Quality in modular program design
K. Monk, A. Stavrinides

School of Mathematical and Information Sciences, Coventry University, Priory Street, Coventry, CV1 5FB, UK

ABSTRACT

This paper describes current work at Coventry University aimed at developing a methodology and creating a suitable environment and tools to support methodical program design at the modular level. The work focuses on undergraduate teaching of programming, an important area which does not always receive the attention it deserves, though it is also relevant to the commercial environment. For many students the concepts of program design and testing typically degenerate to those of coding and debugging, while documentation is paten an afterthought. The results of poor teaching are reflected in equally poor practices in the graduates' early employment.

The approach we have taken attempts to fill the gap between systems development methodologies and programming. While the former can tackle problems on a larger scale, once functional specifications have been produced for individual software components there is little guidance for the programmer to follow in designing high quality reusable and well documented program units and this can lead to problems in implementation and later maintenance. In order to address this area, initially in the educational context, we are developing an integrated system based on the use of a new design language, provisionally named MPC (Modular PseudoCode), which in some respects resembles Modula-2 or Ada.

The MPC tools include support for syntax-oriented editing, interface checking, chart generation, etc. The language itself offers facilities to support stepwise refinement and lends itself easily to block-structured and modular approaches. However, the most interesting aspect of the work is in developing and prototyping a CASE tool to control the design process, keep track of module inter-relationships and navigate through complex designs. In this paper we present an overview of the system and of the philosophy guiding its development.
INTRODUCTION

The work at Coventry University is targeted at the provision of a methodology and a suitable CASE environment and tools to aid in the creation of modular programs. This is currently designated the Program Design Workbench (PDW). It was felt that there was a considerable gap between many of the standard design methods used and the final code produced - students finding it hard to properly document the changes taking place between the production of the initial design and the handing in of the final working code. The excuse offered by many students is that of pressure of time - a reason given by some professional programmers for lack of or poor documentation for their code.

It was felt that there was a need for a system which automated as much of the design refinement and its documentation as was possible. The provision of a suitable design language which could be automatically verified and from which design documentation, both textual and graphical, was produced would allow any programmer to devote more time to the detail design. If the system could also take some of the load of code production, both design and code should be more complete and less prone to error. Such an environment will be suitable for both novice and experienced programmers. Since much of the overall cost of commercial programs can be spent in maintenance, it would also be of great help to professional programmers in allowing their designs to be made available at this stage. If a problem is identified as requiring a design modification this can be easily done and the code reproduced from this updated design created with the minimum of effort. This is much preferable to the alternative - a 'bug fix' which may itself create other problems.

MPC, the language our system is based around, is in many ways conventional. The most striking difference from Pascal/Modula-2/Ada is the lack of semi-colons. Some language aspects which may at first glance seem to be weaknesses are quite intentional. The nesting of procedures is not allowed to permit better visibility and variables cannot be anonymous types but have to be named types.

When importing/exporting an object, only the object need be mentioned in the import/export statement, not its 'owner' module. This will allow the module dependencies to be defined indirectly, unlike Modula-2 and Ada, with the system resolving references as needed - something like 'late binding'. The full Program Design Workbench (PDW) CASE tool will use a design database which will handle 'libraries', but the early prototypes are using directories to group modules.

When the MPC system was at the proposal stage a survey of the literature for an internal paper [AA/KM 1991] was made. This showed that, whilst the idea was not completely new, none of the then current systems were designed
to support the concept of working across multiple modules at the design phase. Some of these findings are summarised below.

Pierre N. Robillard and some colleagues at the Ecole Polytechnique De Montreal developed a Schematic Pseudocode (SPC) which represents the structure of a computer program language but is independent of any particular programming language [ROBI 1985a, ROBI 1985b, ROBI 1985c, ROBI 1986, ROBI 1987]. This has been built into a software tool, named SCHEMACODE, which supports the use of one of several target languages to design a computer program and then automatically generate source code in that target procedural language. While the SCHEMACODE is a useful tool for module design it does not deal with the problem of module interfacing well.

Another area of research encountered was a comparison of the use of structured flowchart (a graphical method) with pseudocode (a textual method) for learning short and relatively complex algorithms [SCAN 1988]. In Scanlan's study considering 292 students (from 24 classes) in 8 application areas, 80% showed preference for flowchart. Examples were taken from Elizy R S: Data Structures for Computer Information Systems. Chicago: Science Research Associates, 1982. The study concludes that graphic methods are superior to pseudocodes for teaching relatively complex algorithms. It also points out that there may be two types of readers, one type preferring flowcharts and the other pseudocodes. Comprehension of short and relatively complex algorithms, and NOT the design of program or algorithm is the point at issue.

B A Benander in another study [BEN 1990] investigated relationships between design techniques used and the correctness of programs, as well as, the program development effort among COBOL programmers. Using 31 students, the study found that for program design, 43% of the students preferred pseudocode, 31% flowchart, and 13% a combination of the two. As for correctness, 59% of programs designed using pseudocode were correct, 52% when using flowchart, and 89% when a combination of the two was used.

These last two studies showed that the design time, debug time and code time were better for the pseudocode method. However, a combination of both the structured flow-chart and the pseudocode gave the best result. The full paper is available from the author.

Many of the current programming languages in common use allow, indeed encourage, modular code. Such systems allow black and white box testing and the production of re-useable modules. We feel that the system we are developing will greatly aid the correct development of such programs.

With the development of different modules often being split amongst a number of programmers some method of checking or verifying the interfaces between the modules is thought to be essential to the production of quality code.
Such interface checking is an important element of our system, as is the ability to navigate through a complex design. These are not features needed by the novice programmer, but will be appreciated by experienced programmers. Also of importance to the latter group will be the ability of the system to show the progress of the refinement of the design.

**THE CASE SYSTEM**

The system is being developed around a Modular Pseudocode [AAS 1991] which uses ideas from Pascal, Modula-2 and Ada. A full version is available from the author. It supports structured programming and modular design including the encapsulation of Abstract Data Types (ADT’s) and runs under the Unix operating system. After the specification of MPC in a formal manner the prototyping of the necessary tools to complete the project is being done by final year students as their final year project.

The comprehensive range of tools proposed is:

For design input:

**Text Editor** is aware of the structure of MPC and will only allow the entry of correct code (as far as is possible). An initial prototype of this has been produced which contains an online help system. This is context sensitive, providing help on the current keyword. This prototype does not, unfortunately, have an expression parser as yet, so does not provide all the assistance we intend to provide in writing correct code.

For static validation:

**Syntax Checker** used to validate the MPC so that all later tools do not have to protect against invalid source. The current version handles the complete MPC syntax and can recover from errors so as not to fail on the first discovered error. It also does some pretty-printing.

**Interface Checker** used to verify the communication between program units and between modules. It stores information for use by other tools. A student project has completed the first phase of this, the production of a 'tags' file. A typical tags file gives the locations of specified objects in a group of files. Each line of such a tags file contains the object name, the file in which it is defined, and an address specification for the object definition.

**Static Analyzer** applies full semantic checks to ensure correct and consistent use of data, code reachability, adherence to standards, etc. It may also generate cross-reference listings and possibly may also
apply design metrics (e.g. complexity measures) to evaluate the design.

For design documentation:

**Chart Generator** creates graphical documentation from the pseudocode, currently Nassi-Schneiderman diagrams and control hierarchies. Will be extended to produce module interface charts, control flow diagrams and data flow diagrams, amongst other graphical forms.

**Formatter or Pretty Printer** used to enforce the layout standards as both an aid to readability and to reduce the work of other tools.

**Translators** are the final stage of the overall system and will automate as much as possible of the process of producing code in the chosen target language from the MPC design. Intended target languages are Modula-2, C and Ada. It may be necessary to allow extensions or enforce restrictions on the MPC design to facilitate the final translation to the target language. A project has produced an MPC to C Translator with limited features. It maps pre- and post-conditions and assertions to C comments. No attempt is made to deal with pointers. The experience gained from this project confirms our opinion that all conventional MPC constructs can be mapped to C.

For design execution:

**Interpreter** executes the whole MPC program. This is seen as a central part of the environment.

**Assertion Processor** will operate as an extension to the Interpreter and will use the MPC pre- and post-condition and assertion statements. A long term aim here is to allow the tracing of conditions through control points (branches and loops) to help 'prove' program correctness. A main reason for including the assertion/condition language features of MPC was to encourage students and programmers to document their algorithms and to argue 'manually' about their correctness.

**Animator** will show onscreen the current program status whilst the program is executing in the interpreter. One idea is to allow the user to define the pictures to be animated. An example would be a data flow diagram where at run time the animator would insert actual data values next to data names (on arrows or stores).
For dynamic validation:

**Debugger** is an extension to the interpreter, allowing single-stepping and breakpoint execution, inspection of variables and parameters, etc.

**Profiler** will assist in program tracing and instrumentation.

**Test Planner** analyses code and data usage using information supplied by the static analyzer to identify test cases, test data classes and suitable test data.

**Test Harness** will be used to control the 'structured' testing of the program within a suitably extended interpreter environment, using information from the test planner.

The initial development work has gone into the core of the system, syntax checking and static analysis and the editor. Some work has been done on the graphical output, but most system parts are currently purely textual. It is envisaged that a terminal-based version of PDW will be used as a teaching aid and a workstation-based version for 'professional' use. The former will provide direct and/or script access to the tools via a screen-oriented front end. This will be suitable for large classes though of limited sophistication. The latter will provide a graphical user interface produced using the IPSYS ToolBuilder and will provide more sophisticated facilities.

**QUALITY IN MODULAR DESIGN**

Both forms of the CASE system will support advanced program design in a number of ways. The first of these is modularity. This will be done by the checking of what each module imports and/or exports. Such checking allows visibility control over the correct use of variables, procedures and functions. Secondly, by encouraging the use of proper modular design the system will also aid the production and control of libraries of source code. This can be of great use to the commercial programmer who has variants of the same program running on a number of platforms.

Thirdly, by allowing the checking of pre- and post-conditions and the testing of assertions about the current state, formal validation becomes possible. This is of enormous benefit in the development of larger systems. A fourth support for advanced program design is in the area of design management tools. Those CASE tools to be provided give some of the expected benefits provided by such systems, such as dependency checking and make, but will also allow the logging of what design changes were made. This will go further in that not just what was changed will be logged, but also who made the change, when and why. This is expected to be of benefit to commercial users, especially as the design documentation for the complete system will be modified to reflect the new state.
One aspect of the MPC design language which we consider to be of importance in tracking the progress of a design is the marking of English style initial pseudocode as requiring development at a future design stage. Such phrases/sentences are indicated initially by a leading '++'. These will be retained as comments in the later pseudocode when they are refined, the '++' becoming '--'. This automatically allows any programmer to follow the progression of the design. It is to be expected that each '++' design stage would be refined through several more similar stages for a typical project. All of the phases of this multi-stage refinement process are retained and may be navigated through by folding and unfolding levels as with an 'outliner' tool. The editor, when fully developed, will provide this function. This is expected to make maintenance programming easier as, once a design flaw is found in a project, our system will document the changes made, check their correctness in the context of the complete project and update all the associated project documentation.

CONCLUSIONS

We feel that the system as currently being developed is a pointer to the future of modular program design and development, though current work is focused on the teaching elements of the system. As code becomes larger and more complex there is a need for tools to aid the programmer in code development. It has been shown in past research [BEN 1990][SCAN 1988] (summarised above) that more correct code is produced when at least two different forms of design are used. Our system will produce both textual and graphical design detail, the combination which was found to be most effective in correct code production.

Since all aspects of program production are covered by our system, it is to be expected that the final code produced will be more correct than from many current systems. It will also be more maintainable as the complete design detail is available. These two factors should reduce the considerable overheads involved in code maintenance.
REFERENCES


