Proc. EUUG
Conference, Dublin
Sept. 1988

DES - Support for the Graphical Design of Software

1987a

Stephen Beer and Ray Welland
Dept. of Computer Science
University of Strathclyde
Glasgow G1 1XH, Scotland
stephen@uk.ac.strath.cs

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

Software design methods such as JSD, MASCOT and Structured Design have been in existence for some time now and most of these methods utilise graphical as well as textual notations for describing designs. These graphical forms quickly convey the overall structure and interconnections of a design more easily than straightforward textual descriptions. However, the full impact of design diagrams employing these graphical notations has been severely restricted in the past due to the lack of automated facilities for production and maintenance of such diagrams. This contrasts markedly with CAD developments in other engineering fields.

This paper describes DES (the Design Editing System) - a system which investigates how such graphical support may be provided in a *generalised* way for software engineering purposes. DES comprises of 3 tools: a shapes editor (SHAPES) for defining the shapes of a method, a language (GDL) for describing a software design method and a graphical design editor (DE) which is driven by tables generated from the first 2 tools. The system is implemented in C on a Sun workstation using the pixrect graphics layer and the panel user interface package.

At a very high level of abstraction a design diagram can be viewed as a number of symbols and a number of rules concerning the physical and logical constraints on those symbols. A novel feature of DES is that it is not geared towards any specific method. Rather, the tool builder defines the syntax, semantics and shapes of a design method using the high level tools SHAPES and GDL. This increases the applicability of the system since MASCOT and JSD users, for example, may utilise the same system facilities.

This paper presents each of the tools, emphasising how method specific checks may be specified in GDL and enforced during design editing sessions.

1 Introduction

A major aim of software engineering is concerned with bringing methodical practices into the various phases of the software life cycle. At the present moment there are a very wide variety of methods within the phase of software design. Examples of such methods include JSD [Jackson83], Structured Design [Constantine79], Petri-Nets [Peterson81] and MASCOT [MASCOT80]. Many of these design methods have associated graphical techniques to complement or replace textual design descriptions. Structured design, for example, provides for two diagram types: structure charts and dataflow diagrams.

These graphical techniques have been in use for some time now but are not as widespread as one initially might expect. One reason often put forward is that a design diagram lacks formality and cannot capture the same amount of detail as a straightforward textual description. This is true in many cases, but their major use is in conveying overall system structure and interconnections in a more easily digested form. The absence of *automation* from the design process, we believe, is a key reason as to why these graphical techniques are not in common use.

To date, support for the creation and maintenance of design diagrams in other fields of engineering has been extensive. Electronics has seen systems supporting the design of logic circuits which can then generate pin connections. CAD in the area of mechanical engineering allows metal components to be designed graphically; the design information is then passed to a cutting machine which automatically cuts the part to specification. Support for graphical techniques within the area of software engineering, however, has been acutely scarce.

One reason for this is the sheer number of methods that already exist and the number of methods likely to be developed in the future. Many reasons can be stated for the existence of each, among these being the area of application. For example, JSD is more suited for the area of data processing tasks whilst MASCOT is aimed particularly at real time embedded applications.

The work described in this paper has been taking place within the context of ECLIPSE [Alderson85] which aims to provide an integrated project support environment for different approaches to the software process and tools to assist in software design. A requirement of ECLIPSE is that many different graphical techniques should be supported and that designs created can then be captured in the project database and manipulated by other tools. A common user interface across design support tools is another requirement.

One approach to providing design support is to build a specific design editor for each method from scratch. However, given that we have a situation where many different methods *do* exist and more *will* be devised it seems much more sensible to abstract the general concepts of design methods into a design editing system which has the capability of being tailored for a specific method.

Our work here at Strathclyde has been concerned with just that. We have been experimenting and building a design editing system which can be tailored at a very high level to cater for many different methods. This paper will describe our system and emphasise how a design method can be described more formally.

The next section in the paper presents an overview of our system and compares some related work in the area of describing software design methods. This is followed by brief descriptions of each of the three constituent tools. The task of checking a design diagram finally concludes the paper.

2 A Design Editing System

A large number of graphical design techniques can be characterised as exhibiting a graph-like structure composed of design objects. Each design object can be classified into one of the following:

node	denotes a software component, state, etc.
link	denotes flow of control or data, etc.
label	denotes the textual and graphical annotations of a design

These techniques whilst sharing the common properties of graphs tend to differ with respect to:

- the actual symbols used to represent design objects
- rules concerning how a design diagram may be constructed

Some of these rules refer to syntactic constraints such as ensuring that the name label of a node appears completely within the perimeter of the nodes' symbol. Other rules refer to the semantic properties of a design - for example, in designing a dataflow diagram it is generally considered "good" design to have no more than 8-10 nodes on any one diagram.

The DES approach to providing design support for different methods is to provide tailoring tools to describe the essential features of a method and a generic design editor which is configured by this method specific information. This is achieved by providing the following three distinct tools (depicted in Figure 1):

SHAPES	a graphical shapes editor allowing the representation of each design object to be defined.
GDL	a language for specifying the "grammar" of a method.
DE	a generic design editor configured by tables from the previous tools

In this type of system there are two types of user. The **method administrator** is the person responsible for *setting up support* for a design method. The **end-user** is a person who uses a design editor to *create design diagrams*. The tasks performed by the method administrator in tailoring an editor for a specific method are:

- Describe the types of design objects (node, links and labels) by writing a GDL description
- Define how each object is to be represented using the SHAPES editor
- Tables created using the GDL compiler and SHAPES are then used as input to the generic design editor, thus configuring it for a specific method.

End-users can now create design diagrams using the tailored DE. Diagrams may be stored and retrieved at any time thus ensuring that the task of diagram maintenance is made easier.

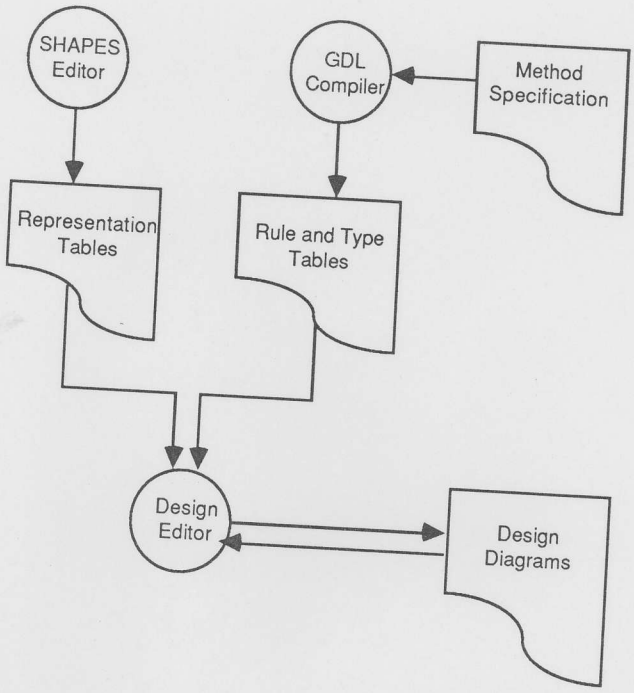


Figure 1 - The Design Editing System

The tailoring technique has been used successfully in many other areas where general properties are abstracted and specific differences described. A particularly useful and commonly known analogy is available from *compiler-compiler* systems. Such systems are used to automatically generate compilers for languages specified with high-level tailoring tools. In this area the observation has been made that compilers in general possess the same common stages of scanning, lexical analysis, semantic analysis, etc.. What they actually differ in are:

- the lexical tokens of the language (eg. keywords, identifiers, operators, etc.)
- rules referring to how a syntactically legal program may be constructed from the available lexical tokens

The **lex** [Lesk75] and **yacc** [Johnson75] tools of UNIX provide a means of defining such language specific features. The lex tool takes a specification of the legal tokens of a language and generates C code to perform the scanning and lexical analysis phase of compilation. The yacc tool takes a specification, describing the grammatical rules for constructing a legal program, and generates code to accomplish the remaining stages of compilation. The lex and yacc generated code can then be linked, together with user-written code, to produce a working compiler for the specified language.

The DES approach in providing method support is similar. The major difference is that the output from the tailoring tools are *tables* encapsulating method information rather than C code. These tables are then used to "drive" a generic design editor.

Related work in providing graphical support for design methods shows slightly different approaches. The work of [Woodman86] is inspired from the development of picture grammars in the field of pattern recognition. The principle idea is that specific patterns can be described using a grammar and the task of recognising a new scene corresponds with trying to parse it according to known pattern definitions. Rather than return a value of "parse failed" or "syntax error", a weighting is returned and used in deciding how close a scene fits a known pattern. Their application to software engineering diagrams is similar. Essentially it consists of specifying what a dataflow diagram, for example, should look like. The grammar approach in DES is similar except that the definition of the methods' lexical tokens is separated from the specification of the method rules. In their work the symbol definition is an integral part of the grammar.

From Figure 1 it can be seen that the DE is essentially syntax driven - the syntax of the method "drives" the editor. Another grammar based approach in defining a method is contained in the SEGRAS-Lab [Kramer86]. This system provides graphical support for Petri nets within a syntax directed editing environment. Their grammar normally used for generating textual syntax directed editors has been extended to allow context-sensitive constraints to be specified. The end product here is a syntax-directed graphical editor. Our approach differs here in that actual diagram construction by an end-user is accomplished with a non-restrictive interface.

3 Defining Method Symbols - SHAPES

The purpose of the SHAPES editor is to allow a method administrator to define the representation (symbols) of each design object of a design method. Once created, the symbols can be stored in method specific libraries which are used as input to the generic design editor. The SHAPES user interface (Figure 2) consists of:

- a control panel for selecting commands
- a **shapelist** for storing/selecting shapes
- a scrollbar for scrolling through the shapelist
- a drawing area for constructing new shapes (symbols)

Most interaction is via a mouse device except for the task of naming a symbol prior to storing it on the shapelist. The shapelist initially consists of a number of **primitive** shapes. This list can be extended by adding new user-defined symbols. The primitive shapes provided are text, ellipse, rectangle (right-angled and round cornered), triangle, diamond and line with circle and square being special cases of ellipse and rectangle. Whilst it is not possible to produce every conceivable shape of every method from a combination of these primitives it is possible to generate a very large percentage of them.

To define a new shape the method administrator first selects a primitive shape from the shapelist using the mouse. This shape is then instantiated in the drawing area by defining its enclosing boundary, again with the mouse. Any number of shapes can then be added and all, or any subset, of these can be moved, stretched or deleted. Once created, a symbol can be entered into the shapelist along with its name.

At any time during a SHAPES session the user may store the current shapelist in a shape library for later use as input to the DE.Therefore a shapes library consists of a number of user-defined shapes each identified by a name and described in terms of basic, primitive geometric shapes.

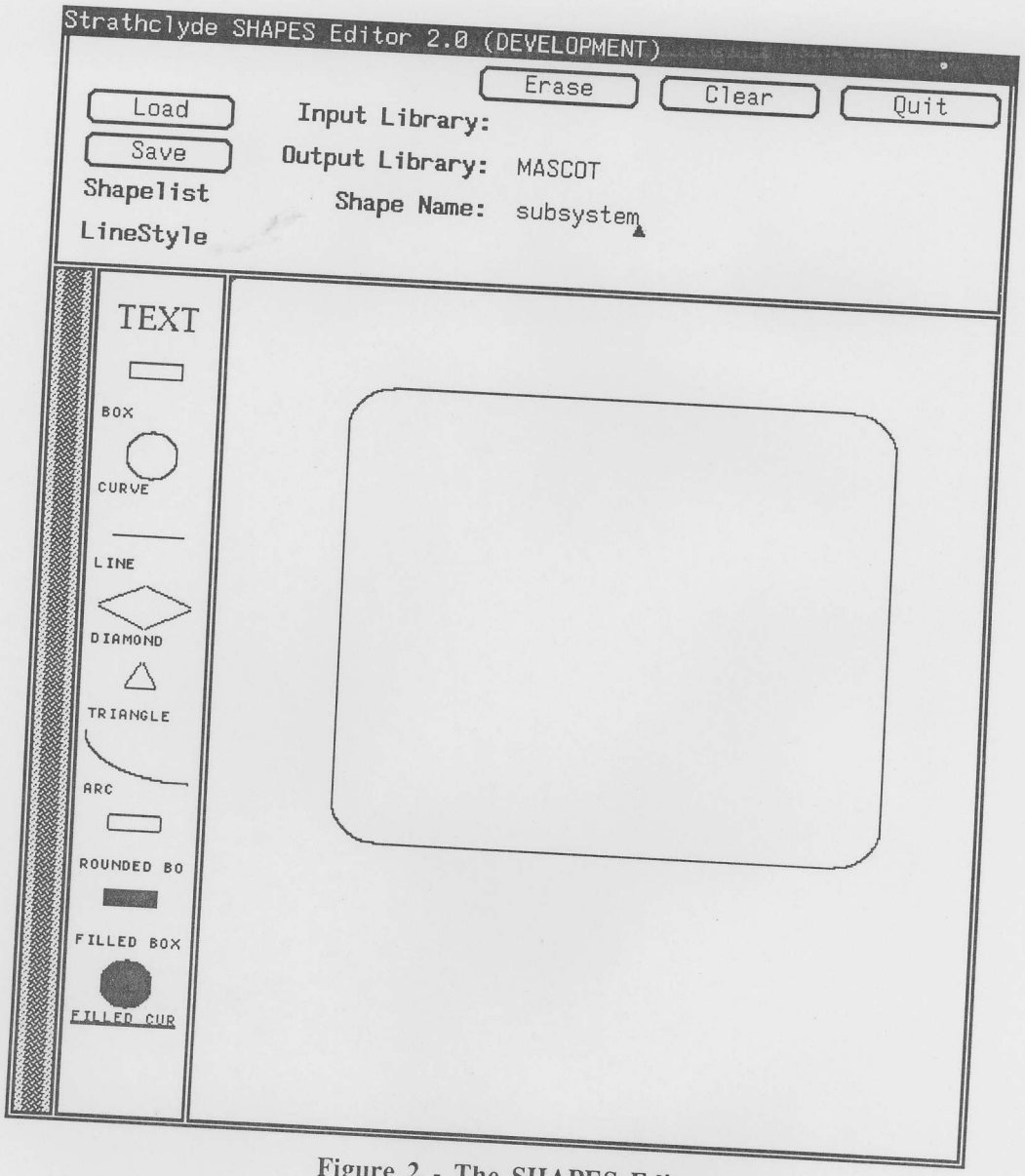
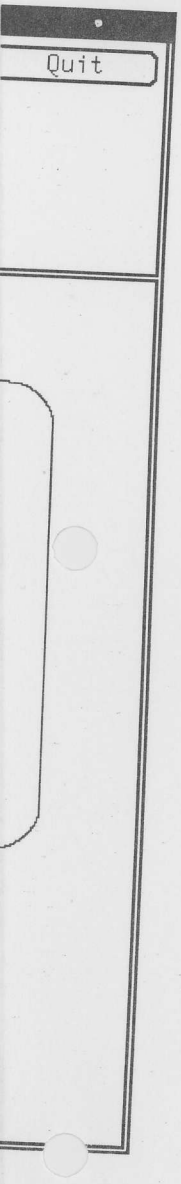


Figure 2 - The SHAPES Editor

4 Describing a Method - GDL

The notation we have developed for describing a software design method is the Graph Description Language. A method administrator describes a method by writing a GDL



description which is then transformed into tables using the GDL compiler. The compiler has been constructed using lex and yacc and also employs the facilities of cpp - the C preprocessor.

The best way of describing the GDL is by example and the following fragment describes some of the design objects of the MASCOT design method used in an editing session shown in Figure 3.

```
type PATH           is LINK ( src : NODE; dst : NODE )

type JUNCTION       is NODE ( parent : owned by  SUBSYSTEM;
                              in_path : in set of PATH;
                              out_path : out set of PATH )

type PORT           is JUNCTION
type WINDOW         is JUNCTION

type SUBSYSTEM      is NODE ( junctions : owner of set of JUNCTION )

for PORT            use SYMBOL ( MASCOT.port )
                    ++ ACCESS_INTERFACE ( STRING )
                    ++ JUNCTION_NAME ( STRING )

for  SUBSYSTEM      use SYMBOL ( MASCOT.subsystem )
                    ++ TEMPLATE_NAME ( STRING )
                    ++ COMPONENT_NAME ( STRING )

assertion Junc_name_enclosed ( SUBSYSTEM ):
                    inst i:
                    forall j; Member ( Dependents ( i ), j ) and GetType ( j ) = JUNCTION:
                    Encloses ( GetLabel ( i, SYMBOL ), GetLabel ( j, JUNCTION_NAME ) )
```

The type JUNCTION is in fact a place holder in the type hierarchy so as to avoid unnecessary repetition.

Types

In describing a design method the main task is to assign some type to each design object of the method (ie. node, link or label). The base types of the language are **node** and **link** (ie. the basic constructs of any graph) and a hierarchy of types is formed from these. In the example shown the type PATH is introduced stating that any instance of a PATH should have parameters *src* and *dst* of type NODE. The type SUBSYSTEM is based on NODE and has only dependent (or child) nodes. A child node is always associated with some parent node and any operations affecting the parent also affects its children. In our example a child node of a SUBSYSTEM node is of type JUNCTION which in effect is a PORT or a WINDOW. Having introduced the node and link types attention is now turned on how label types are defined.

Labels

The *for-use* declaration is the construct for specifying the label types to be associated with a node or link type. In the example, a PORT type has one symbolic (iconic) label and two textual labels.The reserved word SYMBOL indicates that this label is the one used to represent an instance of the PORT type on a diagram. The name *MASCOT.port* refers to a symbol named *port* stored within a shape library named *MASCOT*.

Assertions

Graph
GDL

The assertion construct is used to define the syntactic and semantic constraints of a design method. In the example, we have chosen to describe the constraint that the JUNCTION_NAME label of every child JUNCTION node of a SUBSYSTEM must be completely enclosed within the SUBSYSTEM boundary. This is a syntactic constraint which ensures that in Figure 3, for instance, the labels "access1" and "access2" both appear within the "Processing Subsystem" symbol boundary. Semantic constraints could refer to the number of JUNCTION's present within a SUBSYSTEM, so as to reduce design complexity, for example. Further details of the GDL and its capabilities can be found in [Beer87 and Sommerville87].

5 The Generic Design Editor (DE)

The envisaged user of a DE tailored for a certain method is a software designer having knowledge of the design method. Rather than interacting via simple graphical drafting terminology (eg. box, circle) the DE uses the same terminology as the designer. A designer selects a TEMPLATE or a PROCESS, for example, and adds it to a design diagram as opposed to selecting a box or a line. Hence the designer interacts in terms of the **design objects** of a method.

The major facilities of the DE allow the user to:-

- add, move or delete images representing design objects
- create a design diagram larger than the designers' workstation window,
- view the total diagram at a reduced size
- annotate the diagram only, not the underlying design, with text, boxes or lines.
- utilise a grid facility for aligning objects

The object oriented approach to user interaction has been pursued throughout the development of the DE. The functions available to the designer are applied to a currently selected object from the design. The current selection may consist of a node, link, label or combination of. Functions are applied consistently across all objects wherever it is sensible to do so. It is nonsense to edit the graphical symbol of a node but sensible to textually 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.

The DE has been implemented on a SUN workstation running under the Suntools window environment. The user interface (see Figure 3) consists of a tool window subdivided into the following four subwindows:

- drafting area where a design is constructed,
- a control panel giving access to the editing functions and object types,
- two subwindows containing scroll bars. These allow the the drafting area to be moved around the total area of the diagram.

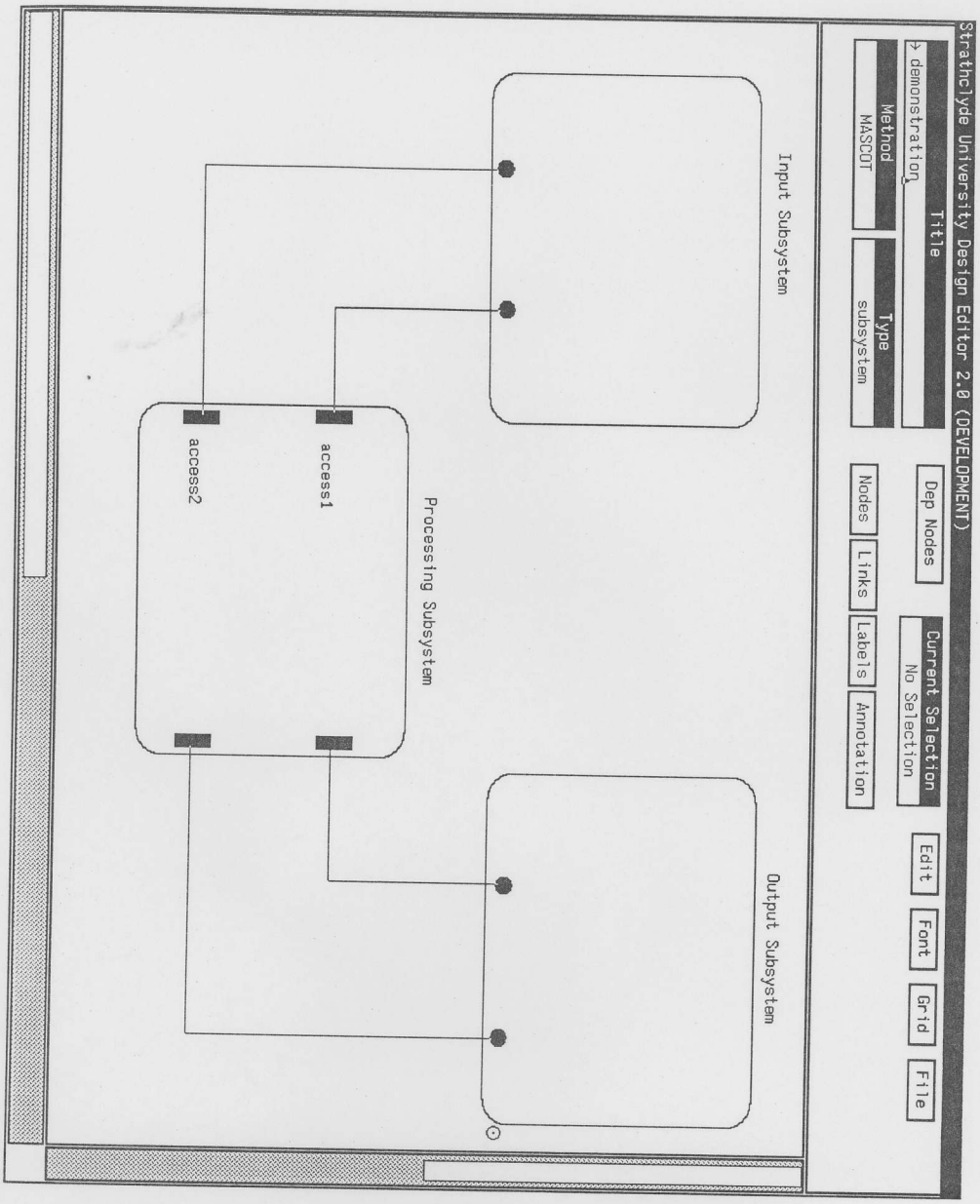


Figure 3. The DE User Interface

Editing functions are selected from a control panel [Reid86] containing pull down menus and "soft" buttons. The designer selects an node, link or label 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.

As mentioned previously, the DE has built-in knowledge that the design must be in the form of a graph. The one restriction implicitly enforced by the DE is that a link must originate from and end at a node. This prevents a design from being created where dataflow links, for example, lead to or originate from nowhere.

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 selected as the current object then its links and labels are automatically selected as well. 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 reference a non-existent node. In the same way a move operation automatically moves all links and labels associated with a node.

6 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 neat looking diagram is of no use, and indeed may be harmful, if it is incorrect. Design checking therefore is a major function of the DE and is one aspect in which it differs from a straightforward drafting tool.

Other design support tools have demonstrated the need for method specific checking of a design [Jones86, Stephens85] but a 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 :-

- "connectedness"
- layout and semantic constraints
- completeness checking

In the case of a node the strong 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.

The method rules to be enforced in the DE are called **assertions**. These are compiled by the GDL compiler into rule tables which drive the DE. 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 object 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 object to correct the design.

Type checking can be enforced mainly through assertions if the link parameters are specified as the generic type NODE and appropriate assertions on connections are written. Alternatively, a strongly typed description removes the need for such assertions but increases the number of checks each time a link is added to the design. This GDL trade-off effectively means striking a balance between continuous checking (closer to syntax-directed editing) and user-initiated checking.

In a GDL representation expression a label can be specified as being either compulsory or optional. This information on the optionality 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.

In interacting with the DE the end-user has freedom to construct a design diagram in any appropriate manner. This is in contrast to other editing systems which provide a syntax-directed user interface [Kramer86]. Design checking occurs in a non-obstructive manner. If an assertion is violated then an appropriate warning message is output and the offending object highlighted. The end-user can then choose to fix or ignore this error rather than being forced into a fixing it before proceeding with the design.

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:

1. **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.

2. **immediate** As soon as an editing operation alters a design certain assertions will be executed immediately. For instance, these would include checking that a link has source and destination nodes of the correct type.

3. **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 freedom to develop a design in whatever manner is favoured. Checking can be called at the designers' convenience and may include assertions concerning completeness and consistency, assertions about spatial arrangement of objects and completeness of label sets.

7 Conclusions

Presented in this paper has been a description of some applicative research into providing automated support for graphical diagrams within software design methods. The system can be tailored at a very high level to support different methods. So far, descriptions for methods such as JSD, MASCOT, state transition diagrams and dataflow diagrams have been defined.

The novel aspect of this work is the ability to be able to specify and enforce method specific checking within a **generic** design environment. The true worth of a production-level version of this work would be obtained within an integrated project support environment requiring many different design methods to be supported.

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.

Personal thanks are also due to Alistair Blair, Stevie Keith and Jim Reid for providing good, constructive criticism of this paper.

References

[Alderson85]
Alderson A., Falla M. E., Bott F., *"An Overview of ECLIPSE"*, Integrated Project Support Environments, J. McDermid (ed) Peter Perigrinus London (1985)

[Beer87]
Beer Stephen, Welland Ray, Sommerville Ian, *"Software Design Automation in an IPSE"*, To appear in Proc. of 1st European Software Engineering Conference, Strasbourg, France (Sep 1987)

[Constantine79]
Constantine L. L., Yourdon E., *"Structured Design"*, Prentice-Hall (1979) Englewood Cliffs, NJ

[Jackson83]
Jackson Michael, *"System Development"*, Prentice-Hall(1983)

[Johnson75]
Johnson S. C., *"Yacc: Yet Another Compiler Compiler"*, Computing Science Technical Report, No. 32 Bell Laboratories Murray Hill, New Jersey (1975)

[Jones86]
Jones John, *"MacCadd, An Enabling Software Method Support Tool"*, Proc of 2nd Conference of British Computer Society Human Computer interaction Specialist Group, Harrison (ed) pp. 132-154 Cambridge University Press (23-26 Sep 1986)

[Kramer86]
Kramer Bernd, *"Interactive Graphical Specification Using he Syntax-Directed SEGRAS* Nineteenth Annual Hawaii International Conference on System Sciences, Bruce D. Shriver (ed) pp. 420-429 (1986)

[Lesk75]
Lesk M. E., *"Lex - A Lexical Analyser Generator"*, Computing Science Technical Report, No. 39 Bell Laboratories Murray Hill, New Jersey (Oct 1975)

[MASCOT80]
MASCOT, *"The Official Handbook of Mascot"*, Mascot Suppliers Association Malvern, UK (1980)

[Peterson81]
Peterson James L., *"Petri Net Theory and the Modeling of Systems"*, Prentice-Hall (1981)

[Reid86]
Reid P., Welland R., *"Software Development in View"*, Software Engineering Environments, Ian Sommerville (ed) Peter Peringrinus London (1986)

[Sommerville87]
Sommerville Ian, Welland Ray, Beer Stephen, *"Describing Software Design Methodologies"*, The Computer Journal, Vol. 30 No. 2 pp. 128-133 (1987)

[Stephens85]
Stephens M, Whitehead K, *"The Analyst - A Workstation for Analysis and Design"*, Proc of 8th International Conference on Software Engineering, IEEE Computer Society Press London (28-30 August, 1985)

[Woodman86]
Woodman M, Ince D, Preece J, Davies G, *"A Grammar Formalism as a Basis for the Syntax-Directed Editing of Graphical Notations"*, Open Univeristy Technical Report 86/19, (1986)