PCL: a language for modelling evolving system architectures

by Ian Sommerville and Graham Dean

The paper describes a language called PCL, which has been designed to model the architecture of multiple versions of computer-based systems (system families). PCL has evolved from module interconnection languages, and its novel features include the ability to model the variability between members of a system family and facilities for the integrated modelling of hardware, software and documentation structures. The features of PCL are illustrated using a number of simple examples. A supporting toolset for PCL has been implemented and is briefly described.

1 Introduction

A key problem in supporting the evolution of computer-based systems is the management of variability. As systems evolve, new versions are created so that, rather than a single system, a *family* of systems is created. In general, the different versions of the system will share components. Some of these components will be stable across all versions. Others will vary depending on the specific requirements of a particular version.

This variability causes problems as more and more versions are produced. It becomes increasingly difficult to consider the system as a single entity, rather than a disparate collection of loosely related systems. Where changes may affect more than one version, the processes of traceability, change analysis and change implementation may have to be repeated for each version even when these versions have much in common.

Variability across versions of a software system occurs at a number of different levels.

- *Structural variability*: the designs of different versions may have different architectures, i.e. different versions will be composed of different configurations of components. In some cases, interface structure may also vary depending on version functionality.
- *Implementation variability*: the implementation of system components may vary depending on non-functional requirements such as performance and differences in the implementation platform, i.e. the same logical design component may have different source code representations depending on what machine they are designed for, how fast they must run etc.
- *Installation variability*: the runtime configuration of the system may vary depending on the execution platform where it is installed. For example, in a distributed system, the number of copies of a process may depend on the number of processors. In a heterogeneous system, these copies need not all be the same.

The variability in versions of application software system often directly reflects variability in underlying hardware platforms and support software. Furthermore, for each version of the application software and underlying platform, different documentation may be required. Therefore, as well as software variability, there may be versions of a system created as a result of *system* variability:

- ☐ *documentation variability*, where different versions of a system have different documentation structures and links to software and platform components.
- ☐ *platform variability*, where different platforms for the system have different hardware structures, operating systems and support software.
- ☐ *tool variability*, where different tools are used to create different executable system versions and the associated documentation depending on the system execution platform.

The principal objective of the work described here therefore was to develop a way of modelling, in an abstract way, all the different versions of a system. Such a description provides a basis for change prediction because it identifies the stable and variable parts of the system. It reduces the costs of change analysis across versions and facilitates the reuse of parts of the system. Variability modelling was therefore a key requirement for our work and is its most important novel characteristic.

As well as the need to represent variability, we also identified further requirements for a notation used to model the family of variants which make up a system. These were

- *integrated systems modelling*: the language must be able to model all of the entities and dependencies which make up a system. This means that the system model should include information about the software, hardware and documentation structures. It must also support different 'views' of the system structure. These include views

defining the interface offered to other systems, the static and dynamic composition of the system, and how a software system or component is represented as a set of physical files in a filestore or version management system.

- *object-oriented modelling*: the language must be able to model object-oriented systems. This was an important requirement as partners in this work were concerned with developing object-oriented extensions to existing design methods: HOOD [1], SDL [2] and Modular Design (an extension of Structured Design).
- *design method integration*: designers need to reflect design method-specific types and relationships in their architectural description. The architectural description language should not therefore have a fixed and pre-defined set of classifications and relation types, but must allow multi-dimensional, extensible entity classification and user-defined relations.

To address these requirements, we have developed a language for describing system families called PCL (Proteus Configuration Language) [3]. The work took place in the context of a collaborative project called Proteus, whose objective was to develop improved methods and tools for system evolution. As well as the work on modeling evolving systems described here, the Proteus Project has developed tool support for system building and version management, object-oriented extensions for design methods, process support for software evolution and tools for managing design rationale.

PCL is a dual-purpose language. It is a modelling language which may be used to describe, at an abstract level, the architecture of different system versions, the system documentation structure, the system hardware etc. It is also a configuration language as the software description of PCL specifies component dependencies, and hence may be used as the basis of a system building and version management system. This paper concentrates on the architectural modelling capabilities of PCL, rather than on its configuration capability. These aspects of the language which allow modelling of storage structures in a repository, the description of tools which are used to build a version of a system and relationships to support system building are not covered here. They are described elsewhere by Tryggeseth and Gulla [4].

2 Related work

The modelling of system architectures has now become an important research topic [5, 6], but the field is still relatively immature. There are no widely used architectural modelling languages, and work in this area is being carried out under a number of different headings.

☐ *Module interconnection languages*: these are languages which are used to describe the composition of a system in terms of its modules. Typically, these languages support a system description of the static structure of modules and their interfaces with no embedded control information.

☐ *Configuration languages for version description*: these languages are intended to describe the different versions of a system for configuration management. Their principal focus is the modelling of the physical structure of a system (i.e. the code structure) so that the required version can be built on demand.

☐ *Component description languages*: these are languages which have been developed to support software reuse and the construction of systems from reusable components written in different languages. In essence, the component description is the 'glue' between the reusable components. Owing to the need to ensure that the components work properly together, detailed interface description and checking is an important part of component description languages.

☐ *Languages for dynamic system re-configuration*: these are languages for describing systems which may be re-configured without interruption of service. The principal requirement here is to ensure that components are independent so that languages provide isolation constructs to ensure that components may communicate without the need for mutual knowledge of the component structure or location.

Graphical notations as used in design methods such as Hood [1] may also, of course, be used for describing the abstract structure of a system. However, these notations do not provide adequate facilities for describing system variability. Either they provide no support for expressing variability (the most common situation) or, if they are object-oriented, the only support for variability is through the inheritance mechanism. Methods do not allow the association of alternative structures with components, and they assume a 1 : 1 relationship between design and implementation components. Few methods provide support for specifying hardware/software relationships. We required a method-independent description of the system architecture, and so we did not use any specific method as a basis for our work.

The most important influence on the architectural description features of PCL was previous work on module interconnection languages (MILs). These are based on DeRemer and Kron's seminal 1976 work [7], and include languages such as INTERCOL [8], NuMIL [9] and SySL [10]. Apart from SySL, MILs have been solely concerned with software description and have not provided facilities for describing hardware or document structures. Prieto-Diaz [11] provides an excellent and very thorough discussion of the development of MILs and compares those languages which had been designed at the time of that survey. These were all developed before object-oriented concepts were widely known, although their notion of a module (code plus data) is comparable to an object.

MILs are best suited for providing a description of the static system structure and none of the above MILs includes facilities for describing dynamic system models. They provide some interface description but, in general, not to the same extent as the component description languages discussed below. Early MILs concentrated on modelling single versions of a system, but developments by Cooprider [12] and Tichy [8] allowed system families to be described. This notion of a system family has influenced almost all later work in this area.

More recently, Hall and Weedon [13] have investigated object-oriented module interconnection languages, and new architectural description languages such as UniCon

[14] and Rapide [15] have been developed. In contrast to text-based languages, Dean and Cordy [16] have developed a graphical language for architectural description. All of these were published during the development of PCL, and so they did not influence the language design.

As we intended PCL to be used to support system building, languages for version description were also an important input to our work. These languages include simple languages such as Make [17], which describe more abstract system building languages such as DSEE [18] to database configuration languages such as that provided by the Adele system [19]. These languages were relevant because we planned to integrate PCL to version management systems and provide a system building facility. As discussed later, we use the notion of attributes from Adele for version indentification and selection and we generate the notation used by Make as part of the system building process.

We considered component description languages, summarised by Whittle [20], to be of less relevance because of their focus on system assembly. For example, early work in this area by Goguen [21] used formal interface specifications to check the compatibility of components written in different programming languages. This has influenced a number of later developments in this area such as ACT TWO [22], CIDER [23], Meld [24] and LILEANNA [25]. Most of these languages have been developed in the context of research projects on software reuse and do not provide facilities for variability expression or integrated system modelling.

Similarly, languages for dynamic system reconfiguration are not oriented towards variability expression. We examined program configuration languages including Conic [26], its successor Darwin [27] and POLYLITH [28]. The focus of these languages is to provide support for the dynamic modification of system structure, rather than the architectural description of system families. However, a separate development of the PCL language has examined its use in this area [29] and has shown that PCL may be used to support dynamic reconfiguration.

3 Basic PCL concepts

As discussed above, we required concepts in PCL to model the stable and variable parts of hardware, software and documentation entities, and the relationships between them. These entities had to be modelled at different levels of abstraction from generic to concrete. After experimentation, we finally established six basic types of entity which should be included in PCL.

- *Family entities*: used to define the architecture of hardware, software or documentation components in a system. Family entities may incorporate variability, and therefore a single family entity can represent a set of versions of a component.
- *Version descriptor entities*: used to define the specific attributes of a single version of a system.
- *Tool entities*: used to define tools which may be used to build a system whose architecture is modelled in PCL. Tool descriptions include a description of the tool inputs and outputs, and the command syntax required to execute these tools.
- *Classification definitions*: used to define classification terms which may be associated with a family entity. All classifications are derived from basic classifications, including *hardware*, *software* and *document*.
- *Relation definitions*: used to define relations which may exist between family entities, family entities and version descriptor entities, or family entities and tool entities in a system description. The relationships derived from these relations are established within entity descriptions, as described below.
- *Attribute type definitions*: used to define attribute types as an enumerated set of identifiers.

Classification definitions, relation definitions and attribute type definitions are used to extend the basic facilities of PCL and to link it with specific design methods. Tool descriptors are used when the PCL architectural model is used as a basis for the system building process. This is not central to the notion of architectural modelling that we cover here, and this concept is explained elsewhere [4].

Family entities are used to model the architecture of a set of systems or components with stable and variable parts. A stable part of a family is that part which is common to all versions. A variable part differs from one version to another. Version descriptor entities are used in conjunction with family entities to define the architecture of a unique version of a system. The family entity describes a version set, and the version descriptor defines a version within that set. When a version descriptor is associated with a family, the information in the version descriptor is used to remove variability from the family description. We have developed tools to support this process, as described later.

Family entities may be related to other family entities in a number of different ways.

☐ They may inherit information from other family entities.
☐ They may be logically composed of other family entities.
☐ They may participate in a variety of user-defined and built-in relationships with other family entities.

Within each family entity, there are a number of different sections which provide different types of family information. Fig. 1 shows these different sections and illustrates that the application of a version descriptor results in the generation of a single system version.

The different sections associated with a family entity are as follows.

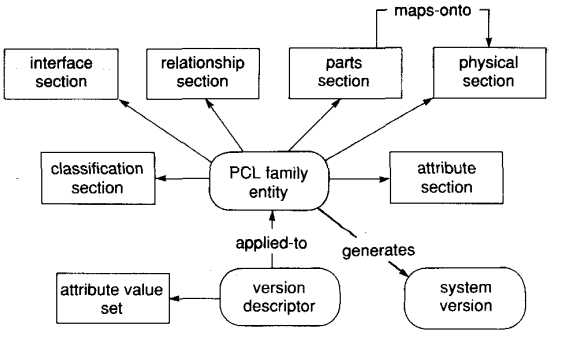


Fig. 1 PCL entity structure

- A classification section is used to classify an entity in a number of different dimensions. For example, we can classify a single family entity as *software* (meaning it can be represented electronically in some storage system and processed by tool), *process* (meaning it can be thought of as an executable entity in its own right) and *primary* (meaning that sources are available and the entity can be the input to a translation process).
- An attributes section defines the attributes associated with an entity. The attributes may be used to provide entity information such as the date of creation, the author etc. Attributes may also be used as variability specifiers where the different versions of a system are characterised using a set of attribute values.
- An interface section defines the names exported by the entity to other entities. This is equivalent to the *provides* construct which is supported in other MILs.
- A parts section is used to specify the composition of the entity in terms of other PCL entities. The parts section is therefore a list of the components which make up an entity.
- A physical section is used to specify the physical entity names (e.g. source code, object code) where the entity is stored. This section links a PCL entity to one or more files which contain the code implementing the entity.
- A relationships section is used to set out the relationships between PCL entities. In a system there are many relationships between different parts such as *calls*, *passes-data-to* etc., and it is important to be able to represent them in a system model. There are built-in relations in PCL such as *implemented-on* and *requires*, but most relations are application-specific and a facility to define these relations is provided in the language.

In all of these sections, conditional inclusion is supported, thus allowing variability between different versions of a system to be specified. We illustrate this and other features of PCL with examples below.

4 Modelling system families

System families are made up of different versions of a system where each version differs, in some way, from the others. These reflect different logical system structures where system compositions include different components, or different implementation structures where different versions include different implementations of the same component. To support these, PCL structural descriptions include a description of the logical system composition and a mapping between this and the components (usually stored in files) which implement it. An abstract description of a component interface and a description of the relationships between a component and other components may also be defined.

There may be variability in any of these structural descriptions. To support the modelling of this variability, we have incorporated two constructs in PCL for variability expression. These are

☐ *conditional structural expressions*: we assume that system families are normally composed of a stable part, which is common to all versions, and a variable part which is version-dependent. The variable part is described by using conditional expressions to show when parts of the structure are and are not included.

☐ *object-oriented modelling*: the stable parts of system components can be reflected in some common ancestor and inherited by specific components. These are then supplemented with details of a specific component.

PCL supports all three types of variability identified in the introduction namely structural, implementation and installation variability. Structural variability is supported by conditional structural expressions and inheritance. Implementation variability is supported using comparable conditional binding to the physical section of a description. We also provide a further level of implementation variability support by allowing different *implementations* of a PCL family entity in a version management system to be identified using a set of attributes. The attributes specified in a version descriptor are passed to a PCL tool, called the repository manager, which retrieves the appropriate files from a version management system. Our current implementation of this is based on RCS [30].

4.1 Conditional structural expressions

Conditional structural expressions may be used to describe the variable parts of a system family. They resemble conditional expressions in a programming language, in that they have a selection condition which is used to choose a 'true' or a 'false' part. Whether a component is included in a system depends on the values of attributes used in the selection condition. We illustrate both the structural parts of a PCL description and the use of conditional structural expressions to describe variability using a (simplified) example of a print server (Fig. 2).

The print server offers various services through its interface (print file, examine print queue, remove from print queue) and is composed of several parts which are themselves described by PCL entities. There are two versions of this system, depending on whether the system must handle multiple paper types.

Structural sections in a PCL description are used to describe interfaces, the composition structure and the physical structure of entities. Each section is presented as a list of slots with associated slot values. Slots act as placeholders identifying information that should be provided. They may be left unfilled at one level of description and instantiated at later levels. They allow descriptions to be modified when they are used as a parent for some other description (see the discussion on inheritance later).

Slot names are to the left of ⇒ in the examples, with slot bindings to the right. Therefore, in Fig. 2, the slot PRINT in the parts section has an associated PCL entity called 'printer-controller'. The slot SOURCE in the physical section is bound to a filename ('print_sever'), where the source code is stored.

The binding associated with a slot may be conditional on some attribute value. Therefore, the SELECT-PAPER-TYPE slot in the parts section of Fig. 2 is only bound to the component 'set-input-tray' if the attribute 'multiple-paper-types' is true. Otherwise, the component 'set_input-tray' is not included in the system. We discuss how variability is removed from a family description to create a unique system description below.

```
family print_server

attributes
        multiple_paper_types: boolean;
end
interface
        PRINT ⇒ print_file;
        DISPLAY_QUEUE ⇒ show_print_queue;
        DEQUEUE ⇒ delete_print_job;
        SELECT_PAPER_TYPE ⇒
        if multiple_paper_types then
          set_input_tray endif;
        SELECT_PRINTER ⇒ set_printer;
end
parts
        PRINT ⇒ printer_controller;
        QUEUE ⇒ queue_manager;
        SELECT_PAPER_TYPE ⇒
        if multiple_paper_types then
          set_input_tray endif;
end
physical
      SOURCE ⇒ "print_server"
        attributes
            repository := "/usr/src";
        end
      EXECUTABLE ⇒ Print_man
        attributes
            repository := "/usr/utils/bin";
        end
   end
end//print_server
```

This attribute is set true when a printer has multiple paper trays.

The interface view sets out the names which are exported by the print server's interface.

Notice the variability here. The interface set_input_tray is only provided if the attribute multiple_paper_types is true.

This section defines the logical components of the printer server. Notice that these slot names and component names can be (but don't have to be) distinct from the interface names. The name of the component 'set_input_tray' is the same as the interface name which is exported.

This is the file where the source code of the print server is stored.

This defines the name of the file which is the executable version of the system.

Fig. 2 PCL description of print server

The files that contain the source code of the print server program are specified in the physical part. The storage model that we assume is that all source files are held in some repository file system and the keyword **repository** identifies their location in that system. When these are compiled, we assume that they are 'checked out' of this repository into some other file system for compilation. The **repository** keyword defines where the managed copy of the file is maintained. As discussed above, different implementations of the same PCL entity are identified by attribute values and so, implementation variability may be supported without undue clutter of the structural description.

PCL family descriptions always include an attributes section which allows for the declaration of two different types of attribute.

☐ Identifying attributes are used to distinguish between different versions of a PCL entity. In Fig. 2, the attribute *multiple_paper_types* is an identifying attribute, which can be used to select the version of the description for a particular printer. Identifying attributes may be used as selectors in conditional structural expressions. Identifying attribute values are set in a version descriptor and assigned values when variability is removed from a PCL description.

☐ Documentation attributes (identified by using = to associate values with them) are used to add information to the system model. Identifying attributes are given a value when they are defined. In Fig. 8, for example, the attribute *author* is a documentation attribute that states who was responsible for developing the system being described.

When a specific version of a system family is required, the conditional structural expressions must be evaluated and variability removed. Fig. 3 illustrates this process of structural variability removal. The binding of a value to a slot or to an attribute depends on the evaluation of the associated conditional expression. The attributes referenced in the conditional expression are not evaluated when they are defined but at a later stage, which we call bind-time. At that stage, the PCL user defines a set of attribute values in a version descriptor. A tool called PCL_bind processes the PCL description, discovers the attribute values which are set in the version description and then assigns these values to the appropriate attributes. The conditional expressions may then be evaluated.

The effect of this application is to resolve conditional expressions and generate another PCL system model without variability. This represents a single version of the system. This model and the attribute values are then passed to the Repository manager, which selects the required files to build a unique system version.

Version identification is a well known configuration management problem, and version identifiers often become

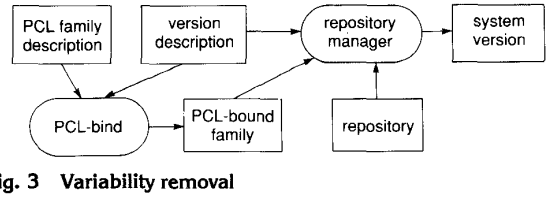


Fig. 3 Variability removal

version Acme_version **of** Some_system **attributes** language := "Ada"; manufacturer := "HP"; customer := "Acme_widgets_corp"; **end** **end**//Acme_version	Some_system is the name of the family to which this version descriptor is applied. Acme_version is the name of the version descriptor. These attribute values must uniquely define a single member of the family.

Fig. 4 PCL version description

long and obscure (e.g. version 4.1.3.2). A version numbering scheme gives no clue as to the distinguishing characteristics of that version. Rather than use numbering therefore, we identify versions using the approach adopted in the Adele system [31], i.e. attributed identification. If we have a version of a system which has been developed for a particular type of machine, using Ada and is delivered to a particular customer, we can identify this as the version which is programmed in *Ada*, for *HP workstations* and which was delivered to *Acme widgets corp*.

We incorporate these in a version description (Fig. 4) which can be applied to a family to remove variability.

The attribute values set in the version descriptor are propagated to the family 'Some_system' and attributes which are identifying attributes are assigned these values. These attribute values are also propagated to all components which are part of 'Some_system' so that variability in these components is also controlled by these values.

The support of installation variability also depends on setting attribute values in version descriptors. Assume that a system is composed of a number of switch controllers which run on separate processors. The family description is structured so that the number of processes, described in the parts section, is left variable:

SWITCHES ⇒ Switch_procs[NumbProcessors]

In the version descriptor, the actual number of instances is instantiated and the attributes of each instance is set. These can be set individually or in groups. Assume that the attributes 'Proc' and 'Connections' must be set for all switches. This would be written as shown in Fig. 5. Individual instances of family entities can thus be configured when the system version is created.

4.2 Object-oriented modelling

As part of the Proteus project's support for system evolution, we have extended several design methods to support object-oriented development. It was therefore a requirement on PCL that an inheritance relation should be provided so that the design structures could be modelled in PCL. Inheritance, however, is a generally useful facility in creating architectural models as many systems may be described as modifications to some generic base system.

The inheritance facility is, of course, an alternative construct for representing variability. It is possible to define generic components at the base of an inheritance hierarchy and to extend these in different variations. However, variability may be controlled by multiple attribute values (e.g. **if** a **and** b **and** c..). This complex conditional variability is very difficult to express using inheritance, and our experience has been that conditional structural expressions lead to a more compact description of system families.

Fig. 6 is a simple illustration of the use of inheritance in PCL. In this example, we have taken the description of the print server shown in Fig. 2 and added a new facility to reorder to queue of jobs to be printed.

5 Integrated system modelling

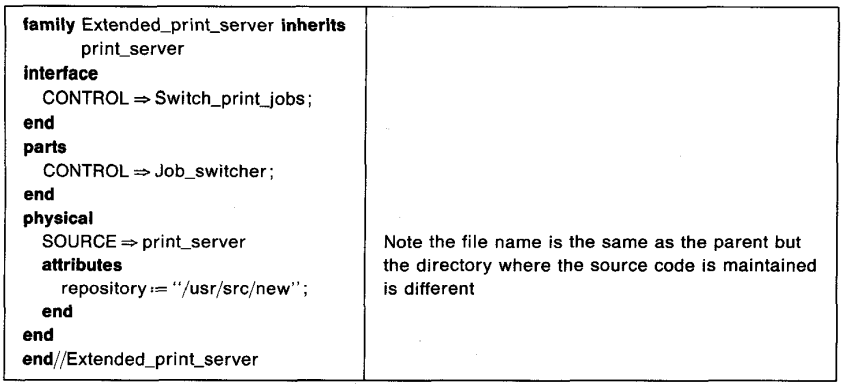
A significant problem that arises during system evolution is discovering the hardware dependencies which are implicit in many system components and tracing the documentation which must be changed when a software change was made. Furthermore, system evolution sometimes involves the re-implementation of software parts of a system in hardware, or *vice versa*. Different system versions might require different hardware configurations. We wished to integrate hardware, software and documentation descriptions in a single description.

The following facilities in the language support integrated system modelling:

- an extensible classification scheme which allows family entities to be classified. These classifications are derived from three fundamental base classes: *hardware*, *software* or *document*. We can therefore represent dependencies

version XYZ_telecom **of** Switch_controller **attributes** NumbProcessors := 10; Proc := (1..5 ⇒ "M68020", 6..10 ⇒ "M68030"); Connections := (8, 16, 3..10 ⇒ 32); **end** **end**//XYZ_telecom	Switch_controller is the name of the family to which this version descriptor is applied. There are 10 processors in the system. In switches 1 to 5, Proc is set to M68020, in 6_10 it is M68030 In switch 1, Connections is 8, in switch 2, it is 16, in all others it is 32

Fig. 5 Version description showing how attributes of a collection are assigned values



```
family Extended_print_server inherits
       print_server
interface
  CONTROL ⇒ Switch_print_jobs;
end
parts
  CONTROL ⇒ Job_switcher;
end
physical
  SOURCE ⇒ print_server
  attributes
    repository := "/usr/src/new";
  end
end
end//Extended_print_server
```

Note the file name is the same as the parent but the directory where the source code is maintained is different

Fig. 6 Extending the print server using inheritance

between these classes of entity and browse them using the editing facilities described in Section 9.

- build-in relations which are used to link entities of different types. For example, the relation *executes_on* links hardware and software entities.
- separate physical and logical descriptions so that the logical structure of an entity can be maintained while its physical realisation as a source program or even as a hardware component evolves. Multiple physical implementations can be associated with a single logical part. In this respect, PCL is distinct from the configuration languages associated with version management system such as DSEE, which assume a 1 : 1 mapping between the logical structure and its physical realisation.

As an illustration of these facilities, the PCL descriptions in Figs. 7 and 8 show how hardware and software can be described and linked. Fig. 7 is a description of the hardware platform for the set of tools developed to support PCL, and Fig. 8 is a top-level structural description of that toolset. These are linked through the *executes_on* relation in the relationships section of Fig. 8.

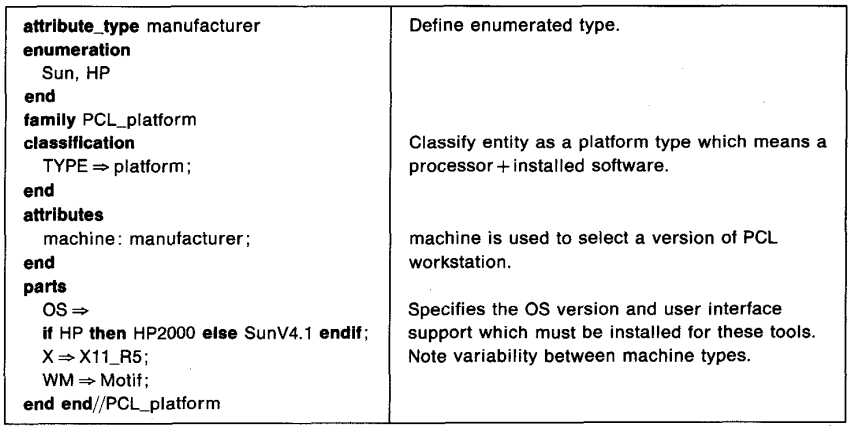
Fig. 7 is a PCL description of the platforms on which the set of tools to support PCL have been implemented (currently UNIX workstations from HP and Sun). The platform description specifies the operating system version and the user interface software that must be installed. At this level of platform description, variability is confined to different operating systems. However, at more detailed levels, variability in terms of hardware components (e.g. graphics cards etc.) can easily be expressed.

Fig. 8 shows the software description of the PCL toolset. This toolset is composed of a number of different parts, each of which is described as a separate PCL family. The overall family description here summarises this structure. Note that, at this abstract level, there is no structural variability (all versions have the same abstract architecture) but there is documentation variability. Each toolkit version has a different user manual, as shown by the *documented_by* relationship in the **relationships** section of the description.

The relationships section is used to write down the relationships between a family entity and other components. Slots are also used with the relation name declared immediately after the slot name, and the entities participating in the relationship follow the ⇒ symbol. The pre-defined relation *executes_on* is used to link the hardware and the software descriptions.

The relationships section is also used to define new structural views of a system which supplement the built-in interface, composition and implementation views. This is most commonly used when we wish to describe a dynamic view of a system (i.e. as a set of processes) and link this with a static system view as a collection of components (Fig. 9).

We can model these different structures using separate



```
attribute_type manufacturer
enumeration
  Sun, HP
end
family PCL_platform
classification
  TYPE ⇒ platform;
end
attributes
  machine: manufacturer;
end
parts
  OS ⇒
  if HP then HP2000 else SunV4.1 endif;
  X ⇒ X11_R5;
  WM ⇒ Motif;
end end//PCL_platform
```

Define enumerated type.

Classify entity as a platform type which means a processor + installed software.

machine is used to select a version of PCL workstation.

Specifies the OS version and user interface support which must be installed for these tools. Note variability between machine types.

Fig. 7 PCL model of the platform for the PCL toolset

family PCL_Toolkit **classification** TYPE ⇒ CPP_source **end**	Classify as C++ source. This is derived from the base type *software*. This information is used by the system builder to chose a compiler for the system represented here.
attributes author: STRING = "Jim Crow"; machine: manufacturer; repository: STRING **default** "/usr/Proteus/PCL_tools"; **end**	The repository attribute may be set in the version descriptor. If it is not set then the default value specified here is used.
parts EDITOR ⇒ PCL_edit; COMPILER ⇒ PCL_compile; BINDER ⇒ PCL_bind; GENERATOR ⇒ Makefile_generator; REPOSITORY ⇒ Repository; **end**	There are five logical parts to the PCL toolset as described in Section 9 of this paper. This section shows that the toolset is composed of these parts, each of which is described by a PCL entity.
physical SOURCE ⇒ "PCL_main" **end**	These is the name of the file where the C++ source code of the system is held. The value of the *repository* attribute is automatically prefixed to this by the system building tools.
relationships MACHINE: executes_on ⇒ (PCL_platform); LIBS: requires ⇒ (C_lib, RogueWave) DOCS: documented_by ⇒ **if** machine = HP **then** (user_manual, Design_docs, HP_changes) **else** (user_manual, Design_docs) **endif**; **end end**//PCL_Toolkit	MACHINE states that the system will only execute on a PCL workstation. LIBS states that this component requires 2 libraries (RogueWave and C) to be available. DOCS states that its documentation depends on whether it is a Sun or an HP version. The entities user_manual, Design_docs and HP_changes are also PCL descriptions.

Fig. 8 PCL model of the PCL toolset

PCL family descriptions and link these using relationships which are either predefined in PCL or are user-defined. For example, in Fig. 9, the predefined relation *is_implemented_by* is used to show which of the components in the static decomposition structure implements each process in the dynamic structural description. This would be expressed as follows in PCL:

relationships
R1: is_implemented_by ⇒ A, (X);
R2: is_implemented_by ⇒ B, (Y);
R3: is_implemented_by ⇒ C, (P, Y);
end

Sometimes, different versions of a system have different mappings of processes to modules. This can easily be expressed in PCL using the support for variability. For example, in a different version of the above system. Process C mapped onto modules Y and R. This could be expressed as follows:

R3: is_implemented_by ⇒ **if** Version
= 'v1' **then** C, (P, Y) **else** C, (Y, R) **endif**;

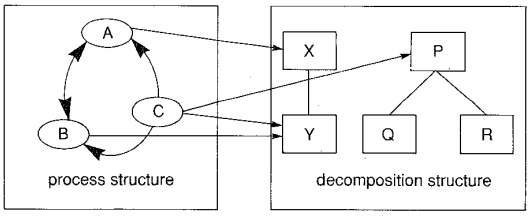


Fig. 9 Mapping from dynamic to static structure

6 Design method integration

Most previous work on languages for describing system structure has taken place in isolation, with little regard for integration with methods of structured design. An important requirement for PCL was that it should not be an isolated system but should be used to supplement design descriptions expressed in graphical notations. In essence, we should compensate for the limitations of design methods in variability description and integrated system modelling by supplementing them with PCL descriptions.

To link a PCL description with designs from a structured method, two requirements must be fulfilled.

☐ It must be possible to reflect the types and relations as used in the design method in the system modelling language. We do this by providing constructs for users to define family entity classifications and attribute types.

☐ It must be possible to express additional relationships between entities over and above those which could be expressed in the method. Method designers usually limit the relations which can be expressed and do not support any variability in relationships between entities. PCL includes a relation definition facility which can be used to create libraries of relations that are conformant with the relations supported by different design methods.

Fig. 10 illustrates examples of the class and relation definitions. We have already seen attribute type definitions in Fig. 8.

When combined with variability expression features, user-defined relationships may be used to link implementa-

class CPP_source **inherits** software
  suffix ".C";
  tool CPP_compile;
end
	Defines a software class which represents C++ source programs. The class definition also specifies a filename suffix to allow inferencing in system building and the name of a PCL tool entity (in this case the C++ compiler) used to process entities of this type.

relation Generates
  domain MD_design, HOOD_design,
  SDL_design;
  range CPP_source;
end
	Defines a relation which links C++ source and the corresponding design description expressed in some design method. The C++ can be automatically generated by design tools. In specifying relations, the domain (i.e. the source of the directed relation) and the range (the destination) are specified.

family MD_design_moduleA
  classification
  TYPE ⇒ MD_design;
  end
  relationships
  R1: Generates ⇒ CPP_moduleA;
  end
end
	This shows how the user-defined relationship is used to link entities. A C++ code skeleton may be automatically generated with further information filled in manually. This entity is described by the PCL entity called CPP_moduleA.

Fig. 10 Class and relation definition

tions which are automatically generated for specific platforms with a common source design. For example:

relationships
R1: Generates ⇒ **if** HP **then** CPP_moduleA
elseif Sun **then** CPP_moduleB
else CPP_moduleC **endif**;
end

Linking with designs is realised through the method support toolkit. In Hood, for example, PCL annotations may be associated with Hood objects, and we have introduced the notion of a 'unbound' object in a design to reflect the variant part of a family description. This has a PCL description associated with it. The PCL toolset is called from Hood when this description is to be edited or analysed.

7 PCL toolset

Tool support is required to manage the complexity of system models and to ensure model consistency. Our work in this area has been the development of a graphical browsing and editing tool for PCL, which presents a graphical view of a PCL description. Fig. 11 illustrates the user interface of this editing system.

The PCL editor provides structured editing and browsing facilities for PCL descriptions through a graphical interface. This aids comprehension and is a basis for navigation around a PCL description. By selecting an iconic representation of a component in the graphical structure display, its description can be displayed. The editor also includes an incremental compilation facility so that PCL descriptions can be modified without regard for syntax details or the compilation cycle.

The structure shown in Fig. 11 represents the architecture of a compilation system. The code generation part of this system (the leftmost icon) has an alternative structure, which is illustrated using query icons annotated with the appropriate condition. More detailed information about the entity is displayed in appropriate fields of the form. Different sections of the description can be selected for display.

The PCL browser and editor is one component of a comprehensive PCL toolset. The other components of the toolset were developed by CAP Gemini Innovation and by the Technical University of Trondheim. These include

- a language analyser (PCL_compile). This processes PCL text to create an abstract syntax tree. The compiler checks the syntax and semantics of PCL descriptions.
- a system binder (PCL_bind). Given a version description, this tool takes a PCL system model including variability and removes some or all of that variability depending on the specified version description. The removal of variability is an essential first step to automatic system building.
- a makefile generator (Make_gen). This takes a bound PCL description with variability removed plus tool descriptions and creates UNIX makefiles representing the system. These can then be processed to create an executable system version.

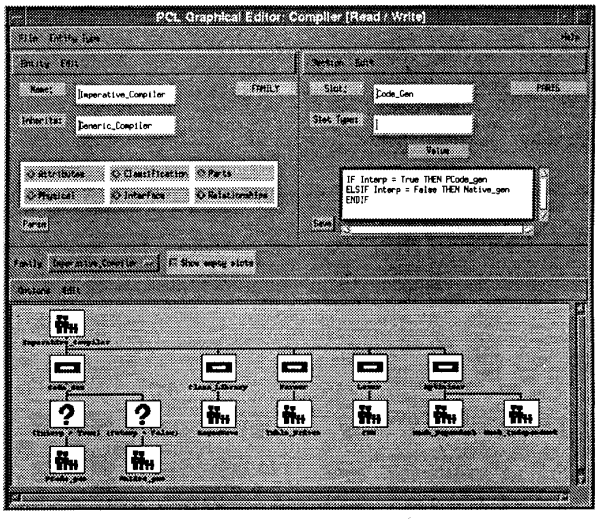


Fig. 11 PCL graphical editor

• a version management system (Repository). This extends the facilities offered by the RCS [30] version management system to provide the attributed identification of versions managed by RCS. We can thus use RCS to store and manage the source code of the different versions of a system implementation, and retrieve that code by specifying attributes in a version description.

Fig. 12 illustrates how these tools are integrated. Round-edged boxes represent tools, rectangles represent information exchanged.

PCL can, of course, be written by hand but this is both laborious and error-prone. We therefore normally expect the outline of a PCL description to be generated automatically. Further information is then added manually. This automatic generation is supported in two ways.

□ Where systems are under development, the system design may be used to generate the initial PCL description. We have modified CASE tools to generate such descriptions from designs expressed in SDL and in the notations used in the Modular Design method.

□ Where systems already exist, we can generate a PCL description from their file system representation. This uses a tool called PCL_reverse, which creates a PCL description to reflect the storage structure of components in a directory hierarchy. We have found that this is a common method of representing systems, and that this simple approach is a low-cost and effective method of generating PCL descriptions of existing systems.

The costs of documenting system structure have meant that structure description languages, such as MILs, have not been widely used. The facilities for automatic PCL generation significantly reduce the costs of generating integrated system descriptions so make PCL more cost-effective than alternative approaches.

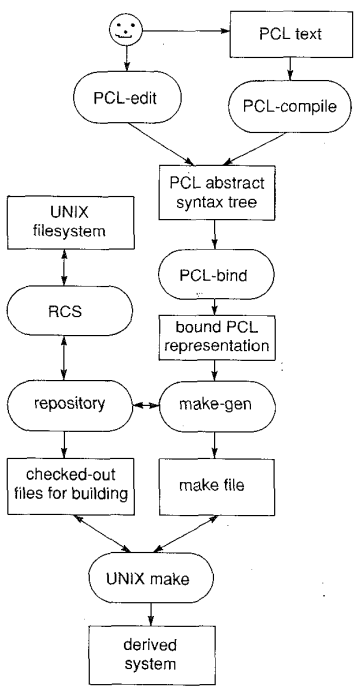


Fig. 12 PCL tool integration

8 Conclusions

We have briefly described a language called PCL, which has been designed to model the architecture of evolving systems. The novel features of PCL, in this context, include the modelling of variability between the different versions of a system, support for object-oriented models, support for the specification of relationships between different parts of the model, and version identification and binding through attributes.

PCL has been evaluated across a range of different applications. These include support software for satellites, CASE tools for a widely used European design method, a network management system and software for telephone switching systems. We have also used it to model a distributed system of workstations with installed software. The language has been integrated with the HoodNice tools for HOOD design produced by INTECS Sistemi, with the ProMod Plus tools for Modular Design, produced by Debis Systemhaus SSP, and with tools for design in SDL.

We have found that PCL may be used effectively for modelling variability, and that system models incorporating several hundred components may be produced and managed. These models have proved to be effective high-level descriptions of system families. However, as the size of a system increases, so does the cost of developing and managing architectural models. While we would like to think that this expense is, in the long term, a worthwhile investment, we do not have sufficient experience with the language to make such a claim. As with all such languages, scaleability is undoubtedly a problem and we do not yet know if very large systems may be usefully modelled with PCL.

PCL was designed as a documentation, rather than a design language. We envisaged that a PCL description would be created (with tool assistance) when an initial system version was produced, and this would be supplemented with hardware, documentation and variability information. Accordingly, the language does not include facilities to specify configuration rules and checking to ensure that specified configurations satisfy these rules.

Further work in the development of PCL is focusing on adding additional checking capabilities (such as interface checking) to the language and to an exploration of its use in different domains. We have already investigated the use of PCL for dynamic configuration programming [29], where a PCL system model is interpreted to identify system components for replacement. We are examining how to enhance the interface description capabilities and use PCL in conjunction with different programmming languages. Finally, we are exploring the use of PCL for describing virtual reality models, where its variability description facilities allows different graphical representations of objects to be specified.

9 Acknowledgments

The work described here has been partially supported by the European Commission's ESPRIT programme (Project 6067).

The authors would like to thank the other members of the PCL development team: Bjorn Gronquist, Gilbert Rondeau and Ariane Suisse, CAP Germini Innovation, Grenoble; and Bjorn Gulla, Eirik Tryggeseth and Reidar Conradi, Norwegian University of Technology, Trondheim.

10 References

[1] ROBINSON, P.J.: 'Hierarchical object-oriented design' (Prentice-Hall, Englewood Cliffs, New Jersey, 1992)
[2] BRAEK, R., and HAUGEN, O.: 'Engineering real-time systems' (Prentice-Hall, Englewood Cliffs, New Jersey, 1993)
[3] SOMMERVILLE, I., GULLA, B., and GRONQUIST, B.: 'PCL-V2 reference manual' Lancaster University, 1994
[4] TRYGGESETH, E., and GULLA, B.: 'Comprehensive variability modelling with the proteus configuration language'. Proc. 5th Workshop on Software Configuration Management, Seattle, Washington, 1995
[5] GARLAN, D., TICHY, W., and PAULISCH, F.: 'Daghstul software architecture workshop summary', *ACM Softw. Eng. Notes*, 1995, **20,** (3), pp. 63–83
[6] GARLAN, D.: 'ICSE-17 Software Architecture Workshop Summary', *ACM Softw. Eng. Notes*, 1995, **20,** (3), pp. 84–89
[7] DEREMER, F., and KRON, H.H.: 'Programming in the large versus programming in the small', *IEEE Trans.*, 1976, **SE-2,** (2), pp. 80–86
[8] TICHY, W.F.: 'Software development control based on module inter-connection'. 4th Int. Conf. on Software Engineering Munich, 1979 (IEEE Press)
[9] NARAYANASWAMY, K., and SCACCHI, W.: 'Maintaining configurations of evolving software systems', *IEEE Trans.*, 1987, **SE-13,** (3) pp. 324–333
[10] SOMMERVILLE, I., and THOMSON, R.: 'An approach to system evolution', *Comp. J.*, 1989, (5): pp. 386–398
[11] PRIETO-DIAZ, R., and NEIGHBOURS, J.: 'Module interconnection languages', *J. Syst. Softw.*, 1986, **6,** (4), pp. 307–334
[12] COOPRIDER, L.W.: 'The representation of families of software systems'. PhD Thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, 1979
[13] HALL, P., and WEEDON, P.: 'Object-oriented module interconnection languages'. Proc. 2nd Int. Workshop on Software Reuse, Lucca, Italy, 1993, (IEEE Press)
[14] SHAW, M. *et al.*: 'Abstractions for software architecture and tools to support them', *IEEE Trans.*, 1995, **21,** (4), pp. 314–335
[15] LUCKHAM, D. *et al.*: 'Specification and analysis of system architecture using Rapide', *IEEE Trans.*, 1995, **21,** (4), pp. 336–355
[16] DEAN, T.R., and CORDY, J.R.: 'A syntactic theory of software architecture', *IEEE Trans.*, 1995, **21,** (4), pp. 302–313
[17] FELDMAN, S.I.: 'MAKE—a program for maintaining computer programs', *Softw. Pract. Exp.*, 1979, **9,** (4) pp. 255–265
[18] LEBLANG, D.B., and CHASE, R.P.: 'Parallel software configuration management in a network environment', *IEEE Softw.*, 1987, **4,** (6), pp. 28–35
[19] ESTUBLIER, J.: 'A configuration manager: the Adele data base of programs'. Workshop on Software Engineering Environments for Programming-in-the-Large, Harwichport, Massachusetts, 1985
[20] WHITTLE, B.: 'Models and languages for component description and reuse', *ACM Softw. Eng. Notes*, 1995, **20,** (2), pp. 76–89
[21] GOGUEN, J.A.: 'Reusing and interconnecting software components', *IEEE Computer*, 1986, **19,** (2), pp. 16–28
[22] LOWE, M. *et al.*: 'On the relationship between algebraic module specification and program modules', *Lect. Notes Comput. Sci.*, 1991, **494,** pp. 83–98
[23] WHITTLE, B., and RATCLIFFE, M.: 'Software component description for reuse', *BCS/IEE Softw. Eng. J.*, 1993, **8,** (6)
[24] KAISER, G., and GARLAN, D.: 'Melding software systems from reusable building blocks', *IEEE Softw.*, 1987, **4,** (4), pp. 17–24
[25] TRACZ, W.: 'LILEANNA: a parameterized programming language'. Proc. 2nd Int. Workshop on Software Reuse, Lucca, Italy, 1993 (IEEE Press)
[26] MAGEE, J., KRAMER, J., and SLOMAN, M.: 'Constructing distributed systems in CONIC', *IEEE Trans.*, 1989, **SE-15,** (6)
[27] DULAY, N.: 'A configuration language for distributed programming'. Department of Computing, Imperial College, London, UK
[28] PURTILLO, J.: 'The Polylith software toolbus', *ACM Toplas*, 1994, **15,** (1)
[29] WARREN, I., and SOMMERVILLE, I: 'Dynamic configuration abstraction'. Proc. 5th European Conf. on Software Engineering, Sitges, Spain, 1995, (Springer)
[30] TICHY, W.: 'RCS—a system for version control', *Softw. Pract. Exp.*, 1985, **15,** (7), pp. 637–654
[31] BELKHATIR, N., and MELO, W.L.: 'Supporting software development processes in Adele 2', *Comput. J.*, 1994, **37,** (7), pp. 621–628

The paper was first received on 3 January 1995 and in revised form on 21 November 1995.

Ian Sommerville is with the Computing Department, University of Lancaster, Lancaster LA1 4YR, UK, email is@comp.lancs.ac.uk; Graham Dean is currently with Marex Technology Ltd., 34 Medina Road, Cowes, Isle of Wight PO31 7DA, UK, and was formerly with the Computing Department, University of Lancaster.