A structure diagram design tool for teaching embedded systems software design

John Collins, Mark Beckerleg
Engineering Research Institute
School of Engineering
Auckland University of Technology

Abstract

In this paper we describe an educational software tool designed and developed to help students create design structure diagrams and structured C code for embedded systems applications using a microcontroller. The software allows the easy creation of structure diagrams using standard symbols (sequence, selection and iteration), line drawing and text editing. Multi-page diagrams may be drawn, the pages being linked by interpage connectors or function calls. When the structure diagram is complete, an outline of the corresponding C code can be automatically generated. The generated C code includes comments relating to each symbol in the structure diagram, partly coded selection, iteration and function definition statements, and automatic creation of blocks of code corresponding to the structure diagram. The student then edits the C code by completing the partly coded statements, adding sequence statements and other details such as include, define, variable definition and function prototype statements. If the program structure needs to be changed during debugging, the structure diagram can be modified and new C code generated. This new code can be merged with the previous C code version, so that any code changes already made by the student are not lost. The application of this design tool in a taught paper is discussed, and student feedback is presented. It is shown that this design tool has resulted in much improved student focus on software design before coding.

Keywords: design structure diagram, programming, education, software design, program code generation.

Introduction

In recent years, a number of computer-based tools have been developed to help students learn programming concepts and skills [1] - [6]. These tools have concentrated on teaching the syntax and semantics of various programming languages. The tool described in this paper helps students design their programs before program code is generated.

A major problem for students learning computer programming for the first time is that they need to learn how to design programs at the same time as they consolidate their understanding of the syntax and semantics of the programming language [3]. The aim of this tool is to separate program design and coding so students can concentrate on one development phase at a time. We want our students to design their programs before they start writing the program code.

The Auckland University of Technology School of Engineering graduates who specialise in software development are most likely to be employed by companies developing embedded systems applications using the C programming language. For this reason we use the C programming language for most of our courses. For the same reason, we concentrate on structured programming instead of object-oriented programming. Most potential employers of our engineering graduates require C programmers for embedded systems, not C++ or Java programmers.

In their programming assignments, students are required to develop programs containing a small number of C functions that interact in fairly simple ways. In their first programming course, as students learn the syntax and semantics of the C programming language, they are provided with design structure diagrams from which they write their program code. This allows the students to become familiar with the programming language and design structure diagrams before being required to design programs themselves. This conforms to the educational model developed in [7].

Once students have been introduced to the fundamental language syntax, we introduce program design when the students design their programs before coding. The design structure diagram conforms to the ISO/IEC 8631:1989 standard [8]. This is based on the United Kingdom standard BS6224:1987 [9].

Historically, when students are given a design specification, most students will code the program and then attempt to make the program work correctly, before documenting their program as a design structure diagram. Experienced programmers often code like this, without a detailed written design, because they have the design in mind and the design is often based on familiar programming patterns. These programmers are also familiar enough with the language syntax that this does not obstruct their thinking about the design. However, experienced programmers also recognise when they need to design and would be expected to complete a design before coding in these circumstances.

Some students postpone designing because it is hard, and they would rather deal with the coding which they find easier. These students have great difficulty producing a program that meets the specification. Other students complain that writing the design by hand is time-consuming because the design diagram needs to be completely redrawn if there are more than a few errors, as is usually the case.

This tool addresses these issues by:
(1) making it easy to draw, save and modify the design diagram,
(2) generating a program code outline,
(3) allowing the design and code to be kept consistent as modifications are made.

Method

A typical design structure diagram is shown in fig 1. This structure diagram is for a simple reaction timer device. The user presses a start button to arm the system. Then after a random time delay of 2 to 6 seconds, a led is activated and the user presses the reaction button as quickly as possible. The reaction time is displayed on a two digit seven segment display, as a time from zero to ten seconds in steps of tenths of a second.

When drawing the design structure diagram, the student selects diagram symbols from the toolbar and places them on the page. The student then draws lines to connect the symbols, and can add or edit text in each symbol on the diagram. The tool ensures only valid line connections are drawn.

Program execution starts at the START symbol and moves sequentially down the symbols off the vertical line below the START symbol. Function calls are shown as symbols with double vertical lines. Program execution transfers to another part of the diagram headed by the function symbol. When the END symbol of the function is reached, program execution returns to the symbol following the function call. Selection statements are shown by the small diamond shaped symbol, and iteration statements are shown by the looping arrow symbol.

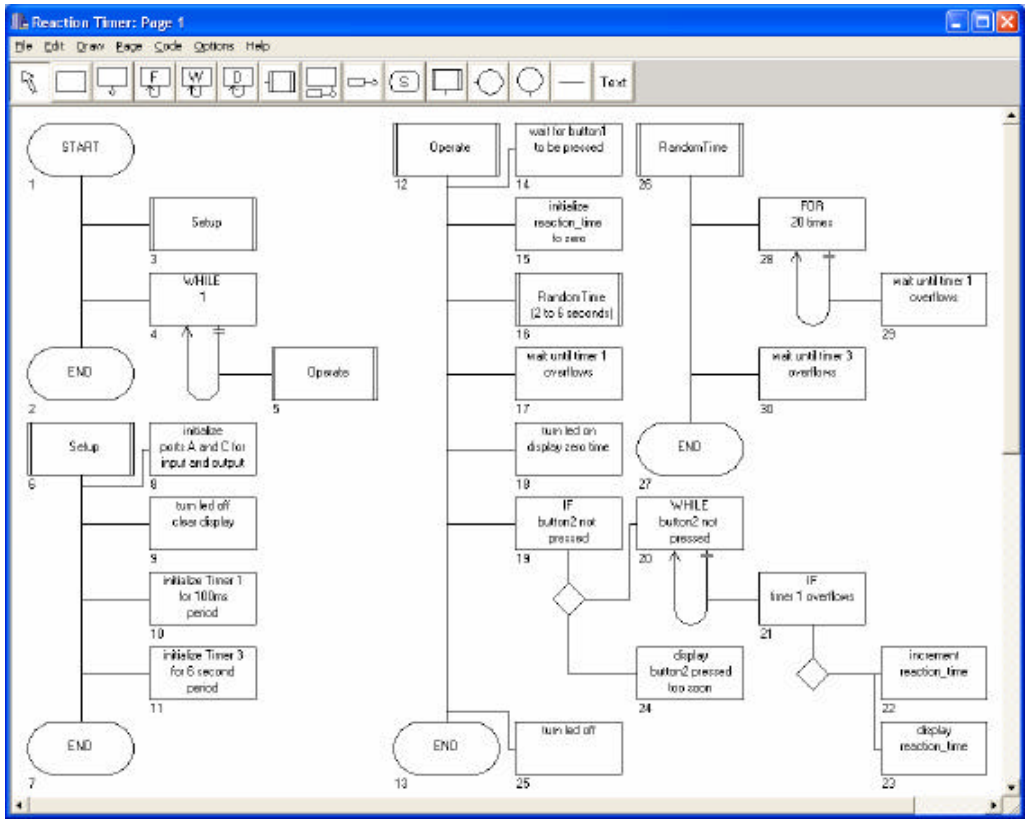

Figure 1: A typical design structure diagram

When the diagram is complete, the tool can generate C code from the diagram. Code from the above diagram is shown in appendix A. Each symbol is numbered, and the code is commented with these numbers so each block of program code is associated with a symbol on the diagram. The student then completes the generated C code required for the application. The writing of code has been separated from the program design, so at this stage the student can concentrate on the syntax of the code being written. The code is compiled and tested in the compiler development environment, as usual.

If any design errors are found at this stage, the student can return to the diagram, modify it, and merge the resulting diagram with the code already written. In this way, already completed

code does not have to rewritten. If symbols are rearranged on the structure diagram, then the code will be rearranged in the same way, based on the symbol numbers and corresponding comments in the code.

Results

A few students were selected to trial the tool during beta testing. These students were not beginners, because we wanted to get knowledgeable feedback about the tool operation. The feedback from the beta test students was very positive. They all stated that they wished the tool had been available for the entire time they had been enrolled.

The tool has been released for general use this year. A survey of two classes (38 students) with access to the tool was conducted. All 38 students had used the tool. The students were asked which of the following features had been used. The number of "yes" responses is listed below:

37 drew structure diagrams,
34 printed diagrams for reports,
35 saved program code from diagrams,
17 merged program code from modified diagrams.

When asked what other features had been used, 2 students (from different classes) stated they had drawn diagrams across multiple pages. They considered this was a feature significant enough for additional comment, although we had not considered it significant enough to include in our list of features.

Students were asked whether the tool had helped their understanding:

29 stated it had helped their understanding of structure diagrams,
27 stated it had helped their understanding of computer programming.

3 students stated they already had a good understanding before using this tool.

When asked what they liked about the tool, the responses were:

25 easy to use
7 produces program code
2 saves time/more time to debug/fast to use
1 setting out discipline

The generated program code indents and lays out lines of code according to the style recommended for student use. This imposes the "setting out discipline" on the program code.

When asked what they disliked about the tool, the responses were:

14 need to fit larger diagrams on a page/smaller symbols /smaller snap grid
9 implement all editing commands
5 comments added to generated program code
3 Delete key operation in text editing
1 mixed up code

The following improvements were suggested:

2 right click menu
1 print preview
1 enter program description

Some responses indicated a lack of understanding of the role of design structure diagrams, for example:

more help with compiler errors
hard to define and use variables

Students were asked if they had drawn design structure diagrams themselves, before using this tool Most of the surveyed students had previously completed a short introductory programming course in which they were required to write program code from provided structure diagrams. For some students, this was their only previous programming experience.

28 students had previously drawn structure diagrams for design.

When asked if they previously drew the structure diagrams before writing code, these 28 students stated:

25 drew diagrams before writing code
1 sometimes drew diagrams before writing code
1 drew diagrams after writing code
1 did not respond to this question

When asked whether they were now drawing structure diagrams before coding, the 38 students stated:

35 drew diagrams before writing code
2 sometimes drew diagrams before writing code
1 drew diagrams after writing code

Discussion

The survey has identified a number of areas where the tool needs be enhanced:

implement all editing commands
Delete key operation in text editing

These are both areas that were known to require attention.

In addition the following suggestions have been made:

right click menu
print preview
enter program description

These will be fairly easy to add to the tool. These features are standard for modern computer applications and it is not surprising that students expect them to be available.

The main problem identified by students is the need to fit larger diagrams on a page. Related suggestions are to use smaller symbols or a smaller snap grid for positioning symbols on the page. The current layout of the symbols is used to avoid allowing symbols to overlap and to allow lines to be drawn easily to connect the symbols. Implementing resizable symbols or a smaller snap grid would allow some increase in diagram size per page, but the increase would be relatively small (probably up to 25%) and is not really a significant increase in diagram size. In the short term, we will encourage students to split their diagrams across multiple pages rather than make significant changes to the current diagram layout.

The way the diagrams are defined by the standard also affects the use of page space. For example, the start and end symbols are connected by a vertical line that uses a lot of space on the page. However we are reluctant to change the diagram from the standard format.

The survey also shows that the program code modification is a major feature that is only being used by about half the students. This feature was added part way through the semester and it is possible some students were not aware of it. However some students specifically stated they had difficulty with this feature. The tool adds comments to the generated program code so that the structure diagram tool can identify which diagram symbol the program code is associated with, even after the student has modified the program code. These comments are regarded as cluttering the program. An option was added that allows the student to disable writing of comments to the code. This has the side effect of making it impossible to merge diagram changes with existing code. It is not known how many students are using this option.

Several of these factors indicate that students need more training in using the structure diagram tool. Some students still do not understand the role of the structure diagram. Also some features appear to be poorly understood. This certainly applies to the merging of program code after modifying the structure diagram,

Teaching staff report that they now see most students drawing design structure diagrams before writing code, whereas before this tool was available teaching staff reported that most students were not drawing design diagrams before coding. We were surprised that most students stated they were drawing design diagrams before coding, even before this tool was available. This was not consistent with the observations of teaching staff.

Future Work

A further enhancement, not suggested in the survey, is to allow students to test their design by running or stepping through the diagram. If there are any errors in the structure of the design, such as unconnected symbols, then a warning will be issued before running the design.

When stepping, the tool will highlight the diagram symbols in the sequence they will be executed, starting at the START symbol. The stepping may be automatic at a rate controlled by the student, or it may be manual where the student controls the occurrence of each step.

When the simulation reaches a selection (if or switch) or iteration (for, while or do-while) symbol, the student will be asked whether the associated condition is true or false. The simulation will then make the appropriate step to the next symbol. The simulation can be configured to step into functions or step over them, depending on the depth the student wants to go to in the simulation. When running, breakpoints will be set on symbols so the run will stop when it reaches a symbol with a breakpoint set. It will also be possible to set the conditions of selection symbols or the number of times iteration symbols will be executed so the simulation will not stop at these symbols when running automatically.

It is expected these features will improve the student's understanding of the meaning of structure diagrams, as well as enabling students to test the logic of their program designs before generating code and testing the program.

Conclusion

The structure diagram tool has been widely accepted by students of programming courses in the School of Engineering at Auckland University of Technology. A survey indicated students appreciate the value of this tool and that it has improved their understanding of design

structure diagrams and computer programming. The survey also identified a number of enhancements that would improve the usability of the tool. It appears that some students require more training in the role of structure diagrams and the use of the tool.

One of the main objectives of the tool was to get students to draw design structure diagrams before writing program code. Both the student survey and the observation of teaching staff indicate this objective has been achieved.

Appendix A

Program code generated from the design structure diagram in figure 1.

```
//sdg PROGRAM TITLE: Reaction Timer
//sdg DIAGRAM FILE: C:\Micros\ReactTimer.sdg
//sdg CODE FILE: C:\Micros\ReactTimer.c
//sdg DATE: 28 February 2005 10:14am

// INCLUDES

// DEFINES

// FUNCTION PROTOTYPES

// GLOBAL VARIABLES


//sdg PAGE 1 TITLE: Page 1

//sdg00001 START
void main()
{
      //sdg00003 Setup
      Function();
      //sdg00004 WHILE 1
      while()
      {
            //sdg00005 Operate
            Function();
      //sdg00004
      }
//sdg00002 END
}

//sdg00006 Setup
void Function(void)
{
      //sdg00008 initialize ports A and C for input and output

      //sdg00009 turn led off clear display

      //sdg00010 initialize Timer 1 for 100ms period

      //sdg00011 initialize Timer 3 for 6 second period

//sdg00007 END
}
```

```
//sdg00012 Operate
void Function(void)
{
      //sdg00014 wait for button1 to be pressed

      //sdg00015 initialize reaction_time to zero

      //sdg00016 RandomTime (2 to 6 seconds)
      Function();
      //sdg00017 wait until timer 1 overflows

      //sdg00018 turn led on display zero time

      //sdg00019 IF button2 not pressed
      if()
      {
            //sdg00020 WHILE button2 not pressed
            while()
            {
                  //sdg00021 IF timer 1 overflows
                  if()
                  {
                        //sdg00022 increment reaction_time

                        //sdg00023 display reaction_time

                  //sdg00021
                  }
            //sdg00020
            }
      //sdg00019
      }
      else
      {
            //sdg00024 display button2 pressed too soon

      //sdg00019
      }
      //sdg00025 turn led off

//sdg00013 END
}




//sdg00026 RandomTime
void Function(void)
{
      //sdg00028 FOR 20 times
      for()
      {
            //sdg00029 wait until timer 1 overflows

      //sdg00028
      }
      //sdg00030 wait until timer 3 overflows

//sdg00027 END
}
```

References

[1] Satratzemi, M., Dagdilelis, V., and Evagelidis, G.: A system for program visualization and problem-solving path assessment of novice programmers. ACM SIGCSE Bulletin, Proceedings of the 6th annual conference on Innovation and technology in computer science education, (June 2001), Vol 33, No 3, 137-140.

[2] Chang, K., Chiao, B., Chen, S. and Hsiao, R.: A programming learning system for beginners - a completion strategy approach. IEEE Transactions on Education, (May 2000), Vol 43, No 2, 211-220.

[3] Garner, S.: The learning of plans in programming: a program completion approach Proceedings of the International Conference on Computers in Education, (3-6 Dec. 2002), vol 2, 1053-1057.

[4] Daly, C. and Horgan, J.M.: An automated learning system for Java programming. IEEE Transactions on Education, Vol 47, No 1, Feb. 2004, 10 - 17.

[5] Davidovic, A., Warren, J. and Trichina, E.: Learning benefits of structural example-based adaptive tutoring systems. IEEE Transactions on Education, Vol 46, No 2, May 2003, 241 - 251.

[6] Dancik, G. and Kumar, A.: A tutor for counter-controlled loop concepts and its evaluation. 33rd Annual Frontiers in Education, 2003. FIE 2003. 5-8 Nov. 2003, Vol 1, T3C-7 -12.

[7] Buck, D. and Stucki, D.J.: Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development. ACM SIGCSE Bulletin, Proceedings of the thirty-first SIGCSE technical symposium on Computer science education, March 2000, Vol 32, No 1, 75-79.

[8] ISO/IEC 8631:1989. Information technology - Program constructs and conventions for their representation. (1989)

[9] BS 6224:1987. British Standards Institution. Guide to Structure Designs for use in Program Design and Other Logic Applications. (1987)

John Collins received the BSc(Hons) degree in physics and mathematics and the PhD degree in electronic engineering from the University of Auckland. He is currently a senior lecturer in the School of Engineering at the Auckland University of Technology. He has industrial experience in the development of embedded systems and has worked as a consultant on a number of embedded systems projects. His research interests include the design and verification of embedded systems, and embedded systems education.

Mark Beckerleg received the BE(Hons) degree from the University of Auckland. He is currently a senior lecturer in the School of Engineering at the Auckland University of Technology. Prior to that, he worked as an engineer for Television New Zealand and an embedded systems design consultant. He is currently studying for his PhD.