator.h”);
    expr => (“expr.C”, “expr.h”);
  end
end

family mathlib
  attributes
    workspace := ‘workspace ++ “mathlib/”;
    repository := ‘repository ++ “mathlib/”;
  end
  physical
    files => (
      “math_plus.c”,
      “math_minus.c”,
      “math_mult.c”,
      “math_div.c”,
      “math_sqrt.c”,
      “mathlib.h”);
    lib => “libmath.a” classifications status := standard.derived; end;
  end
end
```

For software a physical object is a file in a certain directory on the user’s disk. The directory where a file is located is called the workspace for the file. Typically the files associated with one logical component tend to have the same workspace. Because of this PCL has defined a special attribute *workspace* which can be set in the attributes section of a PCL entity. The value of this will by default be the workspace of all files defined in the physical section. It is possible to override this, e.g. as for “calc.x” in CalcProg.

PCL allows propagation of attribute values along the composition hierarchy to achieve compact and easily manageable models. This is convenient for example when an application is moved from one directory to another. In mathlib we see that the workspace attribute is extended with the string “mathlib/” compared to the CalcProg’s value. The notation ‘<attribute_name> means to use the value of this attribute closest above in the composition hierarchy.

3.2 Entity Attributes

PCL supports annotation of entities with attributes of two different kinds, *information attributes* to provide stable information about an entity, and *variability control attributes* which are determined during system instantiation. Syntactically they are distinguished by using the ‘=’ assignment operator for entity attributes, while ‘:=’ (or no assignment) is used for variability control attributes.

Entity Information Attributes

Entities may be annotated with a number of attributes of type string, integer, or user-defined enumerations. There is a pre-defined enumeration type, boolean, whose members are true and false. We can elaborate the calculator example with attributes for the CalcProg entity:

(iii)

```
family CalcProg
  attributes
    created_by = "Marius Kintel";
    created : string = "94/08/12";
    contract_no: integer = 1643256;
  end
end
```

Since string is the default attribute type, including the attribute type string is not necessary (e.g. for created_by).

Variability Control Attributes

A family in principle represents a set of potential logical components. The differences between the individual members of the family is declared by the use of *variability control attributes.* A specific member is produced by binding values to these.

On the logical or architectural level, the breakdown of functionality may be different according to what situation the entity is used in, and at the physical level, the mapping from logical structure to files may differ, and finally each file may exist in different versions. An example of a variability control attribute is the 'status' attribute in the example below.

(iv)

```
family CalcProg
  attributes
    ...
    status: status_type exported default initiated;
  end
end
```

The 'status' attribute is of an enumeration type. An enumerated attribute type can be declared as:

(v)

```
attribute_type status_type
  enumeration initiated, module-tested, system-tested end
end
```

This attribute is not assigned a value as the other attributes in the example, but is rather given a *default value*. The default value may be overridden by a new value, taken from a *version description*, during system instantiation. See Section 3.7 for an explanation of the 'exported' qualifier.

3.3 Expressing Variability

The parts section in a family entity defines what parts the entity consists of. Each subpart in a family is declared in a slot which syntactically is on the form:

<slot> => <family-entity>
| <conditional-expr> // eventually referencing family-entities

The types of variability we want to show here are (1) variability in the logical composition structure of a system family, (2) variability in the mapping from the logical composition to the physical objects, and (3) variability in attribute assignments.

Structural Variability

Variability in the logical composition structure of a system family is expressed by associating a conditional expression to the assignment of a parts slot. Consider again the calculator example. We will extend the example to optionally include a graphical user interface. The original structure of the program is shown in Figure 2. Figure 3 illustrates the modified structure of the CalcProg entity:

(vi)

```
family CalcProg
  attributes
    ...
    xgui : boolean default false;
  end
  parts
    ui => if xgui = true then XGUI endif;
    calc => Calculator;
    math => mathlib;
  end
end
```

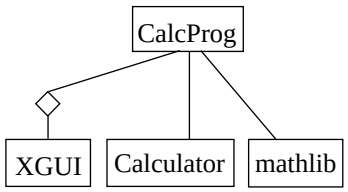

Fig. 3. Extending the structure of the calculator program

PCL also features entity refinement through inheritance. Inheritance might also be used for expressing variability between family entities. Inheritance is however mainly used for achieving economy of description by allowing extraction of common information for a set of family entities. This information can then be declared once in a generic family from which the other families inherit.

Providing these two constructs for expressing variability on the structural level allows declaring variability at different levels as needed.

Variability in Mapping

In (iii) we defined the Calculator entity to have four files, where two are related to expression parsing (“expr.c” and “expr.h”). PCL can express that the mapping from the logical component Calculator into files contains variability. This is expressed by introducing conditions on the slot assignment in the physical section. In the following example we express that the binding of files to the expression slot is dependent on the ‘expression’ attribute. In this case we want to distinguish between infix and reverse polish notation expression parsing.

(vii)

```
family Calculator
  attributes
    ...
    expression : expr_type default infix;
  end
  physical
    calc => ("Calculator.C", "Calculator.h");
    expr => if expression = infix then
              ("expr.C", "expr.h")
            elsif expression = reverse_polish then
              ("rpn_expr.C", "rpn_expr.h")
            endif;
  end
end
```

This allows straight forward and elegant treatment of collapsing, splitting, deleting, and moving files during system evolution. Many traditional configuration management systems have problems with handling this properly.

Attribute Assignment Variability (Constraints)

The last form of variability we present here may be used to represent simple constraints among attribute assignments. I.e. an attribute may take a special value only if another attribute (or a combination) takes a particular value like the CCFLAGS attribute in the following example:

(viii)

```
attribute_type os_type
  enumeration sun, vax end
end
family CalcProg
  attributes
    ...
    expression : expr_type default infix;
    debug : boolean default false;
    os: os_type;
    DEBUG := if debug = true then "-g -Ddebug " endif;
    INCL : string := "";
    CCFLAGS : string :=
      if os = sun then
        if expression = infix then DEBUG ++ "-Dsun4 "
        elsif expression = reverse_polish then DEBUG ++ "-O2 "
        endif
      elsif os = vax then DEBUG ++ "-C -Dvax "
      endif;
  end
end
```

3.4 System Instantiation

A system family described using PCL defines a set of possible system instances. System instantiation is the process of removing all (1) structural variability, and

(2) physical mapping variability, and assigning correct attribute values to the attributes throughout the instantiated system. We call this process *binding*.

A system is bound in an iterative way, in which the three following activities are performed interleaved: (a) Application of a version description on a family entity. (b) Evaluation of attribute expressions. (c) Propagation of attribute values along the composition hierarchy.

A variability control attribute may have its value propagated from another entity in the composition structure. This is a feature which is particular convenient for resolving logical and physical variability to build a consistent configuration. We declare an attribute to take its value from *the nearest entity above* in the logical composition hierarchy which has a value assigned for the particular attribute name. We may extend the Calculator entity with the attributes:

(ix)
```
attributes
   status : status_type := ‘status;
   number : integer := ‘X + 3 default 10;
end
```

Since declarations of the first form are used in most occasions, we allow the shorthand declaration below to mean the same.

(x)
```
attributes
   status : status_type;
end
```

The specification of version descriptors in PCL is intensional, i.e. defined in terms of the desired properties of the final system rather than explicitly enumerating the particular instances for each component. An example illustrates this:

(xi)
```
version my-version of CalcProg
   attributes
      os := sun;
      xgui := true;
   end
end
```

When this version descriptor is applied to the family entity CalcProg in (viii), the following happens during Bind:

- Most attributes are bound to their default values, e.g. expression is bound to ‘infix’.
- The os and xgui attributes are bound to sun and to true.
- The expressions for the DEBUG and CCFLAGS attributes are evaluated to ““ and to “-Dsun4 ”.
- The structural variability on the ui slot assignment is resolved, so the CalcProg entity in (vi) is composed of the XGUI, Calculator and mathlib parts.

Thus the CalcProg entity, after instantiation by applying the my-version version description, looks like:

(xii)
```
family CalcProg
```

```
    attributes
      HOME : string := "/home/ask/proteus/test";
      workspace : string := "/home/ask/proteus/test/calc/src/";
      repository : string := "calc/";
      created_by : string = "Marius Kintel";
      created : string = "94/08/12";
      contract_no: integer = 1643256;
      status: status_type exported := initiated;
      xgui : boolean := true;
      expression : expr_type := infix;
      debug : boolean := false;
      os: os_type := sun;
      DEBUG : string := "";
      INCL : string := "";
      CCFLAGS : string := "-Dsun4 ";
    end
    parts
      ui => XGUI
      calc => Calculator;
      math => mathlib;
    end
  end
```

Assume that both XGUI, Calculator and mathlib declare the attribute INCL := 'INCL;. By default, the value assigned to the INCL attribute in this case is the value of the attribute in the nearest ancestor in the composition structure. Now, for some reason, during system instantiation, it is discovered that the INCL attribute needs to have a different value for the XGUI entity. This is achieved by declaring a "sub" version descriptor specifying the particular bindings for XGUI. Figure 4 shows how this is visualized in the PCL tool set.

(xiii)

```
  version my-version of CalcProg
    attributes
      os := sun;
      xgui := true;
    end
    parts
      ui => ui-version;
    end
  end

  version ui-version of XGUI
    attributes
      INCL := "-I/local/X11R5/include ";
    end
  end
```

Now, when the bind operation is fixing the attribute values for the XGUI entity, it uses the values applied on the entity from the ui-version version descriptor. This has higher priority than the values propagated along the composition hierarchy, which are used e.g. for Calculator and mathlib.

To assist the system instantiation process for large configurations, the PCL

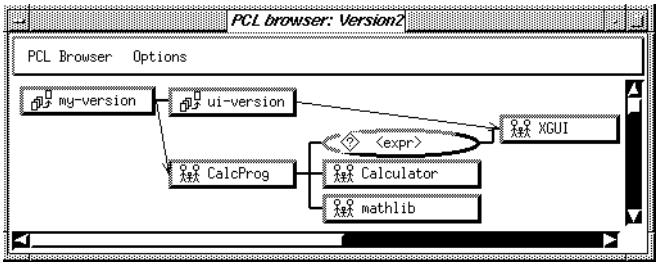

Fig. 4. Visual presentation of a composite version descriptor

tools provide:

- *Partial binding* to iteratively remove parts of the variability in a system model. This is useful for scrutinizing a model covering only a limited set of all possible system configurations.
- *Interactive binding* to aid the process by allowing the user to interactively choose between possible attribute bindings whenever Bind cannot compute a value. This is convenient for large and unfamiliar models, where it might be hard to know which attributes exist and which may or must be assigned a value. Automatic constructions of correct composite version descriptors is provided by the PCL tools to be able to re-instantiate the particular instance made during an interactive bind session.

3.5 Entity Classification

PCL provides a framework for classifying family entities and physical objects. Classifications are also used for defining the domain and range for user-defined relations. Requirements from the partners in the PROTEUS project have shown that entity classification is a complex task. Therefore PCL provides an extensible framework from which users can define their own classification hierarchies. PCL basically allows classification along four different dimensions, distinguished by different slot names:

- **abstraction:** Used to classify the entity according to its level of abstraction. The possible abstractions are *system*, *process* and *component*.
- **type:** Used to classify the entity as either *hardware*, *software* or an *amalgam* (*platform*). *Processor* is a sub-class of hardware, used for entities which can execute software processes. *Platform* is used for entities which are logically considered as a single entity and which include one or more processors and associated software. Application software is installed on a platform.
- **category:** Used to specify whether the entity is a documentation or a representation produced during the system development process. Possible categories are *document* and *program*.
- **status**: Used to specify whether the entity can be automatically derived. Possible status assignments are *primary* or *derived*.

In addition the user may define new classification dimensions, or introduce sub-classes of the pre-defined classes. Default values for classification assignment for

family entities are type => software, abstraction => component, category => program and status => primary.

Classification for Relation Definitions

In structural models of application systems there may be different kinds of relationships between the system entities. Some, such as the part-of relation, is directly provided by the parts construct in PCL. Others are specific to a particular system or a design method used in conjunction with PCL. We allow these relationships to be documented by offering users mechanisms to define binary relationships between family entities in a model. Relationships may be restricted to connect only certain types of entities by declaring the legal domain and range in the relation definition. Only entities defined with classifications matching the classifications specified as domain and range may participate.

Some relations are pre-defined in PCL, such as ‘requires’, ‘implemented-by’ and ‘installed-on’.

Classification for System Manufacture

Classifications are particularly important for system building, as this process is basically to find a relationship between a physical object and a tool that is able to transform the physical object into a new form.

In the calculator example we have

(xiv)

```
physical
   files => (“Calculator.C”, “Calculator.h”);
end
```

To be able to find a tool that may compile the file “Calculator.C”, we must be sure that the file and the input expected by the tool is of the same type.

For the calculator example, a number of sub-classes of software are defined.

(xv)

```
class text inherits standard.software
end

class source-code inherits text
end

class cpp-source inherits source-code
   tools CC end
   physical
      name ++ “.C”;
   end
end
```

From this example, we see that the file “Calculator.C” matches the classification ‘cpp-source’. Figure 5 shows a part of the full classification hierarchy.

The next paragraph explains how we use this classification information to support system manufacture.

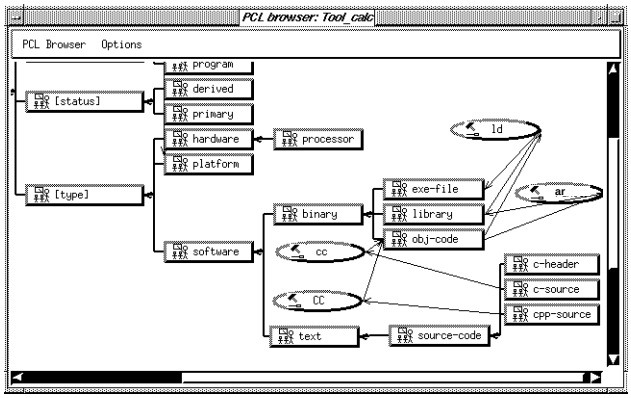

Fig. 5. Extract from the PCL classification hierarchy

3.6 System Manufacture (Building)

Borison [1] defines software manufacture to be the process by which a software product is derived, through an often complex sequence of steps, from the primitive components of a system. PCL provides constructs to define customizable tasks in software system manufacturing. The PCL tools use such descriptions to find the correct steps needed in a particular system manufacture process to build e.g. an executable program.

Section 3.4 describes how variability is removed in a PCL model. This step identifies the system configuration as a set of family members at the logical level. Variability is also removed as physical objects are mapped to file version groups and further to specific versions as described in Section 3.7. The configuration is then completely defined, with all variability removed.

From such a configuration description the system manufacture process may begin. The *tool* entity in PCL defines the signature and behavior of software tools which can transform a representation from one form to another, or more generally, transform a set of input representations to a set of output representations. A C++ compiler may be modelled in the following way using the tool entity:

(xvi)

```
tool CC
  inputs
    InSrc => cpp-source;
  end
  outputs
    OutObj => obj-code;
  end
  attributes
    CC : string default "CC ";
    CCFLAGS : string default "-c ";
    INCL: string default "";
  end
  scripts
    build := CC ++ CCFLAGS ++ INCL ++ "-o " ++ OutObj ++ "-c " ++ InSrc;
```

```
    end
  end
```

The *inputs* section specifies that the CC tool can transform a physical object classified as *cpp-source* into a physical object classified as *obj-code* as specified in the *outputs* section. This constitutes one step in the system manufacture process. The behavior of this step is defined in the *scripts* section, where two pre-defined script slots may be given an expression.

1. The *build script* specifies how the actual tool invocation on the command line is formatted. This is a catenated string expression. The CC tool entity declares three attributes which are used in the string expression. The values of these attributes are propagated from the physical object which the tool transforms. If no value is found there, the value defined for the enclosing family entity is used, or a recursive search along the system composition structure is initiated until a value is found. This facilitates customization of every manufacture step. As an example, the file Calculator.C, if the enclosing family is bound with the version descriptor in (xiii), is transformed by the following command line:

   CC -Dsun4 -o Calculator.o -c Calculator.C

   Since attribute INCL in entity XGUI is bound to another value, the C++ file in that entity would be derived with

   CC -Dsun4 -I/local/X11R5/include -o ui.o -c ui.C

2. The *depend script*, not used in the CC tool description, specifies the command line for (source-level) dependency extraction for the tool. The form of this script is similar to the build script.

As physical objects are transformed to new representations, which again may be further transformed, the system derivation graph is built. This information is emitted to a makefile which can be utilized by the Make program [6]. The makefile generation process can be customized in different ways, as shown in Figure 6.

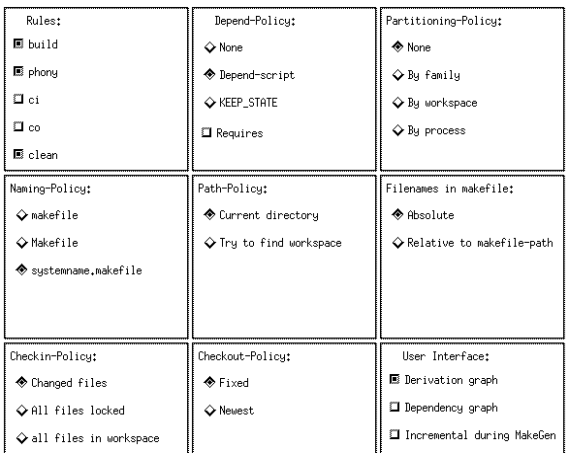

Fig. 6. Menu for customizing makefile generation.

The system derivation graph for the calc program is shown in Figure 7.

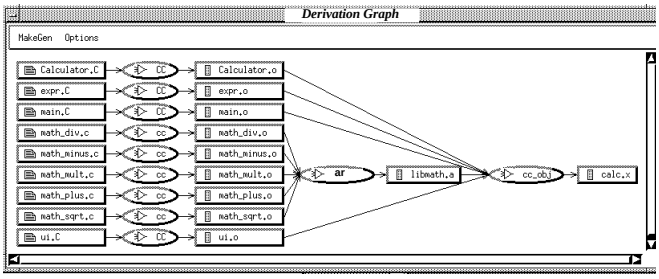

Fig. 7. Derivation graph for the calc example

3.7 Repository Management

A PCL model refers ultimately to a set of physical objects. For elements classified as software, a physical object corresponds to a *file*. These files are typically versioned, since they evolve over time and may exist in several variants. Real systems contain a large number of files, and over time there will be a vast number of file versions with subtle differences. If all file versions and their particular characteristics were represented inside the PCL model, the model would soon become impractically large. In PROTEUS we have therefore chosen a two-tier approach, in which file versions and their properties are managed by a special component library called the Repository.

Version selection is the process of determining a consistent set of versions for all elements in a configuration. Basically, this process consists of finding a unique version identifier for each element, so that the resulting configuration is consistent and possesses the desired properties. PCL supports intensional version selection, adopted from the Adele system [4].

Version selection is done by the *Select* operation. It transforms a bound PCL model to a selected model by adding explicit version identifiers to the description of each physical object stored in the Repository. For example, the following PCL fragment:

(xvii)

```
family CalcProg
  physical
    main => "main.C";
    defs => "defs.h";
    ...
```

might be transformed into:

(xviii)

```
family CalcProg
  physical
    main => "main.C" attributes repository_version := "5.14.2.4"; end;
    defs => "defs.h" attributes repository_version := "4.22"; end;
    ...
```

The intensional, attribute-based version selection works as follows. For each

physical object referenced in the model, the classifications are used to determine if it is supposed to exist in the Repository (i.e. classified as *software* and *primary*). If so, Select queries the Repository for the best matching version for the object. The submitted query includes all attributes defined in the family which are declared with the **exported** qualifier. If successful, a unique version identifiers is returned.

The following example illustrates some of the available operations when stating version selection queries over attributes:

```
(xix)
  version Calc_test of CalcProg
    attributes
      status >= module-tested;
      time := max; // The latest version
      author <> "bj.*"; // Note the use of regular expression
      // The time and author attributes are automatically inserted
      // for any version when checking it into the Repository.
    end
  end
```

This descriptor will select the latest file versions which has reached at least status module-tested and are not entered by a user having a name starting with "bj". The Repository resolves such queries by investigating the properties of all versions of a component. Version properties are expressed as attributes, i.e. user-defined name-value pairs. A user typically associates attributes to characterize a version when checking it into the Repository, or after having tested configurations in which the version occurs. It is the responsibility of the user to choose appropriate attributes which discriminate between versions.

Upon a successful Select, the resulting selected PCL model may be further used to check out the configuration from the Repository and possibly build the configuration. A selected PCL model ensures reproducibility, i.e. it uniquely defines a system instance which may be re-created.

To summarize, the following Repository operations are available for a PCL model:

- Select: invoke intensional version selection.
- Check In: check in all changed files of a configuration, and optionally attach a set of attributes to each new version.
- Check Out: establish or update a workspace by checking out all files of a specific configuration.

4 Tool Support

A comprehensive tool set to support the creation and use of PCL models has been developed. It includes a graphical structural editor for entering and browsing PCL models, an interactive PCL compiler, and a graphical browser for inspecting and manipulating the contents of the Repository. PCL models are organized in libraries with explicit prefixing for inter-library entity referencing. The PCL compiler

supports parsing of textual PCL descriptions, binding of models, version selection, and makefile generation. Figure 8 presents an overview of the core PCL tool set. In addition comes a simple reverse engineering tool for constructing a rudi-

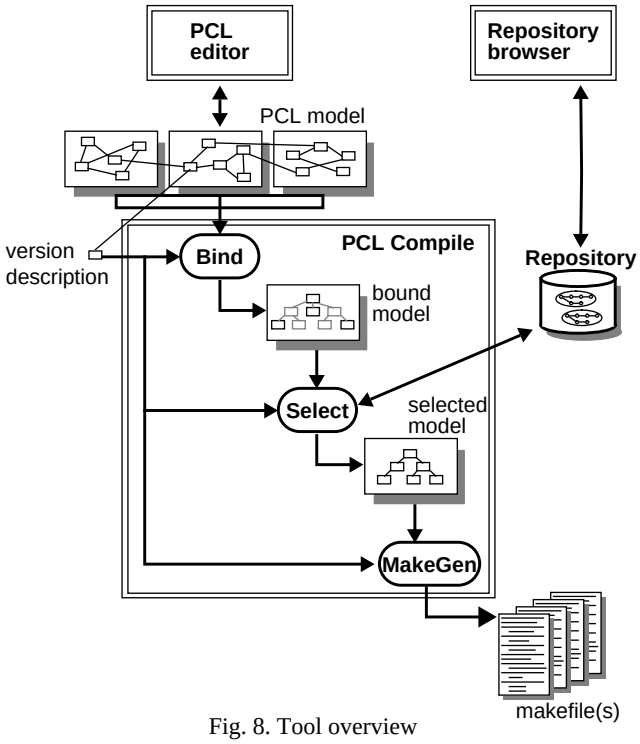

Fig. 8. Tool overview

mentary PCL description for existing software products. Figure 9 shows the user interface for the PCL compiler and the Repository browser.

The tool set is implemented in C++ using X11 and the OSF/Motif™ toolkit. The core tool set is about 60 KLOC. It is currently available for Sun and HP workstations. BMS, a selective multicast package provided by CAP Gemini, is used for tool integration, both of the PCL tool set itself and for integration with external design tools. The Repository is currently implemented on top of RCS [16].

5 Preliminary Experiences

PCL and its tool set is currently being validated at four different partners in the PROTEUS project, on applications ranging from telecommunications software to system development tools. Reported benefits include (see also [7]):

- Increased system *visibility*, i.e. recording and formalizing knowledge previously distributed and unavailable (person dependent). This system documentation is essential for controlling the system evolution (impact analysis of changes), but has in addition proved valuable for internal communication and training.
- Integrating the system manufacture process into the system configuration support

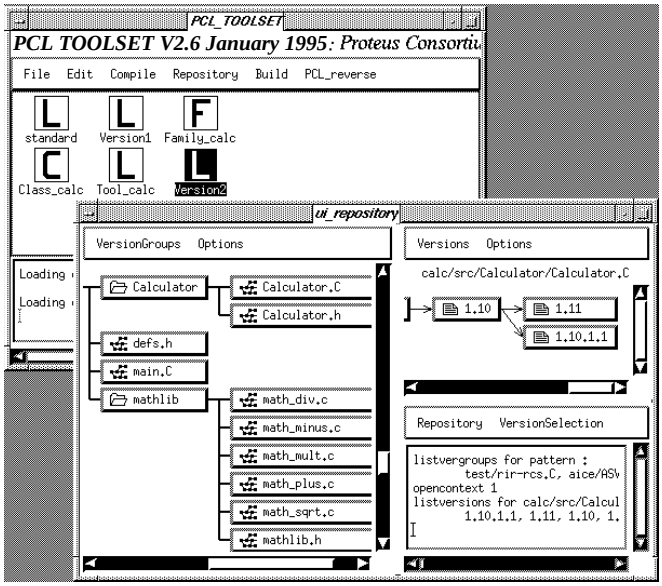

Fig. 9. PCL compile main window and Repository browser.

has been acknowledged by several of our partners. Manual maintenance of make-files and shell scripts for each system variant is avoided.

- People outside the development team may specify and build a release, based on desired properties expressed by customers.
- The *test space*, i.e. the set of configurations which must be tested after changes, is made explicit.

As a system evolves, structural changes need to be reflected in the PCL model. In order to ease the creation and maintenance of PCL system models, different strategies have been chosen. For one partner, a CASE tool has been tightly integrated with the PCL tool set, providing automatic propagation of changes. For file-base software systems, the PCL Reverse tool is able to both generate an initial PCL model and to check consistency between a system model and an actual system version in a workspace.

6 Conclusions

In this paper we have presented the PROTEUS Configuration Language and its supporting tool set. PCL supports comprehensive system modelling and provides expression of variability in the logical system model, in the mapping from the logical model to files, in the version selection, and finally in the system manufacture process. Intensional system configuration using attribute assignment accommodates configuration binding and system building in a concise and reproducible manner.

We have illustrated the important concepts in PCL on a small, but complete

example. The example has been illustrated with screen dumps from the PCL tools.

Experience shows that it requires some effort to build a comprehensive system model, especially if trying to incorporate all potential variability in an industrial product. However, the benefits in terms of improved system visibility and automation are significant.

The PCL tool set will be made available by the PROTEUS consortium. Further information about the PROTEUS project, PCL and its tool set is available at http://www.comp.lancs.ac.uk/computing/research/soft_eng/projects/PROTEUS/.

Acknowledgment

We would like to thank the anonymous reviewers, Richard Sanders and Joe Gorman (SINTEF) for their constructive comments on earlier versions of this paper.

The work reported has been supported by the European Commission's ESPRIT programme. We would like to acknowledge the other members of the PCL development and tools implementation teams, namely Ian Sommerville, Graham Dean (Lancaster University), Björn Grönquist, Ariane Suisse, Gilbert Rondeau, Sergio Calabretta (Cap Gemini Innovation, Grenoble).

References

1. E. Borison: A Model of Software Manufacture. In Reidar Conradi et.al., editors, *Proceedings of the International Workshop on Advanced Programming Environments*, Trondheim, Norway, June 16-18, 1986, *LNCS no. 244, Springer-Verlag, Berlin*, pp. 197-220.
2. L. W. Cooprider: The Representation of Families of Software Systems. PhD thesis, *Carnegie-Mellon University, Computer Science Department*, April 1979.
3. F. DeRemer and H. H. Kron: Programming-in-the-large Versus Programming-in-the-small, *IEEE Transactions on Software Engineering*, SE-2(2), June 1976, pp. 80-86.
4. J. Estublier: A configuration manager: The Adele data base of programs. In *Workshop on Software Engineering Environments for Programming-in-the-Large*, Harwichport, Massachusetts, June 1985, pp. 140-147.
5. J. Estublier and R. Casallas: The Adele Configuration Manager. In W. F. Tichy, *Configuration Management*, John Wiley & Sons Ltd., Chichester, 1994, ISBN 0-471-94245-6, pp. 99-133.
6. S. I. Feldman: Make, a Program for Maintaining Computer Programs, *Software - Practice and Experience*, 9(4), April 1979, pp. 255-265.
7. B. Gulla and J. Gorman: Supporting evolution with a configuration language: industrial experience, 11 pages. Submitted for publication.
8. D. L. Leblang: The CM Challenge: Configuration Management that Works. In W. F. Tichy, *Configuration Management*, John Wiley & Sons Ltd., Chichester, 1994, ISBN 0-471-94245-6, pp. 1-37.
9. A. Lie, R. Conradi, T. M. Didriksen, E.-A. Karlsson, S. O. Hallsteinsen and P.

Holager: Change Oriented Versioning in a Software Engineering Database. In W. F. Tichy, editor, *Proceedings of the Second International Workshop on Software Configuration Management*, pp. 56-65, Princeton, NJ, October 25-27, 1989. ACM SIGSOFT Software Engineering Notes 17(7), November 1989.
10. K. Marzullo and D. Wiebe: Jasmine: A Software System Modelling Facility. In P. B. Henderson, *Proceedings of the 2nd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Palo Alto, CA, December 9-11, 1986. *ACM SIGPLAN Notices*, 22(1), January 1987, pp. 121-130.
11. K. Narayanaswamy and W. Scacchi: Maintaining Configurations of Evolving Software Systems, *IEEE Transactions on Software Engineering*, SE-13(3), March 1987, pp. 324-334.
12. PROTEUS consortium: PCL-V2 Reference Manual, Technical Report P-DEL-3.4.D-1.9, September 1994, 85 pages.
13. I. Sommerville and G. Dean: PCL: A configuration language for modelling evolving system architectures, 21 pages. Submitted for publication.
14. R. Thomson and I. Sommerville: An Approach to the Support of Software Evolution, *Computer Journal*, 32(5), October 1989, pp. 386-396.
15. W. F. Tichy: Software Development Control Based on Module Interconnection. In *Proceedings of the 4th International Conference on Software Engineering*, IEEE, September 1979, pp. 29-41.
16. W. F. Tichy: RCS - A System for Version Control, *Software - Practice and Experience*, 15(7), July 1985, pp. 637-654.
17. D. Whitgift: Methods and Tools for Software Configuration Management, *John Wiley & Sons Ltd., Chichester*, 1991. ISBN 0-471-92940-9

Appendix: The Calculator Example

Below follows the complete PCL description for calculator example. Note that the latter half of this description is independent of the actual example system, allowing it to be shared among different models.

```
version my-version of CalcProg
   attributes os := sun;
              xgui := true;
   end
   parts      ui => ui-version; end
end

version ui-version of XGUI
   attributes INCL := "-I/local/X11R5/include "; end
end

family CalcProg
   attributes created_by= "Marius Kintel";
              created : string = "94/08/12";
              contract_no: integer = 1643256;
```

```
            HOME : string default "/home/ask/proteus/test";
            workspace := HOME ++ "/calc/src/";
            repository := "calc/";
            status   : status_type exported default initiated;
            xgui     : boolean default false;
            expression : expr_type default infix;
            debug    : boolean default false;
            os       : os_type;
            DEBUG := if debug = true then "-g -Ddebug " endif;
            INCL   : string := "";
            CCFLAGS : string :=
               if os = sun then
                  if expression = infix then DEBUG ++ "-c -Dsun4 "
                  elsif expression = reverse_polish then DEBUG++"-c -O2"
                  endif
               elsif os = vax then DEBUG ++ "-C -c -Dvax "
               endif;
   end
   parts      ui => if xgui = true then XGUI endif;
            calc => Calculator;
            math => mathlib;
   end
   physical   main => "main.C";
            defs => "defs.h";
            exe => "calc.x" attributes workspace := HOME ++ "/calc/bin"; end
                        classifications status => standard.derived; end;
   end
end

family XGUI
   attributes INC := 'INCL; end
   physical   files => "ui.C"; end
end

family Calculator
   attributes workspace := 'workspace ++ "Calculator/";
            repository := 'repository ++ "Calculator/";
            expression : expr_type := 'expression default infix;
            status : status_type := 'status;
            INCL := 'INCL;
   end
   physical   calc => ("Calculator.C", "Calculator.h");
            expr => if expression = infix then
                  ("expr.C", "expr.h")
                  elsif expression = reverse_polish then
                  ("rpn_expr.C", "rpn_expr.h")
                  endif;
   end
end

family mathlib
   attributes workspace := 'workspace ++ "mathlib/";
            repository := 'repository ++ "mathlib/";
            INCL := 'INCL;
```

```
   end
   physical   files => ( “math_plus.c”, “math_minus.c”, “math_mult.c”, “math_div.c”, “math_sqrt.c”,
                  “mathlib.h”);
              lib=> “libmath.a” classifications status => standard.derived; end;
   end
end

attribute_type status_type
   enumeration initiated, module-tested, system-tested end
end

attribute_type os_type
   enumeration sun, vax end
end

attribute_type expr_type
   enumeration infix, reverse_polish end
end

tool CC
   attributes CC : string default “CC “;
              CCFLAGS : string default “ “;
              INCL: string default ““;
   end
   inputs     InSrc => cpp-source; end
   outputs    OutObj => obj-code; end
   scripts    build := CC ++ CCFLAGS ++ INCL ++ “-o “ ++ OutObj ++ “-c “ ++ InSrc; end
end

tool cc
   attributes cc : string default “cc “;
              CFLAGS : string default “ “;
              INCL : string default “ “;
   end
   inputs     InSrc => c-source; end
   outputs    OutObj => obj-code; end
   scripts    build := cc ++ CFLAGS ++ INCL ++ “ -o “ ++ OutObj ++ “-c “ ++ InSrc; end
end

tool ar
   attributes AR : string default “ar “;
              ARFLAGS : string default “rv “;
              RANLIB : string default “ranlib “;
   end
   inputs     InObj : multi => obj-code; end
   outputs    OutLib => library; end
   scripts    build := AR ++ ARFLAGS ++ OutLib ++ “ “ ++ InObj ++ ”\n” ++ RANLIB ++ OutLib;
   end
end

tool ld
   attributes LD : string default “CC “;
              LDFLAGS : string default “ “;
              LIBS : string default “-lm “; end
```

```
   inputs     InObj : multi => obj-code;
              InLib : multi => library; end
   outputs    OutExe => exe-file; end
   scripts    build := LD ++ LDFLAGS ++ " -o " ++ OutExe ++ " " ++ InObj ++ " " ++ "'fixlib "
                  ++ InLib ++ "' " ++ LIBS; end
end

class text inherits standard.software end
class source-code inherits text end
class binary inherits standard.software end

class cpp-source inherits source-code
   tools CC end
   physical name ++ ".C"; end
end

class c-header inherits source-code
   physical name ++ ".h"; end
end

class c-source inherits source-code
   tools cc end
   physical name ++ ".c"; end
end

class library inherits binary
   tools ld end
   physical "lib" ++ name ++ ".a"; end
end

class obj-code inherits binary
   tools ar, ld end
   physical name ++ ".o"; end
end

class exe-file inherits binary
   physical name ++ ".x"; end
end
```