CONFIGURATION SPECIFICATION USING A SYSTEM STRUCTURE LANGUAGE

Ian Sommerville and Ronnie Thomson

1. Introduction

The emergence of distributed systems technology has enabled the development and deployment of large-scale computing systems distributed over a local or a wide area. Such systems include both hardware and software components communicating over a network. The size and complexity of such systems means that evolution and management is a difficult task. The ability to re-configure such systems in response to changes on the network and changes to components in the system is an important requirement. The distributed nature of systems enables such changes to be done dynamically without interrupting other systems on the network. However, the difficulty of managing evolutionary change in such an environment is immense. Identifying the different entities that make up such a system and tracking the dependencies between components can be a complex and error-prone task.

We believe that a separate configuration model of the system is an essential aid to the task of system re-configuration. Such a model must identify, unambiguously, the various types and configurations of systems that exist. It should provide the ability to label and track, consistently, information pertaining to system components and manage the process of change to that information. The inherent complexity of systems introduces the problem of information overload. Critical dependencies between components may be hidden making the task of re-configuring such systems extremely difficult. The ability to model components at various levels of abstraction is, therefore, a necessary feature.

This paper discusses a system modelling language and supporting toolkit [1] which was originally developed as part of the Eclipse software engineering environment [2]. It addresses the modelling of large-scale systems made up of many different types of components. The language, called SySL (System Structure Language), enables system structure to be described at various levels of abstraction. A toolkit enables changes to the system descriptions to be made consistently, and automates the process of building an executable version of the described system.Using such a language and toolkit, a systems administrator can control the process of dynamically re-configuring a system in a much more effective way.

SySL was developed in the first place to support the management of software configurations. Recently, however, we have extended the system and are using it in a project investigating the general problems of distributed systems management [3]. The language is also the basis for the configuration language in a European collaborative project which is concerned with providing practical support for dynamic systems evolution.

In the remainder of this paper, we briefly describe some of the features of SySL and illustrate how the language can be used for system configuration specification using an example of a network of workstations. We then briefly describe the toolkit which has been developed to support software configuration management using SySL and our initial work on the use of the language in conjunction with a dynamic configuration manager.

Ian Sommerville is with the Computing Department, Lancaster University, LANCASTER LA1 4YR and Ronnie Thomson is with the Computer and Intelligent Systems Lab., GTE Laboratories Inc., 40 Sylvan Road, Waltham MA 02254, USA.

2. Language Features

SySL is a language for *programming-in-the-large* (PIL) and is a type of module inter-connection language (MIL) [4]. Module Interconnection Languages are notations for explicitly describing a system's structure during its development enabling its evolution to be managed and controlled. The basic concept behind such languages lies in the difference between programming-in-the-large and programming-in-the-small. DeRemer [5] states 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 is required which is capable of representing the entities and abstractions used in PIL.

The most notable work in this field includes languages like MIL75 [5], INTERCOL [6], Jasmine [7] and NuMIL [8]. In such languages the inter-connections 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 certain kinds of consistency and completeness. Other languages and software development tools represent this type of information in different ways. For example, a certain degree of module inter-connection 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 programming-in-the-large. More recently languages such as Polylith [9] have tried to address the issues of module inter-connection in a large-scale, distributed, heterogeneous environment. Therefore the mapping of system structure, as represented in the MIL, onto structures in the software is complicated when these structures are distributed over a network.

We have generalised the ideas in SySL to enable the description of systems made up of hardware, software and documentation components. The language can also be used to highlight the dependencies that exist between components. Such descriptions are useful as an architectural blueprint for designers, programmers and managers involved in the project. Features of the language include:

(i) The ability to model families of systems at various levels of abstraction. SySL encapsulates the idea of a class of systems sharing certain common features. Individual members of a system class can be specified by instantiating this generic structure. For example, we can describe the structure of a generic workstation and instantiate this for individual personal systems. This facility allows us to check that configurations conform to the generic specification and to classify the individual components of the system.

(ii) The ability to describe any structured system whether it be hardware, software, or documentation. The language allows the description of any logical collection of components that can be represented in a project database. This facility means that we can identify which software components or systems are available on which hardware systems and, when used in conjunction with the modelling facilities, allows us to decide which alternative configurations are feasible. Thus, the dynamic re-configuration of the overall system can be supported.

(iii) Facilities to describe the logical system structure. A fine-grain description of logical components and their dependencies may be provided and kept separate from the physical structural description. This facility supports software configuration management as the system configuration need not be specified in terms of the physical storage structure (files, database objects, etc.). Alternative, logically equivalent configurations can be represented and managed.

(iv) Facilities to state constraints on particular combinations of components and component attributes. Invalid configurations can therefore be detected by the SySL language processor before they are created. The problem of detecting configuration errors is therefore reduced.

These facilities are integrated in an object-oriented framework with a simple and consistent syntax.

3. An Example of SySL

To illustrate the features in SySL, we describe several different network prototypes that are the subject of experiments. There are two classes of network with certain commonalities namely those that are experimenting with broadband network technologies and those that are experimenting with multi-media technologies over a telephone network. The following statement introduces a class of systems labelled

NETWORK that is sub-divided into two classes of network called BROADBAND-TEST-NET and MM-NETWORK.

```
class NETWORK is (BROADBAND-TEST-NET, MM-NETWORK )
```

The generic structure of both classes of network can be given as follows:

```
structure NETWORK is
        TRANSPORT
        {SERVER}*
        {MUSE-WORKSTATION}*
        {SIMULATOR}*
        {SWITCH}*
end structure
```

The **structure** statement in SySL defines a template for a generic component or system. Individual instances of components inherit the generic structure. The description of NETWORK shows that particular network configurations comprise a TRANSPORT component and several SERVERs, MUSE-WORKSTATIONs, SIMULATORs and SWITCHs ({ }* means one or more instances may be present). At this level the description is an abstract structural description; specific details of network configurations are not given.

We can further refine the above structure description of a network to reveal more of the structure of network configurations. For example,the class MUSE-WORKSTATION, which represents a workstation running the Athena Muse software, can be described as follows:

```
-- Known types of workstation
class MUSE-WORKSTATION is (vpSun3, rsDec, jdgATT386)

-- Workstation component structure. Assume that the generic
-- structure WORKSTATION is defined elsewhere
structure MUSE-WORKSTATION: WORKSTATION is
        ISDN
        PARALLAX
        ATHENA-MUSE
        X-SHADOW
        COMM-CHANNEL
        [IR-SERVER]
end structure
```

Above we have introduced a number of specific configurations of MUSE-WORKSTATION; *vpSun3*, *rsDec* and *jdgATT386*. The structure MUSE-WORKSTATION inherits attributes from the more abstract structure WORKSTATION. These would normally be attributes such as PROCESSOR, MEMORY, COMMS-INTERFACE, etc. The generic structure of a MUSE-WORKSTATION (i.e. a specific type of workstation) is given, and identifies an ISDN component, Parallax video capabilities (PARALLAX), Athena multi-media authoring software (ATHENA-MUSE) and screen-sharing software (X-SHADOW) as necessary components of a MUSE-WORKSTATION. The final item, IR-SERVER, describes a multi-media information retrieval facility that can be optionally resident on a workstation. This facility allows retrieval of text, still pictures and video segments.

The SySL description of IR-SERVER shows that it is made up of a retrieval component, one or more database components (FAIRS) and a message server. An instance of IR-SERVER is called *aegina*. The SySL description of aegina shows how the components of IR-SERVER are instantiated.

```
structure IR-SERVER is
        MUSE-RETRIEVAL
        {FAIRS}*
        MESSAGE-SERVER
end structure

system aegina : IR-SERVER is
        provides(query, browse, help)
        requires (gte-muse)

        MUSE_RETRIEVAL => (query, query_expansion, browse, help)
        FAIRS => (query_process, locate, rank, present)
        MESSAGE-SERVER => fairs_message_server
end system
```

Here we see how generic attributes (which by convention are always expressed in upper-case characters) are instantiated as actual components (distinguished because they are written in lower-case letters). Hence, MESSAGE-SERVER (for example) is implemented by the component fairs_message_server. Because of the separation in SySL between logical and physical components, the component name used here need not be the same as the name under which the component is stored in the file system or database. Indeed, different versions which are logically comparable may be stored under different names.

The SySL **provides** clause is a very limited form of interface specification. Essentially, an instance of IR-SERVER is an abstract data type which is accessed through query, browse and help operations. This information is not currently used by SySL processing tools but is provided as an aid to readers of SySL descriptions. The components making up aegina are not accessible except through the provided interface. The **requires** clause is used to specify that an optional element in some other part of the system description must, in fact, be present if this system is to work correctly.

We can 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 hierarchical structure of a system. At each level in the tree more details of specific components are given. Therefore, following on from the above description, we can describe the specific structure (i.e. the instantiation of the component types) for a type of workstation:

```
system vpSun3 : MUSE-WORKSTATION is
        provides(aegina.query, aegina.browse, aegina.help)
        requires (stl-isdn-simulator)

        ISDN => gte5-isdn
        PARALLAX => sun-parallax
        ATHENA-MUSE => gte-muse
        X-SHADOW => sh-X
        COMM-CHANNEL => ds3
        IR-SERVER => aegina
end  system
```

The workstation described above makes use of several services available on the network, namely, an ISDN simulator and an audio server. The system 'aegina', a multi-media information retrieval service, is provided by this workstation for use by other devices on the network. The provides clause in the above definition is inherited from the description of aegina and shows that this workstation provides access to the information retrieval facilities provided by the IR-SERVER.

Part of the system is the 'gte-muse' component described below.

```
component gte-muse : ATHENA-MUSE is
        INTERFACE => athena-interface-sw
        ATHENA-WIDGET => (popup-widget, pane-widget,
                event-widget)
        X-PROTOCOL => x-protocol-sw
    end component
```

There is no logical distinction between components and systems in SySL. However, we make the distinction so that we can more readily identify which components may be re-configured. Only systems may be dynamically re-configured. Components represent fine-grain entities which would not normally be executable processes.

The following example describes one of the Athena widget classes, called popup-widget which handles the creation and manipulation of popup menus. For example purposes assume that appropriate class and structure definitions are defined.

```
component popup-widget : ATHENA-WIDGET is
    provides (simpleMenu, menuButton)

    RESOURCES => (   background, borderColor,
                     height, width,labelClass),
    OBJECT => (simpleMenu, menuButton, sme,
                     smeBSB, smeline),
    FUNCTION => (highlight, unhighlight,
                     notify,menuPopDown)
end component
```

In this example component *popup-widget* is an abstract data type representing the component that creates and manages popup widgets in the Athena X-window widget set. The objects *sme*, *smeBSB*, *smeline*, are defined in this component but are invisible to users of this component. The structure representing popup-widget contains a set of resources necessary in X applications, several window objects and a set of functions for creating and managing these objects.

As well as structural relationships, SySL can be used to specify other types of relationships. Implicit relationships between components such as 'partof', 'includes' and 'is_dependent_on', are part of the structural characteristics of the language. To expand on this an assertion capability allows additional properties about a component or class of components to be stated. The following example asserts that for all diskless configurations of MUSE-WORKSTATION an ethernet interface must be present.

```
assert MUSE-WORKSTATION :
    not (Not_present (ETHERNET) and
         Not_present (DISK_SYSTEM))
```

As the assertion applies to a class of systems all members of that class must satisfy the assertion constraint. The language compiler uses such constraints to check the consistency of configuration descriptions.

SySL was originally designed as the basis for a software configuration management system. We designed the notation for describing logical system structure as we believed that this simplified the process of specifying the build instructions for large software systems. Because of the requirement for straightforward system building our principal design objectives were to provide a straightforward and succinct mechanism for specifying component dependencies and to integrate the description of the target of the build process with the software structure specification. We also required a mechanism to tackle one of the most significant problems in system building namely the omission of a necessary component. This checking can be carried out be comparing a system to its structure and ensuring all necessary parts of the structure have been instantiated.

SySL is not, in its current form, a configuration programming language which can be used to compose components or tasks into executable applications. In this respect, it is quite different from configuration languages such as CONIC (10), the Instress notation which is provided as part of the Inscape environment (11) or the Durra notation, described by Barbacci *et al.* (12). All of these languages are geared towards supporting system composition from more primitive components so place grate emphasis on interface specification and on ensuring the consistency of the interfaces of composed components.

This focus on interface specification makes these languages suitable for the construction of distributed applications where components of an application may execute on separate machines in a network. However, to provide this level of functionality, systems such as CONIC must also provide a language for programming components (called task modules in CONIC). This clearly limits the applicability of the system. SySL, by contrast, is language independent and components may be realised in any programming language.

Since 1991, we have been investigating the use of SySL as a language for distributed system or network specification where the system components are themselves applications such as compilers, window management libraries, software tools, etc. We intend that this specification should be used by network managers to manage versions of software systems are available on a network which may be composed of several hundred workstations. Managers need to be able to obtain a picture of the network being managed at a number of different levels of abstraction and the structuring facilities of SySL make it well-suited for network representation.

However, we have just started the development of a derivative of the language for specifying the composition of distributed applications. We recognise that this will require the provision of mechanisms for interface specification which are comparable to those available in other configuration languages.

4. The SySL Toolkit

We have provided an integrated toolset to help maintain the consistency and completeness of SySL descriptions. This can be difficult to manage if the language is detached from the environment containing the project information. Therefore to enhance the usability of the language we have provided tools that automate the task of keeping such a description consistent with the environment it is modelling. In designing SySL, we made a decision that the language should not be divorced from its tool support. Therefore, we have deliberately kept the language as simple as possible and provide functionality which might have been included in the language (such as mapping from logical to physical store) in the toolset.

The following tools are included.

(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. The language processor checks the internal consistency of a SySL specification and is tightly integrated with the browser and structured editor.

(ii) **Language Browser and Structured Editor**: SySL is an important source of documentation on the system. The language browser provides a structured approach to viewing this dependency graph. The editor system allows the user to update the graph and preserve the semantics of the SySL description. The browsing and editing system presents the underlying dependency graph so that logically related entities are grouped on the user's display. Navigation facilities allow the dependency graph to be traversed and holophrasting facilities simplify the information display.

(iii) **Object Name Management**: SySL provides no explicit naming conventions for identifying versions of components. We assume that the underlying environment provides such facilities. This tool lets users map SySL names onto software engineering environment object names and to view the object representation. In essence, this system provides the link between the logical system names and the underlying physical storage system. Multiple versions of the same logical system are supported by using the name management system to create separate directories which map onto the object management system (Figure 1)

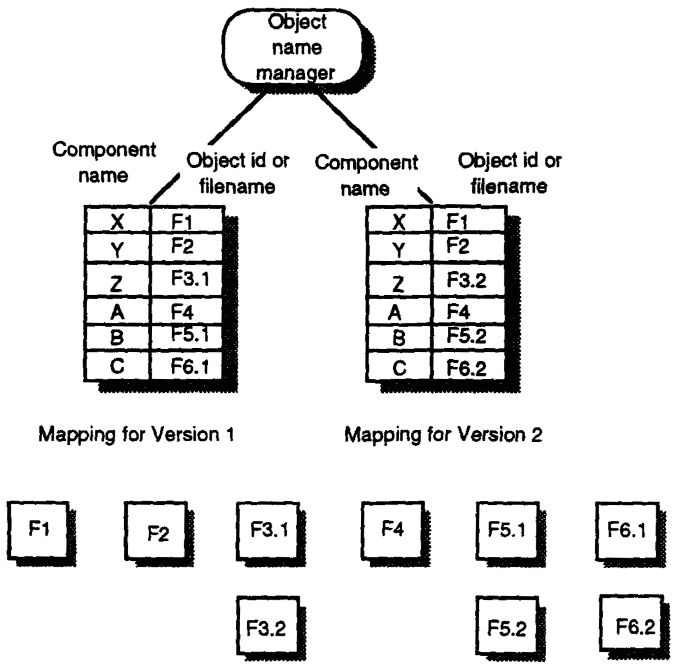

Figure 1 The object name manager

(iv) **System Building Facility**: This tool allows the user to generate the information required to build a software system and dependent components. The system uses a rule base that contains knowledge about the types of component and the information required to build each software component type. The rule base describes which tools are used to transform entities of one type (C-SOURCE, say) to another type (OBJECT-CODE). When used in conjunction with the typing information in the system model and the name management system, this allows system makefiles to be generated. Makefiles are essentially a combination of physical dependency information and (sometimes implicit) system construction specifications so can readily be constructed from the SySL description.

These tools work together as shown in Figure 2.

Work is currently underway to extend the SySL toolkit. A tool to generate graphical views, giving facilities similar to ConicDraw [Kramer 89] shared by several workstations is being developed. The objective of this work is to allow distributed system managers to collaborate in making changes to the system configuration. A query facility is being developed which will allow a SySL description to be queried and modified by a configuration manager as described in the following section.

5. System Re-configuration

We are currently investigating the use of SySL as a configuration language in two areas:

1. In the management of a network of distributed workstations where services such as compilation services, data base services etc. are provided on different machines. We intend to support the upgrading of these services without making them unavailable.
2. In a distributed manufacturing system where the re-configurable components are relatively fine-grain (C++ objects). This project is in its very early stages and working software is not yet available to support interface checking.

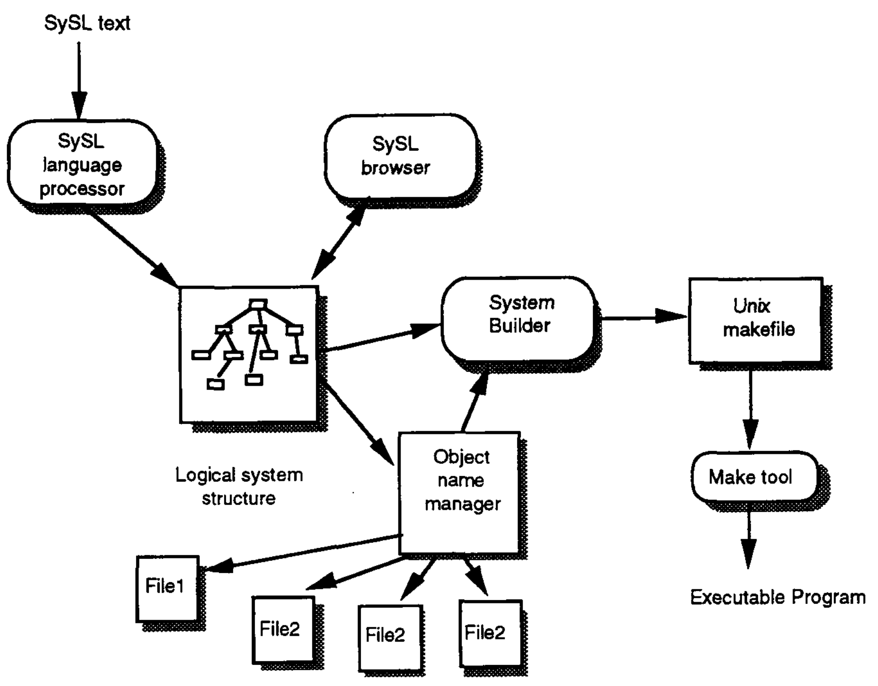

Figure 2 The SySL toolkit

In the first of these areas, we are concerned with a distributed system providing a range of services as separate applications. The granularity of the configuration items is coarse (for example, an item is an Ada compiler) and, unlike CONIC say [10] we are not concerned with applications which are themselves distributed.

The model for re-configuration support for this system is as shown in Figure 3.

Dynamic re-configuration is managed through a configuration description expressed in SySL. We assume that a system provides a set of services and includes more than one processing node which can support these services. Re-configuring the system involves re-defining which processing node provides a specific service.

In our earlier example we described an instance of MUSE-WORKSTATION called 'vpSun3' which contains the system 'aegina', which is a multi-media information retrieval facility associated with that workstation. This retrieval service is shared by all workstations on the network therefore it is only resident on 'vpSun3'. The service doesn't require any dedicated hardware to be attached to the workstation. All the video segments and still pictures come from compressed data files are resident on the workstation. If we decide to re-configure our network such that the 'aegina' service is located on another machine all we need to do is alter the SySL description to reflect this change.

Using the structured editor, provided as part of the SySL toolkit, we can move the 'aegina' component across to another workstation. In executing such a command, the editor will make sure that the semantics of the SySL is preserved by removing the provides line and the definition of 'aegina' from the workstation 'vpSun3' and will add appropriate text to the definition of the workstation which will now provide the service.

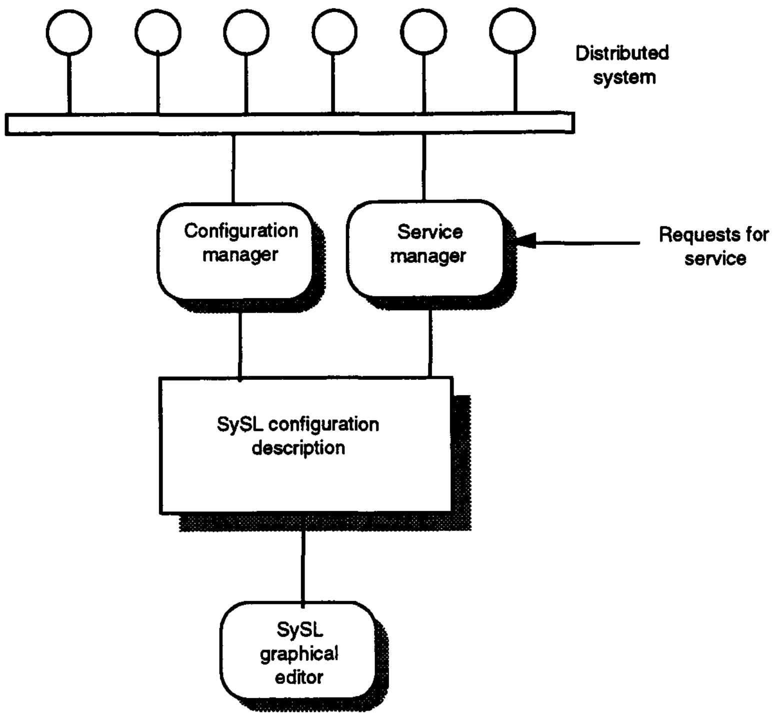

Figure 3 Dynamic re-configuration using SySL

Altering the configuration description automatically invokes the configuration manager. The configuration manager interprets the user's changes, assesses their feasibility and invokes the system builder to prepare and install the modified service. In our current implementation, this involves creating a makefile but we anticipate that dynamic configuration support may bypass this step and translate changes directly into system commands.

We ensure that the system node which is currently providing a service is accessed by routing service requests through a service manager. The service manager may access the SySL configuration description to discover the location of a particular service and then route the request to the appropriate system node. The service manager maintains a queue of requests so can block service requests while a re-configuration is in progress.

We plan to use the same basic model when using SySL as a basis for the dynamic re-configuration of distributed applications. However, we are aware of the need to extend the interface specification facilities in SySL to support distributed applications (we shall make the interface specification compatible with C++) and SySL is evolving towards a language for configuration programming.

6. Conclusions

The process of configuring and re-configuring a distributed system is an expensive and error-prone activity. The work described in this paper has shown that a tailored notation can be used for configuration description and we believe that a configuration description in this notation is easier to manage and evolve than alternative *ad hoc* descriptions. The SySL system allows system managers to document the components that make up a system, record their interrelationships and automatically generate the build files necessary to create a new version of the system. We have used SySL in the configuration management of software systems and are now experimenting with the notation in the wider context of distributed systems management.

7. References

[1] Thomson, R. & Sommerville, I., 1989, 'An Approach to the Support of Software Evolution', *Comp. J.*, 32 (5), October 1989.

[2] Bott, M.F. 1989, *ECLIPSE: An integrated project support environment*, Peter Perigrinus, Stevenage, 1985.

[3] Dean, G., Hutchison, D., Rodden, T. and Sommerville, I., 'Cooperation and Configuration within Distributed Systems Management', Proc. Int. Workshop on Configurable, Distributed Systems, London, 1992.

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

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

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

[7] Marzullo, K & Weibe, D., 1987, 'Jasmine: A Software System Modelling Facility', *ACM SIGPLAN/SIGSOFT*, Vol 22, No 1, 1987.

[8] Narayanaswamy, K., 1985, *A Framework to Support Software System Evolution*, Ph.D Thesis. Computer Science Department, University of Southern California.

[9] Purtillo, J., 1985, 'Polylith: an environment to support management of tool interfaces', *Proc. ACM SIGPLAN Symposium on Languages Issues in Programming Environments*, July 1985, 12-18.

[10] Magee, J., Kramer, J., and Sloman, M., 'Constructing Distributed Systems in Conic', *IEEE Trans. on Software Engineering*, 15 (6), 663-75, 1989.

[11] Perry, D.E., 'The Inscape Environment', *Proc. 11th Int. Conf. on Software Engineering*, 2-12, IEEE Press, 1989.

[12] Barbacci, M.R., Weinstock, C,B., and Wing, J.M., 'Programming at the Processor-Memory-Switch Level', *Proc. 10th Int. Conf. on Software Engineering*, 19-28, IEEE Press, 1988.

[13] Kramer, J., Magee, J. and Ng, K., 'Graphical Configuration Programming', *IEEE Computer*, 22 (10), 1989.