Transactions on Information and Communications Technologies vol 9, © 1994 WIT Press, www.witpress.com, ISSN 1743-3517

A prototype program design workbench

A. Stavrinides & K. Monk
School of Mathematical and Information Sciences, Coventry University, Priory Street, Coventry, CV1 5FB, UK

ABSTRACT

This paper describes current work in defining and developing a CASE tool to support program design. This work is proceeding alongside a parallel effort to produce a comprehensive set of separate but cooperating tools for use in teaching programming to undergraduate students, based on the use of a formally defined program design language, provisionally called MPC (Modular PseudoCode). The CASE tool, which we call the Program Design Workbench (PDW), is intended to help integrate the use of MPC tools through a convenient windowed user interface, but much more importantly it will embody and enforce a disciplined and methodical approach to modular program design.

In addition, the PDW will eventually manage a collection of design databases and allow controlled access to and navigation between design objects. The prototype is being developed using the IPSYS ToolBuilder meta-CASE tool. The target methodology is "captured" in terms of a data model and a frames model. The former defines the structure, attributes and possible inter-relationships of design objects, and is used to generate the PDW design database; while the latter defines the available operations and allowable context transitions in the generated graphical user interface. In this paper we explain both aspects and highlight the continuing development work of the associated methodology.

INTRODUCTION - CASE

Computer Aided Software Engineering (CASE) is concerned with the use of computer-based tools to support the entire spectrum of software development activity. Thus there are, for example, tools supporting particular systems analysis methods, and others supporting particular software design methods. The term CASE is quite justifiably applied to

computer support for software project management and program development activities, although most people would perhaps be reluctant to refer to a simple text editor as a CASE tool. On the other hand, an integrated set of program development tools, where the integration is achieved either through support for a specific programming language (C, Ada, etc), or through close cooperation between the tools, would clearly qualify as CASE.

CASE tools which span several stages of software development activity are referred to as Integrated Project Support Environments (IPSEs). An early example of such a tool is EPOS [1], which has support for at least two stages, namely requirements engineering and software design, plus comprehensive project management facilities. The three main parts of the EPOS project database are linked by cross-reference relations. In the area of software design EPOS offers the ability to adopt different views (eg data-flow oriented) in order to produce specifications which follow particular methodologies. However, this falls well short of offering direct support for any one methodology.

A very different approach is taken in CASE tools which are capable of supporting more than one methodology, by fitting the appropriate supporting tool into a sophisticated and flexible integration framework. An example of such a "toolkit" is Software Through Pictures [2]. An important aspect, common to the IPSE and toolkit approaches, is the possibility of linking the products of different stages (eg requirement specifications, design specifications and source code), so that developers can keep track of the inter-relationships. This is particularly useful when requirements change during development or in later maintenance.

There are, of course, stand-alone CASE tools supporting software design according to specific methodologies. Some of these tools, including for example the IPSYS toolset for HOOD [3], follow the methodology along its full scope from software requirements analysis through architectural design to detailed design and possibly the generation of source code. Again, there are obvious benefits to be obtained from the use of design databases where individual software objects are documented thoroughly, packaging together both informal descriptions and formal specifications as well as their relationships to other objects. Reusability is perhaps the greatest of these benefits.

THE PROBLEM

CASE technology has now reached a reasonable level of sophistication, but there remain areas of deficiency which are probably due more to current attitudes than to technological or other factors. One deficiency lies in supporting the transition from design to implementation. Even where software design tools are able to generate source code, it remains the programmer's responsibility to implement software by compiling

and testing individual units, then progressively integrating and testing the complete software system. Although CASE tools are available to support such programming activities, they generally offer nothing more than that.

However, testing is meaningful only if its results, especially when negative, can be fed back to the design stages. It would clearly be foolish to allow code fixes to invalidate design specifications, *yet this is what happens quite often in industry*. Simply documenting the code fixes and/or reverse-engineering the relevant design specifications is a clear sign of a breakdown in the integration of design and programming environments.

One possible reason for the breakdown is a distinction between the "design" and "programming" activities, perceived as involving different concepts, different methods, different skills and different people. It is interesting that "systems analysis and design" is often seen to be an integrated activity, but "software design and implementation" is more often taken to refer to *two* activities. The problem lies with the word "design", which is applicable (and is therefore applied) to at least three levels of software engineering activity:

- system design - aiming to specify overall system structure
- architectural design - aiming to specify software structure
- detailed design - aiming to specify procedural structure

If "programming" is perceived as an activity distinct from even the lowest level of design, the inevitable result is to employ programmers who are trained in language-specific coding skills. We have all seen advertisements for "Software Engineer with two years' experience of C", and academics in particular will be aware of the demand by students, eager to enhance their employment prospects, to be taught popular programming languages.

At Coventry University, as elsewhere, the view has been taken that programming should always be taught in the context of at least the lower levels of software design, covering modularity as well as procedural aspects. For the purposes of this paper we call it *Program Design*. It is our contention that programmers should be skilled *program designers*, not *coders*. Further, we believe that programmers should have available to them, and be trained in the use of, appropriate program design tools.

Unfortunately, the distinction between design and programming is reflected in a scarcity of CASE tools combining these "two" activities. In the educational context there have been some attempts to create suitable support environments for the teaching of modular program design (we have done a brief survey [4]). A more ambitious project is the TIPSE [5] development at the University of Wales, Aberystwyth, which aims to provide teaching facilities across the spectrum of software

engineering. We note it here primarily for its use of the IPSYS Tool Builder's Kit to generate both the user interface and the underlying database, these being the primary integration mechanisms.

MPC AND ITS TOOLS

Our own attempt at Coventry University began with the definition of a program design language, MPC (Modular PseudoCode). This is described in an internal publication [6], available from the author. MPC is in some respects similar to Modula-2, but has additional features specifically intended to support and enhance the design process. The basic idea is to formalise the top-down "stepwise refinement" approach to program design, covering specifications of the modular structure as well as data structures and detailed processing logic.

Since Modula-2 itself (in common with Ada and other languages) is quite capable of capturing such specifications at source code level, it may be asked what the "additional features" are. Briefly, they include the ability to document the stepwise refinement process, and the ability to associate preconditions and postconditions with procedures, and place assertions elsewhere in the source. But what makes MPC a better vehicle for teaching program design is its *deliberate omission* of those "features" found in Modula-2 and most real languages which tend to distract and detract from the design process. For example:

- no semicolons - at most one statement per line
- no anonymous data types - must be named explicitly
- no nested procedure declarations - simpler visibility

Perhaps the most controversial feature (or omission) of MPC is the lack of separate "definition" modules. These are used in Modula-2 to enable the compiler to check external references, by consulting the definition module named in an IMPORT statement. In MPC, one module cannot specify the name of another from which items are to be imported. This elevates the task of resolving external references to a higher-level tool, namely the Interface Checker, which is aware of all "available" modules and the items each module exports. Thus name-space management is implicit in the MPC toolset, rather than explicit at the source language level.

This brings us to the point that MPC is not just a design notation, it is a *design environment* containing a number of supporting tools. The full toolset is envisaged to include the following:

- text editor
- syntax checker
- interface checker
- chart generator
- static analyser
- interpreter
- assertion processor
- animator
- profiler
- debugger
- test planner
- test harness

Thus the toolset supports design entry, design documentation, design validation and design execution. In addition, there will be one or more translators to enable designs to be converted to source code in "real" programming languages.

Not all of these tools need to be available before MPC and its concepts can be used for undergraduate teaching; after all, many courses use no more than informal "structured English" pseudocode. However, the first step towards a useful toolset is probably the Syntax Checker, whose role will be central in protecting other tools from syntactically invalid MPC sources. From the students' point of view, the use of a syntax-cognisant Text Editor will simplify design entry, while a Chart Generator will automate the production of graphical documentation to complement the textual design specifications.

At Coventry University we have tackled the development of the MPC toolset as final year computer science projects, with some success. We have a usable Text Editor, a good Syntax Checker, a limited MPC-to-C Translator, part of the Interface Checker, and a Chart Generator for an earlier version of MPC. These are Unix-based stand-alone tools, whose usability would clearly be enhanced by providing at least a menu-based integrated front end running on ordinary terminals. However, we have also been looking at the requirements of a more "professional quality" system, running on workstations with bit-mapped displays. This would not be just a cosmetic variant of the terminal-based system, but would in fact be fundamentally different in structure and operation.

GENERATING THE PDW

It is primarily the workstation-based system, provisionally called the Program Design Workbench (PDW), that we would like to describe in more detail. This has been conceived as an integrated environment, a true CASE tool, covering program design in all its aspects. We are using the IPSYS ToolBuilder [7] meta-CASE tool to construct a prototype system, aiming to test the concept and obtain user feedback in an "incremental" or "evolutionary" development process. Prior to the PDW effort, another project at Coventry had successfully used the ToolBuilder to generate a CASE tool to handle EXPRESS textual and graphical specifications [8]. The PDW prototype development was begun as a final year student project, but brought to a reasonable level of usability by a graduating student from the University of San Sebastian, Spain, on an exchange visit to Coventry [9].

The IPSYS ToolBuilder is a development of the Tool Builder's Kit, which was mentioned earlier. Its central component, called METHS, allows the specification of database entities and associated operations (the *data model*), and a set of "frames" or context-points and navigations between them (the *frame model*). The data and frame models are used to generate the target tool's underlying database and its user interface.

Another component, PUBLISHER, allows the specification of the target tool's reports - textual or graphical output documents. There is also a third component, called ERA, which can be used to produce a graphical representation of the entity model (as an entity-relationship diagram) before entering the textual specifications into METHS. In the current version of the ToolBuilder, ERA is a stand-alone tool.

The Data Model

Although the full PDW will require a complex and sophisticated data model to capture not only design specifications but also the associated management information (author, version, status, etc), for the prototype system we decided to model only design structures. A rough entity-relationship diagram was produced on paper and with ERA, showing the main entity types and main composition and reference relationships. The top entity is a Design, which consists (is composed) of a number of DesignObjects. Each DesignObject can be a Module or an ExecUnit, ie the latter two are sub-types of the DesignObject entity type. Graphically we represent composition relationships by using solid-headed arrows, and sub-types by nesting the entity boxes. Reference relationships, connecting entities which reference each other in some way (eg one ExecUnit can call another), are represented by open-headed arrows. Single and double arrow heads indicate one-to-many and many-to-many relationships as appropriate. In Figure 1 we show a slightly simplified version of the entity model, with unlabelled arrows to aid clarity.

The Frame Model

A frame is essentially the context of the user's interaction with the tool, in terms of the current display and available operations, some of which may navigate to other frames. The frame model is used to capture the *methodology* that the target tool will support. For example, in a top-down design methodology, the tool user needs to be able to navigate between the various levels of refinement. If different operations are available at different levels, they are likely to require separate frames.

The PDW is primarily intended to support a top-down approach, with reusability of designs encouraged through the use of "design libraries". The full system will allow for this by distinguishing between *bound* and *unbound* external references (imported items), as well as allowing new variants of existing design objects to be created. An unbound reference will be resolved (at build time) to the latest version of the referenced item. For the prototype system, these facilities have been left out of the frame model. Figure 2 shows the overall frame structure, considerably simplified, while Figure 3 shows in more detail two of the frames together with their operations and resulting navigations. The R or R/W arrow labels indicate the read or read/write mode enabled by the frame change.

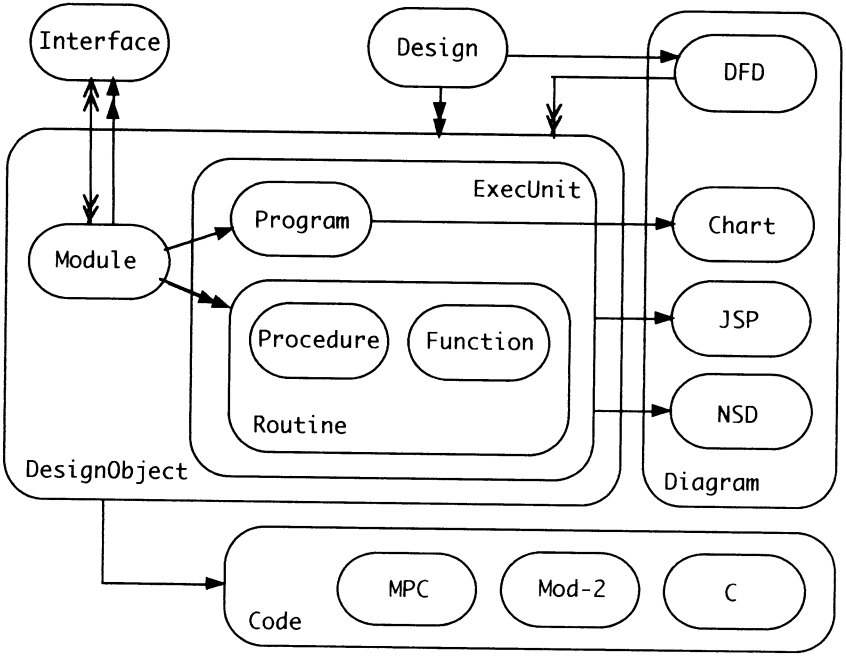


Figure 1: PDW data model. Not all entities or all relationships are shown.

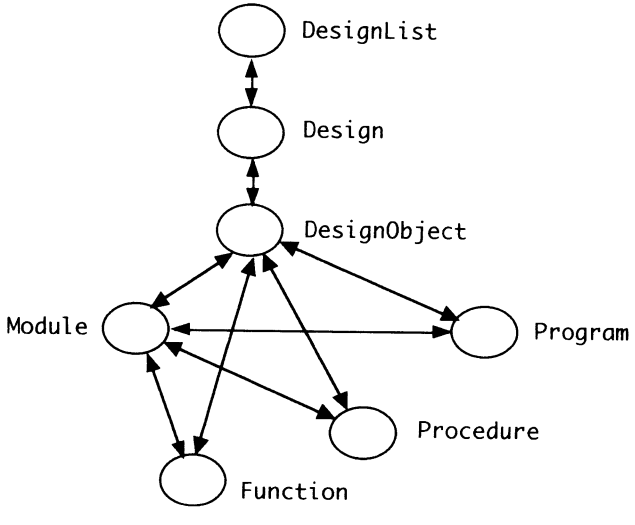


Figure 2: PDW frame model. The links shown are the main bidirectional navigations between frames. Additional links (not shown) navigate from the lower five frames back to the top two.

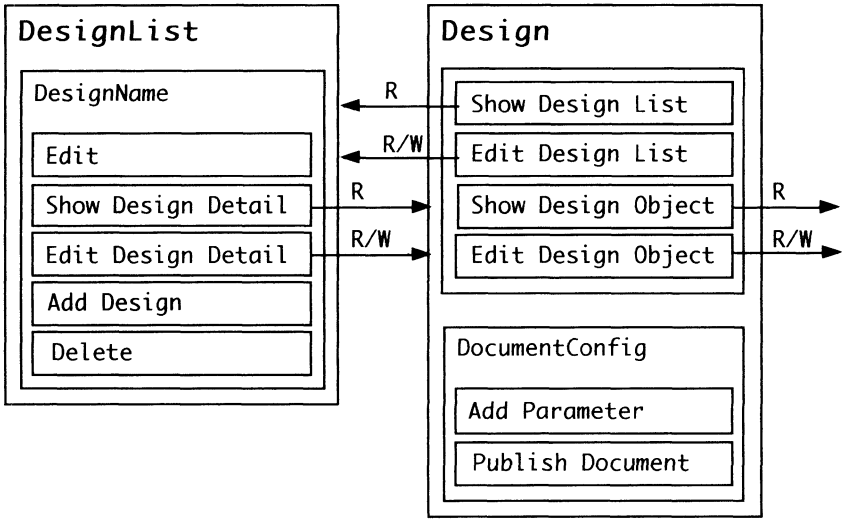


Figure 3: Navigations between the DesignList and Design frames. Operations shown without arrows navigate to the current frame. The unlabelled operation set in the Design frame is invoked from the <<Actions>> menu.

THE PDW USER INTERFACE

The ToolBuilder has an extensive range of defaults which, if not over-ridden, result in the generation of target tools with a uniform "look and feel" compared to other generated tools. The IPSYS HOOD tool, which we also have at Coventry, was produced using the Tool Builder's Kit and therefore has a similar user interface, and the same applies to the METHS, PUBLISHER and ERA components of the ToolBuilder. For the purposes of the PDW prototyping exercise we therefore chose to follow a path of "minimum resistance" to the preset defaults, in order to produce a working system with the minimum effort.

The default user interface of the generated tool has two windows, used for textual and graphical operations. For the prototype PDW, we have concentrated on textual operations, but future development will look at ways to allow graphical input as well as output. The text window includes labelled buttons which offer global options (Print, Save, etc), and an <<Actions>> button giving general navigation control. The rest of the window contains textual specifications depending on the current frame. At the top level, it may simply be a list of design objects whose specifications are held in the design database and which can therefore be selected for display or editing. At lower levels, further detail is shown from particular design specifications. By selecting an appropriate area of the visible text the user can obtain further applicable actions. All selections are achieved by using the workstation's mouse. Figure 4

shows an example of the text window in use (note that this figure was produced as a separate drawing working from a rather less legible screen dump).

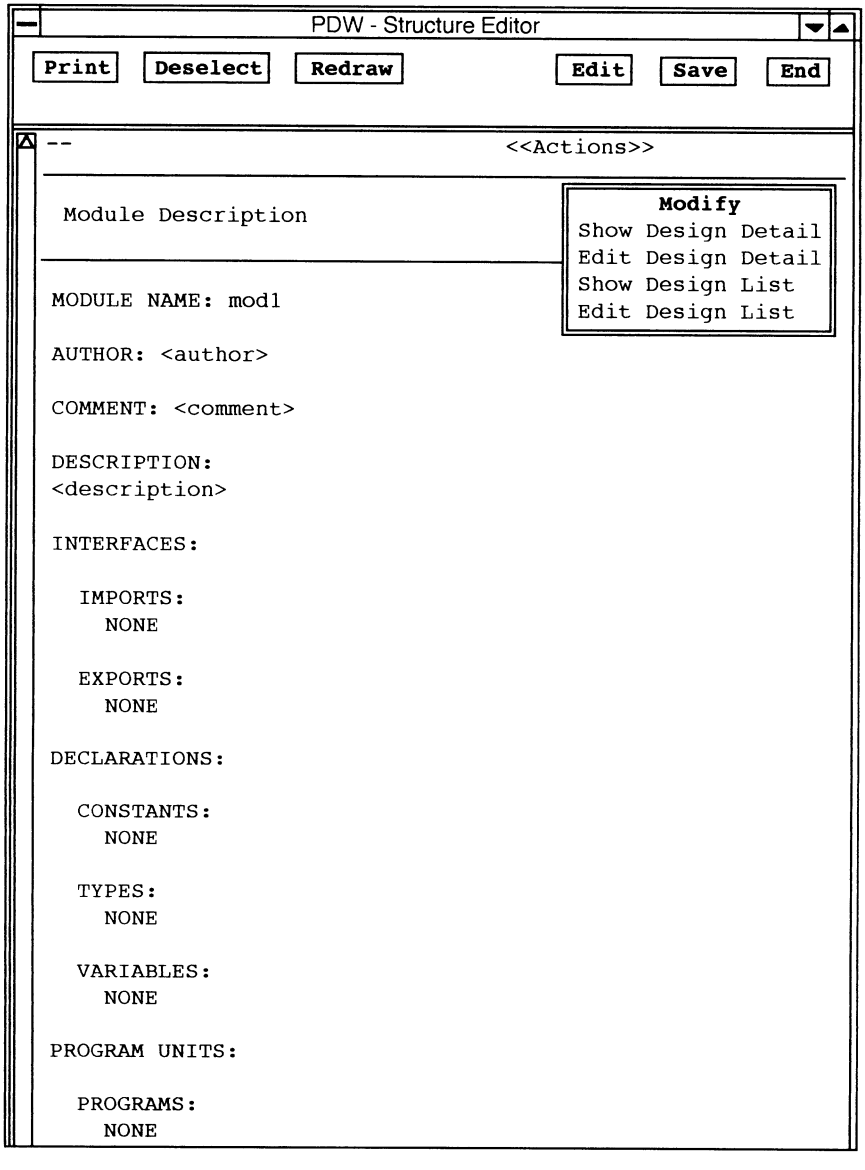


Figure 4: PDW Structure Text window. A new module has been created and is about to be edited. The inner window is the menu called from <<Actions>>.

The text window displays a specification at a certain level of detail. The user can edit the detail by selecting a particular area, obtaining the available menu, selecting Edit, and entering the required text into an edit window. The system can check that the text entered complies with the syntax for that part of the specification. In the simplest case, if only an identifier is expected then the check is against a regular expression, but in general the validation may need to be quite complex. Rather than build a full parser into the system, the idea is to narrow down the range of valid inputs, for example by offering the user a selection of statement type from a menu. A template for that statement type can be placed in the text window, with editable portions enclosed in angle brackets <>. The user next selects and edits each portion separately, with the system checking against specific rules.

Essentially we have here a *structure editor*, a means of ensuring that specifications are valid as they are being entered. This is not quite the same as a language-oriented editor, because the specifications entered are not in MPC syntax, except perhaps at the lowest levels. For example, at module level (as shown in Figure 4), the user can enter text against editable portions, but that will not make the display look like an MPC module source. The MPC source is generated by a separate operation, and indeed it should eventually be possible to generate source code in other languages.

FUTURE WORK

The reader may have noticed that the PDW data model includes entities concerned with other forms of design representation. These are:

- Chart - control hierarchy diagram (program structure chart)
- DFDiagram - data flow diagram (refinable to several levels)
- NSDiagram - Nassi-Schneiderman diagram (structogram)
- JSPDiagram - Jackson Structured Programming diagram
- Code - generated source code in a programming language

The frames and navigations necessary to support these entities have yet to be added. For the graphical entities, the graphics window will be used to display automatically generated diagrams (which may or may not be editable), or to create diagrams using a graphical tool (eg a DFD editor).

A previous section outlined the range of MPC-specific tools which will be included in a terminal-based system. We envisage that the same facilities will be present in the full PDW system, but not necessarily in the same form. For example, the PDW Structure Editor, used for design entry, effectively combines the two functions of Text Editor and Syntax Checker. Interface checking should be implicit in the validation which must be performed whenever a new or modified design object is to be saved in the design database. Some MPC tools, such as the code generators (translators) and static analyser, might be be "adopted" into

Transactions on Information and Communications Technologies vol 9, © 1994 WIT Press, www.witpress.com, ISSN 1743-3517

the PDW as design post-processors. Probably the biggest challenge will be to provide an integrated design execution environment within the PDW. The simple approach would be to adopt the relevant MPC tools (Interpreter, Animator, etc) from the terminal-based system, but clearly the PDW system could be made to support a much more sophisticated arrangement, involving user interaction with multiple views in linked windows.

The use of design libraries (briefly mentioned earlier) is an extension of the database concept, requiring additional entity attributes and relations. Browsing through and selecting libraries or individual design objects must be supported by proper search facilities and navigations, and the build process itself needs to apply a *current context* to the resolution of external references. Although not a main requirement for the PDW as a teaching tool, we would like eventually to provide facilities for both version control and configuration management. Thus our long-term goal is to develop a full "professional" quality design support environment, to meet industrial as well as academic criteria of usability.

CONCLUSION

In its current form the PDW tool is no more than an early prototype, but already its possibilities are becoming clearer. Our immediate aim is to explore these possibilities in an incremental fashion, evaluating each new facility primarily in the academic context. Implicit in that aim is the intention to support a suitable program design methodology, rather than simply developing a range of tools. The MPC design language provides the basis for both a toolset and a methodology. Between the language and its tools, the following concepts are supported:

- structured programming
- top-down design
- modular development
- design reuse
- static and dynamic validation
- automatic documentation

What a good language and a good toolset normally do is to *encourage* good practice in software development. But a CASE tool such as the PDW is intended to *gently enforce* good practice, by guiding the user through the steps of a design process, the output of which is likely to be a well-structured, well documented, at least partially validated modular design.

Transactions on Information and Communications Technologies vol 9, © 1994 WIT Press, www.witpress.com, ISSN 1743-3517

REFERENCES

1. Lempp, P. 'Development and Project Management Support with the Integrated Software Engineering Environment, EPOS', in *Software Engineering Environments* (Ed. Sommerville, I.) , Peregrinus/IEE, 1986.

2. Wasserman, A.I. and Pircher, P.A. 'A Graphical, Extensible Integrated Environment for Software Development', in *Proceedings of the 2nd Symposium on Practical Software Development Environments*, ACM SIGPlan Notices,Vol.22, No.1, pp. 131-142, January 1987.

3. HOOD Technical Group (Dellatte/Heitz/Muller) *HOOD Reference Manual 3.1*, Masson and Prentice-Hall, Paris and London 1993.

4. Amatya, A. and Monk, K. 'A Survey of Modular Pseudocodes', *Working Paper WP-B6*, School of Mathematical and Information Sciences, Coventry University, December 1991.

5. Ratcliffe, M.B., Bott, M.F., Slotter-Brooks, T.J. and Whittle, B.R. 'The TIPSE: an IPSE for Teaching' *Software Engineering Journal*, Vol.7, No.5, pp.347-356, September 1992.

6. Stavrinides, A.A. 'MPC - A Modular PseudoCode' *Working Paper WP-B1*, School of Mathematical and Information Sciences, Coventry University, December 1991.

7. ToolBuilder 1.2 Manual, IPSYS Software plc, Macclesfield, 1992.

8. Law, K.C. 'EXPRESS Tool' *MSc Project Report*, School of Mathematical and Information Sciences, Coventry University, September 1991.

9. Arbelaitz, O. 'CASE Tool Development for the PDW' *Project Report*, School of Mathematical and Information Sciences, Coventry University, July 1993.