Proc. 1st European Conference on
Software Engineering, Strasbourg. Sept. 1987

(R)

1987E

SOFTWARE DESIGN AUTOMATION IN AN IPSE

Stephen Beer and Ray Welland,
Department of Computer Science,
University of Strathclyde,
Glasgow G1 1XH, Scotland.

Ian Sommerville,
Department of Computing,
University of Lancaster,
Lancaster LA1 4YR, England.

ABSTRACT

This paper describes an editing system which is explicitly designed to support the production of graphical representations of a software design. The novel features of the system are that it is table-driven, so that it may be tailored to support most graphical design notations, and that it explicitly includes a means of defining the rules of the design method. These rules may be enforced automatically or checked at user request by the design editing system.

Furthermore, the system is intended to operate within the context of an integrated project support environment called ECLIPSE and the designs generated by the system are stored as objects in the ECLIPSE database. These objects have a defined structure and may be manipulated by other tools (such as a code generator). The editing system is implemented in C and runs on a Sun workstation.

Keywords: Design diagram construction, Method description, Design checking, Design method support, Integrated Project Support Environment.

Automation de la Conception du Logiciel dans un Environnement Intégré

Résumé

Cette communication décrit un système d'annoter, qui est explicitément construit pour supporter la production de representations graphiques d'une conception du logiciel. Les traits nouveaux du système sont qu' on le fait marcher avec des tables pour qu'il soit possible de l'adapter pour supporter la plupart des notations graphiques de la conception et aussi qu'il embrasse explicitément les moyens de définir les règles de la méthode de la conception. Le système peut automatiquement observer ces règles ou il peut les vérifier sur la demande de l'utilisateur.

De plus, le système a l'intention de fonctionner dans le contexte d'un environnement intégré qui s'appelle ECLIPSE et les conceptions, que le système produit, sont gardées comme des objets dans la base de données d'ECLIPSE. Ces objets possèdent une structure bien déterminée et des autres outils (comme un générateur de code) peuvent les manipuler. Le système est écrit en 'C' et il marche dans un poste de travail de SUN.

Mots-clés: La construction de notations graphiques de la conception; la description des méthodes de dévloppement du logiciel; la vérification de la conception; les méthodes de développement du logiciel; l'environnement intégré de soutien de projet.

INTRODUCTION

The work described here is taking place in the context of an integrated project support environment (IPSE) called ECLIPSE (1) where we hope to support different approaches to the software process and provide tools to assist with software design.

Although they have been criticised for lack of formality, graphical approaches to software design such as Structured Design (2) and JSD (3) are widely used and reportedly successful. Thus, it was a requirement of ECLIPSE that it should be possible to incorporate support tools for such methods and to allow designs produced with such tools to be stored and manipulated in the ECLIPSE database. ECLIPSE is an open IPSE so the methods to be supported could not be predefined - indeed as the system construction progressed we changed our mind on which methods should be supported in the initial release of ECLIPSE. A further requirement on ECLIPSE was an integrated approach to the user interface so we came to the conclusion that the most effective way to provide a design support tool for graphical design methods was to produce a generic tool which could be tailored by the system builder for whatever methods were supported in any single release of ECLIPSE.

However, we did not simply want a graphical editing system which allowed us to produce neat diagrams. We also wished to provide as much design checking as possible as the design was created - in essence we wanted to provide a syntax-driven editing system to support whatever graphical method was in use. Thus, we decided that we should define a notation for defining the method to be supported and use this to generate tables to drive the design editor. Furthermore, as we wished to support notations with arbitrary symbolism we also needed a tool with which we could define symbols and relate them to their names used in the design definition language.

The design definition language which we developed is called GDL (4) and is a notation for defining the syntax and partial semantics of software designs which are expressed as directed graphs. This means that most graphical design methods may be supported. The relationship between GDL, the symbol editing system, the design editor and the database is shown in Figure 1.

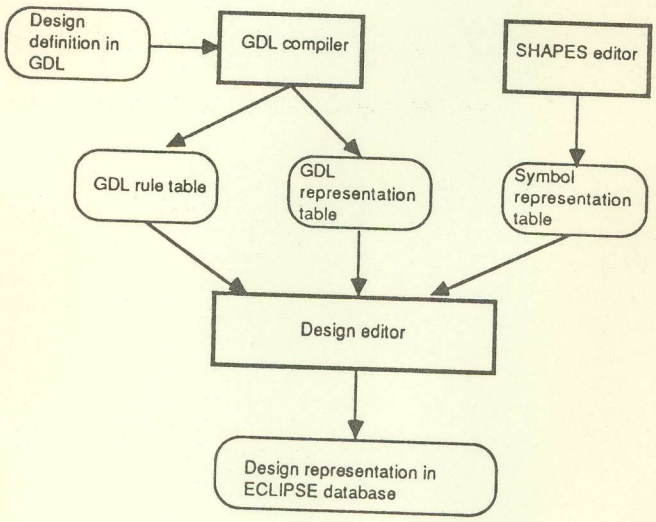

Figure 1. The ECLIPSE design editing system

The mode of use is as follows:

1. The IPSE tool builder defines the syntax and the semantics of the design method to be supported using GDL.
2. The GDL compiler generates tables for input to the design editor.

3. The symbols associated with a method are defined by the IPSE tool builder and tables are generated for the design editor.
4. The design editor uses these generated tables to provide an interface which is tailored to whatever design method is in use.
5. The generated designs are stored in the ECLIPSE database. Notice that we have defined a canonical form for all software designs (as a directed graph) so that generic design processing tools may be produced.

In the remainder of this paper, we describe the notations for defining software design methods, briefly introduce the symbol editing system, illustrate the user interface and facilities of the design editor, paying particular attention to its design checking facilities and, finally, discuss how the editor is used in conjunction with other design method support tools.

GDL

Our basic premise is that a design diagram is a **directed graph.** Therefore, our design description notation (GDL) must be able to define the general structure of a graph, the symbolism and naming of each entity type, and other rules associated with the method. Representing a directed graph, suggests that there should be two basic types of entity: **node** and **link**. From these two basic types a hierarchy of user-defined types can be specified with each level in the hierarchy inheriting properties from its predecessors.

The initial version of ECLIPSE was tested using the JSD method (3) and the examples in this section are drawn from the GDL definitions for JSD. In particular,the main example is of the synchronisation process node of the System Structure Diagram (SSD). A synchronisation process takes a single system input ("time stamp") and produces many data streams ("time grain markers"). A fragment of the GDL which describes a synchronisation process is given in example 1.

```
type SSD_SYNC is NODE ( time_in : in SSD_LINK;
                        times_out : out set of SSD_LINK)
for SSD_SYNC use SYMBOL(JSD_ssd.synchronisation) ++ NAME(STRING)
assertion Input(SSD_SYNC) : GetType(Source(time_in)) = SSD_SYS_IN
assertion Outputs(SSD_SYNC) :  forall i ; Member(times_out, i) :
                               GetType (Destination(i)) = SSD_DATA_STREAM
assertion Name_inside(SSD_SYNC) : inst i :
                                  Encloses(GetIabel(i, SYMBOL), GetIabel(i, NAME))

type SSD_LINK is LINK (start : in NODE; finish : out NODE)
for SSD_LINK use SYMBOL(JSD_ssd.arrow)
```

Example 1.

To declare a node the first thing we consider is its interface to other nodes in the diagram, that is what directed links come into the node and what directed links leave the node. A link is always assumed to have a single source node and a single destination node. The GDL fragment in Example 1 indicates that a synchronisation process, SSD_SYNC, has a single input of type SSD_LINK and a set of outputs, also of type SSD_LINK. The type SSD_LINK is a simple link between two entities of the base type node.

Having established the 'shape' of the graph we can now turn our attention to the symbolism of the entities and any annotation associated with the entities. Any type declaration may have an associated representation expression which gives these details. A representation expression contains a list of entity annotations, which are represented as **labels** in the Design Editor. The SYMBOL part references a component in a shape library (see next section). Thus JSD_ssd is a shape library and 'synchronisation' and 'arrow' are components defined within it.

A synchronisation process also has a name of type STRING. A SSD_LINK is an unannotated arrow. Provision is also made in GDL for optional annotations, which are indicated by enclosing the label definition in square brackets.

The type declaration and associated representation expression establish the basic framework for a design diagram but there are other rules which need to be associated with a design diagram. In GDL these rules are expressed as **assertions** which are written as predicates. Assertions can be roughly divided into two categories, those which express further syntactic or semantic constraints and those which express spatial relationships.

Refering to the example of the synchronisation process, we find that input must come from a system input node, SSD_SYS_IN, and all outputs must be connected to nodes of type data stream, SSD_DATA_STREAM. These constraints can be expressed using the named assertions: Input and Outputs shown in Example 1. In these assertions GetType, Source, Destination and Member are examples of built-in functions. GetType is a generic function which takes an entity reference and returns its type. Source and Destination both take parameters of type link and return a reference to the node at the relevant end of the link. The function Member(X, i) is true if i is a member of the set X and so we can use this in combination with the iterator **forall** to scan all members of a set.

We also note the need for a simple spatial constraint that the name of any instance of a SSD_SYNC should appear within the symbol for the node. This can be expressed as shown in the assertion Name-inside in Example 1. Getlabel and Encloses are further examples of built-in fuctions. Getlabel (A,B) returns a reference to the label named B associated with entity A. One of the attributes of a label is the area which surrounds the label. Encloses (X, Y) is true if the area surrounding Y is enclosed by the area surrounding X. Thus the assertion is true if the NAME annotation is enclosed by the SYMBOL defined for SSD_SYNC.

A facility for including user defined functions is available in GDL and the above spatial constraint occurs so frequently that it is likely to be defined as a named function, as follows:

```
define Name_enclosed (X : NODE)
-- the name of the node must be enclosed by the node's symbol
Encloses (Getlabel(X, SYMBOL), Getlabel(X, NAME))
```

and the assertion in Example 1 then becomes:

```
assert Name_inside(SSD_SYNC) : inst i : Name_enclosed(i)
```

Many design methods include hierarchic decomposition of designs. Thus a node on a diagram at one level may be expanded into a complete diagram at the next level in the hierarchy. We represent this in GDL by associating an **abstraction** with a node type. This allows to maintain two representations of a node, one its simple form as it appears on a higher level diagram and the other its expanded form at a lower level in the hierarchy.

There are also some design methods which allow composite nodes on a single level diagram, for example the Systems Implementation Diagram (SID) of JSD. These composite nodes are represented by extending the node parameters to include the concept of **owner_of** and **owned_by** relationships between nodes.

The GDL definitions for a method are compiled into tables for input to the design editor. The compiler was constructed using the lex and yacc tools available under the Unix operating system (5).

THE SHAPES EDITOR

The SHAPES editor provides a drawing tool tailored for the production of symbols which appear on design diagrams. The symbols which the user creates are stored as components in a method-specific library which is used as input to the design editor and referenced in the SYMBOL label of the GDL representation expression (**for ... use**).

The SHAPES editor is **not** a general-purpose drawing tool, it is designed specifically for constructing the stylised symbols which appear on design diagrams. The user is provided with a set of basic shapes including:

rectangle (square), ellipse(circle), triangle, diamond and line (single-headed and double-headed arrow). These basic shapes are provided in a graphical menu which is positioned at the left-hand side of the editor's drawing area. The user can select a shape from the menu and instantiate it in the drawing area at any required size. Shapes in the drawing area may subsequently be modified by stretching or shrinking in any planar dimension and other shapes may be selected from the menu and added to the existing shape to form composite shapes.

Some shapes are treated as special cases of others, a rectangle is constrained to be a square by holding down the shift key while drawing, similarly a circle is a constrained ellipse. Arrows, single-headed and double-headed, are treated as special cases of line, the user selects the appropriate 'line style' from a menu. When the user is satisfied with a new symbol, constructed in the drawing area, it can be named and added to the graphical menu of shapes. The user can then continue shape development building further symbols from basic shapes or by modifying the additional shapes already added to the menu.

At the end of a SHAPES editor session the user can store the new shapes, currently displayed in the graphical menu, in a shape library for use by the design editor. Shapes are stored in a shape library geometrically rather than as bit-map images. Any shape library can be reloaded into the SHAPES editor to allow further development of the symbols.

THE DESIGN EDITOR (DE)

The envisaged user of the DE is a software designer having knowledge of the particular design method being used within the DE. Rather than interacting in terms of graphical drafting terminology (eg. box, circle) the DE uses the same terminology as the designer. A designer selects a SSD_SYNC or a SSD_LINK, for example, and adds it to a design as opposed to selecting a box or a line and drawing on a diagram. Therefore the designer interacts in terms of the **design entities** of a method.The DE asumes that the diagram is based on a directed, annotated graph and the objects which may be manipulated within the graph structure are:-

- **node** denotes a software component, state
- **link** denotes flow of control or data, etc.
- **label** denotes the textual and graphical descriptions for labelling a node or link

The major facilities offered by the DE allow the user to:-

- add images representing design entities, move and delete such images,
- add, move or delete labels (textual or symbolic) associated with a design entity,
- create a design diagram larger than the designers' workstation window,
- view the total diagram; it is necessary to be able to "step back and look at" large areas of a design. This cuts out the "noise" of detail in order to view the overall structure,
- annotate the diagram only, not the underlying design, with text, boxes or lines.

The object oriented approach to user interaction has been pursued through out development of the DE. The functions available to the designer in constructing a design are applied to a currently selected object from the design. The current selection may consist of a node, link, label or collection thereof. Functions are applied consistently across all objects wherever it is sensible to do so, for example, it is nonsense to edit the graphical symbol of a node but sensible to edit its labels.

The implication here is that the designer points at an object to make it the current selection and then applies some editing function, such as delete or move. The converse to this philosophy is the function-oriented approach where a function is first selected followed by the objects to which the function is to be applied. The choice of interface essentially depends on whether user-interaction is more natural with the object or the function. For a design editing tool we believe that the best interface is via the object-oriented approach.

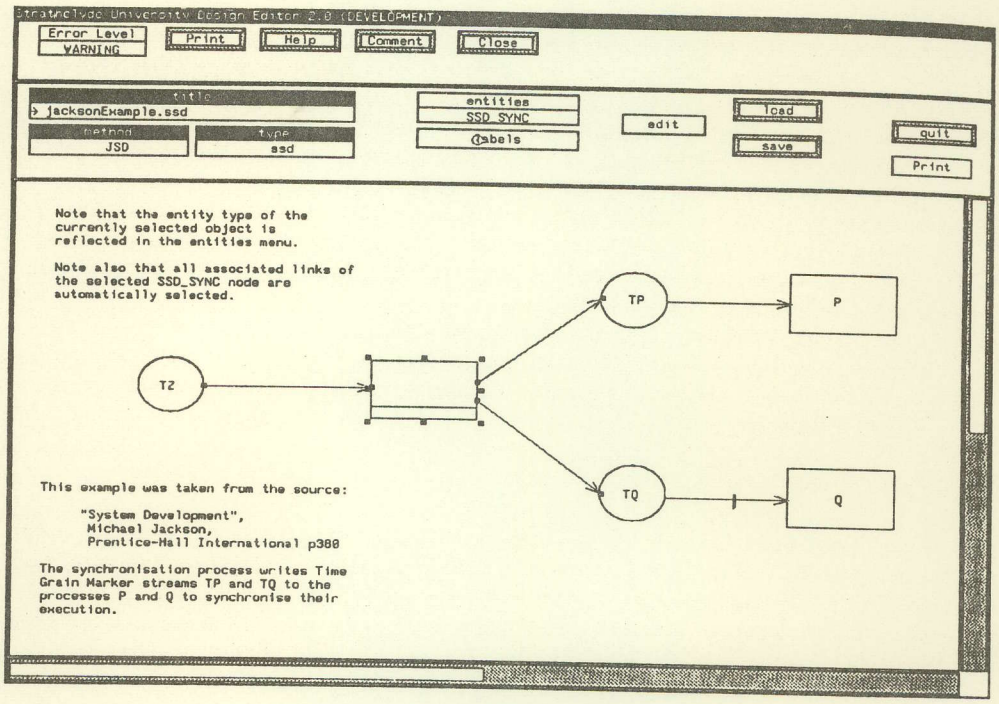

Figure 2. Design editor screen dump

The DE has been implemented on SUN workstations running under the Suntools window environment. The user interface, see Figure 2, consists of a tool window subdivided into five subwindows:

- drafting area where a design is constructed,
- a control panel giving access to the editing functions and entity types,
- an ECLIPSE standard tool subwindow providing functions common to all ECLIPSE tools,
- two subwindows containing scroll bars. These allow the the drafting area to be moved around the total diagram. (Remember a diagram can be larger than the drafting area).

The interface makes full advantage of the mouse pointing device in that all user interaction, except for textual input, is via the mouse. The user interface to the DE follows the WYSIWYG approach where text is input at the position of display rather than a point remote from the actual display.

Editing functions are selected from a control panel (6) containing pull down menus and "soft" buttons. The **entities** menu displays the types of entity available for the supported method. The designer selects an entity type from the menu and then adds this to a design by fixing the position for its graphical image on the diagram.

The concept of a **label** is used within the DE to associate either a name or a graphical symbol with a node or link. Therefore, the label is a generic object for capturing textual and graphical descriptions. Labels have a defined enclosing boundary, contain a value and may be manipulated in the same fashion as other objects of the design.

A design entity (node or link) may have an associated set of labels. The type and number of labels is specified in the representation expression of the entity type in the GDL description of the method. For example, Example 1 shows that the node entity type SSD_SYNC has an associated label type NAME. To associate a NAME

label with an SSD_SYNC instance in the design, the designer first selects the instance, making it the currently selected object. At this point the **labels** menu in the control panel dynamically changes to show the available types for the current object, see Figure 2. The designer selects the label type NAME and types in the name of the SSD_SYNC instance.

The user has complete control over where and when a label should be placed. There is no notion of the editor being syntax directed- it is syntax driven but not syntax directed. The use of a syntax-directed editor would force the user into specific ways of design creation. We feel it more natural that the designer should create a design as he pleases, and that most design checking should be initiated when required rather than imposed at each step of the editing session.

As mentioned previously, the DE has built-in knowledge that the design must be in the form of a graph. Therefore, the DE enforces the restriction that a link must originate from a node and end at a different node. This prevents designs being created where dataflow links, for example, lead to or originate from nowhere. This restriction is enforced implicitly within the DE when the designer attempts to add a link. The DE prevents a source point for a link from being fixed unless it lies within a node boundary, similarly for the destination point.

This graph knowledge is used in other situations. For example, each node can have an associated set of input and output links and labels. When a node is made the currently selected object then its links and labels are also automatically selected. Subsequent functions applied to a node are also applied to its links and labels, for example, if a node is deleted then all its associated links and labels are also deleted- it makes no sense for them to be left hanging in mid-air. In the same way a move operation automatically moves all links and labels associated with a node.

DESIGN CHECKING

In a software design editing system the provision of drafting facilities for automating diagram production is important. What is even more important, from the point of view of ECLIPSE, is that the underlying design is captured in the project database. A beautiful looking diagram is of no use, and indeed may be be harmful, if it is incorrect. Design checking therefore is a major function of the DE and is one of the ways in which it differs from a straightforward drafting tool.

Other design support tools have demonstrated the need for method-specific checking of a design (7) but the novel feature of the DE is that checking is enforced at three levels and at various times throughout an editing session. These checks are all closely integrated with the GDL description of the method being supported and the three levels can be defined as :-

- parameter list checking
- assertion checking
- completeness checking

In the case of a node the typing of parameters can be used to enforce correct design automatically. This is so because each node has associated links defined as being either **in** or **out**. Subsequently, in the DE, at the time when a link is added to a design the parameter lists of both the source and destination nodes can be checked. This check ensures that the link type is consistent with the legal types of the source node's **out** links and also with the destination node's **in** links.

For example, referring to the GDL in Example 1, the designer is allowed to use only links of type SSD_LINK to connect a node of type SSD_SYNC to any other node. This restriction can be enforced by checking that if an SSD_SYNC instance is selected as the source or destination node when drawing a link then the current link type, selected from the entities menu, must be SSD_LINK. If an attempt is made to draw an illegal link then no link is added to the design, the design image remains unchanged and a suitable message indicating a wrong connection is displayed in the control panel.

The rules of any particular method to be enforced in the DE are called **assertions**. These are compiled by the GDL compiler into rule tables which drive the DE. In Example 1 there are three assertions on a node of type SSD_SYNC. They concern the types of the connected input and output nodes, and the placing of the name

label.These assertions could be enforced at different times in an editing session, as described later, but we believe that the best approach is to allow the user to specify when checking should take place. At this time any entity in error is highlighted and made the current selection. An appropriate message is displayed in the control panel and the designer can then apply functions to the erroneous entity to correct the design.

In a GDL representation expression a label can be specified as being compulsory or optional (by using square brackets). This information on the optionality, or otherwise, of a label is transmitted to the DE through the GDL generated tables and the presence of mandatory labels is checked at an appropriate time. This is an example of a **completeness** check which can only be carried out under the control of the user. Again, looking at Example 1, a node of type SSD_SYNC must have a name. The DE therefore has to check each instance of an SSD_SYNC node and if no name exists then an appropriate message is displayed and the offending node highlighted.

The specific timing of checks is a contentious subject. The basic philosophy behind the DE is that the designer should be given as much freedom as possible to construct a design diagram is his or her own way. This is achieved by provided an object-oriented, modeless interface with most of the design checking initiated at user specified times.

Checking in the DE can be classified on its timing during an editing session as follows:

- **implicit/restrictive.** There is implicit continuous checking throughout an editing session because certain editing operations are restricted at certain times. For example, at the point when a node is selected only certain label types are made available to the designer. This ensures that, by restriction, the designer cannot associate a label of incorrect type with a particular node.
- **immediate**. As soon as an editing operation alters a design certain checks will be executed immediately. These include checking that a link has source and destination nodes of the correct type.
- **user-initiated**. Rather than providing a syntax directed editor where the user is forced to construct a design in a certain manner the DE allows the designer the freedom to construct a design in whatever manner favoured. Checking can be called at the designers' convenience and such checking might include: assertions concerning completeness and consistency, assertions about spatial arrangement of objects, and completeness of label sets.

THE DESIGN EDITOR WITHIN A METHOD-SUPPORT SYSTEM

The design editing system is a useful stand-alone tool in its own right but it is really intended for use within a support package geared towards supporting some particular design method such as JSD (supported in ECLIPSE). Such a package would normally include a name management system to provide access to the objects known to the support system, checking tools for classes of checking which are inappropriate to carry out during design creation (level checking for example) and an integrated editor manager which allowed easy transition from graphical to text to forms editing systems. Some such systems might also include code generators which produced skeleton code from the software design.

There is, therefore, a need for the various support tools to communicate with each other and, in this section, we discuss how this is achieved by the design editor. In fact, there are two classes of communication which must be supported namely information passing while editing is in progress and post-processing of the design information generated by the editor. In our case, both of these are simply achieved because we make use of a single canonical representation for all software designs irrespective of the design method used to create that design.

Run-time communication is supported using a technique which was adapted from that used on the Apple MacIntosh, namely a clipboard. This is mainly used in situations where we have named templates for part of the design held in the database and we wish to include these templates within a design being produced. The user simply uses a database selection tool to find the template required, copies this to the clipboard and then uses the paste facility of the design editor to include that template in the design.

Post-processing is supported because the ECLIPSE database has a fine-grain structure and the schema of structured database items is available to any tool using the database. Thus, the design representation may be

accessed by any other ECLIPSE tool and the design updated or checked. This means that integrated graphics/forms/text editing is possible where different parts of the same item may be operated upon by different editors. Thus, a design method which associates a large amount of textual information with a node on the design may use the design editor to create the nodes and links and then move to a text editor to create the text associated with that node.

CONCLUSIONS

This paper has described the ECLIPSE design editor which is a syntax-driven editor suporting the creation and updating of graphical software design representations. We believe that the system we have produced has significant advantages over other editing systems which are geared to a specific method in that it allows the easy definition of the rules associated with the design method. This has had the useful spin-off in that GDL also provides a technique for formalising and producing a succinct definition of what are sometimes informal and wordily described design methods.

At the time of writing an initial version of the editing system is available which supports all of the method definition facilities and editing functions but which only carries out a limited amount of design checking. Development of the checking facilities is currently underway and the final release of the editing system is scheduled for September 1987.

ACKNOWLEDGEMENTS

The work described here was funded by the Alvey Directorate, UK. Thanks are due to our collaborators in the ECLIPSE project namely Software Sciences Ltd., CAP Industry Ltd., Learmonth and Burchett Management Systems Ltd. and the University College of Wales at Aberystwyth.

REFERENCES

1. Alderson, A., Falla, M.E. and Bott, F. (1985) An Overview of ECLIPSE. In: McDermid, J. (ed.) *Integrated Project Support Environments.* London: Peter Perigrinus.

2. Constantine, L.L. and Yourdon, E. (1979), *Structured Design*, Englewood Cliffs, NJ: Prentice-Hall.

3. Jackson, M. (1983), System Development, Englewood Cliffs, NJ: Prentice-Hall.

4. Sommerville, I., Welland R. and Beer S. (1987) Describing Software Design Methods,*Computer Journal.* **30** [2].

5. Johnson, S.C. and Lesk, M.E. (1978) Language Development Tools, *Bell Systems Technical J.* **57 (6)** Part 2.

6. Reid, P. and Welland, R. (1986) Software Development in View. In: Sommerville, I. (ed) *Software Engineering Environments*, London: Peter Peregrinus.

7. Stephens, M. and Whitehead, K. (1985), The Analyst - a Workstation for Analysis and Design, *Proc. 8th Int. Conf. Software Engineering*, London.