Configuration Management using SySL

Ronnie Thomson and Ian Sommerville
Department of Computing,
University of Lancaster,
Bailrigg, Lancaster,
LA1 4YR, United Kingdom.

1. Introduction

Software configuration management is concerned primarily with the consistent labelling and tracking of project information and managing change to that information [1]. Its objective is to try and control the changes that are made to the software in such a way as to preserve the integrity of the system and provide a basis on which to measure quality, both of the system and the development process.

Initially, software configuration management systems were aimed at the results of the software life-cycle itself (i.e. software components) without trying to manage other outputs from the various stages in the process. Systems such as Make [2] and SCCS [3] are indicative of such an approach. However recent research has recognised that software configuration management should be applied throughout the software process. Therefore information such as design diagrams, requirements documents, test data, etc, as well as code, should be under the control of the configuration management system.

System modelling lies at the heart of software configuration management. The system model captures the state of the system, identifies the parts making up the system and specifies how to put these components together. To provide features such as the identification of baselines, the system model must capture the process by which the system is constructed, the identifiable pieces of information which emerge during the development process and their relationships. There are many dimensions to system modelling identified in [4].

(i) ***Organisational*** : Deals with external factors identifying organisational responsibility within the project.
(ii) ***Structural*** : Here we are recording the elements which make up a system and how they fit together.
(iii) ***Spatial*** : Deals with topology of our development environment and issues related to the distributed location of project information.
(iv) ***Temporal*** : This aspect deals with the development history of both the system as a whole and each of the subparts making up the system.
(v) ***Purpose*** : Here we are recording the issues that were discussed during the development of the system.
(vi) ***Procedural*** : This records the knowledge that is necessary to build a release of the system.

This paper discusses a system modelling language and supporting toolkit, called SySL, that was developed as part of Eclipse [5], a joint industry/academia project which developed an software engineering environment (SEE).

2. SySL - A System Structure Language

SySL is used to represent the structure of a software system, highlighting the dependencies and relationships that may exist between structural components. A description of the structure can be used as an architectural blueprint for the system development, providing a reference point for designers, programmers, managers, etc. involved in the project. Different members of software projects adopt different roles within a project and information requirements, within such roles, vary considerably from the very abstract to the specific. SySL provides constructs which allow project information to be presented at varying levels of detail, according to the role of the reader.

Language features include:

(i) ***Description of systems at various levels of abstraction.*** SySL encapsulates the idea of a class of systems which share certain common features. Individual members of this class are described by instantiating a generic structure.
(ii) ***Constraint association on particular combinations of entities.*** A SySL description of a system can validated against the project database representation. Attributes and relationships defined in the SySL description can be checked against those which are assumed to exist in the database.
(iii) ***Description of any structured system.*** All information generated during a project should be under the control of the configuration management system. SySL has been generalised to embrace any logical collection of components in a project database.
(iv) ***Description of the logical system structure.*** SySL is used to present the user with view of logical system structure as opposed to the physical structure that exists in a project database. This allows the designer of the system to describe the system in terms of logical entities. Further refinement of the description during the implementation phase maps this logical structure onto physical entities in the SEE.

Large software systems are made up of thousands of entities participating in many different types of relationships and

held in the project database. The database schema provides the user with access to this information.

SySL includes constructs to define the following: (1) Classes of database items which are defined by enumeration of explicit items or by class unions, (2) The structure associated with all members of a particular class, (3) The specific configuration of individual members of a class, (4) The interface published by individual members of a class, (5) Constraints on classes in general or on individual class members.

SySL is a language for *programming-in-the-large* (PIL) [6] and as such is an extension of module interconnection languages (MIL) [7]. Work in this field has developed from a practical point of view as a method of representing the relationships between components. MILs are notations for describing a system during its development and of representing the numerous versions which comprise a system. In [6] it was stated that the task of "structuring a large collection of modules to form a system is an essentially different intellectual activity from that of constructing the individual modules". It is then argued that a different type of notation capable of representing the entities and abstractions used in PIL.

The most notable work in this field includes languages like MIL75 [6] and INTERCOL [8], etc. In such languages the interconnections between software components, representing different types of dependencies, are described explicitly. The structure of the system is in static terms and this allows the system structure to be checked for consistency and completeness. Other languages and software development tools represent this type of information in different ways. For example, a certain degree of module interconnection can be represented in languages such as Ada, Modula-2, etc. However these languages do not support concepts like versions and configurations, which are important for PIL.

2.2. A SySL Example

To illustrate SySL, we present an example describing the SySL tool support system. This shows how the language supports views at different levels of abstraction and shared configurations.

Classes of systems which share certain common features can described via the class declaration facility which allows systems to be defined by explicit enumeration of class members which can be specific systems or subclasses. Each system named as a member of a class is said to be a particular configuration of the class. For example

```
class STRUCTURED_VIEWER is
        ( ADA_VIEWER, SYSL_VIEWER)
```

defines a class STRUCTURED_VIEWER which has two members, namely ADA_VIEWER and SYSL_VIEWER both of which are subclasses. Each subclass can be further defined in terms of specific entities or subclasses.

To describe the common features or characteristics of a group of systems the language contains a structuring facility whereby a generic structure describes the structure of the entities in terms of other classes of entities. This structure is associated with a class through name association.

```
structure STRUCTURED_VIEWER is
        [COMPILER],
        SETUP_ROUTINE,
        INTERFACE,
        VIEWER,
        EDITOR,
        {DATABASE}*
end structure
```

The specification of STRUCTURED_VIEWER indicates that it comprises a number of other classes of components some of which are optional, indicated by the square brackets, and some of which are repeated, indicated by the curly brackets. The asterisk after the closing curly brackets as in the DATABASE class above indicates that there may be a number of such components in a system of the type STRUCTURED_VIEWER. The description is abstract and details as to specific systems or configurations of systems has not been given.

Any defined class can have a structure, therefore classes, which are defined as part of a structure declaration, can themselves have a structure. For example, the class VIEWER which is part of the class STRUCTURED_VIEWER above can be described thus.

```
structure VIEWER is
        {MENU_SETUP}*,
        {EVENT_HANDLER}*,
        {OPERATIONS}*,
        {TEXT_SETUP}*,
        DISPLAY
end structure
```

We continue to refine the structure of a system or class of systems in the above manner. The result of this process is a tree-like structure defining the overall structure of a system.

We can further refine class definitions by partitioning each subclass into further subclasses or by naming specific systems within a class. For example the subclass SYSL_VIEWER which was named as part of the STRUCTURED_VIEWER class can be defined as follows.

```
class SYSL_VIEWER is
                (sun_viewer, vax_viewer)
```

Here we have identified two systems namely, *sun_viewer* and *vax_viewer* which are members of the class SYSL_VIEWER. Both *sun_viewer* and *vax_viewer* above correspond to objects in the SEE, i.e. they are actual software systems. The structure of both systems is inherited through the class hierarchy already defined. However if we wish to display more details of these systems we can do so by instantiating the structure tree that was inherited with SYSL_VIEWER as follows;

```
structure SYSL_VIEWER :
                STRUCTURED_VIEWER is
        COMPILER => sysl_compiler,
        SETUP_ROUTINE => sysl_main,
        INTERFACE,
        VIEWER,
        EDITOR,
        {DATABASE}*
end structure
```

In this definition, the COMPILER component has been instantiated to the entity *sysl_compiler* and the SETUP_ROUTINE to the entity *sysl_main*. All instances of systems of the class SYSL_VIEWER include these components.

Instances of *sysl_viewers* are described by instantiating the other elements of SYSL_VIEWER:

```
system sun_viewer : SYSL_VIEWER is
   INTERFACE => setup_interface,
   VIEWER => sysl_viewer,
   EDITOR => sysl_editor,
   DATABASE => sysl_database
end system
```

We do not need to repeat part of the structure which defines the use of components *sysl_compiler* and *sysl_main* as they are inherited by all systems in the class SYSL_VIEWER.

Each part of the structure tree is instantiated with details of specific components making up a system. Therefore following on from the above description, the structure of the VIEWER component given in sun_viewer above, sysl_viewer, is:

```
system sysl_viewer : VIEWER is
   provides (edit_win_event)
   requires (object_def)
   MENU_SETUP => menu_setup,
   EVENT_HANDLER => (edit_win_event,
                     menu_event, keyboard_input),
   TEXT_SETUP => (text_structure, text_index),
   DISPLAY => display,
   OPERATIONS => (object_commands)
end system
```

Software components may use components or resources which are logically unrelated to the part of the system being described. This usage is like a horizontal flow of resources across the system. These resources may correspond to functions and procedures, data types, etc. To enable this horizontal usage of resources to be specified, component descriptions have resource clauses which specify the interface to the component and specifies dependencies between components.

As well as the structural aspects of a system, SySL can be used to record other types of relationships. Such implicit relationships between components include 'partof', 'includes' and 'is_dependent_on', etc. The language includes assertions which can record other types of relationships by simply attaching attributes to components. Assertions provide the ability to state some property about a component or class of components, define component relationships or retrict certain combinations of components in a particular configuration. For example, in the class STRUCTURED_VIEWER the COMPILER and DATABASE components can theoretically be missing any configurations of that class. If we wish to enforce the rule that both components must be present in a particular subclass of systems, for example within ADA_VIEWER, then the following rule would be used.

```
assert ADA_VIEWER :
        forall i: member(i, ADA_VIEWER)
        and not(not_present(COMPILER) and
                not_present(DATABASE))
```

Which states than no configuration of the class ADA_VIEWER can have both the COMPILER and DATABASE components missing for the configuration to be a valid one.

3. The SySL Toolkit

A SySL description must reflect project structure to be of any use. This is difficult if the language is detached from the environment containing the project information. Therefore we have provided tools which automate the task of keeping such a description consistent with the project it is modelling. Among the tools provided are a powerful language-oriented editing system which provides a structured approach to viewing and editing descriptions.

Other facilities, apart from the language, that are provided by the language toolkit are described below. Figure 1 provides a logical view of the language and the tools.

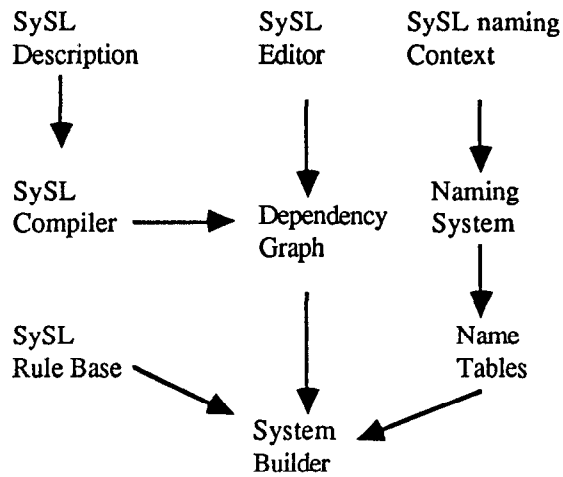

Fig 1 : A Logical view of the SySL Toolkit

(i) ***Language Processor and Graph Generator***: The environment uses a dependency graph generated by the language processor. This graph is a logical representation of the equivalent SySL description. Nodes in this graph represent entities in the system description and links between nodes represent relationships between entities.

(ii) ***Language Editor***: SySL is an important source of documentation on the system. The language editor provides a structured approach to viewing and editing this dependency graph.

(iii) ***Name Management***: SySL provides no explicit naming conventions for identifying version of components. We assume that the underlying SEE provides such facilities. This tool provides the user with a means to map the SySL names onto the SEE database names.

(iv) ***System Builder***: Allows the user to generate the information required to build a system. The tool is based on a rule base which contains the knowledge about the types of component and information required to build the components.

3.2. Generating a System Release

The most important feature of the SySL toolkit is its ability to generate, automatically, an executable version of

the system descrbed by the SySL description. As part of the SySL toolkit we have provided a tool which takes a SySL description, a set of rules describing how to build different types of components and generates the information to build the system. The prototype generates a Unix makefile [2]. Three important subsystems on which system building is based include;

(i) ***The dependency graph***: Contains logical representation of the system structure.
(ii) ***Name management system*** : Provides mapping between the logical representation in graph onto the physical representation in project database.
(iii) ***System builder***: Takes the logical representation and the information contained in the name management system and, using a rule base of translation rules, generates a Unix makefile.

Each logical item in the SySL description is mapped onto a physical entity in the project database. This mapping identifies the particular version that is to be used. The mapping process is aided by the use of a context. This contains a list of names and versions. The Name management system takes the context and binds the SySL description. This system is similar to configuration threads in the DSEE system [11] and generic configurations in Adele [12]. The build tool takes both the graph, and the mapping information to generate a Unix makefile. A rule base, containing the build information, is used to generate the correct makefile rules.

4. Conclusions

This paper has demonstrated the feasibility and applicability of a number of different features of the SySL system.

(i) ***Building from a logical system description.*** Software is designed and implemented in a modular fashion, with components in the software system representing logical entities in the design. SySL allows a system to be built from this logical description.
(ii) ***Abstract description of complex system structures.*** Software systems have a long life-time and as such are implemented once but read many times and maintained over many years. Therefore the presentation of complex system information at a more understandable and abstract level is essential.
(iii) ***Re-configuring Software.*** The task of re-configuring a system is difficult and error-prone. The provision of a powerful language-oriented editor and object name manager, allows a designer to easily re-configure the system.

We have evaluated our system by describing hardware, software and documentation systems and found it sufficiently general purpose to describe any type of system which may be represented in an Eclipse database. At the time of writing, an initial version of the system is complete. To demonstrate the ideas we have developed a version of the system builder which generates UNIX makefiles. We have also completed a port of the system into the PCTE [10] domain.

5. References

[1] Bersoff, E. H., Henderson, V. D., Seigel, S. G. Software Configuration Management: A Tutorial. *IEEE Computer*, Vol 2 No 1, 1979.

[2] Feldman, S.I. MAKE - A program for maintaining computer programs. *Software Practice and Experience*, 9, 255-265. 1979.

[3] Rochkind, M. J. The Source Code Control System. *IEEE Transactions on Software Engineering*, Vol SE-1, No 4, Dec 1975.

[4] Chase, R. P. and Fuchs, M. System Modelling. *Proceedings of the InternationalWorkshop on SoftwareVersion and Configuration Control*. Grassau, W. Germany. Jan 1988.

[5] Alderson, A., Bott, M.F and Falla, M. An Overview of Eclipse. *Proc. 1st UK Conf. on Integrated Project Support Environments*, York, April 1985. To be published by Peter Perigrinus, London, 1985.

[6] DeRemer, F. and Kron, H.H. Programming in the large versus programming in the small. *IEEE Transactions on Software Engineering*. SE-2 (2), 80-86, 1976.

[7] R. Prieto-Diaz and J. Neighbors. Module Interconnection Languages. *Journal of Systems and Software*. Vol 6, 1986.

[8] Tichy, W.F. Software Development Control Based on Module Interconnection. *Proc.4th Int. Conf. on Software Engineering*, 29-41, Munich, 1979.

[10] Campbell, I. PCTE proposal for a public common tool interface, in *Software Engineering Environments*, edited by I. Sommerville, published by Peter Pererinus, 1986.

[11] Leblang, D. B, Chase, R. P. and Spilke, H. Increasing Productivity with a Parallel Configuration Manager, in *Proceedings of the International Workshop on Software Version and Configuration Control*, Grassau, W. Germany, 1988.

[12] Estublier, J. A Configuration Manager: The Adele data base of programs, in *Workshop on Software Engineering Environments for Programming-in-the-large*, Harwichport, MA, June 1985.