Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences, 1986.

1986 b

THE ECLIPSE SYSTEM STRUCTURE LANGUAGE

Ian Sommerville and Ronnie Thomson
Department of Computer Science
University of Strathclyde
Glasgow, Scotland.

Abstract

This paper introduces a notation for the description of the static structure of systems whose components are represented in a software project database system. It is being implemented as part of the ECLIPSE integrated project support environment and is supported by a toolset made up of tools for description preparation, database checking and system building. The ECLIPSE environment is described briefly and the language is illustrated by example.

1. INTRODUCTION

It is now fairly generally accepted that software environments which are intended to support the development of large software systems should be based on some form of project database system where all of the information associated with the software development is recorded. In general, the associated data base is very complex and is made up of a large number of entities and tens or hundreds of relationships. Each entity/relationship may have associated attributes. Usually, the software engineer working on part of a project is aware of only a relatively small number of entities, attributes and relations and it is very difficult for him or her to form an overall understanding of the system structure recorded in the data base.

The aim of the work described in this paper is to develop a notation which will allow the structure of systems which are held in a project database to be described. This description should be understandable to the software engineers involved in software development and maintenance. The notation, described here, is called SySL (System Structure Language).

In DeRemer and Kron's terminology (1), SySL is a notation for programming in the large. It is intended to describe the structure of any system (not just software systems) which is recorded as attributed entities and relationships in the database of the project support system called ECLIPSE which is currently being built in the UK as a joint industry/university research project.

The system structure language is intended to describe the static structure of large software systems and may be looked upon as a development of module interconnection languages such as those described by DeRemer and Kron (1) and Tichy (2). Module interconnection languages are used for specifying the static structure of systems in terms of how one system module makes use of other modules. They are principally intended as a high-level description of the software system. The features which distinguish SySL from simple module interconnection languages and notations for configuration description such as the Cedar system modeller (4) are as follows:

1. SySL is tightly integrated with an underlying database. It describes configurations of database items rather than files. In essence, a SySL description is a description of database structures in a form which is readily understandable without detailed knowledge of the underlying DBMS.

2. SySL allows the description of both specific systems and classes of system which may be represented in the database. Thus the user may view system configurations at varying levels of detail.

3. SySL has an associated toolset which supports the construction of SySL descriptions, system building and database checking. The language was designed with automated tool support in mind and it's practical usability is dependent on its toolset.

One example of a class of tool which we intend to build as part of the SySL project is a system building tool like the Unix MAKE (3) system or the Cedar System modeller (4). These tools use explicit notations for describing the build system structure. It seems to us that by extending static structural descriptions in SySL with build information abstracted from the project database, it will be possible to automate the process of system building.

In the remainder of this paper, we briefly describe the ECLIPSE environment (5) in which our work is to be incorporated. We pay particular attention to the object management system which controls the ECLIPSE data base as SySL reflects some of the concepts of this system and allows an abstract approach to configuration description. We then go on to illustrate SySL by example and describe, very briefly, the toolkit which we intend to associate with the system structure language. Space does not allow a full description of SySL here - further information is available from the authors (6).

2. THE ECLIPSE ENVIRONMENT

ECLIPSE is a large-scale integrated *project* support environment (IPSE) which is being implemented in the UK by a consortium made up of three universities and three software companies. It is designed to offer support for all stages of the software life cycle from project conception through to operation and maintenance. The system will provide automated support for a number of software development methodologies, tools to encourage the reuse of software components, fast prototyping and an Ada support system including cross-development facilities. The first version of ECLIPSE is scheduled for delivery in early 1986 with subsequent releases planned in 1986 and 1987.

In itself, ECLIPSE is a large-scale software project. It involves almost 200 person-years of work undertaken by over 60 staff based at seven different sites. Different types of computer are in use (DEC VAXes, SUN workstations) and three different programming languages are being used for system development (Ada, C and Prolog). Equipment is connected using local area networks (Ethernet) and wide-area X-25 packet switched networks.

ECLIPSE is explicitly intended to be a *project* support environment rather than a programming support environment so emphasis has been placed on developing tools to support life cycle phases such as requirements definition and software design and on providing facilities which allow existing tools for programming support to be integrated with ECLIPSE. The ECLIPSE team is developing tools to support three different methodologies namely JSP/JSD (7), MASCOT (8) and LSDM/SSADM (9). In addition, it is hoped that third-party developers will integrate tools, with ECLIPSE, to support other methdologies including formal methodologies such as VDM (10). We are also developing tools for fast prototyping based on the reuse of existing software components.

The user interface with ECLIPSE is via personal workstations which support high-resolution graphics, a pointing device such as a mouse and multiple window working. Some ECLIPSE tools will be explicitly written for workstations (such as a generalised design graphics support system) and some tools (such as the SySL toolkit) will run on both the workstations and the computers hosting the ECLIPSE database system. Wherever possible, processing will be distributed to the user's workstation. Limited database distribution will also be supported.

The database management system which is used in ECLIPSE is a software product, called SDS-2 (11) developed by one of the project collaborators. SDS-2 is a network database system which has been explicitly designed to support the development of large-scale software projects. SDS-2 provides basic DBMS facilities, allows complex relationships amongst database entities to be expressed and provides a query language allowing the database to be examined and modified. The system includes facilities to support project management such as the generation of plans, schedules and costs but does not include configuration control facilities which are provided by the object management system described in the following section.

2.1 The ECLIPSE object management system

Configuration management in ECLIPSE is handled by a generalised object management system which is built on top of the SDS-2 database management system. This object management system has been designed in such a way that it avoids imposing any particular approach to configuration management on its users and simply provides fundamental facilities to support user's own in-house configuration management standards.

An ECLIPSE database is made up of a large number of *items* where an item is some logical real-world entity such as a chapter of a document, a software component, a description of a hardware component, a project plan, etc. Each item has a project-defined type and the notion of a type network exists where one type is deemed to be a specialisation or a generalisation of another type. For example, the type C-Code is a specialisation of type Source-code which is a generalisation of any other language type such as Pascal-code or Ada-code.

ECLIPSE items may be versioned or unversioned but in most databases most items exist in a number of different versions. Unversioned items are normally transient and are of no real interest to us here. As ECLIPSE runs under an operating system, ECLIPSE items may make use of operating system entities which are not controlled by ECLIPSE . Examples of such items might be Unix command processes or library procedures.

Each version of a ECLIPSE item is termed an *object*. Typically, each item has several associated objects and at any one time the user of an ECLIPSE database (which may be a person or a tool) sees a universe of objects which may be manipulated. These objects hold the actual data which is of interest to the user. Objects are read-only entities so that whenever an object is modified, a new object (that is, a version of the item) is created and associated with the original object's source item.

The particular objects which are accessible at any one time, depends on the user's *context*. A context is, simplistically, a list of names and object references where the name identifies the object of interest. The same name may be used in different contexts so that it may refer to different objects at different times.

In general, cross-references are made from object to item. Thus, if procedure A uses procedure B, an object associated with item A would include a reference to the item B not a particular version (object) of B. The version of B which is actually used is not built in to the item reference but is selected using the user's current context.

Say the production version of A made use of the production version of B which had the object identifier B1. Assume also that the development version of A makes use of the development version of B (say B2). In each case, the object representing procedure A would simply hold a reference to B (not B1 or B2). However, when the system came to be built, the user's context would either contain a reference to B1 or B2 so that the appropriate version of procedure B would be selected.

3. SySL - A SYSTEM STRUCTURE LANGUAGE

The ECLIPSE system allows very complex relationships between database entities to be set up with many of these relationships actually created by software tools. Although the user of the system may have access to the database schema and to the query language, it is, in general, extremely difficult for him or her to get some kind of overall picture of the part of the database of interest and the relationships which are (or which ought to be) maintained there. In many cases, it is also very difficult to check that relationships between items which should exist actually do exist and that items have a particular set of expected attributes. Thus, the aim of our work was to develop a notation which allowed the static structure of systems to be made apparent and to develop an associated toolkit to assist with activities such as system building, database validation and system viewing.

The fundamental assumption underlying our work is that a universe of entities exists in a project database and users are interested in particular configurations of these entities, the classes to which these entities belong, the structure of

members of these classes and constraints on individual entities and entity configurations.

SySL includes constructs to define the following:

(i) Classes of database items which are defined by enumeration of explicit items or by class unions.

(ii) The structure which is associated with all members of a particular class.

(iii) The specific configuration of individual members of a class.

(iv) The interface published by individual members of a class.

(v) Constraints on classes in general or on individual class members.

A SySL description consists of a number of class definitions where individual classes are defined, structure definitions where structure is associated with particular classes, assertions which place constraints on the attributes associated with class members and a large number of component descriptions where the static structure of the components making up a system is described.

A component description is a description of the static structure of a particular software component which is part of the system whose structure is being defined. Depending on the underlying implementation, a component may be a procedure, an Ada package, an 'include' file or whatever - SySL does not proscribe how a component is represented. As far as SySL is concerned, any item in the database may be considered as a system component.

However, we do support the notion of a primitive components and compound components. We define a primitive component to be a component where the database item represents a single logical system element such as a procedure or a function. A compound component is a component where the database item is a packaging of a number of logical components. Examples of this type of component include Ada packages which make several procedures/functions available through their interfaces and C 'include' files where several functions are contained in the same file.

We believe that users of an IPSE take on different roles at different times which means that they are interested in understanding the structure of systems at different levels of detail from the very abstract (what is the general standard structure for a specifications document? say) to the concrete (what is the particular organisation of the SySL specification document?). Thus, a key feature of SySL is the ability to express system structure at different levels of detail. For example, the description below might be part of the SySL specification of a personal workstation.

```
structure WORKSTATION is
        PROCESSOR,
        MEMORY,
        BACKPLANE,
        DISPLAY,
        -- square brackets mean optional item
        [DISK_SYSTEM]
        -- []* means 0 or more items of this type allowed
        [ETHERNET]*
        POINTING_DEVICE,
        KEYBOARD,
        -- tape is optional
        [TAPE],
        OPERATING_SYSTEM;
end structure
```

This specification of the structure of a workstation indicates that it is made up of a number of components some of which are optional (indicated by square brackets). In this abstract description, the names used in the description are class names (distinguished by the fact they are written entirely in upper-case letters). Thus, workstations in general are made up of processors, memories, backplanes, displays, etc. The writer of the description has not chosen to associate particular item names with these class names, that is, to specify which particular processor is used, what sizes of memory are available, etc. This specific information is provided as part of a concrete description of a specific workstation.

Classes are defined explicitly by enumeration and may have an associated structure as defined above. For example, the class WORKSTATION and the class PROCESSOR may be defined as follows:

```
class WORKSTATION is (MOON_WORKSTATION,
VENUS_WORKSTATION, MARS_WORKSTATION)

class PROCESSOR is (m68010, m68020, ns32032)
```

The class WORKSTATION is defined in terms of sub-classes MOON_WORKSTATION, VENUS_WORKSTATION and MARS_WORKSTATION (any unescaped sequence of upper-case letters, underscores and digits is taken as a class name) and the class workstation is defined as the union of the classes MOON_WORKSTATION, VENUS_WORKSTATION and MARS_WORKSTATION. The class PROCESSOR, on the other hand, is described explicitly as being made up of the items identified by the names m68010, m68020 and ns32032.

A structure may be associated with any class via a structure declaration. This means that the structure of all items in that class must correspond with that defined. For example, the declaration below associates a structural description with the class MEMORY:

```
structure MEMORY is
        MEMORY_MANAGEMENT_UNIT,
        -- + means that there must be one or more items
        [MEMORY_BOARD]+;
end structure
```

This definition states that items of class MEMORY are composite items made up of an item of class MEMORY_MANAGEMENT_UNIT and a number of items of class MEMORY_BOARD. Notice that following a bracketed item with a '+' means 1 or more instances of that item, following it with a * means zero or more instances. The upper bound on the number of instances may be specified by following the '+' or '*' with an integer as shown in the MOON_WORKSTATION example below.

The definition of a workstation structure allows for the possibility that some elements of the structure may be empty. Thus, not all workstations need be equipped with a tape drive, some may be connected in a network and may be diskless, etc. However, not all possible options are allowed so we allow the

user to make assertions defining disallowed structures. For example, the following assertions would normally be associated with the structure WORKSTATION.

```
-- Diskless workstations must have an ethernet interface
assert WORKSTATION: not (Not_present (ETHERNET)
                         and Not_present (DISK_SYSTEM))

-- Workstations with a tape must also have a disk
assert WORKSTATION: not
        (Not_present (DISK_SYSTEM) and Present (TAPE))
```

Assertions are always associated with particular classes and the assertion mechanism is a general one. As well as specifying constraints on system compositions, it may also be used to associate particular attributes with classes. For example, the following assertions state that items of class DISPLAY must have an attribute called RESOLUTION which may have the value '1192/768' or the value '1000/800'

```
assert DISPLAY:  Has_attribute (RESOLUTION)
assert DISPLAY:  Has_value (RESOLUTION, '1192/768')
                 or Has_value (RESOLUTION, '1000/800')
```

The workstation description above is very general indeed and might apply to any of the personal workstations which are now available. We can define a more detailed workstation description as shown below.

```
structure MOON_WORKSTATION is
        PROCESSOR => (m68010, m68020)
        MEMORY => MOON_MEMORY
        BACKPLANE => multibus
        DISPLAY => (monochrome_display, colour_display),
        DISK_SYSTEM => (d71Mb, d130Mb, d430Mb,
                        null)
        POINTING_DEVICE => OPTICAL_MOUSE,
        OPERATING_SYSTEM => (unix4.2bsd, unixsV)
end structure
```

Again, this is a generic description but at a less abstract level than the description of workstations in general. Parts of the structure of class WORKSTATION have been instantiated either to specific database items or to sub-classes. Class names which are associated with a previously declared type name introduce a sub-class. Thus, MOON_MEMORY is a sub-class of the class MEMORY, OPTICAL_MOUSE is a sub-class of the class POINTING_DEVICE, etc. This class network is reflected in the ECLIPSE database where each item must have an associated type (class name) and specialisations and generalisations of types are supported. It is assumed that those parts of the structure of workstation which have not been instantiated are unchanged and are inherited directly from the description of the structure of WORKSTATION.

The description may be further refined to refer to specific individual workstations. For example:

```
structure IANS_MOON: MOON_WORKSTATION is
        PROCESSOR => m68010,
        MEMORY => s2mbytes,
        DISPLAY => monochrome_display,
        DISK_SYSTEM => d71mb,
        ETHERNET => e12345,
        OPTICAL_MOUSE => m1234,
        KEYBOARD => k1234,
        OPERATING_SYSTEM => unix4.2bsd;
end structure
```

This defines a specific workstation configuration where all of the names must refer to specific items in the ECLIPSE database rather than item classes. In practice, of course, the items in the database for hardware systems are descriptions of the hardware.

We have deliberately introduced this section on SySL with a hardware example to illustrate the flexibility of the notation. However, we anticipate that the language will be most used for describing the structure of software and documentation systems. As part of the evaluation of SySL, we have written a complete description of the structure of the Unix kernel and anticipated that this might serve as a source of useful examples for this paper. In fact, because there is no associated database, the Unix system is represented as a large collection of files and its structure is flat and uninteresting. It has not proved particularly useful for assessing features of the language.

Instead, we use examples taken from an electronic mail system which is currently being developed in Ada. This system provides the usual mail facilities of sending and receiving mail, uses menus for user interaction, provides comprehensive on-line help facilities and makes use of a number of abstract data types. Initially, it is possible to define the expected structure of (generalised) systems written in Ada:

```
structure ADA_SYSTEM is
        -- packages of shared constants, types, variables, etc.
        [SHARED_COLLECTION]*
        [ABSTRACT_DATA_TYPE]*,
        [SUB_SYSTEM]*,
        MAIN_PROCEDURE;
end structure
```

This specifies that a system of class Ada system should be made up of a number of items defining shared collections of types, constants, etc, packages defining abstract data types, packages defining sub-systems and a single item of type 'main_procedure'.The classes declared here are s as a union of sub-classes as follows:

```
class ADA_PACKAGE is  (SHARED_COLLECTION,
                       ABSTRACT_DATA_TYPE,
                       SUB_SYSTEM)
```

A possible assertion which might be associated with the class ADA_PACKAGE is:

```
assert ADA_PACKAGE: Has_attribute (Ada_package_spec)
                    or Has_attribute (Complete_Ada_package)
```

This assertion states that to be used as part of a system, the Ada package must either be a specification of a complete (specification + body) package. Package bodies alone may not be included.

The highest-level description of the electronic mail system is:

```
system Electronic_mail_system:  ADA_SYSTEM is
        external (print_spooler, sort_command);
        provides (Em_system);
        SHARED_COLLECTION => Em_types
                             Em_constants
-- No abstract data types visible at this level in this system
-- Define the sub-systems
        SUB_SYSTEM =>   Menu_manager
                        Send_mail
                        Receive_mail
```

```
                    Help
                    Mailbox_manager
MAIN_PROCEDURE => Em_system

end system
```

This description starts with a SySL **external** declaration which specifies that this system requires access to items which are not controlled by the ECLIPSE database. Here, it makes use of a print spooler and a sort command which are presumed to be provided by the operating system. Declaring these entities as external indicates that they are an integral part of the system but are not ECLIPSE database items.

The **provides** declaration is used in a SySL description to specify which names used in the package are made visible to system users. Here, we specify that the name Em_system is published as the interface to Electronic_mail_system and may be used in other SySL definitions. The use of a name in a **provides** declaration implies that that component is part of the component item and that its use in a structure description does not represent a reference to another ECLIPSE item. Thus, the item identified by the name Electronic_mail_system contains the main_procedure called Em_system. The description continues with a list of the ECLIPSE items which make up the electronic mail system. Thus, the sub-systems Menu_manager, Send_mail, etc are separate and distinct items for which SySL definitions should exist.

Notice that the Electronic_mail_system has been identified as a **system** whereas the Mailbox_manager below has been identified as a **component**. The difference between these is that components may be parts of systems but systems may not be part of components. Whilst this distinction is not strictly necessary, we believe that it reflects a natural human structuring mechanism.

The components of the electronic mail system are also described using SySL. For example, the Mailbox_manager sub-system may be described:

```
component Mailbox_manager: SUB_SYSTEM is
    provides ( Get_from _mailbox_of,
               Discard_expired_mail_from,
               Put_item_in_mailbox,
               Move_mail_to_mail_list
             );
    SHARED_COLLECTION => Em_types
                         Em_constants
    ABSTRACT_DATA_TYPE => Mailbox
                          Mail_list
    PROCEDURE =>    Get_from_mailbox_of
                    Discard_expired_mail_from
                    Put_item_in_mailbox
                    Move_mail_to_mail_list
end component
```

Here, the component description specifies which procedures may be called by users of that component and list ECLIPSE items which are used by the Mailbox_manager sub-system. Mailbox_manager provides four procedures called 'Get_from_mailbox_of', 'Discard_expired_mail_from', 'Put_item_in_mailbox', and 'Move_mail_to_mail_list'. It makes use of the ECLIPSE items named Em_types, Em_constants, Mailbox, and Mail_list.

The Receive_mail component is described in SySL as follows:

```
component Receive_mail: SUB_SYSTEM is
    provides (Present_mail, File_mail, Redirect_mail);
    requires ( Mailbox_manager);
    SHARED_COLLECTION => EM_types
                         EM_constants
    SUB_SYSTEM => Send_mail
    PROCEDURE =>
        Mailbox_manager.Discard_expired_mail_from
        Mailbox_manager.Get_from_mailbox_of
        Present_mail
        File_mail
        Redirect_mail
end component
```

In this description, the **requires** declaration specifies the name of an item that is required and implies that the component being described makes use of facilities provided by that item. It is rather like an Ada **with** clause. Here we specify that the item Mailbox_manager is required and, within the definition, we specify which procedures from Mailbox_manager are actually used, namely Discard_expired_mail_from and Get_from_mailbox_of. These names should be published in the **provides** declaration of Mailbox_manager. The **provides** clause specifies that the Receive_mail sub-system provides the procedures Present_mail, File_mail and Redirect_mail.

In our final example, we show the SySL description of the Mailbox abstract data type.

```
component Mailbox: ABSTRACT_DATA_TYPE is
    provides (Append_item, Remove_item, Clear,
              Is_empty, Get_owner);
    PROCEDURE =>    Append_item
                    Remove_item
                    Clear;
    FUNCTION =>     Is_empty
                    Get_owner
end component
```

This component is entirely self-contained and does not use any other ECLIPSE item. All of the modules making up Mailbox are specified in the **provides** list and are thus assumed to be part of the Mailbox item itself.

3.1 Binding SySL to the ECLIPSE database

The writer of a SySL description may chose any names that he or she likes for SySL classes, components, etc. and we assume that sensible and meaningful identifiers will normally be used. When the SySL description is processed, these names are either implictly or explicitly bound to objects in the ECLIPSE database. Explicit binding is accomplished using a rename declaration which has the form:

rename <SySL_name> **as** <ECLIPSE_name>

The ECLIPSE name must appear in the user's current context as defined in section 2 above and must refer to an object in the ECLIPSE database. The rename construct is used to associate some SySL name with an ECLIPSE name which is not appropriate in the context of the SySL description. Binding of names to objects is carried out dynamically when a structure description containing the name is used.

More commonly, binding of SySL names to ECLIPSE names is implicit where the names used in a SySL description themselves appear in the user's context. Thus, when a tool

processes a SySL description it does so with the current context as input and dynamically binds SySL names to database objects as these names are encountered. These database objects may then be accessed by the SySL processing tools.

3.2 SySL and configuration descriptions

As can be seen from some of the above examples, SySL allows the user to specify alternative compositions for particular items. Thus a MOON_WORKSTATION may include the item d71mb of type DISK_SYSTEM, the item d130mb of the same type, the item d430mb, or no DISK_SYSTEM at all. However, it must be emphasised that SySL only allows for alternative items to be specified so that similiar system structures can be defined. SySL does not provide facilities to specify that a particular version of a system or a component is made up from particular versions of its parts.

In fact, the facilities provided by the ECLIPSE system make such implicit version specification unnecessary. As discussed above in the section on ECLIPSE, it is normal for all items in a ECLIPSE data base to be versioned with each distinct version termed an object. In all cases then, an item name really means one of the objects associated with that item but the binding of the name to a particular object is not made when the system is defined but is made at run-time when the object is actually referenced. Thus, if some name X, say, is used in a SySL system, the user's current context is evaluated and the database object named X in that context is selected. If some other version of the component X is required, the context must be changed (using system commands) to ensure that it refers to the version of X which is required.

The tools which process a SySL description always do so in an environment defined by the user's current context. The current context contains the names of items which are referenced in the SySL description plus the association of these item names with particular objects. The evaluation context may be saved with a SySL description so that future processing will always access the expected objects but the normal default for initial SySL descriptions is to bind the name to the latest version of the referenced item. Thus, one context may refer to the system being developed, another may refer to the previous version of the system, another may refer to the system delivered to some customer T. Thus, a complete specification is made up of a SySL description and the context which binds this to the ECLIPSE database.

4. The SySL TOOLKIT

SySL is a notation for describing systems represented in a ECLIPSE database and is thus useful in its own right. However, SySL was designed to be supported by automated tools and it is our intention to develop a number of tools which use SySL descriptions as their input. Indeed, without such tools, the preparation of the descriptions of large systems using SySL would be a tedious and error-prone activity. Our work so far has concentrated on the development of a SySL 'compiler' which translates a SySL description to an internal directed graph representation but we intend to build several other SySL processing tools. These include:

1. A SySL viewing and editing system

This is a tool which will make use of the windowing facilities on a workstation to help the reader of a SySL description to prepare and to read and understand that description. We may also include facilities for displaying part of a SySL description as a diagram.

This is an absolutely critical part of the SySL system as a powerful language-directed editing system makes it possible to include language facilities which enhance readability but which are burdensome to maintain without automated support. For example, consider the language feature where classes are defined in terms of sub-classes. A complete description requires that all sub-classes of a class should be specified in the class definition yet a common operation is to add a new sub-class to an existing class (a SATURN_WORKSTATION to the above example, say). An editing tool can add this automatically on user confirmation whereas manual update is liable to error.

2. A database checking system

This is a tool which takes a SySL description of a ECLIPSE database and which interacts with the database to check that the SySL structure is a correct representation of the database structure. This is necessary as the database structures are usually built by other tools whereas the SySL description is a perception of the system structure as expected by the user.

3. A system builder

The information captured in a SySL system description records the dependencies of one system component on another and appears to be sufficient to allow us to build a tool which, if the item types allow, will automatically build a system from its components. This will involve adding some extra information to a SySL description (in the form of formal comments) but much of the build information for an item is actually recorded as item attributes.

5. CONCLUSIONS

The language which we have developed appears to be a useful one to describe the structure of complex systems. It has been evaluated by describing hardware, software and documentation systems. and is sufficiently general purpose to describe any type of system which may be represented in a ECLIPSE database. At the time of writing, the development of a SySL compiler is in progress and we hope to include the first of our SySL tools, namely the system editor and viewer, in the first release of ECLIPSE.

6. ACKNOWLEDGEMENTS

The work described here is funded by the Alvey Directorate, UK. Thanks are due to our collaborators Software Sciences Ltd, CAP Group UK Ltd., Learmonth and Burchett Management Systems Ltd., the University of Lancaster and the University College of Wales at Aberystwyth.

7. REFERENCES

1. 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.

2. Tichy, W.F. Software Development Control Based on Module Interconnection. *Proc. 4th Int. Conf. on Software Engineering*, 29-41, Munich, 1978.

3. Feldman, S.I. MAKE - A program for maintaining computer programs. *Software - Practice and Experience*, 9, 255-265. 1977.

4. Lampson, B.W. and Schmidt, E.E. Organising Software in a Distributed Environment. *SIGPLAN Notices*, 18 (6), 1983.

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. Sommerville, I. and Thomson, R. *SySL - A System Structure Language*. Research report CS/ST/9/85, Software Technology Research Group, University of Strathclyde, Glasgow. 1985.

7. Jackson, M.A. *System Development*. London: Prentice-Hall. 1982.

8. Jackson, K. Language design for modular software construction. *IFIP 77*, 577-582, Amsterdam: North-Holland. 1977.

9. Maddison, R.N. *Information Systems Methodologies*. London: Wiley Heyden. 1983.

10. Jones, C.B. *Software Development - A Rigourous Approach*. London: Prentice- Hall. 1980.

11. Software Sciences Ltd. *The SDS-2 system*. Farnborough, Hants. UK. 1985.

8.BIOGRAPHY

Ian Sommerville and Ronnie Thomson are members of the Software Technology Research Group at the University of Strathclyde. The group is primarily involved in research in support environments and has specific interests in programming in the large, software reuse and user interface design for software engineers.

Ian Sommerville has been a lecturer at Strathclyde University since 1978 and is the author of the widely used textbook *Software Engineering* published by Addison Wesley. Ronnie Thomson is a graduate of Strathclyde University and is currently a research associate with the Software Technology Research Group.