Development and Assessment of a Tool to Support Pattern-based Code Generation of Time-Triggered (TT) Embedded Systems

Thesis submitted for the degree of

Doctor of Philosophy

at the University of Leicester

Chisanga Mwelwa BEng (Hons)

Department of Engineering, University of Leicester

Leicester, United Kingdom

October 2006

Development and Assessment of a Tool to Support Pattern-based Code Generation of Time-Triggered (TT) Embedded Systems

Chisanga Mwelwa

Department of Engineering, University of Leicester, Leicester, United Kingdom

Abstract

This thesis is concerned with embedded systems which employ time-triggered software architectures and for which there are both severe resource constraints and a requirement for highly-predictable behaviour. The thesis discusses design patterns and their benefits to software development and reviews a pattern language (the PTTES collection) previously assembled to support the development of time-triggered embedded systems.

As embedded systems become ever more complex and – in many cases – take on an increasing role in safety, it is widely recognised that developers require tools and techniques that support the 'automatic' generation of such designs. This thesis makes a novel contribution to the field of pattern-based automated code generation and illustrates the capabilities of this approach in the development of reliable time-triggered embedded systems. Specifically, the approach described in this thesis addresses a key limitation of previous work in this area, namely the challenge of implementing the 'one pattern, many implementations' relationship. Furthermore, unlike previous pattern tools, the approach described in this thesis is based on a substantial pattern language: this paves the way for the generation of coherent application code from groups of related patterns.

To test the above ideas, the thesis describes PTTES Builder, a pattern-based code generation tool based on the PTTES collection. In an empirical study, the effectiveness of the PTTES Builder approach is compared with an equivalent 'manual' approach. The results obtained demonstrate that time-triggered embedded systems can be created using this approach. There is also some evidence that the use of the tool is likely to lead to improved code reliability and quality. In a second study discussed in the thesis, there are indications that the approach implemented by PTTES Builder is robust enough to support the evolution of its underlying pattern collection.

The thesis concludes by making a number of suggestions for future extensions to this work.

Acknowledgements

I would like to thank a number of people who have supported me in one way or another during the course of this project.

First and foremost, Dr Michael Pont (my supervisor) for introducing me to *patterns,* an inspiration of this work. Also, for his continued enthusiasm, guidance and valuable support during the course of this project. It has been a sincere pleasure working with you.

My sincere thanks go to the EPSRC and Dr David Ward (on behalf of MIRA Ltd) for financially supporting this project.

I would also like to thank Dr Fernando Schlindwein and Dr Da-Wei Gu for their useful feedback at various stages of this work.

Thanks also go to the members of the Embedded Systems Laboratory (ESL) whom I have had the pleasure of knowing. Devaraj Ayavoo for his co-operation during a number of research collaborations and the other members of the group who have been good company when work was not on the agenda.

I would also like to thank my parents for their support and encouragement. My brothers: Pilula, Mulenga and Chishimba for having an interest in my work.

Last but not least, Micah for being an untimely blessing and pleasant distraction during the course of this write up, I hope this inspires you some day.

Contents

List of Abbreviations and Acronyms

ACM	Association for Computing Machinery
ADC	Analogue to Digital Converter
CASE	Computer Aided Software Engineering
CCS	Cruise Control System
CMFD	Condition Monitoring and Fault Diagnosis
CPU	Central Processing Unit
DAC	Digital to Analogue Converter
EDF	Earliest Deadline First
ESL	Embedded Systems Laboratory
IDE	Integrated Development Environment
ISR	Interrupt Service Routine
J2EE	Java 2 Platform Enterprise Edition
LCD	Liquid Crystal Display
LOC	Lines of Code
MDA	Model Driven Approach
OMG	Object Management Group
OOPSLA	Object-Oriented Programming, Systems, Languages & Applications
PDM	Platform Definition Model
PID	Proportional Integral Differential
PIE	Pattern Implementation Example
PIM	Platform Independent Model
PLoP	Pattern Languages of Programming
PSM	Platform Specific Model
PTTES	Patterns for Time-Triggered Embedded Systems
RAM	Random Access Memory
RM	Rate Monotonic
ROM	Read Only Memory
SLOC	Source Lines of Code
TDMA	Time Division Multiple Access
TT	Time-Triggered
TTC	Time-Triggered Co-operative
UART	Universal Asynchronous Receiver Transmitter
UML	Unified Modelling Language
XML	Extensible Mark-up Language
XSLT	XML Style-sheet Language

List of Figures

List of Tables

List of Associated Publications

Below is a list, in reverse chronological order, of publications produced during the course of this research. Material from some of these papers has been included in this thesis and where applicable this is noted in a footnote at the beginning of the chapter concerned. Abstracts of these publications are presented in Appendix D.

Pont, M.J., Mwelwa, C., Bonthonneau, L., Ayavoo, D., Athaide, K., Mearns, D., Kurian, S. and Ward, D., "Pattern-based development of time-triggered embedded systems using software tools: Challenges and solutions," Journal of Systems and Software, submitted 2006.

Mwelwa, C., Athaide, K., Mearns, D., Pont, M.J. and Ward, D., "Rapid software development for reliable embedded systems using a pattern-based code generation tool," In-vehicle software and hardware systems, In: Society of Automotive Engineers (Eds.), Paper presented at the Society of Automotive Engineers (SAE) World Congress, Detroit, Michigan, USA, 2006. [ISBN: 0-7680-1763-7].

Mwelwa, C., Pont, M.J. and Ward, D., "Developing reliable embedded systems using a pattern-based code generation tool: A case study," Proceedings of the 2nd UK Embedded Forum, In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.), pp. 177-193, Birmingham, UK, 2005. [ISBN: 0-7017-0191-9].

Mwelwa, C., Pont, M.J. and Ward, D., "Code generation supported by a pattern-based design methodology," Proceedings of the 1st UK Embedded Forum, In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.), pp. 36-55, University of Newcastle upon Tyne, Birmingham, UK, 2004b. [ISBN: 0-7017-0180-3].

Mwelwa, C., Pont, M.J. and Ward, D., "Using patterns to support the development and maintenance of software for reliable embedded systems: A case study," Proceedings of the IEE/ACM Postgraduate Seminar on "Systems-on-Chip" Design, Test and Technology, Loughborough, UK, IEE, 2004a. [ISBN: 0-86341-460-5].

Mwelwa, C. and Pont, M.J., "Two simple patterns to support the development of reliable embedded systems," 2nd Nordic Conference on Pattern Languages of Programming (VikingPLoP), Bergen, Norway, 2003.

Pont, M.J. and Mwelwa, C., "Developing reliable embedded systems using 8051 and ARM processors: Towards a new pattern language," Proceedings of the 2nd Nordic Conference on Pattern Languages of Programming (VikingPLoP), Bergen, Norway, 2003b.

Mwelwa, C., Pont, M.J. and Ward, D., "Towards a CASE tool to support the development of reliable embedded systems using design patterns," Proceedings of the 1st International Workshop on Quality of Service in Component-Based Software Engineering (CBSE), Toulouse, France, CEPADUES-EDITIONS, pp. 67-80, 2003. [ISBN: 2-85428-617-0].

PART I: INTRODUCTION

This first part gives an introduction and an outline of the work presented in this thesis.

1 Introduction

This chapter gives an introduction to embedded systems and an overview of their importance in modern day society. Some of the challenges faced by developers of these systems are also discussed.

1.1 Embedded systems and their importance in modern society

An embedded system is an application that contains at least one programmable computer and is encapsulated by the device it controls. It is dedicated to a specific function performed by pre-defined tasks it executes (Bolton, 2000, p. 64). Although embedded systems contain a programmable computer, they do not fall into the same category as other computing devices such as personal computers, palm computers etc; these devices have a more general application in comparison to embedded systems that tend to be specialised applications. For instance, desktop computers come with a range of functionalities such as spreadsheets, word processing etc. On the other hand embedded applications have specific functionality, for example washing machines are designed specifically for laundry purposes (Bolton, 2000, p.64; Ayala, 1991, pp.1-2).

Increasingly today, a major distinction between embedded systems and general purpose computers relates to the likely consequences of incorrect operation. It is unlikely that a user will suffer any harm from the incorrect operation of a personal computer e.g. a Windows application such as Excel that is likely to crash. However, the failure of a safety-critical embedded application could result in direct and possibly very serious harm to human beings. For instance, the failure of an automotive engine management or braking system could risk the lives of several people (Storey, 1996, p. 1).

The Apollo Guidance Computer (AGC), developed for NASA's Apollo programme (1963 – 1972) by the MIT Instrumentation Laboratory, was probably the first recognisable modern embedded system (Computer History Museum, 2005). It was used by astronauts to collect and provide flight information and to automatically control all navigational functions of the Apollo spacecraft (NASA, 2005). The first version of the AGC (Figure 1-1 is a picture of the AGC) used 4,100 integrated circuits and it is believed to have been a major impetus to the development and improved manufacturability of integrated circuitry (Computer History Museum, 2005).

Figure 1-1 The Apollo Guidance Computer. This image has been used with permission from the Computer History Museum (Computer History Museum, 2005)

The first microprocessors used in embedded systems appeared on the market in 1971 when Intel released the world's first commercial microprocessor, the 4-bit 4004. It was developed for a Japanese calculator company, as an alternative to integrated circuit packages, to read and execute a set of instructions stored in memory as software. Intel's idea was that the software would give each calculator its unique set of features (Ayala, 1991, p.1).

By integrating the processor into a single integrated circuit, the cost of processor power was greatly reduced. The microprocessor allowed computers to be smaller and faster than ever before (IBM, 2006; Intel, 2005). As a consequence the microprocessor was an overnight success, and its use increased steadily over the next decade. Other more well established semiconductor firms soon followed Intel's pioneering technology so that by the late 1970s one could choose from a dozen or so microprocessor types (Ayala, 1991, p.1). Faster 8-bit microprocessors followed with more and more facilities added to fill what were seen as gaps in a highly competitive market. In recent years 16-, 32- and 64-bit devices have become available with 16-bit devices dominating in automotive applications (Edwards, 2006, pp. 46-47).

The evolution in microprocessor technology has come to be known as 'Moore's Law' (IBM, 2006; Intel, 2005). In 1965, Gordon Moore predicted that the number of transistors the industry would be able to place on a microprocessor would double every year, Figure 1-2 is a graph of this prediction. In 1975, he (Gordon Moore) updated his prediction to once every two years. While originally intended as a rule of thumb in 1965, it has become the guiding principle for the semi conductor industry to deliver powerful processors at proportionate decreases in cost. This predicted exponential growth has driven the semiconductor industry from start-up some fifty years ago to more than $200 billion in annual revenue which today accounts for over a quarter of the annual sales in the semiconductor market (Intel, 2005; SIA, 2002; Gupta and Micheli, 1997).

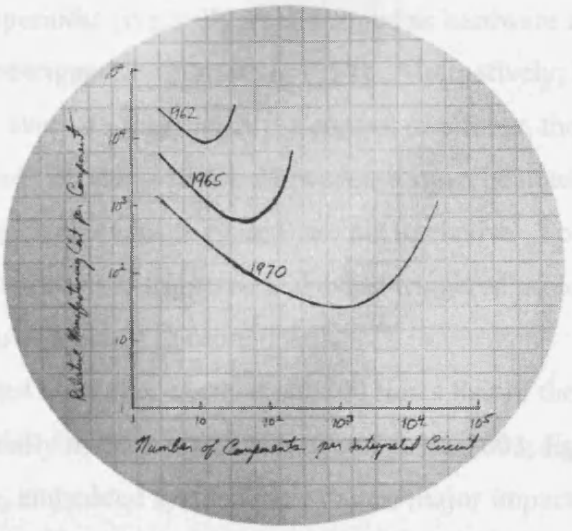


Figure 1-2 Moore's Law. In 1965, Gordon Moore sketched out his prediction of the pace of silicon technology. Decades later, Moore's Law remains true. This image has been used with permission from Intel (Intel, 2005)

Microprocessors are today generally known as central processing units (CPUs). Microcontrollers are by-products of microprocessors; they incorporate all the features of a microprocessor including: ROM (Read Only Memory), RAM (Random Access Memory), parallel and serial Input/Output (I/O), counters and a clock circuit (Antonakos, 1993, p. 6). In essence, a microcontroller can function as a computer without the addition of external peripherals whereas the microprocessor must have certain external integrated circuits (ICs) in order to operate. Figure 1-3 is an overview of the architecture of a microcontroller.

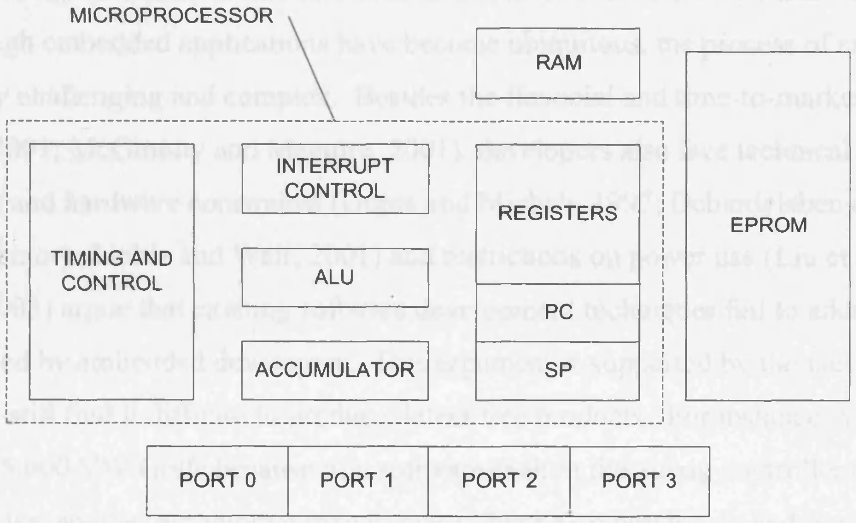


Figure 1-3 Overview of a microcontroller's architecture (adapted from Warnes, 1998, p. 442)

Technically, embedded systems are composed of both software and hardware elements and are often designed and implemented as a collection of tasks that share resources and interact with the system or environment in which they operate (Hsiung et al., 2002; Mullerburg, 1999). The various possible system architectures may then be characterised in terms of their tasks. For example, if the tasks are aperiodic (typically implemented as hardware interrupts) the system may be described as 'event-triggered' (Nissanke, 1997). Alternatively, if all the tasks are invoked periodically (e.g. every 15 ms), under the control of a timer, then the system may be described as 'time-triggered' (Kopetz, 1997). However, it must be noted that these two architectures (time-triggered and event-triggered) are not exclusive. Today it is common to find complex systems in which both time-triggered and event-triggered protocols coexist. Chapter 2 discusses these software architectures in more detail.

Today the embedded market is worth about 100 times that of the desktop market and is forecast to grow exponentially in the next decade (Graaf et al., 2003; Eggermont, 2002). As a result of this market share, embedded systems now have a major impact in many areas of product development (Heiner and Thurner, 1998; Gupta and Micheli, 1997; Storey, 1996). For example, in the automotive industry, embedded systems were initially used in engine control units (ECU) in order to reduce exhaust emissions and improve fuel efficiency of vehicles. This was viewed as a practical and cost effective means of meeting the United States engine emission and fuel economy requirements imposed in 1981 (Bereisa, 1983; Flis, 1983; Marley, 1978). Since then it has been estimated that nearly a third of the cost of developing high-class passenger cars is spent on electronic and software systems (Bouyssounouse and Sifakis, 2005). For example, a luxury BMW today contains approximately fifty embedded applications serving either comfort or safety e.g. window and engine control and anti-braking systems (Mullerburg, 1999). In fact it has been argued that these applications increase overall vehicle safety by liberating the driver from routine tasks and providing assistance to find solutions in critical situations (Heiner and Thurner, 1998).

Although embedded applications have become ubiquitous, the process of creating them remains highly challenging and complex. Besides the financial and time-to-market constraints (Jenko et al., 2001; McGinnity and Maguire, 2001), developers also face technical challenges such as timing and hardware constraints (Gupta and Micheli, 1997; Debardelaben et al., 1997) e.g. limited memory (Noble and Weir, 2001) and restrictions on power use (Liu et al., 2001). Graaf et al. (2003) argue that existing software development techniques fail to address the challenges faced by embedded developers. This argument is supported by the fact that manufacturers still find it difficult to produce defect free products. For instance, Volkswagen had to recall 35,000 VW Golfs because of a software fault in the airbag controller (Mullerburg, 1999). Mercedes, another automotive manufacturer, have also previously had to recall 1.3 million cars (E, SL, and CLS class models), the biggest recall in the company's history. This was due to a fault that caused some vehicles to switch from their advanced electronic braking system into a 'fail-safe' mode (Hutton, 2005). Most recently thousands of Segway Human Transporters were recalled because of a software bug that unexpectedly made their wheels reverse causing users to fall off (Sanderson, 2006). Such eventualities result in increased overheads and the potential loss of business.

Despite the challenges currently faced by embedded developers, the European Commission expects embedded systems to play a key part in achieving their main transport policy goal, which is to reduce fatalities on European roads by 50% by the year 2010 (EU eSafety Working Group, 2003). Achieving these targets will be challenging for developers as in the years to come the key to success will be the ability to successfully produce highly reliable embedded systems within time-to-market constraints. Therefore, to optimise the timeliness, productivity and quality of embedded software development, appropriate development support is required for developers (Graaf et al., 2003; McGinnity and Maguire, 2001; Henderson et al., 2001; Cuatto et al., 2000; Camposano and Wilberg, 1996).

1.2 Developing embedded systems

Similar to desktop computers, embedded systems require some form of operating system to guarantee the allocation of computational and data resources to tasks in order to satisfy all timing and functional requirements (Hsiung et al., 2002). A scheduler is an algorithm concerned with the organisation and allocation of these resources. It manages an embedded system's resources such as power and memory, determines the order in which tasks are executed and also provides a means of predicting the worst-case behaviour of a system when the scheduling algorithm is applied (Bate, 1999; Buttazzo, 1997, p. 23). Consequently, the software architecture of an embedded system is determined by its underlying scheduler (Locke, 1992).

There are a wide range of different types of schedulers. Two widely used scheduling algorithms are the pre-emptive and co-operative (also known as non-pre-emptive) schedulers. In a pre-emptive scheduling environment, tasks can be interrupted at any time and the processor assigned to more urgent tasks ready to execute. On the other hand, in a co-operative scheduling environment, tasks execute uninterrupted to completion (Bannatyne, 1998; Buttazzo, 1997). Schedulers are discussed in more detail in Chapter 2.

Previous studies have demonstrated that compared to pre-emptive schedulers, time-triggered co-operative (TTC) schedulers have a number of desirable features, particularly for use in safety-related systems (Bate, 2000; Nissanke, 1997; Ward, 1991; Allworth, 1981). However, if designed without due consideration of the task durations, a TTC system is likely to prove extremely unreliable. Hence, with the level of sophistication of modern embedded systems, such as those found in the automotive industry, developers need design techniques that can sustain the technical and commercial development constraints they face. Furthermore, in most cases the design of an embedded system requires a range of different expertise for a successful design. For instance, in the design of a steer-by-wire application the design team not only require software and hardware expertise; they also require knowledge from different engineering fields such as digital signal processing and control and instrumentation. Therefore, to ensure that valuable experience and knowledge is present during the development of these complex systems, what is needed is a means of 'recycling design experience' i.e. a means of facilitating the reuse of solutions from previously successful TTC designs.

In recent years software reuse has been seen by many as an important factor in improving software development productivity and quality (Mili et al., 2003; Voros et al., 2003). These observations have been supported by empirical evidence (Basili et al., 1996; Moser and Nierstrasz, 1996; Sparks et al., 1996). Design patterns have emerged in the software community over the last two decades as a means of facilitating design reuse or simply - 'design recycling' (Fowler, 2003). The use of design patterns has been observed to have a positive impact on the way software is developed. They encourage both experienced and inexperienced developers to reuse best practices and they also allow for good communication among members of a software development team (Beck et al., 1996).

This thesis reviews previous and current work on patterns. In particular a collection of patterns for time-triggered embedded systems (PTTES), assembled in the ESL at the University of Leicester, is reviewed. The thesis explores how this collection of patterns can be used to support automated code generation for the development of reliable time-triggered embedded systems. To do this the thesis first seeks to address the challenges of pattern-based code generation. The next section gives an overview of this thesis's contributions.

1.3 Overview of thesis contributions

This thesis seeks to make the following contributions:

- It presents the first assessment of the effectiveness of the PTTES collection.
- It describes two new PTTES patterns (HEARTBEAT LED and ERROR LED).
- It develops a novel approach to pattern-based code generation that utilises a meta-model for the PTTES collection to create the architecture of a potential CASE tool.
- It describes the design of the meta-model itself.
- It describes the design of a CASE tool that is based on the above meta-model.
- It describes and evaluates a prototype implementation of the CASE tool (PTTES Builder).

1.4 Thesis structure

The thesis is divided into five parts. Part I consists of Chapter 1 (this chapter) that gives an introduction to the thesis.

Part II consists of four chapters that present the literature review. Chapter 2, the first chapter in this part, gives an overview of an embedded system's architecture and the design constraints faced by developers of these systems. The focus is on small reliable embedded systems with a time-triggered architecture. Chapter 3 discusses the origin of design patterns from the field of architecture and their adoption in the software community. Chapter 4 reviews a collection of 'Patterns for Time-Triggered Embedded Systems' (PTTES) developed in the Embedded Systems Laboratory (ESL) at the University of Leicester. Chapter 4 also highlights the challenges of applying these patterns 'manually'. Based on the experience of creating two new patterns for the PTTES collection, Chapter 4 goes on to discuss the strengths and weaknesses of the pattern creation process. Chapter 5 reviews previous work involving both traditional and pattern-based code generation.

In Part III, Chapter 6 discusses the challenges of pattern-based code generation and proposes a different approach based on a pattern language that utilises an intermediate representation of patterns. Chapter 7 puts this approach to test by applying it in the development of a pattern-based code generation CASE tool based on the PTTES collection described in Chapter 4. Chapter 8 illustrates the application of this CASE tool by describing its use in the development of a non-trivial embedded application.

Part IV of the thesis assesses the effectiveness of the CASE tool in an empirical study described in Chapter 9. Chapter 10 presents another study aimed at assessing the extensibility of the CASE tool i.e. its ability to support the evolution of the PTTES collection.

Part V concludes the thesis by giving a summary of the thesis and its contributions in Chapter 11. A discussion of ongoing and future work in this area is also given in Chapter 11.

PART II: LITERATURE REVIEW

This part of the thesis reviews background material to the research presented in this thesis. Three areas are of particular interest: i) software and hardware architectures for embedded systems with severe resource constraints and a requirement for highly-predictable behaviour. This is covered in Chapter 2; ii) software design patterns (Chapters 3 and 4) and iii) automated code generation (Chapter 5). The literature review analyses the work in these three areas and outlines the gaps this research aims to fill.

2 The Architecture of an Embedded System

This chapter gives an overview of the hardware and software architecture of an embedded system. In going with the theme of the thesis, the primary focus of this chapter is on software architectures that have severe resource constraints and high reliability requirements. Some of the challenges of developing these architectures are also discussed.

2.1 Terminology

This section defines key terminology used in this chapter and the remainder of the thesis.

- **Context switch:** The process of ending a task execution and starting a new one. During this process current information is stored away for later retrieval and information relating to the new task is retrieved from memory. Context switching takes up processor time and consequently reduces the available computing time. It is therefore regarded as a system overhead (Cooling, 2003, p. 202).
- **Distributed system:** An embedded system with two or more processors or microcontrollers communicating with one another. For example, a brake-by-wire system commonly found in modern luxury vehicles may have an actuator connected to a microcontroller on each of the vehicles wheels (Hedenetz and Belschner, 1998).
- **Earliest deadline first scheduling (EDF):** This rule stipulates that priorities are assigned to tasks dynamically and are inversely proportional to the deadlines of their current requests. At any instant, the task with the highest priority and yet unfulfilled request will be executed (Liu and Layland, 1973, p. 55).
- **Jitter:** The variability of the transmission time i.e. maximum transmission time - minimum transmission time. The transmission time is the time delay between presenting a message to the sender's interface and receiving it at the receiver's interface. Embedded applications tend to be sensitive to jitter. Jitter is therefore an important parameter in the development of an embedded system (Bannatyne, 1998; Torngren, 1998).
- **Node:** A physically independent processor or microcontroller in a distributed embedded system.
- **Polling:** The process by which the status of an external device is scanned at regular intervals by the system it is connected to (Cooling, 2003, p. 209).
- **Rate Monotonic scheduling (RM):** This scheduling assigns priorities to tasks according to their request rates, independent of their run-times. Tasks with higher

request rates get assigned higher priorities (Liu and Layland, 1973, p. 50).

- **Task overrun:** A task overrun occurs when a task exceeds its predicted execution time. If such a situation arises and is not aborted, a domino effect on any subsequent tasks could occur (Buttazzo, 2005).

2.2 Overview

Embedded systems are composed of hardware and software components. As such a characteristic of embedded systems is the interaction between their hardware and software components. Because of this, the hardware and software of these systems should never be designed in isolation (Mooney and Micheli, 2000; Ernest, 1998; Balarin et al., 1997). Figure 2-1 is an overview of the hardware and software composition of an embedded system.

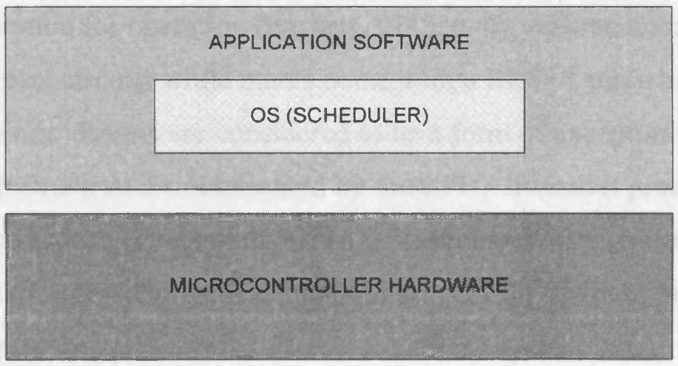


Figure 2-1 An overview of the hardware and software composition of an embedded system

The remainder of this chapter gives an overview of the hardware architecture of an embedded application and thereafter discusses software architectures.

2.3 Hardware architecture

A processor (or microcontroller) is fundamental to the operation of an embedded system. It provides the hardware platform on which such systems are built (Heuring and Jordan, 1997). In Section 1.1 it was noted that the main components of a microcontroller are its CPU, input and output ports and memory - these are important resources used to store and move large amounts of data and execute tasks as fast as possible during the lifetime of an embedded application.

The choice of processor (or at least the family of processors) to be used must be made at an early stage of development as it has an impact on the design of an embedded application. The choice of processor not only has an impact on the CPU load but also the available on-chip resources (e.g. ROM and RAM), programming language and the development tools used. Modern microcontroller families such as the 8051 offer some flexibility as the CPU is generally the same for a given microcontroller family thus the investment in software and development tools is not lost by selecting different members of that family (Morton, 2001, p. 18).

Similar to any synchronous sequential digital circuit, a microcontroller requires a system clock to operate (Baron and Higbie, 1992, p. 9). Whereas the microcontroller may be regarded as the 'brains' of an application, the clock frequency can be viewed as the 'heartbeat'. The clock frequency impacts other aspects of the design such as task execution rates and power dissipation. It is therefore very important that an appropriate system clock is implemented as any unexpected variation in frequency would have repercussions on the operation of the application. There are a wide range of design solutions for system clocks that can be used in line with design requirements, for instance, ceramic resonators are a least-expensive solution whereas crystal oscillators are generally a common but expensive solution. Morton (2001, pp. 384-386) discusses system clocks in more detail.

A microcontroller begins to execute a program immediately following a reset. A reset routine defined in the microcontroller is required to execute at start up as a means of initialising the hardware in preparation for operation (Barnett, 1995, p. 4). Some microcontrollers come with built in internal reset circuits while others come with a RESET pin which is used to connect to an external reset circuit. Resets are considered to be a form of exception because when they are detected the next instruction is not executed by the CPU. Instead it processes the exception. Resets can therefore be used to detect faults by using techniques such as watchdog timers that reset the microcontroller unless the program periodically notifies the watchdog (Morton, 2001, p. 387; Barnett, 1995, pp. 4-5).

Figure 2-2 is a hardware schematic of a simple 'flashing LED' embedded application. The application is composed of a simple reset circuit made up of a capacitor and resistor connected to Pin 9 of the applications microcontroller. In addition, the application has an external ceramic resonator connected to Pins 18 and 19. The flashing LED is connected to Pin 6.

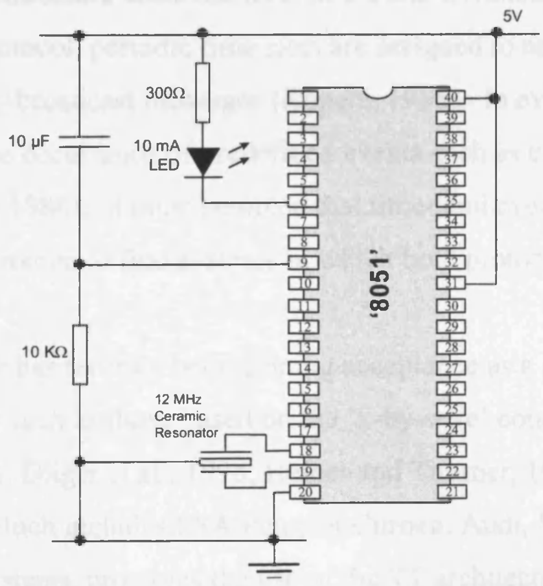


Figure 2-2 Hardware schematic of a 'flashing LED' embedded application

Microcontrollers are programmed by machine code that is stored in ROM. Developers tend to first write program code in a high-level programming language such as C, which is then translated into machine code using a compiler. Section 5.1 gives an overview of this process. A program called a loader is then used to load the machine language into ROM (Lippiatt, 1981, pp. 18-20). Although the cost of embedded hardware is generally low, development tools tend to be costly. In an attempt to keep development costs at a minimum, manufacturers tend to specialise in developing embedded applications for a particular microcontroller family (Eggermont, 2002, pp. 50-53; Debardelaben et al., 1997; Orlikowski, 1993). As a result they avoid purchasing licenses for different sets of tools. Furthermore, training costs for developers are kept low as they only get trained for one microcontroller family.

For extra functionality, embedded designs generally include additional hardware components such as UARTs and ADC circuits or have distributed designs. If implemented appropriately a distributed embedded system may help increase the reliability of safety-critical applications such as planes that generally have redundancy. Redundancy is the ability of a system to maintain functionality in the event of a component failure by having backup components that perform duplicate functions. Tanenbaum (1994) discusses the advantages of implementing distributed designs in more detail.

2.4 Software architecture

There are two fundamental software architectures that are generally implemented in embedded systems: event- and time-triggered architectures.

In a time-triggered (TT) architecture, all actions are derived from the progression of a globally synchronised clock (accessible to all nodes in a distributed system). In a distributed implementation, the TT architecture takes the form of a Time Division Multiple Access (TDMA) protocol. In the TDMA protocol, periodic time slots are assigned to each node at design time and are used to periodically broadcast messages (Kopetz, 1988). In event-triggered systems, all actions are derived from the occurrence of predefined events such as external interrupts (Bannatyne, 1998; Kopetz, 1988). It must be noted that time- and event-triggered architectures are not exclusive as it is common to find systems in which both protocols coexist (Pop et al., 2002).

The TT architecture has recently been gaining acceptance as a generic architecture for highly dependable systems such as those based on the 'x-by-wire' concept in the automotive industry (Blanc et al., 2004; Dilger et al., 1998; Heiner and Thurner, 1998). Also, the Time-Triggered Group (TTG), which includes PSA Peugeot Citroen, Audi, Volkswagen, Honeywell and Delphi Automotive Systems, promotes the use of the TT architecture across industries, e.g. the aerospace, automotive and railway industries, where requirements for safety-critical applications have to be fulfilled at low costs (TTA-Group, 2006). In the ESL, previous research has described how TT techniques can be applied in various automotive applications (Ayavoo et al., 2005; Short et al., 2004a), a wireless electrocardiogram (ECG) monitoring system (Phatrapornnant and Pont, 2006) and various control applications (Bautista and Pont, 2006; Edwards et al., 2004).

The TT architecture has been shown to be more suitable than the alternative event-triggered architecture as it satisfies the requirements of safety critical communications systems by being deterministic and predictable (Karlsson, 2002). The architecture is also compose-able i.e. it allows independent coding, testing, validation and certification of nodes thereby supporting the co-ordination of development teams. Furthermore, this architecture allows the behaviour of an overall system to be predicted by assessing the properties of the subsystem (Bannatyne, 1998).

2.4.1 Schedulers

In Section 1.2, it was noted that among other things the reliability of an embedded application relies on the careful scheduling of its tasks in order to meet deadlines (Liu and Layland, 1973, p. 46). An algorithm known as a scheduler manages the execution schedule and resource utilisation of these tasks. The architecture of an embedded application is characterised by this underlying scheduling algorithm (Locke, 1992).

A 'super loop' is the most basic form of a scheduler that may be used in very simple embedded systems in which tasks are only programmed to execute once or periodically in a predefined sequence. Because of its simplicity, a super loop tends to be used in applications that have limited memory and CPU resources but where accurate timings are not a key requirement. Super loop schedulers are therefore not appropriate for complex embedded systems with high reliability or time critical requirements, instead they are used in much simpler scheduling algorithms such as the cyclic executive described in Section 2.4.1.2. In order to meet complex functional requirements, high reliability embedded applications require scheduling algorithms that are far more sophisticated than the super loop.

The following subsections describe the pre-emptive and co-operative schedulers - two scheduling algorithms commonly used in embedded applications.

2.4.1.1 Pre-emptive scheduler

In a pre-emptive scheduler, tasks can be interrupted during their execution i.e. pre-empted by other scheduled tasks. A pre-empted task is placed in a queue and resumes its execution at its next allotted time slot, exactly where it previously left off (Cooling, 2003, pp. 201-202; Liu and Layland, 1973, p. 48). Figure 2-3 shows the execution of a Task D pre-empted during its execution by an interrupt service routine (ISR). Once the ISR has been executed, Task D continues its execution from where it was pre-empted.

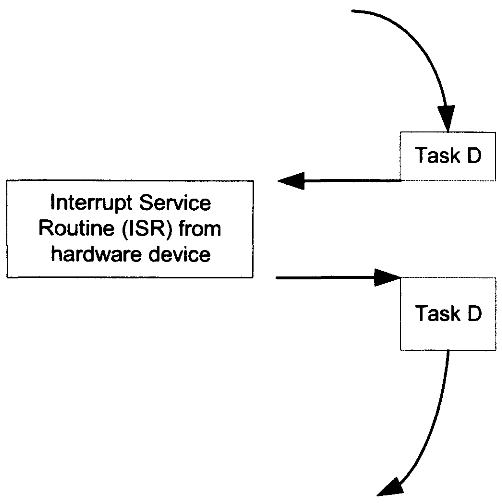


Figure 2-3 A pre-emptive task execution interrupted by an ISR (adapted from: Kalinsky, 2001)

Liu and Layland (1973) have previously described 'priority-driven' pre-emptive schedulers in which tasks are allocated priorities in order to meet predefined execution orders. This means that whenever there is a request for a task that is of higher priority than the one currently being executed, the running task is pre-empted and the newly requested task is started.

If a pre-emptive execution schedule remains fixed during the life of a program then it is classed as a fixed (or static) priority scheme. In this scheme, tasks are assigned priorities using the RM priority assignment (Liu and Layland, 1973, p. 50). Alternatively, a pre-emptive task execution schedule that can be changed during program run time is said to be a dynamic priority scheme. The task priorities can be altered by an external event or running task. This scheme has tasks configured using the EDF assignment rule (Liu and Layland, 1973, p. 55).

Pre-emptive scheduling enables external hardware devices to either be polled or interrupt driven if need be and their information acquired and processed immediately. However, a pre-emptive design has to consider the tasks that can be pre-empted and those that can pre-empt others and those that cannot be allowed to pass information to other tasks by writing or reading shared data. Furthermore, a developer has to devise an algorithm suitable for passing information among tasks e.g. message queues or semaphores, this complexity makes the implementation of pre-emptive schedulers a challenging task.

2.4.1.2 Co-operative or non pre-emptive scheduler

In a co-operative scheduler, a task's schedule is explicitly predefined before run time to produce a feasible execution schedule that is fixed for the entire history of a program (Shaw, 2001, p. 19; Bate, 1999, p. 51; Locke, 1992, p. 39). A key feature of a co-operative scheduler is that tasks are executed to completion uninterrupted in accordance with their predefined schedule; this is therefore sometimes referred to as non-pre-emptive scheduling (Bate, 1999, p. 51).

A cyclic executive (Kalinsky, 2001; Locke, 1992; Baker and Shaw, 1989) is an example of a co-operative scheduler based on a time-triggered architecture. Figure 2-4 is an illustration of a cyclic executive execution model. Automotive (see, Ayavoo et al., 2004) and medical monitoring applications (see, Phatrapornnant and Pont, 2004) are examples of systems in which a time-triggered co-operative (TTC) scheduler may be implemented.

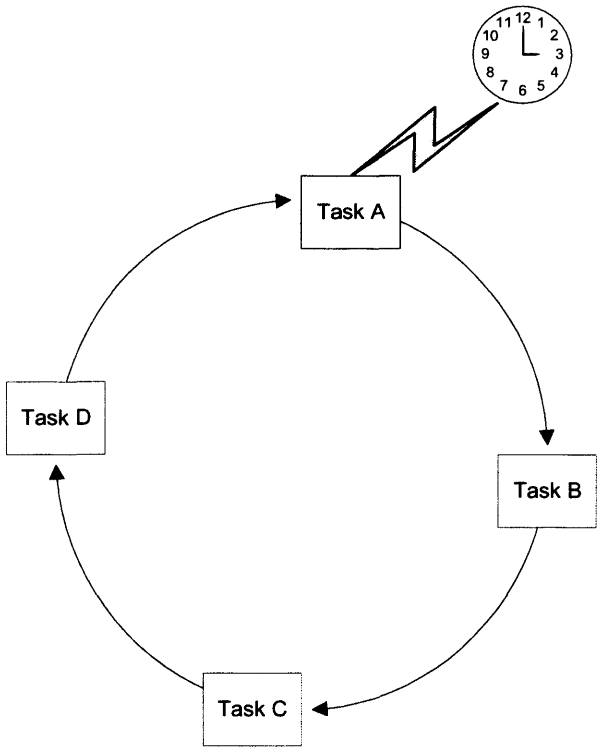


Figure 2-4 A 'time-triggered' cyclic executive execution model (adapted from: Kalinsky, 2001)

The co-operative scheduler has many attractive features, it is simple and straightforward to implement in comparison to the pre-emptive scheduler. Co-operative tasks will always execute in their pre-allocated slots, so jitter levels are reduced (Buttazzo, 2005). The non-pre-emption ensures executive overhead is kept low, as there are no unexpected context switches. Furthermore, tasks can communicate with one another through shared data without special concern about data integrity because every task always runs to completion before another task begins running. There is therefore no danger of tasks getting inaccurate data from other tasks. This reduces resource overheads as there is no requirement to protect the integrity of shared data structures or other resources by the provision of mutual exclusion algorithms such as semaphores or monitors.

Co-operative schedulers therefore pave the way for simple but yet efficient software (Locke, 1992, p. 42). The pre-run-time scheduling of co-operative tasks also means that during execution a system's timing behaviour can be easily predicted (Shaw, 2001, p. 22). In the context of scheduling, predictability is the ability to state at any time during execution the task that will be executed next, this allows analysis that demonstrates whether timing requirements are met (Bate, 1999, p. 26). It is therefore possible to predict the entire future history of a co-operative system provided task overruns do not occur (Locke, 1992, p. 42).

Although the co-operative scheduler may seem like an attractive option, a number of limitations of this algorithm have been noted by various researchers. Most importantly, the scheme assumes that accurate estimates of execution times for each task are available. These however are not easy to ascertain (Shaw, 2001, p. 23). For instance, a co-operative system may become fragile during overload situations since a task exceeding its predicted execution time could generate (if not aborted) a domino effect on subsequent tasks, causing their execution to exceed their deadline. It has also been argued that systems using this method are inflexible and difficult to maintain (Bate, 1999, p. 52). For example, creating a new task or altering a task rate is more than likely to affect an entire execution schedule, resulting in a complete redesign of the scheduling plan (Buttazzo, 2005; Shaw, 2001, p. 24). Furthermore, interrupts from external hardware devices cannot communicate directly with tasks, they would normally have to be polled to interact with the scheduler tasks (Kalinsky, 2001).

2.5 Discussion

The fact that tasks can be pre-empted (by other tasks) at any time during run-time makes the pre-emptive scheduler more flexible and responsive in comparison to the co-operative scheduler (Locke, 1992). However, these features come at a premium; a pre-emptive scheduler consumes lots of RAM and processor power through 'context switching' and (as discussed in Section 2.4.1.1) is generally more complex to design and implement than a co-operative scheduler. Furthermore, an important requirement of embedded systems, in particular safety critical systems, is predictable behaviour. A system based on a co-operative scheduler exhibits more predictable behaviour than one based on a pre-emptive scheduler (Albert, 2004; Kopetz, 1988).

However, a concern often raised about the co-operative scheduler is that long tasks can impact the predictability of a system. A major implication of this is that a co-operative scheduler may not respond to changes in its environment if the duration of the tasks have not been carefully considered at design time (Allworth, 1981). Therefore, despite the design complexity of a pre-emptive scheduler, it has previously been argued that priority-based pre-emptive scheduling is a better scheduling solution than co-operative scheduling (Buttazzo, 2005). In particular, the static priority scheme has been observed to be more suitable than a co-operative design (Bate, 1999; Locke, 1992). In his thesis Bate (1999) makes a case for static priority scheduling in safety critical systems. He argues that the principal difference between the co-operative and static priority-based approach is that the co-operative approach is deterministic (i.e. the ability to state before execution commences the run time ordering of tasks) while the static priority-based approach is predictable. It can however be argued that a co-operative scheduler also exhibits predictability provided worst-case transmission time and jitter are known and accounted for at design time.

Building on previous work in this area, researchers in the ESL have demonstrated design techniques that address the challenges of designing predictable embedded systems based on a co-operative scheduler (Phatrapornnant and Pont, 2006; Hughes et al., 2005; Key et al., 2003; Pont, 2003; Pont and Ong, 2002). Pont and Banner (2004) have demonstrated that using a TTC scheduling approach can provide a relatively simple and robust scheduler. By contrast, the increased complexity of even a comparatively simple pre-emptive environment results in a much larger code framework (Pont and Banner, 2004).

In similar research, Xu and Parnas (2000) have expressed their preference for static as opposed to dynamic schedulers. Xu and Parnas (2000) argue that with static scheduling, context switching is greatly reduced therefore run time resources such as power and memory are preserved. Furthermore, task deadlines can be predicted at design time therefore making static scheduling the approach of choice for safety critical applications as opposed to dynamic scheduling where arrival times and deadlines cannot be predicted before or during run-time (Xu and Parnas, 2000; Xu and Parnas, 1990). Meeting deadlines and achieving high resource utilisation are the two main challenges of task scheduling in embedded applications. Therefore, where appropriate, a static scheduling algorithm such as that provided by a TTC scheduler and static priority-based pre-emptive scheduler is generally preferred (Fredriksson et al., 2003; Pop, 2000).

Xu also notes that a major benefit of static scheduling is the ability to predict all the possible cases of the actual time-critical software code's timing behaviour through rigorous inspection and verification (Xu, 2003; Xu and Parnas, 1990). Nakata et al. (2006) have successfully demonstrated the applicability of inspection and verification techniques, described by Xu (2003), to ensure that the re-scheduling of a system's tasks does not impact its external behaviour.

2.6 Chapter conclusions

This chapter has given an overview of the hardware fundamentals of an embedded application and discussed how this impacts the design of an embedded system.

The software architecture of an embedded application has also been discussed with respect to pre-emptive and co-operative schedulers. Some of the challenges of implementing these scheduling algorithms have also been discussed. It has been argued that the choice of scheduler generally depends on the system requirements. However, if developing a safety-critical system this may not be an appropriate way of deciding which software architecture to implement. Instead, as a TTC scheduler provides a more deterministic and predictable architecture (if developed correctly), it may be appropriate that a developer first establishes whether a TTC scheduler meets the design requirements. By so doing, the decision of which software architecture to implement is primarily based on system reliability.

However, even though TTC schedulers can provide robust software architectures, it is clearly not easy to implement as task durations and any potential task overruns need to be known at design time in order to guarantee reliability. Techniques that can ease the development of applications with TTC architectures are therefore required.

3 The Origin of Design Patterns

This chapter gives an overview of the origin of design patterns. The adoption of design patterns in software engineering is also reviewed.

3.1 Christopher Alexander

The patterns[1] concept stems from the work of the architect - Christopher Alexander. Alexander graduated from Cambridge University, where he studied Mathematics. He later took his doctorate in Architecture at Harvard University (the first PhD in architecture ever awarded at Harvard). In his doctorate thesis, later published in 1964 as a book, Notes on the Synthesis of Form (Alexander, 1964), Alexander proposed a rigorous approach to design that earned him instant recognition and the first Gold Medal award for research by the American Institute of Architects (Alexander, 2005).

3.1.1 Patterns in architecture

In his work, Alexander argued that contemporary methods in architecture, urban planning, and construction failed to generate products that satisfied user requirements due to the growing complexity of design problems. At the time there was also a vast amount of useful information and specialist experience that was not archived and therefore unreachable or misused by designers (Alexander, 1964, p. 4). Alexander believed that this resulted in inappropriate design models that led to solutions that did not meet user requirements. Therefore, in an attempt to improve contemporary design at the time, in Notes on the Synthesis of Form, Alexander proposed, at the time ground-breaking, work on design processes based on structure or what he called 'form'. Through this work Alexander attempted to support top-down design methods (Alexander, 1964, p. 15).

Alexander never quite completed his work on form, he instead went onto describe patterns, an entity that, *"...describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"* (Alexander et al., 1977, p. x). Patterns articulate a solution to a recurring problem in a given context; the relationship among a problem, its context and solution is described as, *"...a three-part rule, which expresses a relation between a certain context, a problem, and a solution"* (Alexander, 1979, p. 247). Overall, Alexander's intention was to conserve the knowledge and experience of architects into a set of patterns that he believed could, *"...provide a complete working alternative*

[1] Patterns and design patterns are used interchangeably throughout the thesis.

to [our] present ideas about architecture, building and planning... " (Alexander, 1979, p. ii). It would appear, from his early work that Alexander hoped that this 'alternative' approach to architecture would supersede the ideas practised at the time.

Alexander's initial work on patterns was published in three volumes. In the first volume, The Timeless Way of Building, the rationale for the patterns concept was presented as a modern theory of planning and building (Alexander, 1979). In the second volume, A Pattern Language, Alexander described what he called 'a pattern language', a language consisting of patterns and their relationships among various architectural (and planning) problems and solutions (Alexander et al., 1977). In essence a pattern language describes a collection of related practices (in the form of patterns), that allow bigger and complex problems to be solved *"... no pattern is an isolated entity. Each pattern can exist in the world, only to the extent that it is supported by other patterns: the larger patterns in which it is embedded, the patterns of the same size that surround it, and the smaller patterns which are embedded in it"* (Alexander et al., 1977, p. xiii).

In A Pattern Language Alexander described how the world, in his own view, should be divided into independent nations, and nations into cities and towns, and how buildings and streets should be arranged, right down to details of how buildings should be decorated. To illustrate this, 253 coherent and detailed patterns for architectural and urban planning were described in A Pattern Language (Alexander et al., 1977). Alexander believed that professional designers and architects should not have the responsibility of designing communities. Instead Alexander believed that the occupants of these communities, with the support of A Pattern Language, should be capable of designing their own communities as only they understand their needs better. A Pattern Language is divided into three parts. The first part defines how towns and communities should be designed (Alexander et al., 1977, pp. 10-457). An example of a pattern in this category is INDEPENDENT REGIONS, summarised in Figure 3-1. This pattern suggests the demarcation of the world into independent regions with a population of between two and ten million inhabitants each. Other related patterns in this category are THE DISTRIBUTION OF TOWNS, CITY COUNTRY FINGERS, AGRICULTURAL VALLEYS, LACE OF COUNTRY STREETS, COUNTRY TOWNS and THE COUNTRYSIDE.

The second part of the language describes design solutions for both individual and groups of buildings (Alexander et al., 1977, pp. 467-931). An example of one such pattern is SMALL MEETING ROOMS that describes how meeting rooms can be designed in order to allow participants to get the most out of their meetings (Alexander et al., 1977, pp. 712-716). The rationale behind this pattern is that the larger meetings are the less people get out of them. The solution presented by SMALL MEETING ROOMS therefore stipulates that at least 70% of meeting rooms should be small i.e. allow 12 or less occupants because the smaller the group the easier it is for all participants to get involved. The pattern goes on to specify that meeting rooms should be evenly distributed within a building so that meetings are held close to participants' offices. By so doing, discussions that begin in a meeting room can continue into participants' offices, thereby promoting a good working environment (Alexander et al., 1977, p. 715).

The third and last part of the language presents patterns that specifically deal with the structure and construction of buildings (Alexander et al., 1977, pp. 939-1166). ROOF LAYOUT, FLOOR AND CEILING LAYOUT, THICKENING THE OUTER WALLS, COLUMNS AT THE CORNERS and FINAL COLUMN DISTRIBUTION are some of the patterns intended to help designers work out the complete structural layout of a building before it is constructed.

THE DISTRIBUTION OF TOWNS

Context

This is a pattern intended for the even or fair distribution of a population in a region. It is applicable to town or city planning.

Problem

If a region's population is not uniformly distributed, it will go to ruin because its population is not where it needs to be, to take care of it.

Solution

Encourage a birth and death process for towns within the region, which gradually has these effects:

- The population is evenly distributed in terms of different sizes - for example, one town with 1,000,000 people, 10 towns with 100,000 people each, 100 towns with 10,000 people each, and 1,000 towns with 100 people each.
- These towns are distributed in space in such a way that within each size category the towns are homogeneously distributed all across the region.

This process can be implemented by regional zoning policies, land grants, and incentives that encourage industries to locate according to the dictates of the distribution.

Figure 3-1 The Distribution of Towns summarised from A Pattern Language (Alexander et al., 1977, p. 16)

From the summary of THE DISTRIBUTION OF TOWNS presented in Figure 3-1, it is observed that patterns are descriptive and hence describe solutions in an abstract form. This abstract form therefore leaves the actual pattern implementation to the designer. As a consequence, since a pattern language *"...is in truth a network, there is no one sequence which perfectly captures it..."* (Alexander et al., 1977, p. xviii), a designer is expected to tailor the implementation of a pattern or collection of patterns in whatever order or way that matches the context of the problem(s). Patterns therefore do not impose any restrictions on the way in which they are applied, thereby allowing designers to maintain their individual creativity in problem solving.

To manifest his ideology, in the third volume of his work: The Oregon Experiment (Alexander et al., 1975), Alexander presented an experiment in which he attempted to provide a comprehensive real life example of applying his patterns. In this experiment, patterns were used to help expand the University of Oregon (in the USA) in order to sustain its growth (Alexander et al., 1975). However, the University of Oregon presented, *"... a very special kind of community. Unlike most communities, it [had] a single owner (The State of Oregon) and a single, centralised budget"* (Alexander et al., 1975, p.3). Alexander did not believe in centralised planning, instead he believed that within a region, each city and town should be responsible for its own land; and within each city and town, each community or individual should similarly be responsible for their own habitat (Alexander et al., 1977). Nevertheless, despite these perceived discrepancies, Alexander was able to adapt his pattern language in order to meet the requirements of this 'special kind of community' (Alexander et al., 1975, p. 3). This ability to adapt patterns to different contexts is desirable to designers as it provides them with flexibility in a design setting.

3.1.2 Have patterns been accepted in architecture?

In his key note address to the 1996 Association for Computing Machinery (ACM) Conference on Object-Oriented Programs Systems Languages and Applications (OOPSLA), Alexander admitted that he had initially believed that he would be able to influence the world that the use of patterns could help develop better communities (Alexander, 1996). But he conceded that he had practically failed to convince people to take to patterns even though his patterns had influenced, *"...a few thousand buildings..."* (Alexander, 1996).

However, interestingly enough, anyone reviewing A Pattern Language is more than likely to find a pattern they can relate to. For instance, THE DISTRIBUTION OF TOWNS (summarised in Figure 3-1) is a potential solution to rural-urban planning in the developing world. In Zambia (a developing country in southern Africa) for instance, one of the problems faced by urban planners is the 'rural-urban drift'. Rural-urban drift is the term given to the mass exodus of people from rural areas (e.g. farms, villages and countryside) to urban areas such as towns and cities mainly in search of employment, education and in general a much better life compared to that offered in the rural areas. Not only does the resulting over population in urban areas cause a strain on urban resources, it also leaves the rural areas depopulated and under maintained. THE DISTRIBUTION OF TOWNS presents a solution to this problem by suggesting an appropriate statistical distribution of towns (by size) and an even spread of towns across regions. However, though this may seem an ideal solution to rural-urban drift, the cause of this problem arguably has little to do with the planning of rural or urban areas. The underlying cause of this problem, faced not only by Zambia but also other developing countries, is the lack of financial resources to make the rural areas more habitable (Saasa and Carlsson, 2002).

Over the years Alexander's campaign to influence the use of patterns in architecture design has led to him being isolated from the, *"...mainstream commercial architectural community..."* where it has eventually become clear that he did not succeed in replacing contemporary architecture ideas and practices (Lea, 1994). The architectural ambitions expressed in his work have been seen as alienating from the practice and as a result have led to some attacks on him. For instance, the New York Times, in an article entitled, 'Architecture's

Irascible Reformer', described Alexander's work as, *"...a quixotic campaign of messianic ambition: to heal the world by reforming the way it builds"* (Eakin, 2003). Despite these negative views, it can be argued that architects and civil engineers apply certain aspects of Alexander's work unsuspectingly. For instance, buildings are generally built on foundations. The process and the design decisions that go with building a foundation e.g. the height, location and capacity of the building, could be described as design patterns.

While some of his peers may have dismissed his work, it is obvious that Alexander is today well-respected in other circles of society. Not least, the fact that he is today Professor Emeritus at the University of California where he devotes most of his time to writing and presenting invited talks. Alexander's work has also been recognised by the Royal Family in the United Kingdom where the Prince of Wales has previously invited him to serve as a Trustee of The Prince of Wales Institute of Architecture. Furthermore, in 1996 Alexander was elected Fellow of the American Academy of Arts and Sciences for his contributions to architecture (Salingaros, 2006).

3.2 Software patterns

Despite his perceived failure to influence the adoption of patterns in architecture design, Alexander has over the years gained devoted followers, more so from the software community where the use of patterns has been welcomed. The adoption of patterns in the software community has been influenced by the need for software reuse. Software reuse was one of the goals of the architects of the object-oriented methodology; programmers created libraries of reusable code and these consisted of classes that could be reused across applications (Rogers, 1997).

The first use of patterns in software development can be traced back to Cunningham and Beck (1987) who had been using the Smalltalk programming language for designing Windows based user interfaces. They adopted Alexander's techniques as the basis for a small pattern language (consisting of five patterns) intended to provide guidance to novice Smalltalk programmers. Following a promising outcome, they presented their results at OOPSLA '87 and outlined their adaptation of Alexander's pattern language concept to object-oriented programming (Cunningham and Beck, 1987).

3.2.3 The Gamma pattern collection

Cunningham and Beck's (1987) work was subsequently built upon by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (today known as the Gang of Four) who, in 1995, published a book on general-purpose object-oriented software patterns: <u>Design Patterns: Elements of Reusable Object-Oriented Software</u> (Gamma et al., 1995). This is arguably the most influential book on software patterns published to date and was consequently merited with the Excellence in Programming Award by Dr Dobbs Journal in 1998.

Table 3-1 Format used to define patterns in the Gamma collection (Gamma et al., 1995)

Element name	Description
Pattern Name	A handle that can be used to describe a design problem, its solution and consequences in a word or two. It also contributes to the vocabulary of designers
Intent	What design issue or problem does it (the pattern) address?
Also Known As	Lists any alternative names the pattern may have
Motivation	A scenario that illustrates an example design problem and how it may be solved using the pattern
Applicability	Describes situations in which the design pattern can be applied?
Structure	A UML representation of the pattern's components
Participants	The classes and/or objects participating in the design pattern and their responsibilities
Collaborations	Describes how the participants (previous element) collaborate to carry out their responsibilities
Consequences	Describes the pattern's benefits, trade-offs and drawbacks
Implementation	Lists the pitfalls, hints or techniques for implementing the pattern (e.g. language specific issues)
Sample Code	Example code fragments that illustrate how one might implement the pattern in C++
Known Uses	Examples of instances where the pattern has been or can be applied
Related Patterns	Lists other patterns that are closely related to the described one and describes their differences and similarities

In their book, Gamma et al. (1995) present 23 design patterns. Each pattern, *"...names, abstracts and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design...Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in view of other design constraints and the consequences and trade offs of its use"* (Gamma et al., 1995, pp. 3-4). The Gamma collection is organised into three categories based on the pattern attributes: creational patterns are concerned with object creation, structural patterns address the composition of classes and objects and behavioural patterns are concerned with the manner in which objects interact. Similar to Alexander, Gamma et al. (1995) did not describe new or unproven design solutions, instead they presented designs that had never been documented before but had previously been applied time and time again to common problems (Gamma et al., 1995, p. 2).

Although most of the patterns in the Gamma collection include example code to illustrate their implementation, it is important to note that patterns are not code. *"Patterns are half-baked – meaning you always have to complete them yourself and adapt them to your own environment"* (Fowler, 2003), i.e. patterns should be abstractly written in order to allow a user to easily adapt them to their own requirements. OBSERVER (summarised in Figure 3-2) is an example of a pattern from the Gamma collection, it is commonly used to design graphical user interfaces – it helps separate the presentational aspects of an application from the underlying application data. Therefore, classes defining application data and presentations can be reused independently, thereby promoting software reuse (Gamma et al., 1995, pp. 293-303). Table 3-1 lists and describes the elements that define the form of the patterns in the Gamma collection.

OBSERVER

Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Also Known As

Dependents, Publish-Subscribe

Motivation

A common side effect of partitioning a system into a collection of co-operating classes is the need to maintain consistency between related objects. You do not want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

The key objects in this pattern are subject and observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronise its state with the subject's state.

Applicability

Use this pattern when:

- An abstraction has two aspects, one dependent on the other.
- A change to one object requires changing others and you do not know how many objects need to be changed.
- An object should be able to notify other objects without making assumptions about who these objects are.

Participants

Subject, Observer, ConcreteSubject and ConcreteObserver

Consequences

- The coupling between subjects and observers is abstract and minimal.
- Broadcast communication i.e. the notification that a subject sends need not specify its receiver.
- Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject.

Related Patterns

MEDIATOR and SINGLETON

Figure 3-2 A summary of Observer (Gamma et al., 1995, pp. 293-303)

Unlike A Pattern Language (Alexander et al., 1977) that is composed of interrelated design patterns that can be applied to solve recurring problems in architecture, the Gamma collection is not a 'pattern language'. This claim is supported by the authors who make it clear that their collection, *"...doesn't have any application domain-specific patterns. It does not tell you how to build user interfaces, how to write device drivers or how to use an object-oriented database. Each of these areas has its own patterns and it would be worthwhile for someone to catalogue those too"* (Gamma et al., 1995, p. 2).

3.2.4 The patterns community

It is arguable that the biggest patterns community is today rooted in the object-oriented software domain. This community includes the authors of Design Patterns: Elements of Reusable Object-Oriented Software (Gamma et al., 1995) and The Hillside Group that organises the world wide Pattern Languages of Programs (PLoP) conferences at which pattern enthusiasts and practitioners meet to share and discuss patterns. The Hillside Group was established under the auspices of Kent Beck and Grady Booch (The Hillside Group, 2005). In addition to the pattern conferences, several mailing lists and websites have been set up over the years and remain very active. This has allowed for the sharing and dissemination of patterns among both experienced and inexperienced pattern members around the world.

Patterns are not invented; they emerge from design knowledge (Fowler, 2003). Therefore, identifying and documenting patterns is the responsibility of domain experts. However, before a pattern is published it goes through a rigorous process of 'refinement' by senior members of the patterns community, this process is called Shepherding. Shepherding involves a series of alterations of a proposed pattern based on the reviews, comments or suggestions of a designated Shepherd. A Shepherd is a peer selected from within the patterns community based on their patterns experience. Following the Shepherding process, if the proposed pattern is considered worthy of publication, it is recommended for further scrutiny at a PLoP conference. PLoP conferences have several criteria that pattern papers should meet before they are published (The Hillside Group, 2005).

Rather than being presented by the individual authors, the patterns are discussed in writers' workshops. Richard Gabriel introduced writers' workshops at the first PLoP conference in 1994. The primary focus of these workshops is to ensure that the pattern under scrutiny communicates its ideas effectively to its users. It is therefore assumed that the pattern author is a domain expert (Rising, 1998, pp. 79-80). During these workshops, the pattern authors have the opportunity to incorporate all the comments and insights before presenting the patterns in their finished form. From inception a pattern therefore undergoes a crystallising process that seeks to refine it to a standard of quality common with all other patterns in the community.

Over the years, there have been many patterns that have come out of the PLoP conferences and the focus has gradually shifted from the use, assessment and refinement of individual object-oriented patterns, to the creation of complete pattern languages for various software domains. For example, the telecommunications domain has a collection of patterns that have been published by Linda Rising (Rising, 2001). These patterns deal with issues such as high-availability requirements and jitter guarantees in communications systems. Besides software development patterns, process and organisational patterns to support the software development process have also been written (e.g., Coplien and Harrison, 2004). Recently the patterns community has also been considering ways in which good design practices can be extracted from legacy code with the intention of documenting them in the form of design patterns, this process is known as 'pattern mining' (Garofalakis et al., 1999).

Beyond the patterns community, industry has also taken to patterns. For instance, Microsoft has 'Patterns & Practices' guidelines. These are recommendations for how to design, develop, deploy and operate applications for the Microsoft platform. Apart from providing technical guidance, these recommendations also come with source code that has been put to test in real-world applications (Microsoft, 2006). IBM also uses patterns within its organisation. They are referred to as 'IBM Patterns for e-business'. These are a collection of reusable assets meant to speed the development of web-based applications (IBM, 2006).

As the numbers of patterns and pattern collections for various software domains, have increased over the years, researchers have attempted to categorise patterns in relation to the software development process. For example, Riehle and Zullighoven (1996) have defined conceptual patterns, design patterns and programming patterns. A conceptual pattern is a pattern whose form is described by means of terms and concepts from its application domain. On the other hand a design pattern is a pattern whose form is described by means of software design constructs such as objects, classes, inheritance and aggregation. A programming pattern is a pattern whose form is described by means of programming language constructs; they can be viewed as low-level derivations of design patterns for a specific programming language.

Buschmann et al. (1996) have also attempted to categorise patterns by defining architectural patterns, design patterns and idioms. An architectural pattern defines the fundamental structural organisation of a system's sub-systems and design patterns define the actual sub-systems (or components) and relationships among them. An idiom is a low-level pattern specific to a programming language that describes how to implement sub-systems and their relationships using the features of the language.

Buschmann et al.'s (1996) categorisation of patterns is comparable to that of Riehle and Zullighoven's (1996). However, Buschmann et al.'s (1996) architectural pattern differs from Riehle and Zullighoven's (1996) conceptual pattern in that an architectural pattern is concerned with the structural organisation of a system's sub-systems whereas a conceptual pattern is concerned with the design analysis of a system's domain.

3.2.5 Do patterns enhance software development?

Though Alexander may not have succeeded in convincing his peers that patterns could be used to improve architecture design, the use of patterns has surprisingly gained acceptance in the software engineering community over the last two decades. However, like in architecture, there is yet to be any credible empirical assessment of their overall value to software development. In fact Schmidt (1995) feels that this is because a pattern cannot be verified or validated from a purely theoretical framework i.e. *"Patterns are validated by experience rather than by testing, in the traditional sense of 'unit-testing' or 'integration-testing' of software"*. Schmidt (1995) has also suggested that one way of being reassured of a patterns usefulness, is by referring to its 'Known Uses' attribute as the number of known uses can serve to indicate a patterns value i.e. the more times a pattern has been used the more likely it is valuable.

Nevertheless, in recent years there have been studies performed to asses the impact of patterns on software development. However, these assessments have focused on particular aspects of the software development process as opposed to the entire process. For instance, Prechelt et al. (2002) previously assessed the effect of using the Gamma collection of patterns on software maintenance using unpaid professionals as subjects. The results from the study suggest that patterns facilitate relevant maintenance tasks with fewer errors. In support of these results, another study performed by Torchiano (Torchiano, 2002) concluded that pattern-specific documentation affects the comprehension process during software development making it more structured and disciplined. Therefore, as software systems are known to evolve into much larger and complex systems through maintenance, it can be argued that the use of patterns does enhance the software development process to a certain extent.

In a replication of the study carried out by Prechelt et al. (2002), Vokac et al. (2004) used paid professionals as subjects in a real programming environment. Apart from drawing similar conclusions and increasing the experimental realism and applicability of the results from the Prechelt et al. (2002) study, Vokac et al. (2004) came up with new observations that indicate that some patterns are much easier to understand and use than others, thereby concluding that each pattern cannot be classified as either 'good' or 'bad' in general terms (Vokac et al., 2004). These observations could either suggest that certain patterns are more complex than others or not well presented thereby suggesting that they may not have been created well.

Despite the lack of credible empirical evidence showing the overall benefits of patterns, Beck et al. (1996) have previously observed that patterns do have a positive impact on the way a team develops software. In particular, the following advantages have been observed (Beck et al., 1996):

- Patterns encourage documenting and reusing best practices for both experienced and inexperienced software developers

- Inexperienced software developers can produce better designs with patterns
- Patterns enhance communication among members of a development team, from designers to maintainers, as they provide a common vocabulary

With the limited studies carried out to date on the usefulness of patterns, it is difficult to establish whether they do actually enhance software development. Furthermore, the few studies that have been preformed have mainly focused on the Gamma et al. (1995) patterns, it could therefore be argued that the results from these studies may not be applicable to other pattern collections. To this effect there is a need for empirical evidence that can clearly demonstrate the effectiveness of pattern collections on the software development process.

3.3 Chapter conclusions

This chapter has given an overview of design patterns and discussed their origin in architecture and adoption in the software field. Patterns describe a solution to a frequently recurring design problem that can be applied in different contexts. An important aspect of the application of patterns is that the intellectual decisions to be made lies with the user. Therefore, users bring innovation to the solution and new experiences can be used to further improve a pattern.

Despite the lack of empirical evidence that suggests that patterns are beneficial to software development, it has been argued that the use of patterns does have some obvious benefits to software development. In particular, patterns facilitate design reuse by providing developers with previously successful design solutions. Patterns also ensure best practice (for both experienced and inexperienced users) and provide users in a team setting with a common vocabulary. The fact that IBM and Microsoft, who are at present two of the largest software companies in the world, are using patterns in their organisations is also a testament to the usefulness of patterns. However, because of the tendency of corporations to protect their intellectual property rights, it is difficult to asses the impact of patterns in industry.

The next chapter presents a domain specific pattern collection and attempts to assess the usefulness of the collection to its domain users.

4 Patterns for Time-Triggered Embedded Systems[2]

In Chapter 2, it was argued that TTC schedulers provide predictable and therefore reliable software architectures for embedded systems. However, it was noted that designing TTC systems does pose certain challenges to developers if they are to meet reliability requirements. This chapter presents a collection of patterns that have been produced in the ESL to support the development of TTC applications.

4.1 The rationale for the PTTES collection

In Chapter 2, one concern raised about TTC designs is that failures tend to manifest themselves as task overruns. This concern is crucial as any co-operative system that has been designed without due consideration of task durations is likely to prove extremely unreliable as the scheduling approach employed must provide adequate control of task overruns commensurate with the reliability of the system (Bate, 1999, p. 43).

One possible way of tackling task overruns is to adapt the hardware to match the needs of TTC designs: in other words, to create a 'time-triggered processor' and write custom code for it. This is an approach that is currently being explored in the ESL (Hughes et al., 2005). Nevertheless, despite the promising prospects of producing custom hardware designs, at present this may not be an economically viable approach for hardware manufactures as off-the-shelf processors and microcontrollers remain popular amongst developers (Edwards, 2006, pp. 46-47; Gupta and Micheli, 1997). For instance, Bannatyne (2004) has predicted that the automotive sector will see a steady growth of microcontroller use in vehicles over the next decade. Besides the automotive sector, off-the-shelf processors are also common in other application areas such as wireless networks used in environmental monitoring (Martinez et al., 2004) and industrial applications such as automation robots (Apneseth, 2006).

There are a number of other alternative techniques that may be used to implement TTC designs (Pont and Banner, 2004):

- Using a faster processor, or a faster system oscillator to reduce the duration of long tasks
- Making use of an additional processor to obtain a multi-tasking capability
- Using 'time out' mechanisms to ensure that tasks complete within their allotted time
- Splitting long tasks (triggered infrequently) into shorter 'multi-stage' tasks (triggered frequently) so that the processor activity can be evenly distributed

[2] Some of the contents of this chapter have previously been published in (Mwelwa and Pont, 2003) and (Mwelwa et al., 2003).

- Employing a 'hybrid' scheduler to retain most of the desirable features of the (pure) co-operative scheduler, but still allow a single long (pre-emptive) task to be executed

In the right circumstances, each of these techniques could prove useful. However, these solutions do not necessarily make it easier for developers to implement TTC applications. What is needed is a means of 'recycling design experience': specifically, a way of allowing less experienced software engineers to incorporate solutions in their systems from previous successful TTC designs (Pont and Banner, 2004). The need for design experience can be illustrated by the fact that in many embedded applications, the task duration is extremely brief. However, where the system does have long tasks, this is often because the developer is unaware of some simple techniques that can be used to break down these tasks in an appropriate way and - in effect - convert 'long tasks called infrequently' into 'short tasks called frequently' (Pont, 2001, p. 252). Furthermore, Bate (1999, p. 47) has previously stated that the design of schedulers relies on informal techniques i.e. the synthesis is performed using estimated values for task worst-case execution times. When there is a requirement change or task overrun, the task scheduling is manually altered. It is without a doubt that appropriate design experience would therefore improve the development of TTC systems. In fact, Voros (2003) has previously argued that design reuse is the key to the design and development of reliable systems and that it is a key technique in the reduction of the development time of complex systems.

In Chapter 3, it was argued that patterns offer a means of facilitating design reuse. To this effect, a collection of patterns to support the development of TTC systems has been assembled in the ESL under the auspices of Dr Michael Pont. Pont (2001), inspired by the work of Erich Gamma and colleagues (1995), adapted the patterns concept to the time-triggered embedded systems domain. Pont believes that using these patterns can allow for TTC architectures to be simply and cost-effectively implemented in a myriad of embedded projects.

Pont began to assemble the patterns in 1996 and the first drafts of the patterns were used internally (within the ESL), primarily for teaching and training purposes. In these drafts Pont initially attempted to demonstrate that patterns could form the basis for solutions to some of the problems faced in the development of simple software-based control systems (see, Pont et al., 1999; Pont et al., 1998a; Pont, 1998b). For example, it was demonstrated that patterns could be used to effectively develop applications for condition monitoring and fault diagnosis (CMFD) in diesel engines (Parikh et al., 1998).

Later drafts of the patterns were presented and discussed more widely. At the fourth European Conference on Pattern Languages of Programming and Computing (EuroPLoP 1999), Pont (2000a) presented more patterns but this time the focus was on assisting developers with desktop experience adapt to embedded systems development . Over the years during which these patterns underwent continuous improvement, the focus shifted towards use in the development of safe and reliable microcontroller-based embedded systems such as those found in the automotive industry e.g. cruise control systems (Pont, 2000b; Wong and Pont, 2000; Pont, 1999). Through this continuous process of refinement, a great deal of useful feedback was obtained from the patterns community and various internal (within the ESL) and external projects. Based on this feedback, Pont subsequently focused on producing patterns to support the development of embedded systems with TTC architectures. The eventual result was a pattern collection consisting of more than 70 patterns aimed at supporting the development of TTC embedded systems. These patterns are referred to as the Patterns for Time-Triggered Embedded Systems (PTTES) collection (Pont, 2001).

4.2 The PTTES form

One of the key characteristics of a pattern is its format and concise description of a problem and its context and solution. Pattern authors tend to adapt Alexander's (1977) pattern format to their domain. For instance, because the Gamma et al. (1995) patterns are associated with object-oriented software design, their form has an element known as 'Structure' (see, Table 3-1 in Section 3.2.3) that illustrates the pattern's solution using class diagrams. In a similar manner Pont (2001) presents the PTTES collection in a form that is associated with its domain. For example, LOOP TIMEOUT (in Figure 4-1) presents hardware and portability implications. These are important design issues that are considered during the design of an embedded system and have an influence on a system's maintainability and production cost. Appendix A contains a list of the patterns that currently constitute the PTTES collection.

Loop Timeout

Context

- You are developing an embedded application using one or more members of the 8051 family of microcontrollers.
- The application has a time-triggered architecture, constructed using a scheduler.

Problem

How do you ensure that your system will not 'hang' while waiting for a task (such as a switch read, an analogue-to-data conversion, or serial data transfer) to complete?

Background

To understand the need for LOOP TIMEOUT, consider an example.

The Philips 8Xc552 is an EXTENDED 8051 device with a number of on-chip peripherals, including an 8-channel, 10-bit analogue-to-digital converter (ADC). Philips provides an application note (AN93017) that describes how to use this microcontroller. This application note includes the following code:

```
// Wait until AD conversion finishes (checking ADCI)
while ((ADCON & ADCI) == 0);
```

Such code is not intended to be of 'production' quality. However, its structure is not unusual in embedded systems. The problem is that there are circumstances under which our application may 'hang'. This might occur for one or more of the following reasons:

- If the ADC has been incorrectly initialized, we cannot be sure that a data conversion will be carried out.
- If the ADC has been subjected to an excessive input voltage, then it may not operate at all.
- If the variable ADCON or ADCI was not correctly initialized, they may not operate as required.

Such problems are not, of course, unique to this particular microcontroller, or even to ADCs. Such code is common in embedded applications.

If your application is to be reliable, you need to be able to guarantee that no task or function will 'hang' in this way. Loop timeouts offer a simple but effective means of providing such a guarantee.

Solution

A loop timeout may be easily created. The basis of the code structure is a loop delay, created as follows:

```
unsigned int Timeout_loop = 0;
...
while ((++Timeout_loop) > 0);
```

This loop will keep running until the variable Timeout_loop reaches its maximum value (assuming 16-bit integers) of 65535, and then overflows (to 0). When the variable overflows, the program will continue. Note that without some simulation studies or prototyping, we cannot easily determine how long this delay will be. However, we do know that the loop will eventually 'time out'.

Such a loop is not terribly useful. However, if we consider again the ADC example given in 'Background', we can easily extend this idea. Recall that the original code was as follows:

```
// Wait until AD conversion finishes (checking ADCI)
while ((ADCON & ADCI) == 0);
```

Here is a modified version of this code, this time incorporating a loop timeout:

```
tWord Timeout_loop = 0;

// Take sample from ADC
// Wait until conversion finishes (checking ADCI)
// - simple loop timeout
while (((ADCON & ADCI) == 0) && (++Timeout_loop != 0));
```

We now know that the loop cannot go on 'for ever'. Note that we can vary the duration of the loop timeout by changing the initial value loaded into the loop variable.

Hardware resource implications

LOOP TIMEOUT does not use a timer and imposes an almost negligible CPU and memory load.

Reliability and safety implications

Using LOOP TIMEOUT can result in a huge reliability and safety improvement at minimal cost. However, if practical, HARDWARE TIMEOUT is usually an even better solution.

Portability

Loop timeouts will work in any environment. However, the timings obtained will vary dramatically between microcontrollers and compilers.

Related patterns and alternative solutions

As mentioned under Reliability and Safety Implications, HARDWARE TIMEOUT is often a better alternative to LOOP TIMEOUT.

Overall strengths and weaknesses

☺ Much better than executing code without any form of timeout protection.

☺ Many applications use a timer for RS232 baud rate generation, and another timer to run the scheduler. In many 8051 devices, this leaves no further timers available to implement a HARDWARE TIMEOUT. In these circumstances, use of a loop is the only practical way of implementing effective timeout behaviour.

☹ Timings are difficult to calculate and timer values are not portable. HARDWARE TIMEOUT is always a better solution, if you have a spare timer available.

Figure 4-1 Loop Timeout (adapted from Pont, 2001, p. 298)

In recent work, Kurian and Pont (2005) have revised the form of the PTTES patterns. In this revised structure each pattern consists of three layers:

- Abstract patterns
- Patterns, and,
- Pattern implementation examples (PIEs)

For example, Kurian and Pont (2005) have presented the abstract pattern TTC PLATFORM. This abstract pattern consists of patterns: TTC-SL (SUPER LOOP) SCHEDULER, TTC-ISR (INTERRUPT SERVICE ROUTINE) SCHEDULER and TTC SCHEDULER that describe some of the ways in which TTC PLATFORM can be implemented. In each of these alternative patterns, the abstract pattern is referred to for background information (Kurian and Pont, 2005).

As discussed in Section 2.3, the choice of processor or microcontroller in an embedded system has a profound impact on the implementation of the patterns used in the system design. A side effect of this is that new example code needs to be created for each software pattern in the collection if a new hardware platform is to be supported. In order to avoid the need to edit the patterns in these circumstances, another layer composed of 'pattern implementation examples' (PIEs) is derived from the patterns layer. PIEs are intended to illustrate how a particular pattern can be implemented on different hardware platforms, e.g. on 8-, 16-, 32- and 64-bit processors. In addition, PIEs can be used to illustrate how patterns may be implemented using different programming languages e.g. Assembly and C (see: Key et al., 2003). As an example, Appendix C lists two PTTES patterns: HEARTBEAT LED and ERROR LED with 8051 and ARM PIEs, respectively.

It must be noted that the PTTES collection PIEs are currently based on the C programming language. This is mainly because C is today the dominant language in embedded control systems. C has many fault and failure modes, but this is balanced by the fact that more is known about how C programs fail. By avoiding these fault and failure modes, C is capable of producing reliable systems whilst retaining its many benefits such as efficiency, small footprint, portability, availability of experienced developers and widespread availability of development tools (Pont, 2001; Storey, 1996; Hatton, 1994).

Similar to work by Buschmann et al. (1996) and Riehle and Zullighoven (1996) that categorises patterns (see Section 3.2.4), Pont (2001) has categorised the patterns in the PTTES collection based on their area of application in the embedded systems domain. The following is a description of the categories that define the various types of patterns in the PTTES collection:

- **Hardware foundations:** These patterns support the design and implementation of an appropriate microcontroller hardware platform and associated peripherals e.g. oscillator and reset circuits discussed in Section 2.3 (Pont, 2001, p. 27)

- **Software foundations:** Patterns in this category describe how to create basic 'Super Loop' architectures, how to control microcontroller port pins, and how to create timing delays (Pont, 2001, p. 159).

- **Single-processor time-triggered architectures:** These patterns describe how to implement software architectures with a time-triggered architecture for single processor schedulers such as those described in Section 2.4.1.2 (Pont, 2001, p. 229).

- **Multi-processor time-triggered architectures:** Patterns in this category describe how to implement time-triggered software architectures for distributed systems using a wide range of protocols e.g. RS-232 and CAN (Pont, 2001, p. 537).

- **Specialised time-triggered architectures:** These patterns are based on the single and multi-processor time-triggered architectures. They support the implementation of specialised software architectures for applications with specific resource constraints e.g. applications with a low power resource (Pont, 2001, p. 891).

- **User interface patterns:** Developers use patterns in this category to support the implementation of user interfaces such as switches and keypads. The category also includes patterns that support the implementation of LCDs (Pont, 2001, p. 359).

- **Serial interface patterns:** This category of patterns support the implementation of serial interfaces used to implement devices such as temperature sensors and memory components (Pont, 2001, p. 491).

- **Monitoring and control patterns:** Patterns in this category present techniques for implementing a range of monitoring and control related components such as ADCs, DACs, PWM outputs and the PID control algorithm (Pont, 2001, p. 725).

From these categories it can be noted that the PTTES collection provides patterns that support the design and implementation of both hardware and software architectures that are fundamental to an embedded application (see Sections 2.3 and 2.4). To support the implementation of various functionalities, the PTTES collection also provides a wide range of application specific patterns. With this wide range of patterns, the PTTES collection can support the development of complete TT embedded applications. Unlike the Gamma collection, it can therefore be argued that the PTTES collection is a pattern language as it is domain specific and is composed of inter-related patterns that support the design of complete applications. This is analogous to Alexander's (1977, p. xiii) pattern language, i.e. *"Each pattern can exist in the world, only to the extent that it is supported by other patterns: the larger patterns in which it is embedded, the patterns of the same size that surround it, and the smaller patterns which are embedded in it"*, see Section 3.1.1.

4.3 How useful is the PTTES collection?

To date there have been no studies carried out to assess the effectiveness of the PTTES collection. This section therefore attempts to analyse the usability and effectiveness of the PTTES collection in the development of reliable TT embedded systems.

4.3.1 Usability of the PTTES collection

Over the years, the PTTES collection has received feedback from various users. For instance, in his foreword of the PTTES book, Kent Beck, a pioneer of the software patterns movement, wrote, *"It [the PTTES collection] brought back that feeling of opening up a field of endeavour to someone who just has a problem to solve and who doesn't want to be an expert in the solution. ...these patterns stand as an example of how much more can be done with patterns than is commonly attempted"* (Pont, 2001, p. xiv). Though a good review, this does not give an indication as to whether the PTTES collection does indeed support the development of TT embedded systems, instead Kent Beck's review gives praise to the use of patterns in solving problems in the embedded domain.

Another review by Norm Kerth (2001), on Amazon, suggests that the PTTES collection is a valuable 'handbook' for developers of TT embedded systems. He writes, *"This book belongs on the bookshelf of every programmer engaged in time-triggered embedded systems... It is the most pragmatic guide to building embedded systems programs I have seen in my 25+ years of experience with such systems [...]. Along the way Pont tackles some of the most baffling topics in embedded systems - co-operative, pre-emptive and shared-clock schedulers; and the interfacing of multi-processor systems [...]. If you have experience building embedded systems then this is a book that deserves to be treated as a reference book or a handbook - that is, it* *should be browsed from cover to cover to discover the comprehensive nature of topics; then as new work is assigned to you, or new problems arise, it should be the first book you consult for ideas, pointers and solutions"* (Kerth, 2001). In contrast to Kent Beck, who is associated with the object-oriented software domain and not the embedded domain, Norm Kerth is an embedded developer and this is evident in his review of the PTTES collection in which he expresses his appreciation of the support that the collection provides for the development of TT embedded applications. Though not conclusive, these two peer reviews suggest that the PTTES collection is presented appropriately and at the same time provides good coverage of some of the challenging problems faced by embedded developers.

A review of the PTTES book by Chris Hills, suggests that the PTTES collection may only be best suited for teaching purposes, *"The book itself is, I think, aimed at students and would be very good for hobby users of the 8051"* (Hills, 2001). Even though the PTTES collection has primarily been used to support the teaching of Embedded Systems courses for both MSc and undergraduate students in the ESL, contrary to Chris Hill's opinion, the PTTES collection has been applied successfully in non-teaching projects. For example, a recent research project in the ESL evaluating suitable software architectures for automotive control systems, has used the PTTES collection extensively to develop an adaptive cruise control system test bed (see, Short et al., 2004c; Short et al., 2004b; Short et al., 2004a).

In order to get first hand information from student users on the usability of the PTTES collection, an informal survey was carried out during the course of this research among undergraduate and postgraduate students in the ESL. The subjects in the study were using the PTTES collection to develop various embedded applications during the course of their degree programmes. Five students were interviewed in the survey, two of which were in their 4th year of an undergraduate degree, another two were in their 3rd year of the same degree and the fifth subject was a postgraduate MSc student. The subjects had an average of approximately three years programming experience. The survey was based on a half hour interview with each subject. The questionnaire used for the interview is presented in Appendix B.

From the survey, it was observed that the subjects followed a similar approach when implementing a pattern. For example, users had a tendency to first implement a hardware platform after which an appropriate scheduler would be implemented. In most instances, users would also implement a flashing LED task to test whether the scheduler and hardware were functioning correctly before proceeding with the remainder of the application development. It was also noted that the subjects did not seem to be knowledgeable of the patterns concept. In fact, the approach generally used by the subjects when implementing the patterns would suggest that the PTTES collection was in fact a code library rather than a collection of patterns. The subject's emphasis in the application of the patterns seemed to be on the reuse of the PIEs as opposed to the actual design solutions presented. This is not the intended usage of patterns, as discussed in Section 3.2, the code examples that come with a pattern are merely an illustration of how a pattern can be applied. Nevertheless, the task of implementing the PIEs was observed to be mundane, tedious and as a result error prone as a structured process is not followed. As such the manual application of the patterns is somewhat of an ad hoc process. This is not an appropriate way to develop software especially for reliable embedded systems. In fact, it was observed that the subjects made common errors when implementing the patterns. For example, not updating the microcontroller header or task array size and allocating port pins inappropriately. These errors, though simple would cause users, especially inexperienced ones, to spend unnecessary time debugging the resulting code.

With the potential pitfall of errors during the implementation of the patterns, it could easily be argued that pattern-based design provides an incomplete solution to the problems of TT software development. This is indeed a valid argument; however patterns are not a silver bullet to the challenges faced by embedded developers. Ideally a developer should expect to use a range of other techniques (including, for example, both patterns and time-triggered software architectures) when dealing with complex analysis, design and implementation issues when developing software for reliable embedded applications. Overall, embedded software is complex, often (at least) safety-related, and developed by fallible individuals as Reason (1997, p. 25) has previously argued, *"Human fallibility, like gravity, weather and terrain, is just another foreseeable hazard. The issue is not why an error occurred, but how it failed to be corrected. We cannot change the human condition, but we can change the conditions under which people work"*.

4.3.2 Can the PTTES collection sustain technological advancements?

An important issue to consider when adopting new technologies in software engineering e.g. programming languages or compilers is the implication of any future technological advancement in the field. The same would apply to the adoption of the PTTES collection. For example, when the PTTES collection was first published in 1996, the 8051 family of microcontrollers was the only platform used to illustrate the use of the patterns. However, since the publication of the PTTES collection, the semiconductor industry has continued to evolve, thereby producing more efficient processors and microcontrollers at affordable prices. There has therefore been a need to review the PTTES collection in order to keep up with advancements in the semiconductor industry.

In essence, as patterns are applied in different contexts and their related technologies continue to advance, we learn more about the problems they are intended hence facilitating their continuous refinement. However, because of the relationships among patterns in a pattern language, a change made to one pattern may affect its relationships with its related patterns and potentially the structure of the entire collection. Hence, the restructuring of the PTTES collection described in Section 4.2 is meant to accommodate future changes to the collection. The idea of having PIEs allows the PTTES collection to be extended to support new hardware platforms. For instance, PORT I/O in the original PTTES collection (see: Pont, 2001, p. 174) was recently revised to allow developers to easily adapt it to other hardware platforms such as the ARM (Pont and Mwelwa, 2003b).

Besides adapting the PTTES collection to technological advancements, new patterns are continuously being added to the collection. However, creating patterns through the PLoP conferences (see Section 3.2.4) is a challenging and time-consuming process, an observation made by the author during the creation of two new PTTES patterns. From the observations made during the PTTES user survey described in Section 4.3.1, HEARTBEAT LED and ERROR LED were created (see Appendix C for a detailed description of these patterns). In particular, the use of a flashing LED by the subjects as a means of getting tangible feedback on the status of a microcontroller during the initial stages of development prompted the creation of these two patterns. Through the creation of these patterns the author gained first-hand experience of the pattern creation process at a PLoP conference.

A number of interesting observations were made during the creation of these patterns. One of these was the level of domain knowledge possessed by the workshop participants. For instance, the only domain experts on embedded systems were the two authors that presented the patterns (Mwelwa and Pont, 2003). The other participants came from different domains, for example Grone and Tabeling (2003) presented patterns for client-server development and Marquardt (2003) presented generic patterns to support the design of architectures used in software development. One of the main reasons for this mix of participants is that PLoP conferences generally accept patterns from various domains e.g., client-server web development to embedded programming as was the case at this workshop. It is therefore not always possible to have pattern workshop sessions restricted to specific domains. Good domain knowledge by a pattern author is therefore very important, as during a workshop non-domain experts are unable to question the technical content of a pattern and therefore assume it is credible. Hence, the primary focus of a pattern workshop at such a conference tends to be on the presentation and structure of a pattern and not on its technical content.

From the observations of the pattern creation process, it is clear that patterns go through rigorous and thorough scrutiny before publication. Technical writing can be challenging for technical people such as software developers, therefore the pattern creation process enables experienced writers to tutor inexperienced writers. However, as the primary focus of the pattern creation process is on the presentation and not the technical content of a pattern it can be argued that not every pattern in a collection has to go through the pattern creation process. Instead, a single pattern that has been through the pattern creation process can be used as a template for all the other patterns in the collection. In fact, not all the patterns in the PTTES collection have gone through PLoP conferences. Having initially published patterns through PLoP conferences, Pont has subsequently relied on this early experience to create and publish new patterns. It must also be taken into account that the pattern creation process can take up to six months i.e. from Shepherding to the Writers' Workshop. This can therefore be a costly process. For instance, if Pont had taken every single pattern in the PTTES collection, through the pattern creation process, the PTTES book may have taken longer to publish due to the lengthy pattern creation process. Furthermore, it would have been very costly to present all 72 patterns at PLoP conferences.

With the observations made above, it can be argued that the PTTES collection is manageable and accommodates changes and the addition of new patterns in line with technological advancements. This can be attributed to the fact that Pont has become more experienced with the pattern creation process over the years and has thus been able to appropriately manage the evolution of the PTTES collection.

4.4 Chapter conclusions

This chapter has presented the PTTES collection that is meant to support the development of TT embedded applications. From observations made of the use of this collection, it has been noted that it provides developers of TT embedded applications with a 'best practices design handbook'. The patterns in the collection bridge the gap between expert and non-expert developers of TTC embedded applications. For example, developing an avionics control system requires expertise in technical areas such as control and instrumentation, which some developers may not be familiar with; the monitoring and control patterns in the collection would support a developer in these circumstances.

It has also been argued that the PTTES collection although initially based on the 8051 hardware platform, can be extended to support new hardware platforms. This is a good feature of the collection, as developers do not have to worry about its ability to cope with future technological advancements. This is particularly important in designs where maintainability is paramount.

A disadvantage of the PTTES collection observed is its 'manual' application. The manual application of the patterns involves users adapting PIEs to their design requirements. This is an ad hoc mundane, tedious and as a result error prone process. This is clearly not an appropriate way to develop software especially for systems with high reliability requirements.

5 Automated Code Generation

This chapter reviews previous research in both traditional and pattern-based code generation.

5.1 Programming languages and automatic programming systems

The use of appropriate Computer Aided Software Engineering (CASE) tools can improve software development productivity (Yang, 1999; Aaen et al., 1992). CASE tools support methodological procedures and techniques and have been used in software development for many years (Hoffnagle and Beregi, 1985). A compiler is arguably one of the first CASE tools to have been widely used and accepted by computer scientists; it translates high-level programming languages into machine code (Koffman, 1998).

It is generally suggested that the advent of compilers can be traced back to the inception of FORTRAN – arguably one of the first high-level programming languages to have been developed (Sammet, 1981). Sammet (1981) has argued that a major technical contribution made by the developers of FORTRAN in the 1950s was to demonstrate that a compiler could produce efficient object code. It then became clear that the productivity of programmers using high-level languages could be significantly greater than that of programmers working in Assembly. Consequently, the first FORTRAN compiler was originally called an 'automatic programming system' (Buntine, 1998). Although it is sometimes argued that the emergence of high-level programming languages resulted in the need for automatic code generation capabilities, John Backus (1980), co-inventor of FORTRAN, has previously pointed to the economic demands at the time, in his own words, *"Instead it began with the recognition of a basic problem of economics: programming and debugging costs already exceeded the cost of running a program, and as computers became faster and cheaper this imbalance would become more and more intolerable"* (Backus, 1980, pp. 130-131).

Today there are a wide range of compilers available in relation to programming languages as each programming language requires a specific compiler. As a result compiler technology has evolved as new programming languages have become available over the last half century. Levenez (2006) has described a timeline that illustrates the evolution of more than 50 programming languages from the inception of FORTRAN in 1954 to modern day programming languages like Microsoft's C#. First, second, third, fourth, and fifth generation languages i.e. 1GL, 2GL, 3GL, 4GL and 5GL respectively are groupings used by computer scientists to represent milestones in the evolution of programming languages. 1GL is specifically associated with machine code, 2GL with Assembly (Giloi, 1997; Shapiro, 1997; Allen, 1981). 3GLs are high-level programming languages that are relatively easier to learn than 2GLs as the syntax used is close in similarity to readable language. Examples of 3GLs are FORTRAN, Pascal, C and object-oriented programming languages such as Java and C++ (Allen, 1981; Backus, 1980). Figure 5-1 and Figure 5-2 present code snippets of a simple flashing LED task implemented in Assembly and C respectively. These code examples illustrate the difference in readability between a 2GL and 3GL programming language. After 3GLs comes 4GLs that are usually associated with domain specific languages e.g. database query languages such as SQL and scripting languages such as JavaScript (Heering and Mernik, 2002).

```
;*************************************************************************
; LED_Flash
; Flashes an LED (ON, OFF) on a specified port pin.
;*************************************************************************

TASK1:

LED_Flash:
mov A,LED_Status
jz LED_on
clr LED_PORT ; turn LED off
mov LED_Status,#00
ajmp SCH_Dispatch_Tasks_end

LED_on:
setb LED_PORT ; turn LED on
mov LED_Status,#01
ajmp SCH_Dispatch_Tasks_end
```

Figure 5-1 A flashing LED task in Assembly. Compare with Figure 5-2

```
/*------------------------------------------------------------------*-
  LED_Flash()

  Flashes an LED (ON, OFF) on a specified port pin.
-*------------------------------------------------------------------*/
void LED_Flash(void)
   {
   // Turn the LED OFF to ON (or vice versa)
   if (LED_state_G == 1)
      {
      LED_state_G = 0;
      LED_pin = 0;
      }
   else
      {
      LED_state_G = 1;
      LED_pin = 1;
      }
   }
```

Figure 5-2 The flashing LED task implemented in Assembly in Figure 5-1, now implemented in C

5.2 Model-based code generation

The use of design models to specify complex designs is a long-standing engineering tradition. In model-based software development, models are used to abstract the design of a system thus allowing developers to: 1) design applications with requirements rather than technical specifications in mind, 2) communicate design ideas across the design team and 3) validate the design before it is implemented (Szemethy, 2006; Martin and Müller, 2005).

The Unified Modelling Language (UML) facilitates model-driven software development, it is a language for communicating and organising design ideas (Pawlicki, 2003). The Object Management Group (OMG) adopted the UML in November 1997 (Object Management Group, 2006). The initial intention of UML was for the development of object-oriented software but over the years it has become the de facto technology for the design, analysis and modelling of various software architectures and more recently for model-based code generation. In model-based code generation source code and (or) design documentation are automatically generated from UML design models and in so doing alleviating the tedious and error-prone task of manually transforming the design models into code i.e. hand-coding (Szemethy, 2006; Milicev, 2002; Smith et al., 1999; Auer et al., 1988). XML Metadata Interchange (XMI) is an Extensible Mark-up Language (XML) that is generally used by code generators to facilitate model-based code generation (Martin and Müller, 2005). XMI's representation of a UML model is very rich in detail and can therefore relate to graphical representation of the elements and references between the various classes and objects in a design model (Object Management Group, 2006). The OMG have also defined the Meta-Object Facility (MOF), an extensible model driven integration framework for defining, manipulating and integrating metadata and data in a platform independent manner, as such it is commonly used to integrate tools, applications and data (Object Management Group, 2006).

In recent years the OMG have proposed and adopted the Model Driven Architecture (MDA) that is based on their established standards: MOF, UML and XMI (Object Management Group, 2006). The MDA provides a set of guidelines for structuring specifications expressed as models. The MDA methodology allows a system's functionality to be defined as a Platform Independent Model (PIM) that abstracts design from implementation. Then using formal transformation rules based on a Platform Definition Model (PDM) e.g. Microsoft's .Net technology, the PIM is transformed to a Platform Specific Model (PSM) from which a system can be implemented. The transformations from PIM to PSM and PSM to code are generally facilitated by automated CASE tools.

The adoption of model-based code generation over the years suggests parallels with the move from Assembly (2GL) to high-level programming languages such as FORTRAN and then Java (3GL). With this form of software development, developers can produce remarkably more lines of code than is possible when hand-coding and at the same time eliminate any potential errors during this stage of development (Whalen and Heimdahl, 1999). Model-based code generation also provides support for software maintenance. Without model-based code generation, the benefits of design modelling seldom extend throughout a product's life cycle because during maintenance, these design models may be ignored and the code modified directly. This results in design models falling out of sync and therefore becoming irrelevant. On the other hand a design model and its generated code simultaneously retain there usefulness if used appropriately (Bell, 1998).

Despite the advantages of model-based code generation, this methodology has not quite picked up in embedded systems development. This is mainly attributed to the fact that there currently exists a mismatch between UML design models and embedded software designs (Schatz et al., 2003; Martin et al., 2001). Limitations are due to the UML's inability to address specific constraints such as timing, the handling of periodic time-triggered tasks, limited memory and power use and pre-defined hardware platform technology (Graaf et al., 2003; Torngren and Redell, 2000). Nevertheless, efforts are being made by various researchers to extend the UML to the embedded systems domain (see for example: Kukkala et al., 2005; Vanderperren and Dehaene, 2005). One of these efforts, the Systems Modelling Language (SysML) looks the most promising. SysML is a domain specific language for systems engineering. It supports the specification, analysis, design, verification and validation of a broad range of systems. These systems may include hardware, software, information and processes (SysML Partners, 2005). However, SysML is currently undergoing formalisation. On November 14th 2005, SysML v.1.0 alpha was made available to the public and submitted for review to the OMG (SysML Partners, 2005).

Although the embedded domain currently lacks a well defined modelling language, certain specialist areas within the domain have relied on other code generation techniques. For instance, Simulink, a platform for simulation and model-based design for dynamic systems, provides an interactive graphical environment and a customisable set of block libraries (MathWorks, 2005). Real-Time Workshop Embedded Coder from MathWorks generates C code from Simulink models (MathWorks, 2005). Today the aerospace and automotive industries make extensive use of such code generation tools aimed at control and signal processing systems and many car manufacturers now rely on production code generated using these techniques (Lee et al., 2003; Marsh, 2003; Schatz et al., 2003; O'Halloran, 2000). For instance, Pi Technology and Ford have used MATLAB and Simulink to generate C code for the development of electronic control units for the Ford Focus (Wartnaby et al., 2003). Ken Karnofsky, marketing director for DSP and Communications at The MathWorks, has previously expressed his views on the importance of this form of software development, *"Engineers can't keep writing code manually when [the] programs take up half a million lines of code. The predecessors of the engineers took a while to accept C compilers and hardware synthesis but eventually it became main-stream. It's the same process all over again"* (Marsh, 2003).

Beyond commercial tools, researchers within the embedded systems community have attempted to develop innovative tools. For example, POLIS is a hardware/software co-design tool for embedded systems (Balarin et al., 1997). POLIS allows the partitioning of an application into co-operating software and hardware modules and synthesises both the individual modules and the interfaces among them. It can also perform system-level simulation to support the partitioning of hardware and software. POLIS also provides the automatic generation of a custom scheduler (Cuatto et al., 2000). Overall, POLIS provides an environment where a systems designer can quickly evaluate choices of hardware and software architectures (Balarin et al., 1997).

Ghezzi et al. (2002, p. 510) have previously implied that code generation is a form of software reuse, *"The generation approach to reusing code relies on a generator that generates the application code on the basis of input that specifies the needed components and their integration. In some approaches, the needed components are generated automatically on the basis of their specification, while others assume a pre-existing set of components that are configured together based on the code generator's output."* Reuse of existing carefully designed and extensively used software is generally accepted as the primary means of improving the quality of new software systems and reducing development costs (Mili et al., 2003). In fact, previous empirical studies have demonstrated that reuse is an important factor in enhancing software production (see: Basili et al., 1996; Moser and Nierstrasz, 1996; Sparks et al., 1996).

However, in the case of compiler based code generation, any form of reuse would be at a low level in the form of source code compilation that involves the translation of statements in one programming language into machine code. On the other hand model-based code generation, where UML is used, is based on purposely-designed models. UML serves well for abstracting the design from the problem at hand and is primarily used as a notation; therefore it can be argued that model-based code generation does not necessarily promote software reuse.

5.3 Automating the application of software patterns

Most of the previous work on software patterns, discussed in Section 3.2, has emphasised a manual approach to their application (Henzinger et al., 2003; Pont, 2001; Prechelt and Unger, 2001; Douglass, 1999; Wild, 1996). As such, it is (sometimes implicitly) assumed that a developer will browse a pattern collection, choose appropriate design patterns and – possibly using some code examples or hardware schematics (if using the PTTES collection) as a starting point – assemble a system. However, it does seem possible that we should be able to reduce developer effort and the number of potential errors in a pattern-based software development process by developing suitable CASE tools. Today, it is unthinkable to develop complete software applications without using a tool-supported process (Jacobson et al., 1999). Furthermore, as discussed in Section 5.1, the use of CASE tools has the benefit of improving software development productivity, quality, software reuse, documentation and maintenance.

Over the years, as patterns have become widely used within the software engineering community, various researchers have attempted to automate their application. Budinsky et al.

(1996) were pioneers in this research, they developed a tool that attempted to automate the application of the Gamma collection of patterns described in Section 3.2. Their tool has a 'wizard' that allows a user to enter application specific data and select trade-offs that lead to the generation of corresponding code declarations and implementations. Essentially the tool takes care of the mundane aspects of the transformation of individual patterns into C++ code. However, the tool does not generate complete code; instead it produces skeleton object-oriented classes associated with the pattern implementation. A developer therefore has to manually add application specific code to the generated classes in order to produce a complete application. The tool also incorporates a browser that renders the Gamma collection into HTML to allow easy access to the pattern documentation during development. Overall, the main benefit of this tool is the implementation of individual patterns in C++.

Florijn et al. (1997) have also developed a pattern tool similar to that of Budinsky et al. (1996) in that it is also intended to support the application of the Gamma collection. Similarly to the Budinsky tool, Florijn et al.'s tool does not support complete code generation instead it generates skeleton code that a developer has to add functionality to by adding application specific code. The main difference between the Budinsky and Florijn tools is that Florijn's tool checks for consistency between the generated code and the associated patterns thus ensuring the implementation maintains the pattern specification.

Pattern Wizard is another pattern tool similar in functionality to that of Budinsky's developed by Arnout (2004). The main difference between the two tools is in the programming language used to generate skeleton classes. Whereas Budinsky et al. (1996) used C++, the programming language used in the Gamma collection, Pattern Wizard generates code in the Eiffel programming language. At present Pattern Wizard only supports the implementation of five patterns described in the Gamma collection (Gamma et al., 1995): SINGLETON, ADAPTER, DECORATOR, BRIDGE and TEMPLATE METHOD. Although the purpose of the tool is to simplify the application of the Gamma collection, this work was a result of limitations in Arnout's attempts to transform patterns into software components, Section 6.2 elaborates on this work.

FRED (Framework Editor) is a pattern tool developed by researchers at Tampere University of Technology and University of Helsinki. FRED specialises in the design of applications using architectural patterns (discussed in Section 3.2). This is based on Viljamaa's (2001) belief that there exists a close relationship between architectural patterns and frameworks. In fact, Viljamaa (2001) argues that most pattern tools actually create application frameworks on a small scale. This is indeed a valid argument as the tools described here do not generate complete applications. Instead they generate skeleton code from individual patterns. A developer then has the task of manually completing the code and integrating it with the rest of the application. However, Viljamaa (2001) does emphasise that a framework tool must be capable of managing groups of patterns not just single patterns as is the case with the pattern tools described here.

In recent years, manufacturers of CASE tools have recognised the potential of pattern-based software development such that they have begun to support the use of patterns in their integrated development environments (IDEs). For instance, Borland (2005) have demonstrated this in their Borland Together Developer IDE that allows a developer to browse and select a pattern from a catalogue into a workspace where generic code templates associated with the patterns are generated. A developer then has to add application specific code to the templates in order to adapt them to the application being developed. Visual Studio .Net, Microsoft's latest development environment, also supports the use of patterns to create object-oriented class templates that provide a framework for building applications rapidly (Microsoft, 2006). It must however, be noted that the majority of these commercial tools have primarily focused on the Gamma collection.

Even though some of the work on pattern tools described here has come out of the patterns community e.g. Budinsky et al. (1996), the topic has been a contentious issue and has been viewed as a contradiction of first principles. Coplien, a pattern enthusiast, has expressed critical views on constructing pattern tools or indeed formalising patterns, he argues, *"...by the time technology or understanding matures to the point where we can formally capture an idea, it ceases to have the generative, literary quality that sets patterns apart from other methods"* (Coplien, 2003). Nevertheless, other pattern enthusiasts believe otherwise, for instance, Viljamaa (2001) believes that the application of patterns can benefit from tool support and based on the discussions in Section 4.3, this is a valid argument.

5.4 Pattern-based code generation

It can be argued that the pattern tools described in the previous section have actually attempted to automate the application of individual patterns rather than support automated code generation from patterns. In particular, they generate object-oriented skeleton classes associated with an individual pattern which a developer then has the task of adding application specific code to and thereafter integrating this code into a complete application. These tools do not therefore generate integrated applications, instead they generate code modules.

Although these tools help ease the application of patterns, Heister et al. (1997) have argued that they do not allow for the further exploitation of patterns in latter phases of software development. In particular, the fact that developers are left to complete the implementation of the incomplete code generated by these tools means that there is a high possibility that the generated structures are lost in the process and thereby making it almost impossible to guarantee whether the original design has been implemented. This is a drawback for both software maintenance and for any attempts to reverse engineer an application developed in this manner. To this effect Pelechano et al. (2002) have previously expressed the belief that applying patterns in an automated code generation process would enhance pattern-based software development. Indeed, with a well defined model, patterns are capable of supporting a code generation process. Pattern design solutions provide a basis for the generation of quality software.

Considering the PTTES collection described in Chapter 4, using a manual approach, it has previously been described how TTC software can be created for a range of industry-standard hardware platforms, for example the 8051 microcontroller (Pont, 2001), ARM processor (Pont and Mwelwa, 2003b) or PC platform (Pont et al., 2003a). While such a manual approach can be effective, there is an imperfect match between generic processor architectures and time-triggered software designs. For example, most processors support a wide range of interrupts, while the use of (pure) time-triggered software architectures generally requires that only a single interrupt is active on each processor. This leads to design guidelines, such as the 'one interrupt per microcontroller rule' (Pont, 2001). Although, such guidelines can be supported by patterns, as discussed in Section 4.3.1, the process of applying the patterns can be mundane and tedious with the potential to be error prone and time consuming thus resulting in unreliable systems.

One way in which we might expect to improve the application of the PTTES collection and, at the same time, the reliability of TT embedded systems, is by developing a CASE tool that supports the generation of complete application code from the PTTES collection. Indeed, developers of embedded systems already make extensive use of code generation tools, particularly those supporting the development of control and signal processing systems (see Section 5.2). However, generating code from a collection of patterns is a process that presents a number of challenges: consequently, there is no code generation tool based on the PTTES collection, let alone a widely used pattern tool supporting a widely used pattern collection. Chapter 6 discusses these challenges in detail.

The pattern tools described in the previous section are inflexible and as such do not withhold the desirable attributes of patterns. To genuinely support code generation, a pattern-based CASE tool should be dynamic, i.e. capable of allowing developers to easily adapt the patterns to a particular design context. Furthermore, such tools must have the ability to generate application code from appropriately selected groups of patterns and not be restricted to generating skeleton classes from individual patterns.

5.5 Chapter conclusions

In this chapter, it has been argued that code generation is an important technique in software development. It has also been argued that existing pattern-based CASE tools primarily focus on automating the application of individual patterns and do not consider the generation of complete applications from appropriately selected groups of patterns.

It has also been argued that patterns can offer an alternative to conventional code generation methods such as model-based code generation. Unlike UML, which is generally used as a design notation, patterns are reusable design solutions. Therefore, code generation based on patterns has the potential to produce code of high quality. Furthermore, pattern-based code generation tools would allow for the exploitation of patterns in the later phases of software development, such as maintenance and reverse engineering.

This chapter has also noted that most of the work on pattern tools to date has focused on object-oriented designs and in particular the Gamma collection. But, as discussed in Section 3.2, the scope of patterns has widened over the years such that there are now many pattern collections that focus on a wide range of software domains, for example the PTTES collection described in Chapter 4. Hence, there is a need to consider pattern-based code generation in different domains and as such a suitable approach to pattern-based code generation is required. The remainder of this thesis investigates a different approach to pattern-based code generation.

PART III: A NOVEL APPROACH TO SUPPORT CODE GENERATION USING A PATTERN LANGUAGE

Existing pattern tools have generally focused on the generation of code from individual patterns, they have not supported the generation of code from appropriately selected groups of patterns. Chapter 6 in this part discusses and addresses the challenges of implementing pattern-based code generation CASE tools. Chapter 6 goes on to present a novel approach to pattern-based code generation. In Chapter 7 the validity of this approach is put to test in the development of a prototype pattern-based code generation CASE tool based on the PTTES collection. The application of this tool is described in Chapter 8.

6 Pattern-based Code Generation - Challenges and Solutions[3]

This chapter discusses the challenges of implementing pattern-based code generation CASE tools. A novel approach that addresses these challenges is then described.

6.1 The 'one pattern, many implementations' challenge

In considering automated code generation from pattern-based designs, the objective is not to replace or automate a developer's design deliberations. Instead, the idea of an automated pattern-based code generation process is to assist developers to efficiently develop applications using design patterns. It is believed that such an approach would avoid the often tedious and mundane manual process of implementing patterns. Furthermore, such an approach is unlikely to be error prone, therefore resulting in the production of reliable software.

Patterns are not a set of laws; they are guidelines that suggest general solutions that can be applied to design problems in various contexts. A pattern is therefore expected to have more than one possible implementation (Florijn et al., 1997). Because of this 'one pattern, many implementations' relationship that exists between a pattern and its implementations, users are expected to devise their own custom solutions using the ideas presented by the pattern. Thus the idea of automating this entire process can therefore seem ludicrous, given the amount of human thought and pattern customisation often required (Bulka, 2002). On the other hand, adapting a PIE to a particular hardware platform would require a developer to be familiar with the hardware platform. However, as was highlighted in Chapter 2, there are currently a wide range of microcontrollers available on the embedded market and new ones are constantly becoming available. It is therefore practically impossible to expect developers to keep abreast with all the microcontrollers available on the market.

The 'one-to-many' relationship between a pattern and its implementations stems from Alexander's theory on patterns. Alexander (1979, p. 187) notes that, *"Ordinary languages and pattern languages are finite combinatory systems which allow us to create an infinite variety of unique combinations, appropriate to different circumstances, at will... "*. In other words, patterns describe a set of solutions for a family of related design problems and, as a result, it is difficult (if not practically impossible) to implement an automated process that can generate code for every possible solution (MacDonald et al., 2002). Hence, the one-to-many mapping between a pattern and its implementations presents a fundamental challenge to those who wish to develop

[3] Some of the contents of this chapter have previously been published in (Pont et al., submitted 2006) and (Mwelwa et al., 2006).

automated pattern-based code generation solutions as for every 'good' pattern, a code generating system should be capable of generating a near infinite number of possible implementations. Pattern automation tools are therefore unlikely to be capable of always producing pattern implementations that exactly match a developer's requirements (Bulka, 2002).

A consequence of the 'one pattern, many implementations' relationship is that fitting automated pattern templates to the exact problem at hand might in principle always be a mismatch in the sense that say, hand crafted solutions might always be better. Hence a custom tailored suit is always going to fit someone better than one bought off the rack. Nevertheless, do pattern implementations always need to be 'custom tailored' (Bulka, 2002)? On the other hand, 'hand coded' pattern implementations could result in solutions that no longer meet the structural or semantic intentions of their patterns resulting in what Garlan et al. (1995) describe as an architectural mismatch. An architectural mismatch results from assumptions that a reusable component makes about the structure of the system it is intended for. These assumptions often conflict with those of the other components and are almost always implicit, making them extremely difficult to analyse before building a system (Garlan et al., 1995). Therefore, in order to implement automated pattern-based code generation solutions, it is imperative that the 'one pattern, many implementations' issue is addressed.

6.2 Addressing the 'one pattern, many implementations' challenge

This section addresses the 'one pattern, many implementations' challenge discussed in the previous section.

6.2.1 Can we use software components as the basis of pattern-based code generation?

It can be argued that the usage of PIEs is comparable to that of software components. It could therefore be further argued that a possible approach to pattern-based code generation is the transformation of patterns into software components.

Software components are generally viewed as pre-implemented software modules used as building blocks in the development of software applications (Wang and Shin, 2000). They are designed to be reusable in many applications, some not yet existing. They must therefore be well specified and easy to understand, adapt, deploy and replace (Crnkovic, 2003). Today there are numerous component libraries available to developers across various software domains (Zimmermann, 2005).

Meyer (2003) has described a type of software component called a 'Trusted Component' that he argues can guarantee the high-quality of software generally required in safety critical applications. In fact, Meyer (1997) had previously argued that software components derived from patterns would actually guarantee this degree of quality and make patterns more useful than they are in their general form, in his own words, *"One can hope that many of the 'patterns' currently being studied will soon cease to be mere ideas, yielding instead directly usable library classes"* (Meyer, 1997, p. 735). Arnout (2004) has built on Meyer's work and considered ways in which software components can be derived from patterns, a process she refers to as 'componentization'. The idea behind componentization is that certain patterns should be capable of producing software components that can help simplify future pattern implementations (Meyer and Arnout, 2006; Arnout, 2004). Arnout (2004, p. 44) believes patterns should not be limited to design reuse, instead they should also be used to promote software reuse through derivable software components.

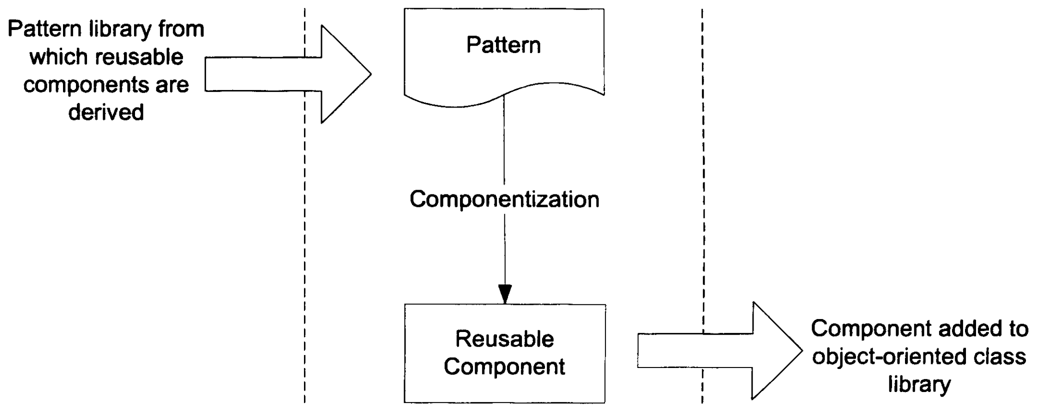

Pattern library from which reusable components are derived
Pattern
Componentization
Reusable Component
Component added to object-oriented class library

Figure 6-1 Overview of the 'componentization' process

Arnout (2004) demonstrates her componentization theory using the Gamma et al. (1995) patterns, Figure 6-1 is an overview of the process. The patterns are transformed to components that are made available as code libraries in the Eiffel programming language (Eiffel Software, 2006). Arnout (2004) argues that this reduces the implementation effort of the patterns in Eiffel as they are not created from scratch each time they are used in their component form.

However, Arnout (2004) realised that it was not possible to componentize all the patterns in the Gamma collection into Eiffel software components. Of the 23 patterns in the Gamma collection, only 15 were componentizable (Arnout, 2004). Several reasons made it impossible to componentize all the patterns. Firstly some of the patterns were too context-sensitive therefore making it impossible to derive components that were not too constrained to a particular context. However, some of the other patterns were less context sensitive but still not componentizable, therefore to support the application of these patterns, Arnout (2004, p. 323) developed Pattern Wizard (this tool is discussed in Section 5.3).

Furthermore, the use of Eiffel as a target programming language made it impossible for some patterns to be componentized. For example, in order to avoid code duplication between different components derived from the DECORATOR pattern, Arnout (Arnout, 2004, p. 78) required the use of genericity[4]. However, this approach was not possible with the version of Eiffel used at the time as all data types need to be known at compile time, hence there was no support for genericity. Arnout (Arnout, 2004, p. 351) also realised that, *"...there are usually many ways to implement a design pattern, which are difficult to capture in a reusable component. It is sometimes feasible to provide several library variants, but it is hardly possible to foresee all possible variations."* This realisation by Arnout adds weight to the 'one pattern, many implementations' discussion in Section 6.1.

Even if componentization was applicable to every pattern, once a component has been packaged and shipped, users of the component are very unlikely to know of how the component was created. In fact users are generally only interested in knowing whether a particular component can solve their problem. Therefore, users wishing to adapt components to design problems similar to those they are intended, are at a disadvantage as they would not have the design solution from which the component was derived. This is typical with object-oriented code libraries commonly used today, for instance Java packages can be easily modified in order to extend their capability within the Java context (Sickle, 1996, p. vi). However, attempting to transform a Java package into a C# package would not be an easy task without the design solution.

Although software components promote software reuse, it is safe to say that a major disadvantage of components is that they are implementation specific and therefore do not promote design reuse. In essence, componentization is based on a 'one pattern, **one implementation**' relationship and not the 'one pattern, many implementations' attribute associated with patterns. A user therefore attempting to use a similar solution in a different context e.g. applying one of Arnout's Eiffel software components in a different programming language such as C# is unable to do so easily without any knowledge of the design solution. This argument is supported by Zimmerman (2005), who argues that adapting software components to different design contexts is not a straightforward task. Hirschfeld et al. (2005) have also suggested that transforming patterns into component packages results in the lack of traceability and therefore unusable design solutions.

To therefore preserve design reuse and at the same time promote code reuse, this thesis proposes a pattern-based code generation approach based on the use of PIEs. With such an approach, a user has the flexibility of selecting an appropriate pattern implementation from a range of PIEs. This approach would adhere to the 'one pattern, many implementations' attribute

[4] Genericity is a technique inherited from generic programming (GP). It is used to create parameterised classes that have different data types (Booch, 1994). A parameterised class must be instantiated before objects can be created.

of patterns. The remainder of this chapter describes this approach in detail.

6.3 A meta-model for the PTTES collection

Heister et al. (1997) have previously argued that if the application domain of a set of patterns is sufficiently narrow, it is possible to define the aspects of a pattern to which code generation principles can be applied. This argument supports the proposition described in this thesis i.e. that a pattern language such as the PTTES collection can be used to support automated code generation by using PIEs as 'first class citizens' in the process. In this context a first class citizen refers to a PIE as a primary entity in the code generation process.

Ideally a pattern-based CASE tool should adhere to a pattern meta-model. A meta-model is a generic model of the building blocks and rules needed to build specific models within a domain of interest (Szemethy, 2006; Saeki et al., 1993). A pattern meta-model provides a unified description of a pattern collection thereby ensuring the integrity of the patterns and their application is consistent. A pattern meta-model can also provide a flexible structure that is adaptable to the evolution of a pattern collection hence supporting the addition of new patterns or indeed PIEs.

Pagel and Winter (1996) have previously described a meta-pattern known as 'Hook & Template' used as a unified description of the Gamma et al. (1995) patterns. They argue that it can be used to establish patterns as first class elements in CASE tools. Hook & Template illustrates the instantiation[5] of individual patterns. However, it does not provide support for the integration of instantiated patterns into an application. Albin-Amiot and Guéhéneuc (2001) have described a similar pattern meta-model for the representation of the Gamma collection. However, Albin-Amiot and Guéhéneuc (2001) argue that besides supporting pattern-based code generation, their meta-model also facilitates the detection of patterns in code.

The meta-models described by Pagel and Winter (1996) and Albin-Amiot and Guéhéneuc (2001) focus on the transformation of individual patterns into code – they do not provide a means of generating integrated applications. A user would still have the task of integrating the generated code from the individual patterns into a single application. This is therefore an incomplete solution that is error prone as users do not adhere to any standard approach when integrating the generated code. Furthermore, such an approach makes it difficult for a pattern-based CASE tool to facilitate reverse engineering as a tool may not be capable of detecting patterns in the application code due to inconsistent implementations. Albin-Amiot and

[5] Instantiation is an object-oriented terminology used to describe the operation of creating an object from its parent class (Sickle, 1996). Instantiation is also commonly used in the patterns community to refer to the implementation of patterns (Wild, 1996).

Guéhéneuc (2001) also make note of these limitations.

To describe their meta-models, Pagel and Winter (1996) and Albin-Amiot and Guéhéneuc (2001) use the UML. This relates to previous work that has attempted to use the UML to represent design patterns (Guennec et al., 2000; Sunye et al., 2000) and recently, work by France et al. (2004) that has proposed a UML-based pattern specification technique that is intended to pave the way for the development of tools that support the rigorous application of object-oriented design patterns. Using the UML to specify the Gamma et al. (1995) patterns has been made possible by the fact that these patterns conform to object-oriented design principles on which the UML is closely aligned with.

For reasons already discussed in Section 5.2, it was decided not to use the UML to specify the PTTES meta-model. Instead, the meta-model described here is based on the PTTES form described in Section 4.2. Figure 6-2 is an illustration of the meta-model used to represent the PTTES patterns. It consists of a design tier that represents the design analysis phase of development based on an appropriate pattern and an implementation tier that represents the implemented pattern as a PIE.

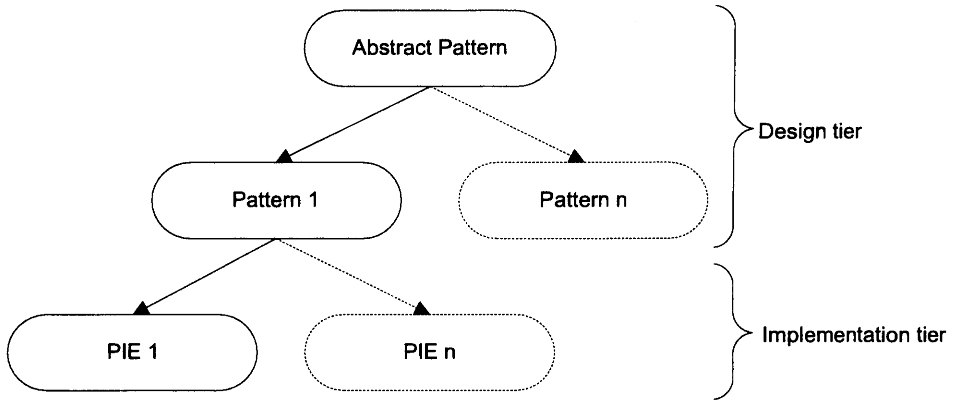


Figure 6-2 The PTTES meta-model

The generic form of the PTTES meta-model makes it possible for a pattern CASE tool to support the application of the PTTES collection as each pattern conforms to it. For example, Figure 6-3 illustrates how the PTTES meta-model can be used to implement TTC PLATFORM. TTC PLATFORM can be applied in various contexts by using its associated patterns such as TTC SCHEDULER generally used to implement applications with high resource constraints. Each pattern then has a number of PIEs associated with different hardware platforms such as the ARM and 8051 microcontrollers (Pont and Mwelwa, 2003b; Pont, 2001).

The PTTES meta-model is comparable to the OMG's MDA methodology described in Section 5.2. In the MDA methodology a PIM is first defined independently of the implementation technology and then transformed into a PSM using transformation rules. This compares to the PTTES meta-model where a pattern is transformed into a PIE based on a specified hardware platform. However, in the PTTES meta-model the PIE generated from a pattern is actual code whereas the PSM is a conceptualised implementation that is later transformed into code. Figure 6-4 compares the two concepts.

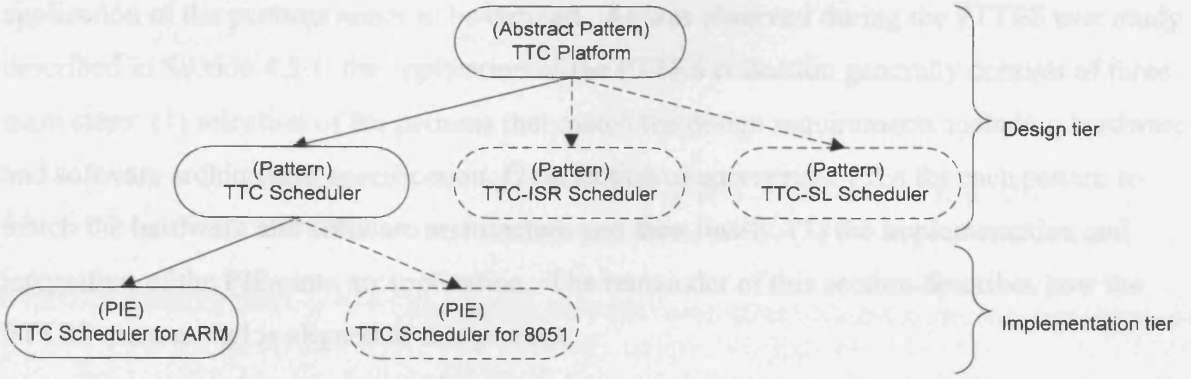

(Abstract Pattern) TTC Platform
(Pattern) TTC Scheduler
(Pattern) TTC-ISR Scheduler
(Pattern) TTC-SL Scheduler
Design tier
(PIE) TTC Scheduler for ARM
(PIE) TTC Scheduler for 8051
Implementation tier

Figure 6-3 Using the PTTES meta-model to define TTC Platform

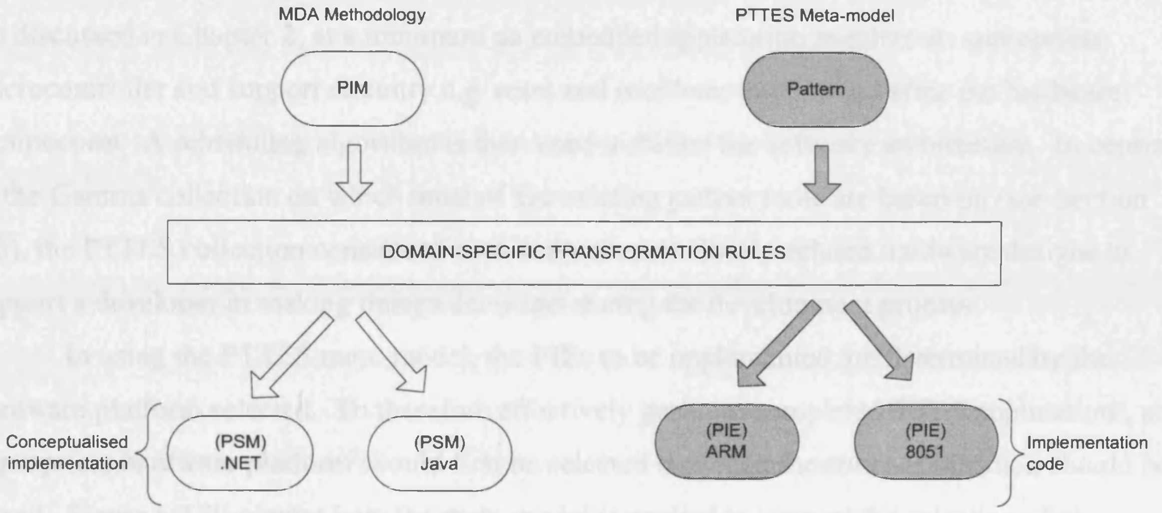


Figure 6-4 Comparison of the MDA and the PTTES meta-model

Most of the pattern meta-models previously described have been based on the Gamma collection that is applicable to high-level programming languages such as Java that generally result in code that is implementation independent. In contrast, as discussed in Chapter 2, the choice of hardware platform impacts the software design of an embedded system. The PTTES meta-model described here therefore uses PIEs as a means of associating a software pattern with a particular hardware platform; this is illustrated in Figure 6-3 where an appropriate PIE for TTC PLATFORM is implemented for a specific microcontroller. As discussed in Section 6.1, it is practically impossible to develop a CASE tool that can generate an infinite number of pattern implementations. Using PIEs as a means of associating software patterns with a particular hardware platform therefore allows a CASE tool to support a finite number of PIEs and in so doing adhering to the 'one pattern, many implementations' attribute of patterns.

6.4 Applying the PTTES meta-model

In order to implement the PTTES meta-model in a CASE tool, a process that describes the application of the patterns needs to be defined. As was observed during the PTTES user study described in Section 4.3.1, the application of the PTTES collection generally consists of three main steps: (1) selection of the patterns that match the design requirements including hardware and software architecture specification, (2) selection of appropriate PIEs for each pattern to match the hardware and software architecture and then finally, (3) the implementation and integration of the PIEs into an application. The remainder of this section describes how the PTTES meta-model is aligned to this process.

6.4.2 Implementing a TT framework

As discussed in Chapter 2, at a minimum an embedded application requires an appropriate microcontroller and support circuitry e.g. reset and oscillator circuits to define the hardware architecture. A scheduling algorithm is then used to define the software architecture. In contrast to the Gamma collection on which most of the existing pattern tools are based on (see Section 5.3), the PTTES collection consists of both software and closely-related hardware designs to support a developer in making design decisions during the development process.

In using the PTTES meta-model, the PIEs to be implemented are determined by the hardware platform selected. To therefore effectively generate complete PTTES applications, an appropriate hardware platform should first be selected on which the entire application should be based. Figure 6-5 illustrates how the meta-model is applied to support the selection of an appropriate microcontroller using the hardware foundations patterns: SMALL 8051, STANDARD 8051 and EXTENDED 8051. These patterns support a user in making appropriate design decisions regarding hardware constraints such as timing and memory that may influence the choice of microcontroller (Pont, 2001, pp. 29-52).

Once a decision has been made on the microcontroller to be used, the next step would be to decide on the software architecture. As discussed in Section 2.4, the software architecture of an embedded application is characterised by its scheduling algorithm. In order to select an appropriate scheduler the PTTES meta-model is applied to an appropriate scheduler pattern, e.g. TTC PLATFORM as illustrated in Figure 6-3.

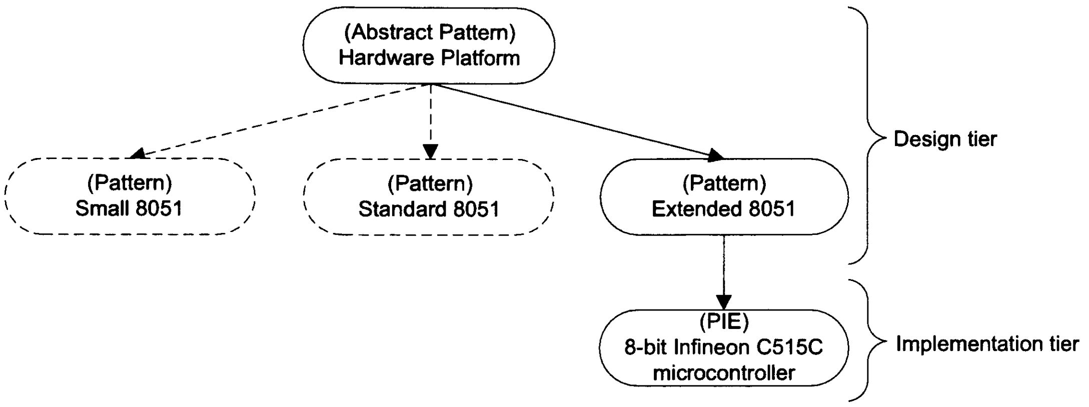


Figure 6-5 Applying the PTTES meta-model to the hardware foundations patterns

A combined hardware and TTC software architecture that meets both design and budget constraints provides a framework on which a TT embedded application can be built. The following sub-section describes how the PTTES meta-model is used to build on the TT framework to complete the development of a fully functional application.

6.4.3 Using PIEs as first class entities in pattern-based code generation

In Section 5.4 it was argued that pattern tools based on the Gamma collection (see Section 5.3) support the implementation of individual patterns without reference to any other part of the design. As a consequence these tools have focused on implementing individual patterns and not on generating application code from groups of patterns. These pattern tools have therefore generally ignored the challenges of integrating pattern implementations.

The inability of these pattern tools to support the integration of patterns can be attributed to the fact that the Gamma collection on which most of them are based is not a complete pattern language; this is discussed in Section 3.2.3. In contrast, the PTTES collection is a pattern language as it allows users to design fully functional time-triggered embedded applications using an appropriate combination of patterns (see Section 4.2). Therefore, an appropriately designed CASE tool should make it possible to generate complete systems from the PTTES collection.

The PTTES meta-model described in Section 6.3 defines how patterns in the PTTES collection are transformed into code using PIEs, however in order to generate complete systems composed of integrated PIEs, a means of integrating these PIEs needs to be defined. Section 2.4, notes the importance of a scheduler to an embedded system, in particular how it is used to manage system resources and task scheduling. When using any form of scheduler (or larger operating system), there is a natural way of integrating tasks. Within the PTTES collection, TTC SCHEDULER provides a mechanism of linking PIE tasks with the system through the use of the 'Add Task' function. Where software patterns have an associated periodic task (as many do), it is a straightforward process to generate an appropriate Add Task function call, based on information provided by the user. Figure 6-6 is an example of how the Add Task function is used to integrate multiple periodic tasks into the same system design.

```
SCH_Add_Task(HEARTBEAT_LED_Update, 0, 1000);
SCH_Add_Task(SWITCH_Update, 0, 1000);
SCH_Add_Task(KEYPAD_Update, 0, 1000);
SCH_Add_Task(LED_MX_Display_Update, 0, 1000);
```

Figure 6-6 Example of the use of the 'Add Task' function to integrate multiple periodic tasks into a system design

It therefore follows that the scheduler implemented in the TT framework (described in Section 6.4.2) is used to facilitate the integration of software PIEs into the system. Furthermore, the microcontroller resources required by each software PIE are accounted for during integration and subtracted as they are used. This helps prevent basic design errors, for example where more than one task has assumed exclusive access to a particular port pin. This is a basic, but useful feature when working with multiple PTTES patterns in a system design.

6.5 Chapter conclusions

This chapter has argued that it is practically impossible to develop a CASE tool capable of transforming software patterns into all their possible implementations. Instead an appropriate means of limiting the number of pattern implementations should be defined. To this effect a PTTES meta-model has been described to uphold the 'one pattern, many implementations' attribute of patterns. In this meta-model, hardware platforms are used as a means of defining the number of pattern implementations that can be generated.

It has also been argued that attempting to generate code from patterns that are not part of a pattern language makes it difficult, if not almost impossible, for fully functional systems to be generated. Using the PTTES collection, a pattern language described in Chapter 4, this chapter has described an approach for pattern-based code generation that can support the generation of applications from patterns. To test this theory the next chapter describes a pattern-based code generation CASE tool that applies this approach to the PTTES collection.

7 PTTES Builder: A Pattern-based Code Generation CASE Tool[6]

To test the applicability of the pattern-based code generation approach described in the previous chapter, a prototype CASE tool: PTTES Builder was developed. The purpose of PTTES Builder is not to aid the design deliberations of a user e.g. which pattern to use and when. This is left to the user, as the objective is not to automate the design of software applications. Instead, the primary goal of the tool is to transform a user's pattern-based design into a complete integrated application, where possible. This chapter describes the implementation of PTTES Builder.

7.1 Overview of PTTES Builder

PTTES Builder, consists of five main modules: (i) a repository used to house the patterns and PIEs (ii) a help facility used to browse the pattern documentation (iii) a GUI-based 'wizard' used to facilitate the selection of patterns and the configuration of their PIEs (iv) a code generator that integrates PIEs into an application (v) a 'code viewer' also integrated with the user interface that is used to view the generated code. Figure 7-1 is an overview of these modules and their associations. The remainder of the chapter describes the design and implementation of these modules.

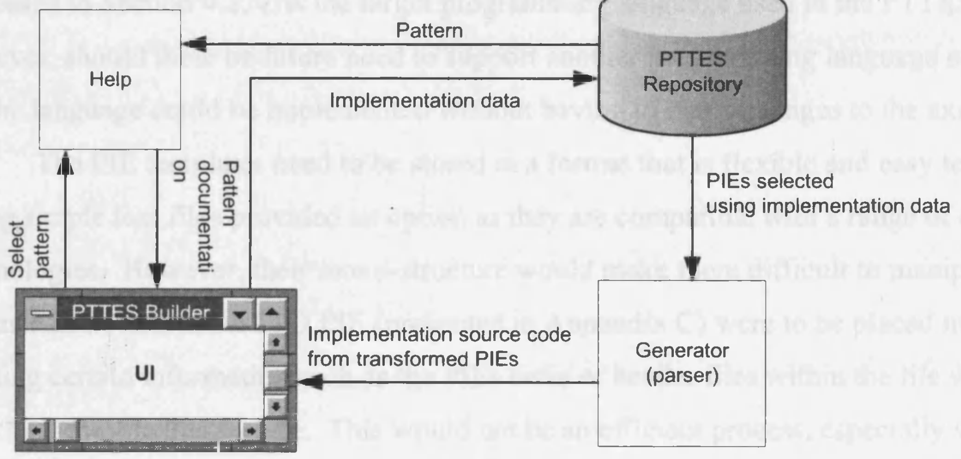


Figure 7-1 Overview of PTTES Builder's architecture

7.2 The PTTES repository

The repository is divided into a design and implementation tier as defined by the PTTES meta-model.

[6] Some of the contents of this chapter have previously been published in (Mwelwa et al., 2006) and (Mwelwa et al., 2004(b)).

7.2.1 Defining the implementation tier

The implementation tier is composed of PIEs that are used as the primary entities in the code transformation process. It was therefore important to find a suitable representation for the PIEs that would facilitate code generation.

The representation of the PIEs also had a bearing on the mechanism used for the code generation process. It was therefore decided to use template-based code generation (this is discussed in detail in Section 7.3). One of the determinant factors in this decision was the fact that using templates for the code generation process would support extensibility. As previously discussed in Section 4.3.2, pattern collections will continue to evolve for as long as they remain useful. Therefore, in order for PTTES Builder to remain valuable as the PTTES collection evolved, the PTTES repository would need to be extensible. Krishnamurthi and Felleisen (1998) define extensibility as the property of an application that allows it to be easily extended without causing undesirable side effects to the application. This is an important feature as it takes into consideration the evolution of an application either through the addition of new functionality or modification of existing functionality.

A template-based design would allow straightforward additions of new patterns or modifications to existing patterns in the repository without having to make subsequent changes to existing parts of the repository (Herrington, 2003). For example, for reasons already discussed in Section 4.2, C is the target programming language used in the PTTES collection, however, should there be future need to support another programming language new templates for the language could be implemented without having to make changes to the existing ones.

The PIE templates need to be stored in a format that is flexible and easy to manage. Using simple text files provided an option as they are compatible with a range of other technologies. However, their mono-structure would make them difficult to manipulate. For instance, if a HEARTBEAT LED PIE (presented in Appendix C) were to be placed in a text file, locating certain information such as the PIEs tasks or header files within the file would require searching the file line by line. This would not be an efficient process, especially when dealing with dozens of files with hundreds of lines of code. Another option would have been to use a relational database to store the PIEs. The ability to use SQL queries to retrieve data is very powerful. However, although databases are the most commonly used mechanism to store data, distributing an application that uses a database can be challenging and costly. Considering that this work was research based as opposed to product development, a database was therefore not used.

Instead, the Extensible Markup Language (XML) was used to define the PIEs in the repository. XML is frequently used to exchange and store structured data and because of its platform independence, it is becoming a standard for data exchange between different platforms and technologies (O'Reilly, 2004). XML also provides features similar to those found in databases e.g. storage in the form of XML documents, schemas and style-sheets (e.g. DTD and XSLT), query languages (e.g. XQuery and XPath) and programming interfaces (e.g. SAX and DOM). Because XML documents have no predefined tags, they can be customised to match the structure of the PIEs. Furthermore, because the essence of XML is the separation of content from presentation, it allowed for the separation of the PIEs from their associated patterns as illustrated in the PTTES meta-model (see Figure 6-2).

The PTTES repository is a file directory based on a simple filing system. Within the repository each pattern has a dedicated directory; Figure 7-2 illustrates this directory structure using HEARTBEAT LED as an example. The 'docs' folder contains the files associated with the design tier and the 'platform' folders correspond to the hardware platforms supported, and as is discussed in the previous chapter, the hardware platforms and associated microcontrollers determine which PIEs are used. In this work, the 8051 platform, in particular the C515C Infineon microcontroller and the AT89S53 microcontroller from Atmel were used. This version of PTTES Builder was therefore capable of implementing two PIEs for each software pattern supported. As illustrated in Figure 7-2, each PIE is associated with a microcontroller and within the associated folder is an XML file and related Extensible Style-sheet Language Transformation (XSLT) files. The XML file is used to define user specified implementation data, for example tags such as `<scheduler_task>` are used as placeholders for data that is specified during the PIE configuration process. Each XML file has a unique namespace used to preserve consistency and avoid redundancy within the repository.

The XSLT files are used during the code generation process (described in Section 7.3). XSLT is primarily used to transform XML documents into other types of documents or formats (Harold, 2002). In this case the XML files were transformed into C source code. The XSLT files are associated with a PIE's header and implementation files. Header (or definition) files usually contain function prototypes and any macro definitions. Ideally they should not contain executable code beyond macro definitions. This is reserved for the implementation files (Sickle, 1996). Figure 7-4 is the XSLT file responsible for the generation of HEARTBEAT LED header file for the C515C microcontroller. Tags within this file are also used as place holders for project specific information e.g. the tag: `<xsl:value-of select="@lastupdate"/>` specifies the date and time that the file was generated. During the code generation process certain data within the XSLT files is obtained from the associated XML file, for instance function prototypes are retrieved from the `<header_file>` element where they are listed (see Figure 7-3).

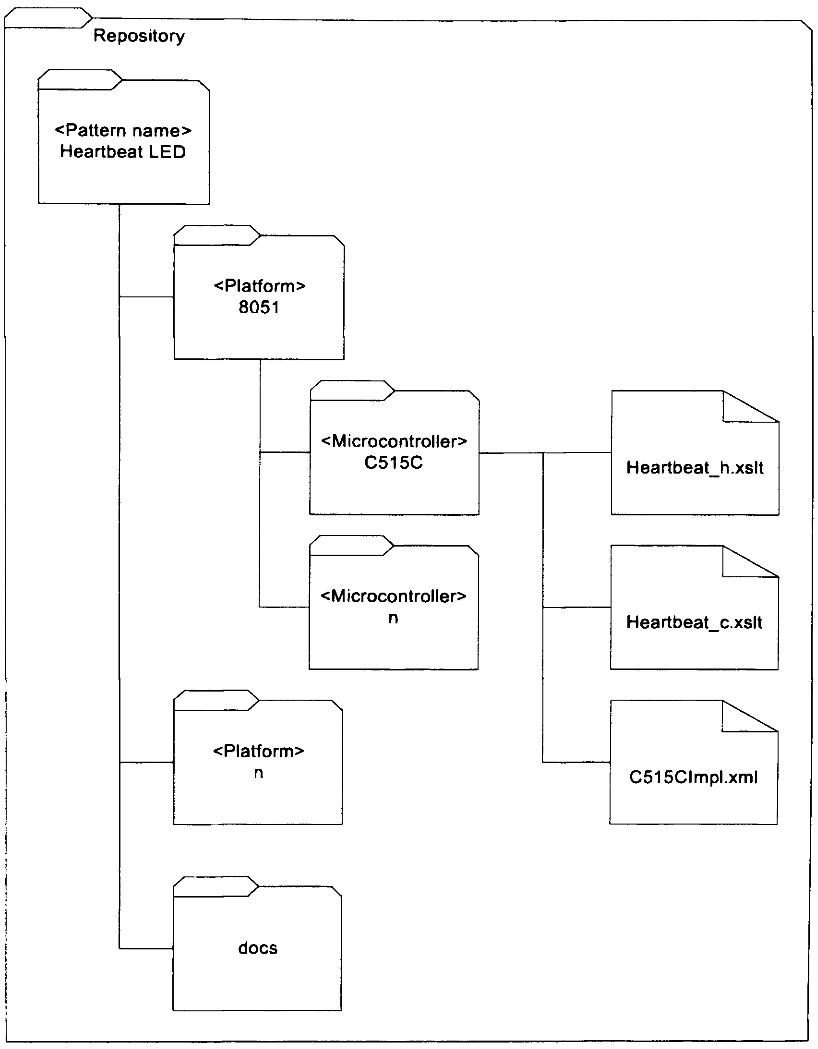


Figure 7-2 The directory structure implemented in the pattern repository

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!DOCTYPE pattern SYSTEM "../pattern.dtd" [
        <!-- Heartbeat LED implementations -->
        <!ENTITY 8051Impl SYSTEM "implementation/8051Impl.xml">
]>
<!-- A list of Heartbeat LED pattern implementations -->
<pattern display_name="&pattern.heartbeat-led;" help_topic="patterns.heartbeat-led.index">
<pattern_implementation xmlns="http://www.le.ac.uk/eg/cm55/heartbeatled/8051Impl">
   <modules>
        <module xmlns="http://www.le.ac.uk/eg/cm55/heartbeatled/heartbeatled">
        <source_file name="heartbeat_led.c">
        <task_header_files>
        <task_header_file>#include &quot;heartbeat_led.h&quot;&#xA;</task_header_file>
        </task_header_files>
        <init_tasks>
        <init_task>&#x20;&#x20;&#x20;HEARTBEAT_LED_Init();&#xA;</init_task>
        </init_tasks>
        <scheduler_tasks>
        <scheduler_task name="HEARTBEAT_LED_Update">HEARTBEAT_LED_Update</scheduler_task>
        </scheduler_tasks>
        <task_pin_variables>
        <task_pin_variable name="HEARTBEAT_LED_pin">sbit HEARTBEAT_LED_pin</task_pin_variable>
        </task_pin_variables>
        </source_file>
        <header_file name="heartbeat_led.h">
        <prototypes>
        <prototype>void HEARTBEAT_LED_Init(void);</prototype>
        <prototype>void HEARTBEAT_LED_Update(void);</prototype>
        </prototypes>
        </header_file>
       </module>
   </modules>
</pattern_implementation>
</pattern>
```

Figure 7-3 A snippet of the XML file that represents a Heartbeat LED PIE

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text"/>
<xsl:template match="/">
<xsl:apply-templates/>
</xsl:template>
<xsl:template match="project">
/*------------------------------------------------------------------
  This code was generated by the PTTES Builder.
  File: heartbeat_led.h
  Project: <xsl:value-of select="@project"/>
  Author: <xsl:value-of select="@author"/>
  Last updated: <xsl:value-of select="@lastupdate"/>
-----------------------------------------------------------------*/
<![CDATA[
#ifndef __HEARTBEAT_LED_H
#define __HEARTBEAT_LED_H

#include "main.h"

/* ------ Public function prototypes ------------------------------- */
]]>
<xsl:for-each
select="pattern_implementation/modules/module/header_file[@name='heartbeat_led.h']/prototypes">
<xsl:value-of select="."/>
</xsl:for-each>

#endif
/*------------------------------------------------------------------*-
---- END OF FILE -------------------------------------------------
-*------------------------------------------------------------------*/
</xsl:template>
</xsl:stylesheet>
```

Figure 7-4 Heartbeat LED XSLT file used to generate a header file from a PIE

7.2.2 Representing the design tier

In the manual application of the PTTES collection, described in Section 4.3.1, users refer to the PTTES book in order to come up with their design decisions. To simplify this book based process, the PTTES documentation was integrated into the CASE tool.

Initially a simple text based format was used to present the patterns, however this did not provide a flexible means of browsing and viewing the patterns – it was too static. XML was not an ideal option as it is specifically used for data exchange and storage. What was needed was a format that can present data dynamically. HTML was therefore found to be appropriate as it supports quicker and flexible browsing with hypertext. For example, when browsing LOOP TIMEOUT (presented in Figure 4-1 in Section 4.2) one can easily browse straight to HARDWARE TIMEOUT by using a hypertext link in the 'related patterns' section. The HTML files were stored in the 'docs' folder within the patterns associated directory as illustrated in Figure 7-5. The pattern documentation is accessed using a custom browser; this is described in Section 7.4.

The current version of PTTES Builder supports eleven patterns from the PTTES collection: STANDARD 8051, EXTENDED 8051, CRYSTAL OSCILLATOR, TTC SCHEDULER, PORT WRAPPER, HEARTBEAT LED, ONE-SHOT ADC, PID CONTROLLER, PULSE COUNT, RC RESET and PC LINK. In terms of hardware, the Atmel AT89S53 microcontroller and Infineon C515C microcontrollers are supported i.e. Standard 8051 and Extended 8051 platforms respectively.

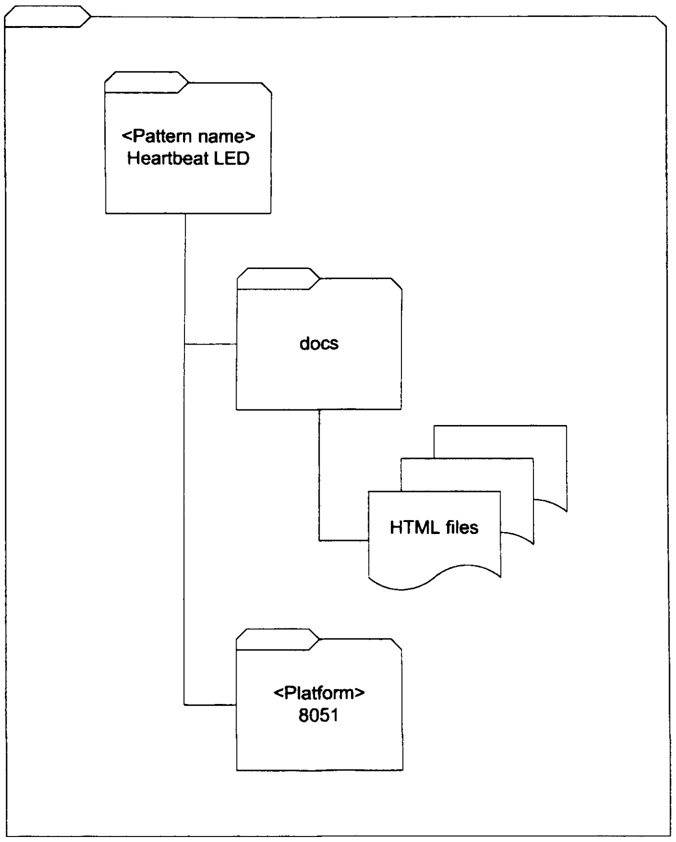


Figure 7-5 The directory structure within the repository used for the pattern documentation

7.3 The code generation process

The code generation process takes place after a user has selected their required patterns and specified the implementation preferences. A project file is created for every application developed within PTTES Builder and is used to keep a record of the selected patterns and implementation data. This file serves as the code generators reference point. The following subsections discuss the project file and code generator in more detail.

7.3.3 The project file

The project file is based on the XML format. Using tags, the project file is able to store project data such as the microcontroller to be used, oscillator frequency and the task execution rates. The project file is populated with data as the user selects patterns and specifies implementation data using the wizard (this is discussed in detail in Section 7.4.5).

For instance, Figure 7-6 lists the contents of a project file after a user has selected an appropriate microcontroller, set the oscillator frequency, and configured a scheduler. Once all the required patterns have been selected and configured, the project file is parsed through the code generator. The code generator is a parser that uses the data in the project file to call the necessary PIE XSLT templates that are then combined with the data in the project file to generate code. This code generation process is described in detail in the next section.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<project author="Jo Bloggs" lastupdate="18/6/2006" project="demo">
<pattern_implementation xmlns="http://www.le.ac.uk/eg/cm55/extended8051">
<modules xmlns="/embeddedbuilder/repository/pttes/extended8051/c515cImpl.xml">
<hardware_listing>
<hardware_item>Infineon C515C development board</hardware_item>
</hardware_listing>
</modules>
</pattern_implementation>
<pattern_implementation xmlns="http://www.le.ac.uk/eg/cm55/cooperativescheduler/ c515cImpl ">
<modules xmlns="/embeddedbuilder/repository/pttes/cooperativescheduler/c515c_implementation
/c515cImpl.xml">
<module xmlns="http://www.le.ac.uk/eg/cm55/cooperativescheduler/c515cImpl/2_01_10i">
<source_file name="2_01_10i.c">
<task_header_files>
<task_header_file>#include "2_01_10i.h"</task_header_file>
</task_header_files>
<init_tasks>
<init_task>SCH_Init_T2();</init_task>
</init_tasks>
</source_file>
<header_file name="2_01_10i.h">
<prototypes>
<prototype>void SCH_Init_T2(void);</prototype>
<prototype>void SCH_Start(void);</prototype>
</prototypes>
</header_file>
</module>
<module xmlns="http://www.le.ac.uk/eg/cm55/cooperativescheduler/c515cImpl/sch51">
<source_file name="Sch51.c">
<task_header_files>
<task_header_file>#include "Sch51.h"</task_header_file>
</task_header_files>
</source_file>
<header_file name="Sch51.h">
<sch_max_tasks>#define SCH_MAX_TASKS (0)</sch_max_tasks>
</header_file>
</module>
<module xmlns="http://www.le.ac.uk/eg/cm55/cooperativescheduler/c515cImpl/main">
<source_file name="Main.c"/>
<header_file name="Main.h">
<microcontroller>#include "REG515C.h"</microcontroller>
<osc_freq>#define OSC_FREQ (10000000UL)</osc_freq>
<osc_per_inst>#define OSC_PER_INST (12)</osc_per_inst>
</header_file>
</module>
</modules>
</pattern_implementation>
</project>
```

Figure 7-6 A snippet of a PTTES Builder project file

Apart from facilitating the code generation process, the project file also enables a user to remove a pattern from a project, change its configuration or even swap it for another pattern. PTTES Builder is able to do this without affecting existing generated code. However, this is only possible within the tool environment, if changes are made to the generated code outside the PTTES Builder environment, the tool cannot keep track of the changes made. The project file also keeps a record of the microcontroller resources such as Port Pins consumed by each pattern thereby reducing any possibilities of hardware conflicts.

7.3.4 The code generator

In order to generate code, the project file is parsed through the code generator. Central to the operation of the code generator is a parser that is used to process the project file. As the project file and template files are based on XML and XSLT technologies, a parser capable of reading these formats was required. At present there are only two major API's that define how XML parsers work: Simple API for XML (SAX) and Document Object Model (DOM). SAX is an event-driven mechanism for accessing XML documents whereas the DOM specification defines a tree-based approach to navigating an XML document at run-time (Harold, 2002).

A DOM tree is stored in memory and can be navigated and manipulated using functions that return parent and child nodes thus providing access to the XML document data. On the other hand SAX does not have a run time representation of the parsed document. Instead, the parser calls handler functions when certain events (defined by the SAX specification) take place. These events include the start and end of the document, finding a text node, finding child elements, and hitting a malformed element (O'Reilly, 2004; Harold, 2002). The strength of the SAX specification is that it parses gigabytes of XML documents without hitting resource limits, because it does not create a representation in memory. It is therefore generally faster and requires fewer resources. However, SAX is much more complex to use as the lack of a run time representation requires writing extra code to manipulate and traverse XML documents (O'Reilly, 2004; Harold, 2002; Harold and Means, 2002). Based on this comparison, the DOM was thought to be the appropriate parser to use in the circumstances.

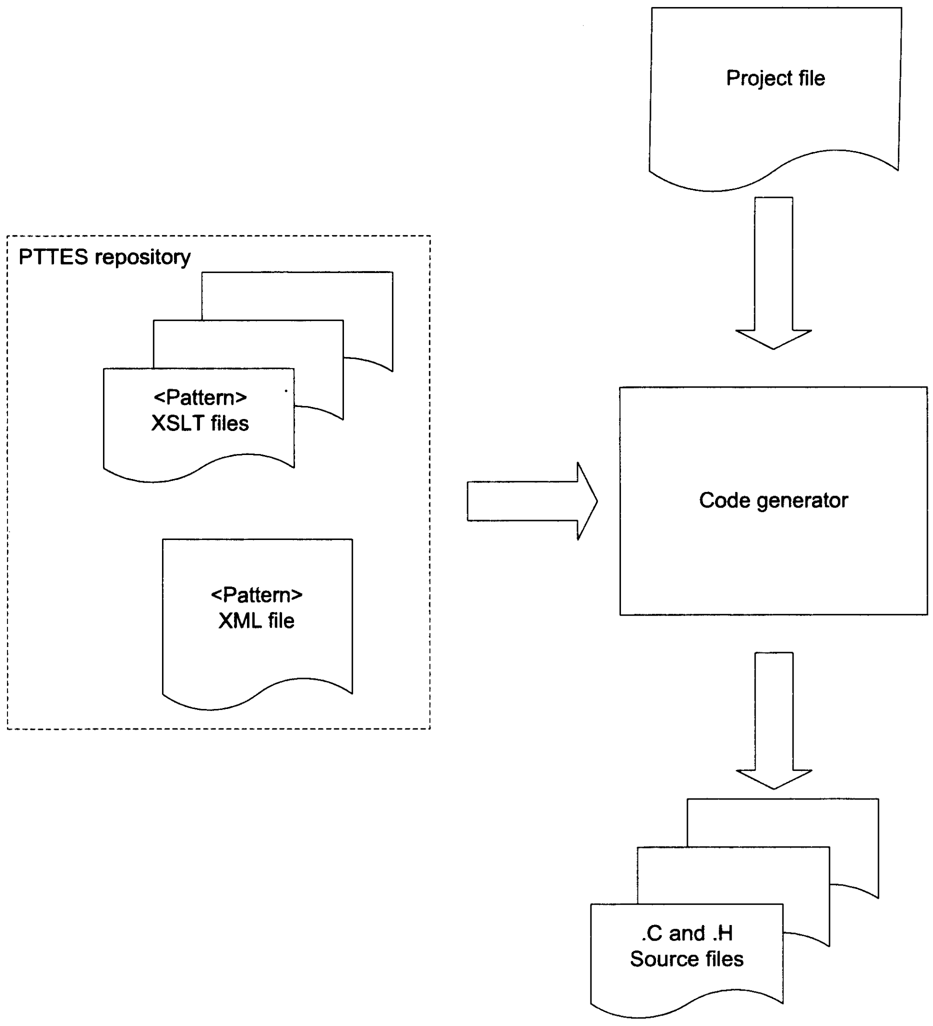


Figure 7-7 The PTTES Builder code generation process

Using the data in the project file, the DOM calls the XSLT templates associated with the selected PIEs. It must be noted that the order of the code generation process is based on the order in which the PIEs are listed in the project file i.e. top to bottom. The first two PIEs listed are associated with the hardware platform and scheduler. This is based on the PTTES meta-model described in Section 6.3 that specifies that a microcontroller and scheduler should always be configured first so as to allow subsequent patterns to be integrated into the system. Figure 7-7 gives an overview of this code generation process.

7.4 The user interface

A text based user interface (UI) was implemented in the Java programming language. Java was used for a number of reasons. Firstly, it is platform independent and supports a range of APIs such as the DOM used in the tool's code generator; therefore XML data can be easily accessed and manipulated in this language. Apart from its compatibility with XML technologies, Java's Open Source status also provides a wealth of useful code libraries such as Swing components used to facilitate the development of graphical components.

The user interface is composed of three main components, a wizard, pattern dialog boxes and a browser. The following subsections describe these components in detail.

7.4.5 The PTTES Builder wizard

In Section 6.4.2 it was noted that a TT framework composed of the hardware and software architecture is required in order to generate complete systems using the PTTES collection. In order to ensure that users do create a basic framework for their systems, the tool requires that a wizard is run at the beginning of each new project. The wizard guides the user through the implementation of a TT framework. Once a TT framework has been implemented a user can then add (or remove / replace) application specific patterns in a project using 'pattern dialog boxes' (discussed in the next section). Figure 8-3, Figure 8-4 and Figure 8-5 in Chapter 8 illustrate the use of the wizard in configuring the hardware and software architecture of an application.

7.4.6 Pattern dialog boxes

Application specific software patterns are added to the project by use of 'Add pattern' and 'Remove pattern' buttons on the front panel of the tool's main user interface (see Figure 7-8). The actual PIE added to the project is determined automatically by the tool by means of the TT framework initially implemented.

The PIEs are configured via dialog boxes that pop up when the 'Add pattern' button is pressed. The dialog boxes are linked to the associated pattern's XML file in the repository. The configuration information supplied by the user is stored in the project file. As a result of using these dialog boxes, most of the (potentially large) list of available PIEs and extraneous pattern information is hidden from the user: instead, the tool takes into account the microcontroller to be used and other context information to present an appropriate list of PIEs applicable to the particular project.

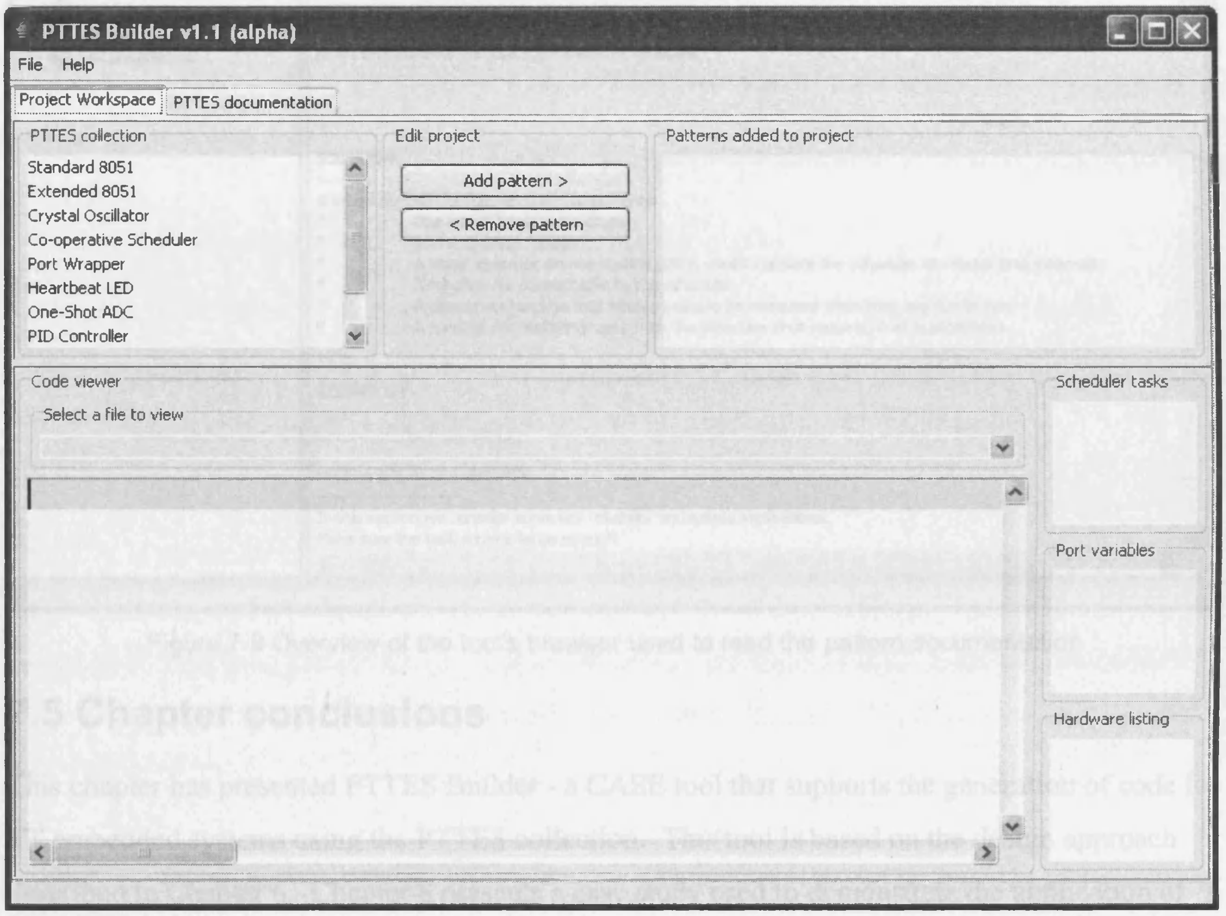
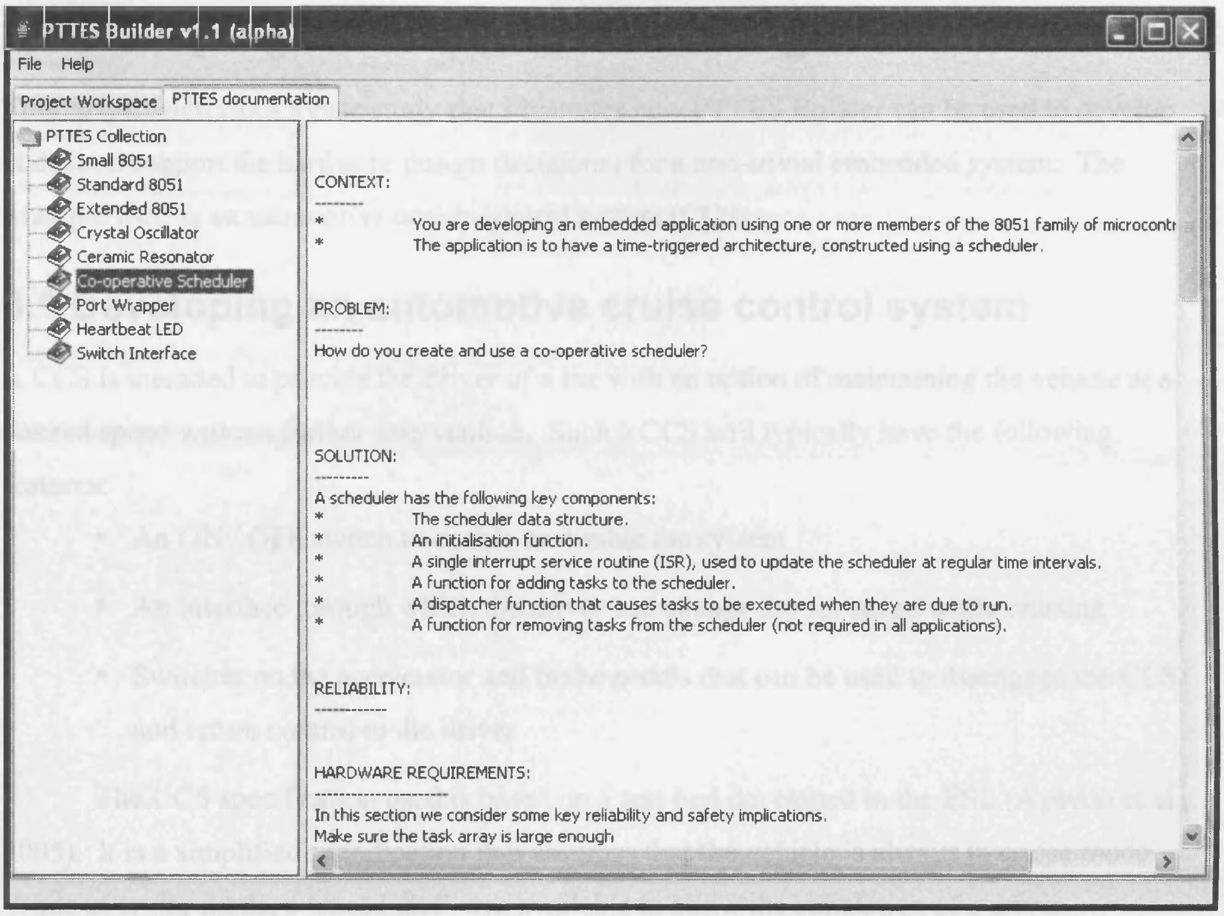


Figure 7-8 The main PTTES Builder UI panel

7.4.7 Code and pattern browser

As each PIE is configured or removed from the project, code is automatically re-generated. This code is viewable in the tool's code viewer (see Figure 8-6, Figure 8-7 and Figure 8-8 in Chapter 8). It must be noted however that this version of the tool does not permit a user to edit the generated code within the tool as it can not keep track of changes made by users to the code.

The pattern documentation can be viewed during the development process within the tool. This is done using a custom HTML browser illustrated in Figure 7-9.



CONTEXT:
----
* You are developing an embedded application using one or more members of the 8051 family of microcontr
* The application is to have a time-triggered architecture, constructed using a scheduler.

PROBLEM:

How do you create and use a co-operative scheduler?

SOLUTION:

A scheduler has the following key components:
* The scheduler data structure.
* An initialisation function.
* A single interrupt service routine (ISR), used to update the scheduler at regular time intervals.
* A function for adding tasks to the scheduler.
* A dispatcher function that causes tasks to be executed when they are due to run.
* A function for removing tasks from the scheduler (not required in all applications).

RELIABILITY:

HARDWARE REQUIREMENTS:

In this section we consider some key reliability and safety implications.
Make sure the task array is large enough

Figure 7-9 Overview of the tool's browser used to read the pattern documentation

7.5 Chapter conclusions

This chapter has presented PTTES Builder - a CASE tool that supports the generation of code for TT embedded systems using the PTTES collection. This tool is based on the design approach described in Chapter 6. Chapter 8 presents a case study used to demonstrate the application of PTTES Builder in the development of a non-trivial TT embedded application.

8 Developing a TT Application Using PTTES Builder[7]

This chapter describes a case study that illustrates how PTTES Builder can be used to develop code (and support the hardware design decisions) for a non-trivial embedded system. The example used is an automotive cruise-control system (CCS).

8.1 Developing an automotive cruise control system

A CCS is intended to provide the driver of a car with an option of maintaining the vehicle at a desired speed without further intervention. Such a CCS will typically have the following features:

- An ON / OFF switch to enable or disable the system
- An interface through which the driver can change the set speed while cruising
- Switches on the accelerator and brake pedals that can be used to disengage the CCS and return control to the driver

The CCS specification used is based on a test-bed developed in the ESL (Ayavoo et al., 2005). It is a simplified specification that assumes that the vehicle is always in cruise mode. While in cruise mode, a 'speed dial' was available to allow the simulation of a driver dynamically changing the car speed. The tasks used to implement the CCS are illustrated in Table 8-1. An Infineon C515C microcontroller (supported by PTTES Builder) was used as the hardware platform.

Table 8-1 CCS tasks implemented

Task Names	**Task Description**	**Task Period (ms)**
LED Flash Update	Flashing LED to indicate that the board is working	1000
Compute Car Speed	Computes the car speed obtained from car model	50
Compute Throttle	Calculates and sends the required throttle to be applied to the car model	50
Get Ref Speed	Gets the desired speed from the driver	1000

[7] Some of the contents of this chapter have previously appeared in (Mwelwa et al., 2006), (Mwelwa et al., 2005) and (Mwelwa et al., 2004(a)).

A computational model was used to represent the environment in which the CCS would operate. The core of the car model was a simplified physical model based on Newton's laws of motion. This model had one input (current throttle) and one output (a train of pulses representing the speed of the car) and was implemented on a desktop PC running the DOS operating system. Figure 8-1 illustrates the CCS setup as described in (Ayavoo et al., 2005).

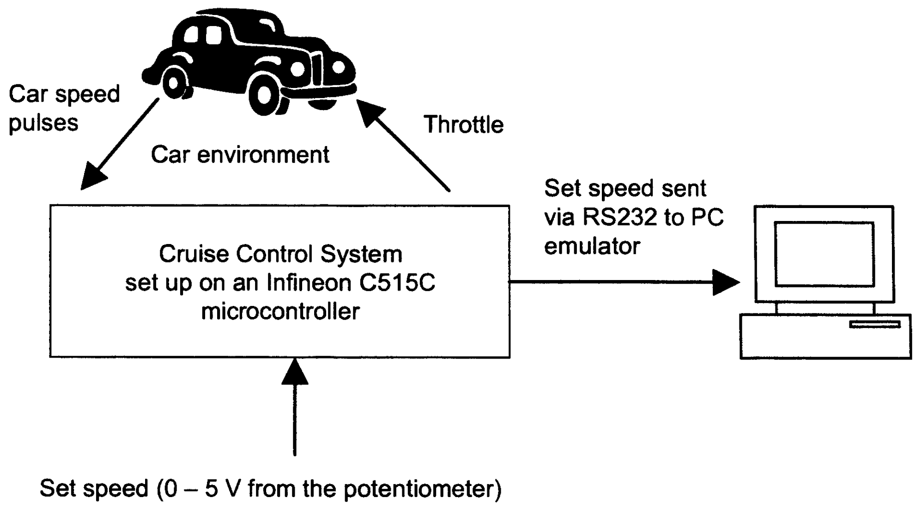

Car speed pulses
Throttle
Car environment
Set speed sent via RS232 to PC emulator
Cruise Control System set up on an Infineon C515C microcontroller
Set speed (0 – 5 V from the potentiometer)

Figure 8-1 An overview of the CCS test bed (adapted from: Ayavoo et al., 2005)

8.2 Using PTTES Builder to develop the CCS

This section describes the use of the PTTES Builder to develop the CCS.

8.2.1 Creating a TT architecture for the CCS

8.2.1.1 Implementing the hardware architecture

The Infineon C515C microcontroller used in the study is an Extended 8051 microcontroller that consists of 3 timers, a UART and an analogue-to-digital converter (ADC), among other hardware features.

The other hardware issues to be considered at this stage are the design of suitable oscillator and reset circuits. The development board used in this project includes these features and - in this project - the only configuration required on the board was the setting of the oscillator frequency (10 MHz). Figure 8-3 and Figure 8-4 illustrate the selection and configuration of a microcontroller using the PTTES Builder wizard.

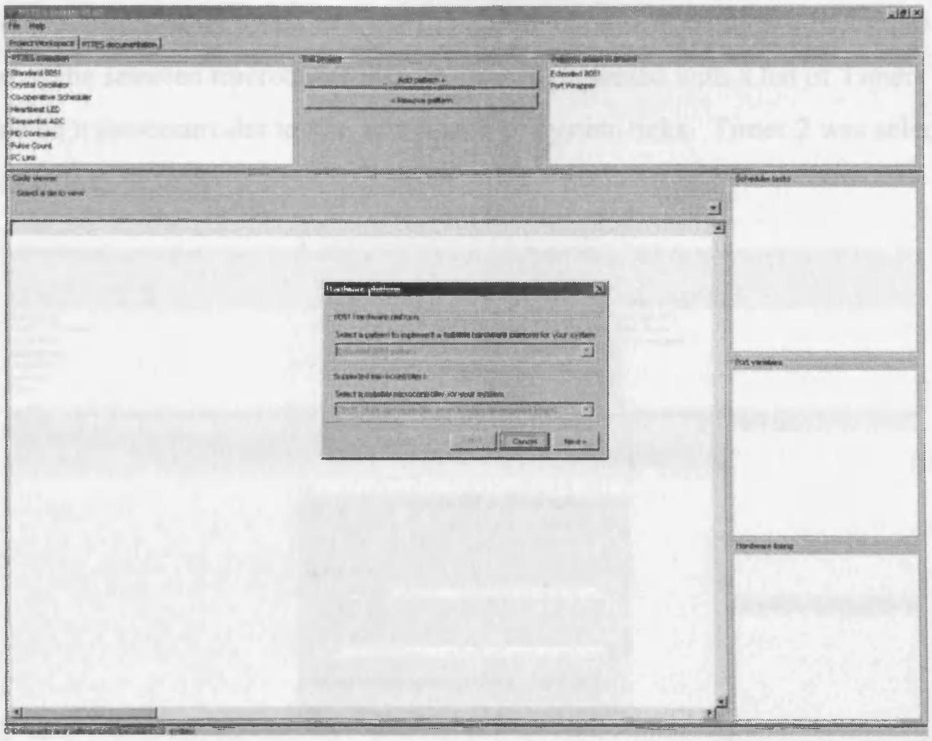


Figure 8-2

Figure 8-3 First step in implementing a system using the tool, is the selection of an appropriate hardware platform, the Extended 8051 was selected

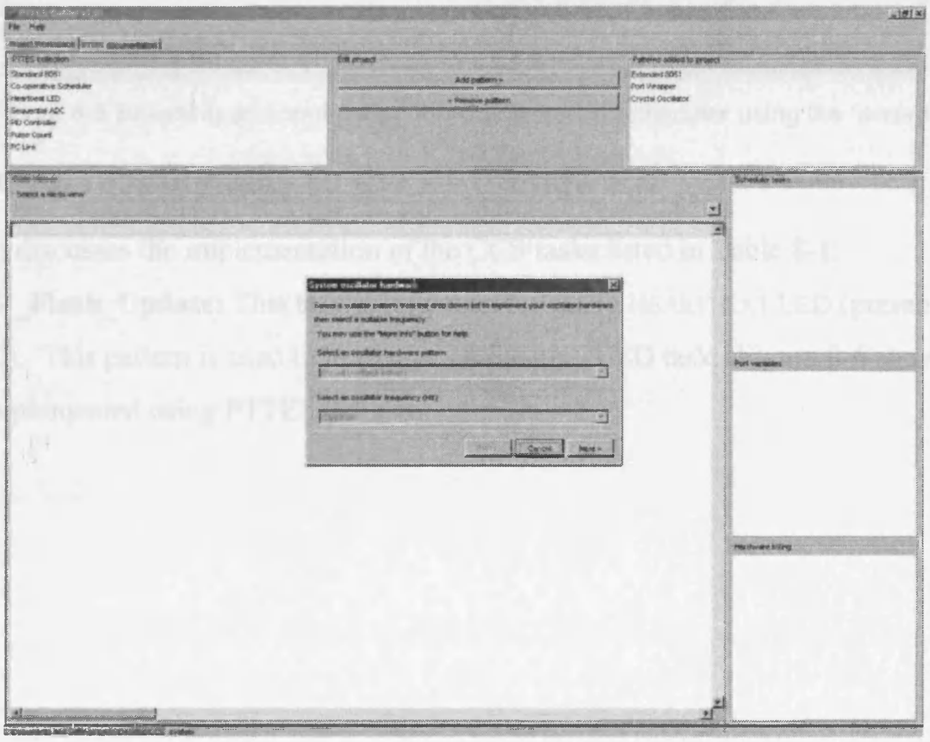
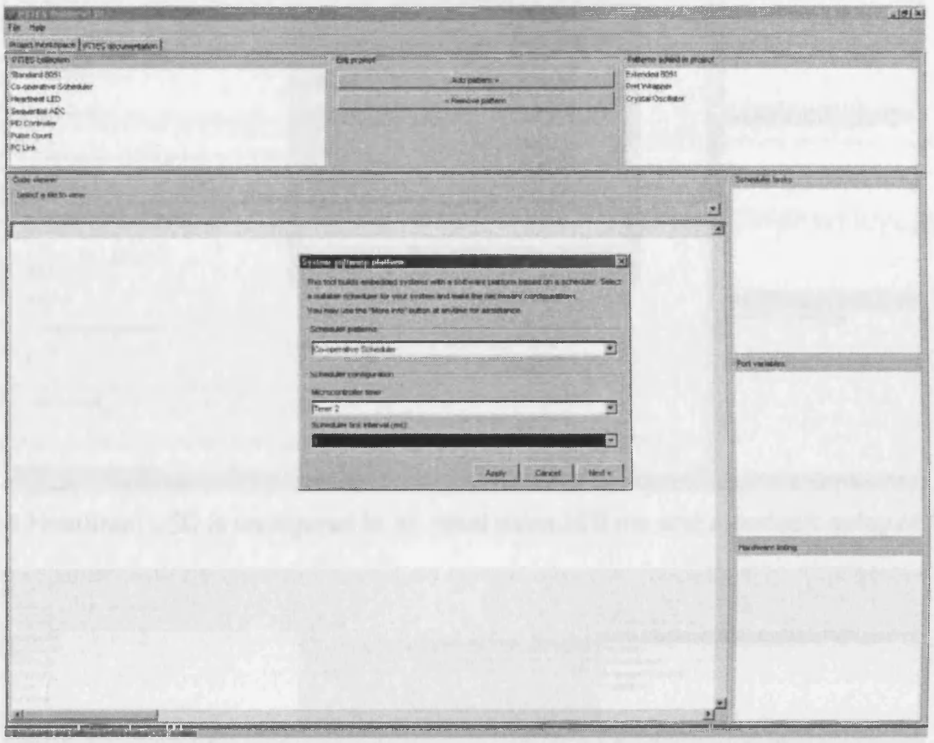


Figure 8-4 Based on the microcontroller selected, an appropriate oscillator frequency is set (10 MHz in this case)

8.2.1.2 Implementing the software architecture

At present, the only scheduler supported by PTTES Builder is TTC SCHEDULER. This was used to implement the software architecture of the CCS. Figure 8-5 shows a TTC SCHEDULER (also known as CO-OPERATIVE SCHEDULER in the PTTES collection) PIE being configured to the specification of the selected microcontroller. A user is presented with a list of Timers available with the selected microcontroller to use as a source of system ticks. Timer 2 was selected with a 1 ms timer overflow.



This tool builds embedded systems with a software platform based on a scheduler. Select a suitable scheduler for your system and make the necessary configurations.
You may use the "More info" button at anytime for assistance.

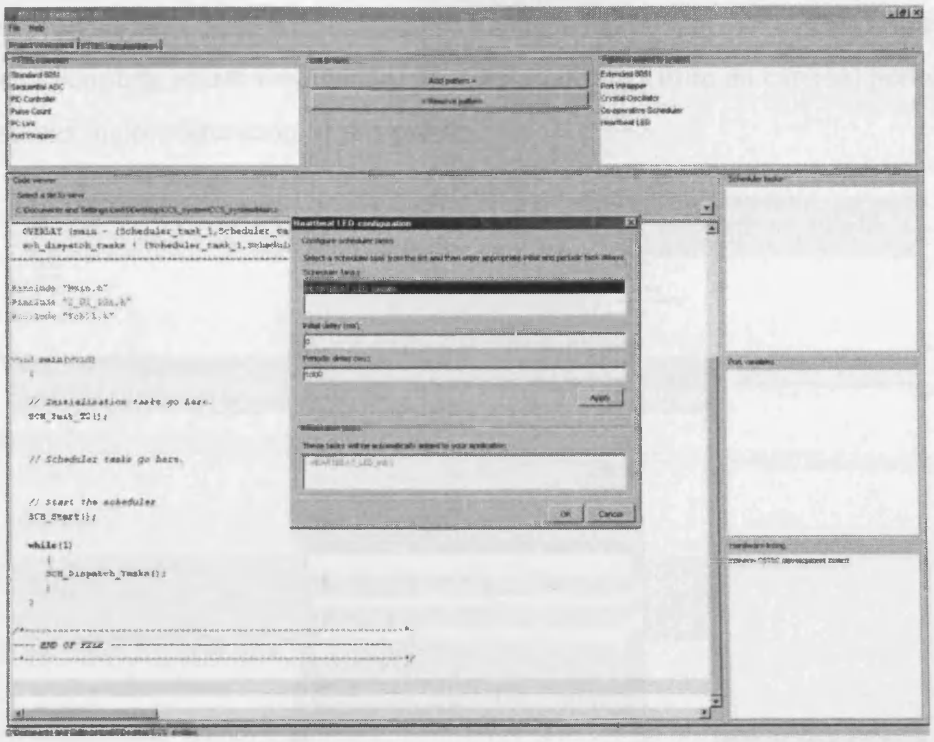
Scheduler patterns: Co-operative Scheduler
Scheduler configuration
Microcontroller timer: Timer 2
Scheduler tick interval (ms): 1

Figure 8-5 Selecting and configuring the Co-operative Scheduler using the 'wizard'

8.2.2 Adding functionality to the TT framework

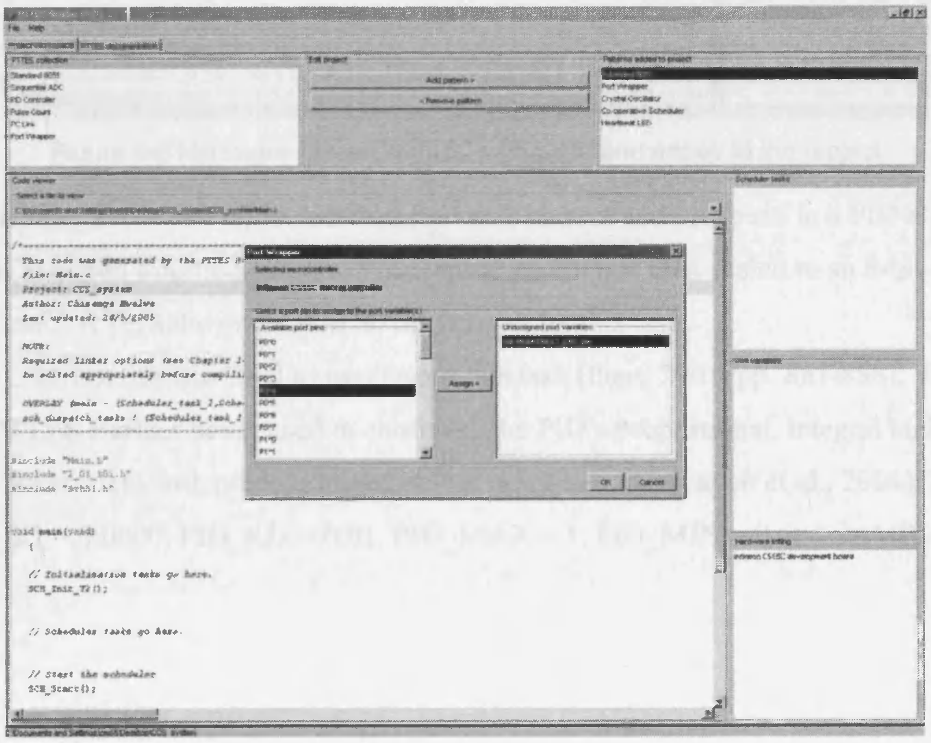
This section discusses the implementation of the CCS tasks listed in Table 8-1.

LED_Flash_Update: This task is implemented using HEARTBEAT LED (presented in Appendix C). This pattern is used to implement a flashing LED task. Figure 8-6 shows how this pattern is implemented using PTTES Builder.



```
OVERLAY (main - (Scheduler_task_1,Scheduler_ta
 sch_dispatch_tasks ! (Scheduler_task_1,Schedul
-----------------------------------------------

#include "Main.h"
#include "2_01_10i.h"
#include "Sch51.h"


void main(void)
   {

   // Initialisation tasks go here.
   SCH_Init_T2();


   // Scheduler tasks go here.


   // Start the scheduler
   SCH_Start();

   while(1)
      {
      SCH_Dispatch_Tasks();
      }
   }

/*-----------------------------------------------------*-
  ---- END OF FILE ------------------------------------
-*-----------------------------------------------------*/
```

Figure 8-6 Heartbeat LED is configured to an initial delay of 0 ms and a periodic delay of 1000 ms

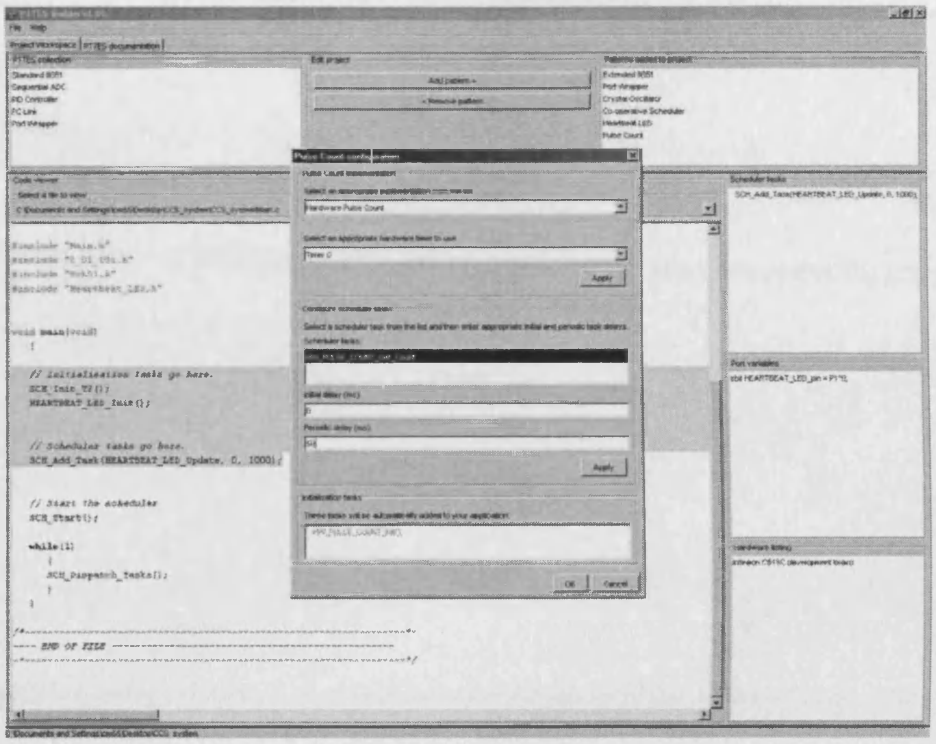


```
/*---------------------------------------------
  This code was generated by the PTTES
  File: Main.c
  Project: CCS_system
  Author: Chisenga Mwelwa
  Last updated: 24/5/2005

  NOTE:
  Required linker options (see Chapter 1-
  be edited appropriately before compili

  OVERLAY (main - (Scheduler_task_1,Sche
  sch_dispatch_tasks ! (Scheduler_task_1
-----------------------------------------------

#include "Main.h"
#include "2_01_10i.h"
#include "Sch51.h"


void main(void)
   {

   // Initialisation tasks go here.
   SCH_Init_T2();


   // Scheduler tasks go here.


   // Start the scheduler
   SCH_Start();
```

Figure 8-7 Port Wrapper is associated with every pattern that interfaces a microcontroller port(s)/pin(s). Here it is used to select a port pin on which to flash the Heartbeat LED task

Sens_Compute_Speed: This task uses an on board timer / counter (Timer 0) to count the number of pulses that have been sent out from the car model. This pulse rate (assumed to arise from an optical or magnetic pulse encoder in a real vehicle) provides an indication of the current speed of the car. The raw pulse count is then scaled to obtain the actual car speed. This task is scheduled every 50 ms.

HARDWARE PULSE COUNT was the pattern used (Pont, 2001, p. 728). This implements a task capable of counting pulses received by the microcontroller from an external peripheral. Figure 8-8 shows the configuration of this pattern.



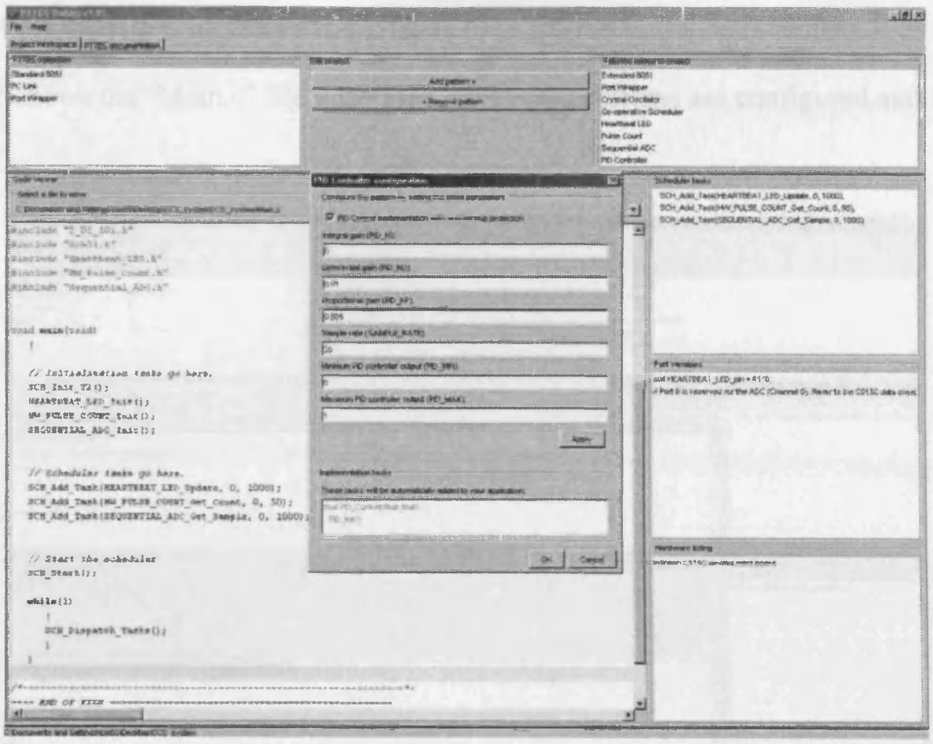
```
#include "Main.h"
#include "2_01_12i.h"
#include "Sch51.h"
#include "Heartbeat_LED.h"

void main(void)
   {

   // Initialisation tasks go here.
   SCH_Init_T2();
   HEARTBEAT_LED_Init();


   // Scheduler tasks go here.
   SCH_Add_Task(HEARTBEAT_LED_Update, 0, 1000);


   // Start the scheduler
   SCH_Start();

   while(1)
      {
      SCH_Dispatch_tasks();
      }
   }

/*------------------------------------------------------------------*-
  ---- END OF FILE ------------------------------------------------
-*------------------------------------------------------------------*/
```

Select an appropriate implementation: Hardware Pulse Count
Select an appropriate hardware timer to use: Timer 0
Select a scheduler task from the list and then enter appropriate initial and periodic task delays.
Scheduler tasks: HW_PULSE_COUNT_Get_Count
Initial delay (ms): 0
Periodic delay (ms): 50
Initialisation tasks: These tasks will be automatically added to your application.
HW_PULSE_COUNT_Init();
Scheduler tasks: SCH_Add_Task(HEARTBEAT_LED_Update, 0, 1000);
Port variables: sbit HEARTBEAT_LED_pin = P1^0;
Hardware listing: Infineon C515C (development board)

Figure 8-8 Hardware Pulse Count is configured and added to the project

Compute_Throttle: This task uses the car's current and set speed in a PID algorithm to calculate the required throttle position. The throttle position is then scaled to an 8-bit value and sent to the 'car'. A periodic interval of 50 ms was set for this task.

PID CONTROLLER was used to implement this task (Pont, 2001, pp. 861-888). Figure 8-9 illustrates PTTES Builder being used to configure the PID's proportional, integral and differential parameters with predetermined values described in (Ayavoo et al., 2004): PID_KP = 0.005, PID_KI = 0.0000, PID_KD = 0.01, PID_MAX = 1, PID_MIN = 0 and SAMPLE_RATE = 20.


Standard 8051
PC Link
Port Wrapper

Add pattern >
< Remove pattern

Extended 8051
Port Wrapper
Crystal Oscillator
Co-operative Scheduler
Heartbeat LED
Pulse Count
Sequential ADC
PID Controller

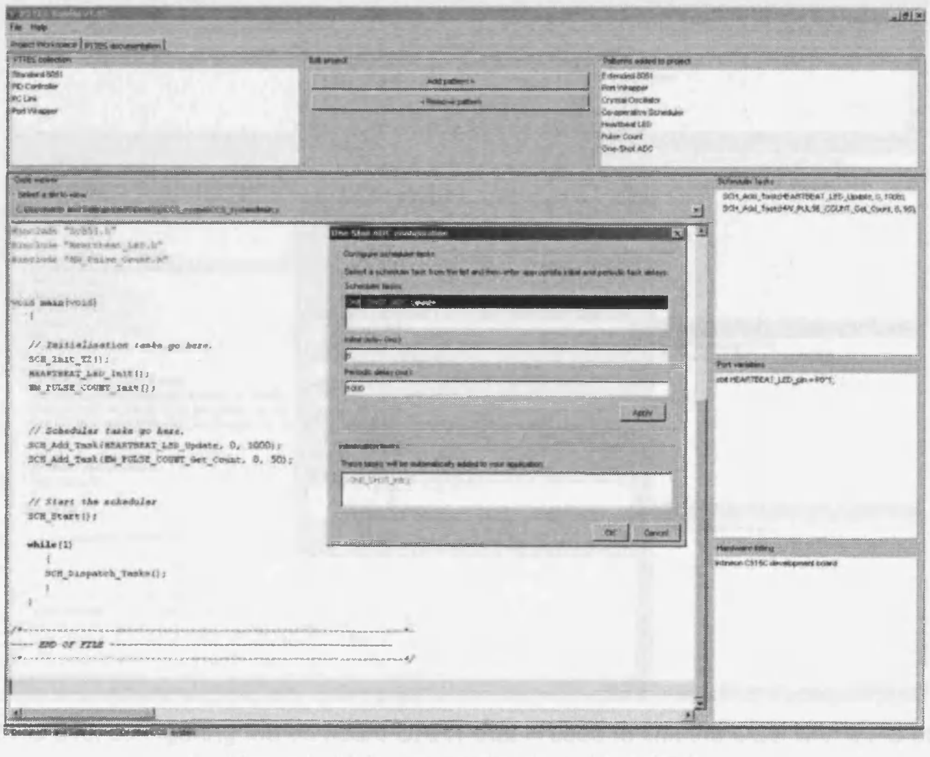
```
#include "2_01_10i.h"
#include "Sch51.h"
#include "Heartbeat_LED.h"
#include "HW_Pulse_Count.h"
#include "Sequential_ADC.h"

void main(void)
   {

   // Initialisation tasks go here.
   SCH_Init_T2();
   HEARTBEAT_LED_Init();
   HW_PULSE_COUNT_Init();
   SEQUENTIAL_ADC_Init();

   // Scheduler tasks go here.
   SCH_Add_Task(HEARTBEAT_LED_Update, 0, 1000);
   SCH_Add_Task(HW_PULSE_COUNT_Get_Count, 0, 50);
   SCH_Add_Task(SEQUENTIAL_ADC_Get_Sample, 0, 1000);

   // Start the scheduler
   SCH_Start();

   while(1)
      {
      SCH_Dispatch_Tasks();
      }
   }

/*------------------------------------------------------------------*-
---- END OF FILE -------------------------------------------------
```

```
SCH_Add_Task(HEARTBEAT_LED_Update, 0, 1000);
SCH_Add_Task(HW_PULSE_COUNT_Get_Count, 0, 50);
SCH_Add_Task(SEQUENTIAL_ADC_Get_Sample, 0, 1000);
```

Integral gain (PID_KI): 0
Differential gain (PID_KD): 0.05
Proportional gain (PID_KP): 0.005
Sample rate (SAMPLE_RATE): 20
Minimum PID controller output (PID_MIN): 0
Maximum PID controller output (PID_MAX): 1

Figure 8-9 Implementation of PID Controller

Act_Get_Ref_Speed: This task uses the C515C's on-chip ADC to read in the voltage from a potentiometer (i.e. Channel 0 on Port 6, Pin 0). This value is then scaled to represent the set speed. This task is scheduled every 1000 ms and was implemented using ONE-SHOT ADC (Pont, 2001, p. 757), see Figure 8-10.


Standard 8051
PID Controller
PC Link
Port Wrapper

Add pattern >
< Remove pattern

Extended 8051
Port Wrapper
Crystal Oscillator
Co-operative Scheduler
Heartbeat LED
Pulse Count
One-Shot ADC

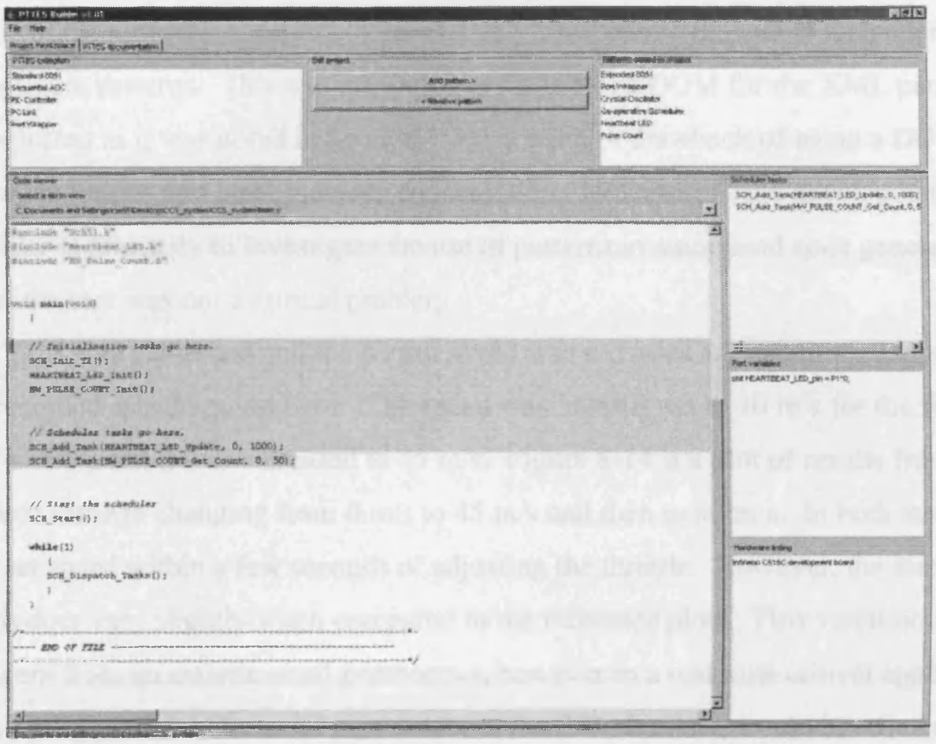
```
#include "Sch51.h"
#include "Heartbeat_Led.h"
#include "HW_Pulse_Count.h"

void main(void)
   {

   // Initialisation tasks go here.
   SCH_Init_T2();
   HEARTBEAT_LED_Init();
   HW_PULSE_COUNT_Init();

   // Scheduler tasks go here.
   SCH_Add_Task(HEARTBEAT_LED_Update, 0, 1000);
   SCH_Add_Task(HW_PULSE_COUNT_Get_Count, 0, 50);

   // Start the scheduler
   SCH_Start();

   while(1)
      {
      SCH_Dispatch_Tasks();
      }
   }

/*------------------------------------------------------------------*-
---- END OF FILE -------------------------------------------------
-*------------------------------------------------------------------*/
```

Initial delay (ms): 0
Periodic delay (ms): 1000

Figure 8-10 Implementing One-Shot ADC

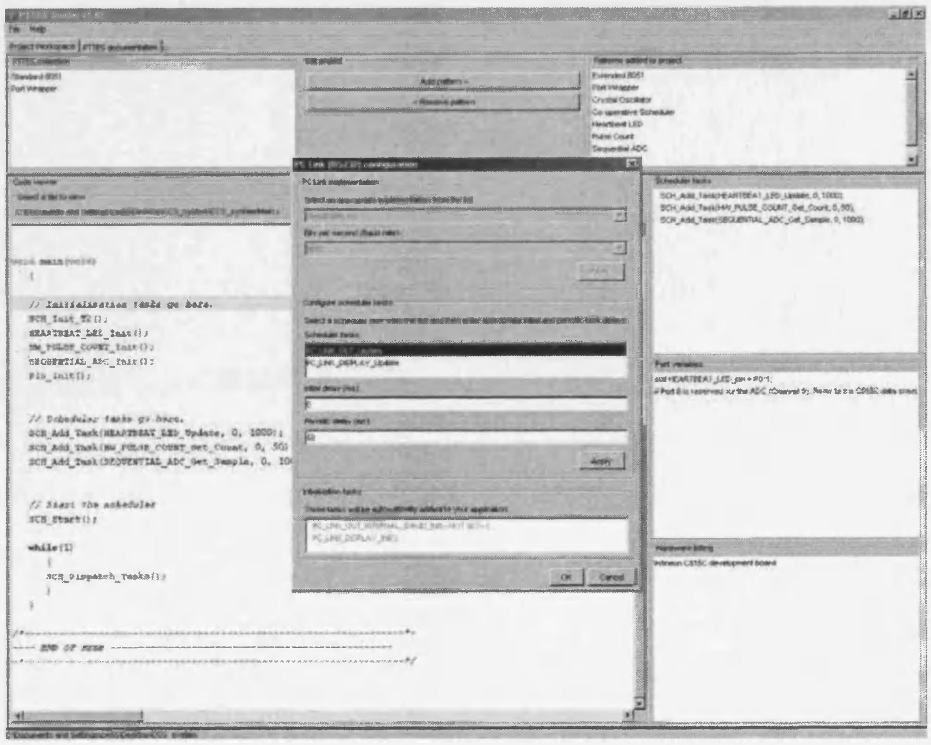
8.2.3 Completing the development

Figure 8-11 shows the 'Main.c' file generated as selected patterns are configured and added to the project.



```
#include "Main.h"
#include "Heartbeat_LED.h"
#include "HW_Pulse_Count.h"

void main(void)
   {
   // Initialisation tasks go here.
   SCH_Init_T2();
   HEARTBEAT_LED_Init();
   HW_PULSE_COUNT_Init();

   // Scheduler tasks go here.
   SCH_Add_Task(HEARTBEAT_LED_Update, 0, 1000);
   SCH_Add_Task(HW_PULSE_COUNT_Get_Count, 0, 50);

   // Start the scheduler
   SCH_Start();

   while(1)
      {
      SCH_Dispatch_Tasks();
      }
   }

/*------------------------------------------------------------------*-
  ---- END OF FILE -------------------------------------------------
-*------------------------------------------------------------------*/
```

Figure 8-11 Main.c file generated as patterns are added to the project



```
void main(void)
   {
   // Initialisation tasks go here.
   SCH_Init_T2();
   HEARTBEAT_LED_Init();
   HW_PULSE_COUNT_Init();
   SEQUENTIAL_ADC_Init();
   Pin_Init();

   // Scheduler tasks go here.
   SCH_Add_Task(HEARTBEAT_LED_Update, 0, 1000);
   SCH_Add_Task(HW_PULSE_COUNT_Get_Count, 0, 50)
   SCH_Add_Task(SEQUENTIAL_ADC_Get_Sample, 0, 10

   // Start the scheduler
   SCH_Start();

   while(1)
      {
      SCH_Dispatch_Tasks();
      }
   }

/*------------------------------------------------------------------*-
  ---- END OF FILE -------------------------------------------------
-*------------------------------------------------------------------*/
```

Figure 8-12 PC Link configuring the on board UART that is used to link the CCS with a PC at 9600 baud

In order to view the set speed, the C515C's UART was linked to a PC running 'HyperTerminal' (see Figure 8-1). PC LINK (RS232) was used to implement the interface between the test-bed and the PC (Pont, 2001, p. 362). Figure 8-12 shows PTTES Builder being used to implement this pattern.

8.3 Observations and results

The response of the tool was observed to be relatively slow during the use of the pattern dialog boxes to configure patterns. This was attributed to the use of a DOM for the XML parsing. This was to be expected as it was noted in Section 7.3 that a major drawback of using a DOM is slow read or write operations and large memory consumption. However, as the development of PTTES Builder is primarily to investigate the use of patterns in automated code generation, the efficiency of the tool was not a critical problem.

The CCS was put to test and the output speed was recorded against time. Figure 8-13 is a plot of the recorded speed against time. The speed was initially set to 30 m/s for the first 25 seconds, at which point it was increased to 45 m/s. Figure 8-14 is a plot of results from a second test with speed settings changing from 0 m/s to 45 m/s and then to 60 m/s. In both tests the CCS reaches the set speed within a few seconds of adjusting the throttle. However, the response time in both cases does vary slightly when compared to the reference plots. This variation does not cause a concern from an experimental perspective, however in a real time critical application the proportional, integral and differential parameters of the PID algorithm could be adjusted in order to refine the response time (Ogata, 2002, p. 701).

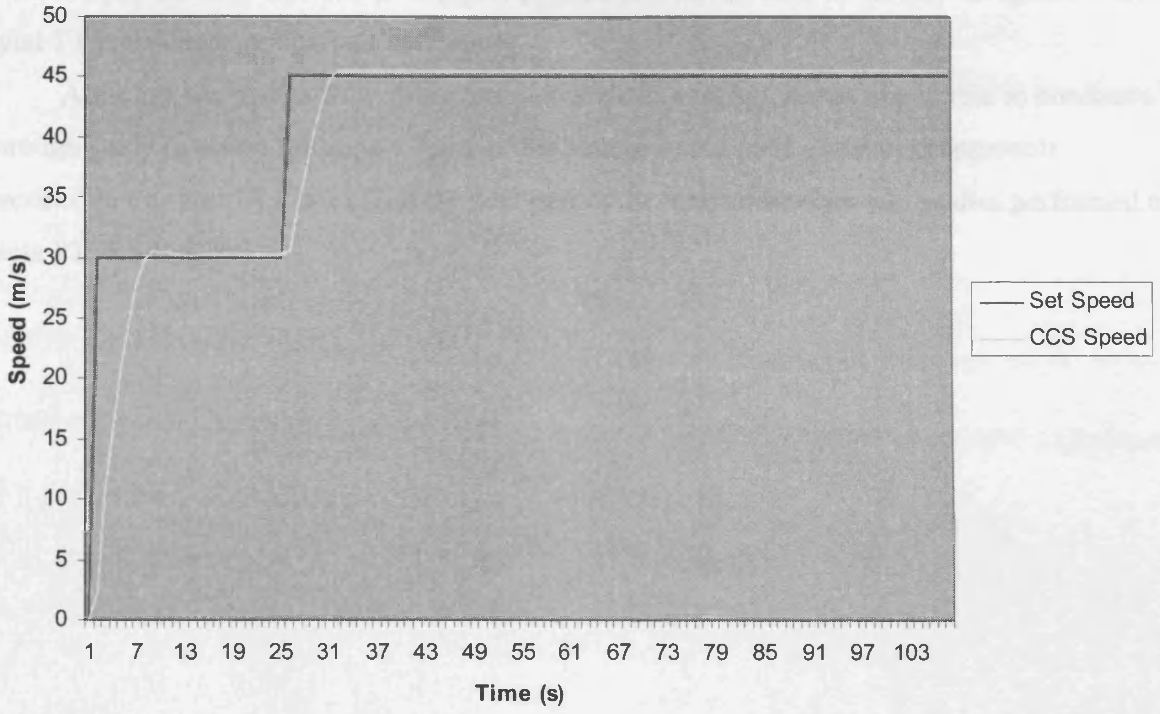


Figure 8-13 The CCS's speed over a period with speed adjustments of 30 and 45 m/s

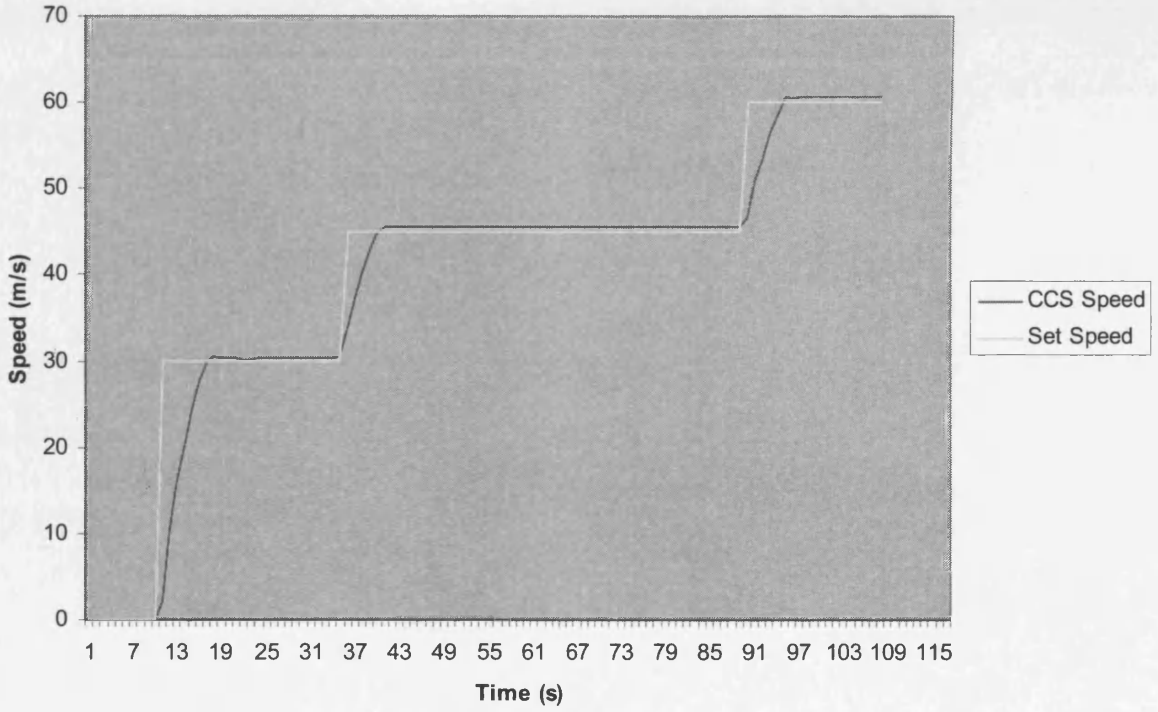


Figure 8-14 CCS speed over time with four changes in speed: 0 m/s, 30 m/s, 45 m/s and 60 m/s

8.4 Chapter conclusions

This chapter has provided an overview of how PTTES Builder can be applied in a non-trivial development exercise. Despite the relatively slow performance of PTTES Builder during usage, the results presented in this chapter suggest that this tool can be used to develop integrated non-trivial TT embedded applications efficiently.

Although the results from this exercise were encouraging, it was imperative to conduct a thorough study to assess the applicability of the pattern-based code generation approach described in this part. To this effect the next part of the thesis describes two studies performed to assess PTTES Builder.

PART IV: ASSESSING THE PATTERN-BASED CODE GENERATION APPROACH

The two chapters in this part of the thesis describe two studies carried out to assess the pattern-based code generation approach described in Chapter 6. In particular, the studies assess 1) the effectiveness of this approach on software quality and reliability 2) the extensibility of the approach with respect to the evolution of a pattern collection.

9 An Assessment of the Approach[8]

This chapter presents a study carried out to assess the extent to which pattern-based code generation, using the approach described in Chapter 6, could support the development of TT embedded systems.

9.1 Overview of the study

This study attempted to assess the effectiveness of PTTES Builder in facilitating the approach described in Chapter 6, to develop a non-trivial TT embedded system using a subset of the PTTES collection. A manual design process using the same set of patterns was used as the control experiment.

To carry out this study, a combination of various empirical techniques was used (Ayavoo et al., 2005; Lethbridge et al., 2005; Basili and Weiss, 1984) and the CCS described in Chapter 8 was used as a test bed.

9.2 A cruise control system test bed

The CCS specified in Chapter 8 was used as a test bed in this study. Similar to Chapter 8, the specification of the CCS was simplified such that the vehicle was assumed to always be in 'cruise' mode. A 'speed dial' was included in the design to allow the car speed to be dynamically changed. The tasks used in the CCS are listed in Table 9-1.

Table 9-1 Tasks implemented in the CCS

Task Names	**Task Description**	**Task Period (ms)**
LED Flash Update	Flashing LED to indicate that the board is working	1000
Compute Car Speed	Computes the car speed obtained from car model	50
Compute Throttle	Calculates and sends the required throttle to be applied to the car model	50
Get Ref Speed	Gets the desired speed from the driver	1000
Display Ref Speed	Displays the desired speed on a terminal emulator	1000

[8] The contents of this chapter have previously been published elsewhere (Pont et al., submitted 2006).

9.3 Assessment methodology

Obtaining suitable subjects for experimental studies is a common problem in empirical software engineering (Kitchenham et al., 2002; Fenton, 1994). With the limited resources available it was decided to use students as subjects. Although the objective of this work was to draw conclusions valid for professional software developers, students were used as subjects as it has previously been demonstrated that the difference in working practices between software engineering students and professionals is minor (Host et al., 2000).

Instead of attempting to use a large group of subjects, a very small group was used. The group of students were selected using the Small Group Methodology (SGM) described in (Ayavoo et al., 2006). This methodology involves the use of small numbers of very well matched student volunteers in empirical studies. In this study four students that had previously taken two modules in embedded systems in the ESL were selected. The students were selected such that their ability and experience in embedded systems was similar. The students all had an average performance in embedded systems modules in the range of 70%-80%. Given the small number of subjects used this ensured that they were well matched in background and ability hence allowing for a reasonable comparison of results.

It was made clear to all the subjects that the CCS was to be implemented using the following eight patterns: EXTENDED 8051, CO-OPERATIVE SCHEDULER, PORT WRAPPER, HEARTBEAT LED, HARDWARE PULSE COUNT, PID CONTROLLER, ONE-SHOT ADC and PC LINK. Each PTTES book handed out to the students was marked to identify these patterns. This ensured that the ability or inability of the students to locate the patterns did not obscure the results.

Students A and B were asked to use PTTES Builder and the PTTES book to implement the CCS whilst Students C and D were asked to only use the PTTES book along with PIEs from the CD accompanying the book.

9.4 Metrics and measurement procedure

The following metrics were used in the study:

- Development time: The amount of time taken for a developer to implement a specified code segment (Solingen and Stalenhoef, 1997; Basili and Weiss, 1984).
- Source code changes: The measure of the number of changes made to a particular segment in the source code. This metric has previously been used as an indicator of system reliability and it has been argued that reliability tends to be at a maximum when code maintenance stabilises (Schneidewind, 1999). It has also been shown that

the more changes made to a module, the more likely it is that bugs will be introduced in the system (Purushothaman and Perry, 2005). In the study described here, the PIEs used are a result of a rigorous development cycle. Therefore, modifying a PIE could potentially introduce bugs into the software resulting in 'less reliable' code.

- Software modularity: This is a measure of the coupling of the various modules in a system with respect to files, functions and variables. For each module observed to be sharing a file, function or variable with another module, a mark of +1 is given to indicate the level of coupling in the system. As the coupling level decreases, the modularity of the system tends to improve (Rosenberg and Hyatt, 1997). It has been argued that this metric can be used to indicate the maintainability and portability of a system (Martin and Shafer, 2006).

To facilitate the use of these metrics, three data collection techniques were identified for the study (Lethbridge et al., 2005):

1. Progress observation: Each team's progress was observed by means of a progress form to visually keep track of the student's activities. Any difficulties observed e.g. misuse of PTTES Builder were also noted on the form.
2. Email: Each team was asked to e-mail their project source code every 30 minutes to one of the observers. The source code was used as the main source of data to measure the metrics described above.
3. Questionnaire and interview: Each subject was given a questionnaire to complete at the end of the experiment and a short recorded interview session was also held for each subject.

9.5 Results

The data obtained from the study, which consisted of source code, completed progress forms and recorded interviews was analysed.

Based on the progress form, it was observed that the students using PTTES Builder generally began developing their source code by using the PTTES patterns as a starting point before writing the relevant tasks for the CCS. By contrast, the students that did not use the tool would work on one pattern at a time, implementing the necessary tasks before moving on to the next pattern.

A synchronisation process described in (Ayavoo et al., 2005) was used to compare the source code submissions of the tool and non-tool users. This involved grouping source files with respect to their associated patterns. The analysis of the source code was used to assess the quality of the code produced. Each source file submitted was analysed and compared with subsequent submissions using Araxis Merge (Araxis, 2005). It was observed that Subjects A and

B who used the tool initially spent approximately sixty minutes using it. They spent fifteen minutes of this time going over a PTTES Builder tutorial. For these subjects, 75% of the software for their final CCSs was generated by the tool.

Table 9-2 summarises the results of the experiment. The coupling level was lower for the subjects who used the tool. However, on closer inspection it was revealed that the two tool users (Students A and B) produced some discrepant results. Subject B's effort and code changes were not as expected when compared to Subject A. Upon further investigation (through the interviews, questionnaire and code analysis), it became apparent that Student B had little confidence in the tool. As a consequence, the subject had attempted to make major changes to the generated scheduler code and its PIE tasks in the Keil IDE. This led to the introduction of bugs and as a result the subject was held back as he attempted to resolve them. This is consistent with the suggestion in Section 9.4 that the more changes made to a PIE the more likely it is that bugs will be introduced in the system. This suggests that such tool support will only be more effective if the users have confidence in the tool. This is more likely to happen with a commercial product than a prototype such as that described here. Nevertheless, like Student A, Student B still produced a complete CCS with better code modularity in comparison to the non-tool users.

Table 9-2 Results of data analysis based on the metrics defined in Section 9.4

Metrics	Tool (A)	Tool (B)	No Tool (C)	No Tool (D)
Total effort for the entire project (in minutes)	150	420	240	510
Total effort to implement the patterns (in minutes)	71	228	145	357
Total changes made to the pattern (in LOC)	11	55	33	127
Total coupling level for all patterns implemented	5	4	11	9

On analysis of the effort involved in implementing individual patterns it was observed that, with the exception of Student B for reasons already discussed above, the level of effort varied considerably among patterns. Figure 9-1 illustrates the effort of the subjects in implementing the individual patterns with the exception of EXTENDED 8051 which is a hardware pattern (Pont, 2001, pp. 29-46). Using McCabe's Cyclomatic Complexity measures (McCabe, 1976), it was observed that PTTES Builder contributed to the reduction in effort in the implementation of CO-OPERATIVE SCHEDULER. However, for less complex patterns (such as PORT WRAPPER and HEARTBEAT LED), the contribution of the tool was not as significant. This may suggest that PTTES Builder is most effective when implementing patterns with a high complexity.

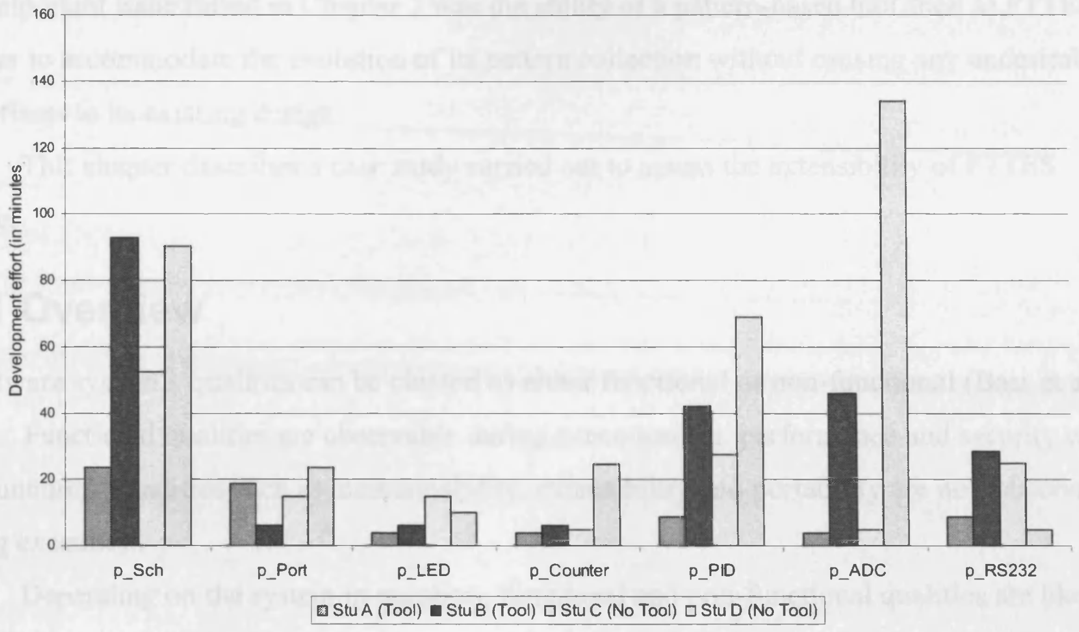


Figure 9-1 Overview of the effort involved in implementing individual patterns

9.6 Chapter conclusions

The results obtained from this study suggest that the use of PTTES Builder can indeed generate integrated TT embedded applications. Furthermore, this form of software development is likely to lead to improved code quality and reliability. The results have also indicated that the contribution of the tool is most significant when implementing patterns with a high level of complexity.

The results of the study presented in this chapter are based on a single study. It would therefore be premature to claim that they provide conclusive evidence. Therefore, a replication of this study is worth pursing in the future with a different set of patterns and subjects.

10 Evaluating the Extensibility of the Approach

One important issue raised in Chapter 7 was the ability of a pattern-based tool such as PTTES Builder to accommodate the evolution of its pattern collection without causing any undesirable side effects to its existing design.

This chapter describes a case study carried out to assess the extensibility of PTTES Builder.

10.1 Overview

A software system's qualities can be classed as either functional or non-functional (Bass et al., 1998). Functional qualities are observable during execution e.g. performance and security whilst non-functional qualities such as maintainability, extensibility and portability are non-observable during execution.

Depending on the system in question, functional and non-functional qualities are likely to either be important or irrelevant. In the case of PTTES Builder, besides its functionality which is primarily the generation of code from patterns, extensibility is an important non-functional requirement. In Chapter 7 it was noted that in order for PTTES Builder to be a useful pattern-based tool, it needed to be extensible i.e. allow for the addition of new patterns and PIEs without affecting the tools existing functionalities.

Maintainability is another non-functional quality that refers to the ability of a system to be modified i.e. the ease with which a software system can be modified to correct faults, improve performance or other attributes (IEEE Std. 610.12-1990, 1993). Although similar in a way to maintainability as both consider future change to a system, extensibility focuses on the ability of a system's design to evolve.

10.2 Case study setup

The case study was performed over a period of 10 weeks during which two developers were employed and assigned the task of adding new patterns and PIEs to the tool. Prior to the study, PTTES Builder supported eleven patterns from the PTTES collection. The microcontrollers supported were the Atmel AT89S53 (STANDARD 8051) and the Infineon C515C (EXTENDED 8051). As is discussed in Chapter 4, the original PTTES collection describes 72 patterns (Pont, 2001), meaning that at the start of this study PTTES Builder only supported 15% of the patterns from the PTTES collection.

During the 10 weeks of the study, the developers were closely observed and data mainly consisting of revision source code was collated and analysed in order to draw conclusions about the tool's extensibility. The following subsections give an overview of the subjects employed in the study and the methodology used to assess the extensibility of the tool.

10.2.1 The subjects

For the same reasons described in Section 9.3, students were used as subjects in this study. The main skills required by the subjects were Java and C programming. The role of the two developers was advertised among penultimate year Engineering and Computer Science students within the University of Leicester. The best five applicants were thereafter interviewed by a panel of four: Dr Michael J. Pont (academic supervisor of this research), Dr David Ward (industrial supervisor of this research), Susan Kurian (PhD candidate in the ESL) and the author. The interview was primarily technical with the aim of establishing the candidates' technical abilities.

The two candidates selected, referred to as KA and DM in this thesis, were highly recommended by their academic referees and this was underlined by their good academic ranking among their peers. During the first four weeks of the study, the two subjects went through a knowledge transfer period which involved familiarisation with PTTES Builder including its capabilities, architecture and source code. This was facilitated by the author.

10.2.2 Metrics and data collection techniques employed

Generally, metrics used in software studies are based on time-oriented data referring to actions performed by developers during development. Because of the qualitative nature of such data it can be challenging to capture and collate; to this effect Fenton (1994) has listed three classes of entities in software measurement that were used in the study, 1) processes: any software-related activities that take place over time 2) products: any artefacts, deliverables, or documents that result from processes and 3) resources: items required by processes. In this study the processes observed were the development activities of the two developers, the source code artefacts were the deliverables produced by the developers and the main source of data used in the analysis. The two subjects were the main resources monitored.

The productivity of the developers was the main indicator of the ease with which new patterns and PIEs could be added to PTTES Builder. In software studies, productivity is generally defined as size divided by effort and traditional software metrics have used the volume of source code produced as a means of measuring size whilst time is usually used to measure effort (e.g., Kitchenham et al., 2002; Basili et al., 1996). However, in this study the number of patterns added to PTTES Builder was instead used as a measure of size. This was mainly due to the fact that the PIEs were already available and therefore the developers were not required to write them.

Each developer was given a set of patterns to implement at the beginning of each week.

At the end of each working day, regardless of whether changes had been made to PTTES Builder, each developer submitted their source code to a secure depot. With each source code submission, the developers made a note of the changes made to the code and any problems experienced. At the end of each work week each developer's working version of PTTES Builder was integrated into a release and the author had an informal interview with each developer to discuss their progress.

10.3 Results

During the study an additional 27 patterns from the PTTES collection were included in PTTES Builder (Athaide et al., 2005). The patterns were added by following the design described in Chapter 7 i.e. adding a pattern and its PIEs to the repository and thereafter implementing its UI dialog boxes in the tool.

To test the new functionality of PTTES Builder, the final release was put through two tests. The first test acted as a means of regression testing as it aimed at establishing whether the existing functionality prior to the new changes was still functioning correctly; it involved the use of PTTES Builder to develop a number of cruise control applications similar to the one described in Chapter 8. In addition to replicating the cruise control application in Chapter 8, multi-processor variants were also developed as a means of testing the application of the new patterns added to PTTES Builder e.g. SCU SCHEDULER (LOCAL) and SCC SCHEDULER implemented UART and CAN based multi-processor cruise control systems respectively (Athaide et al., 2005; Pont, 2001).

The second test involved the development of a speech playback application that replicates a recording of a human voice using stored data (Athaide et al., 2005). This was tested on a range of different hardware implementations added to PTTES Builder (Athaide et al., 2005).

Despite a few minor bugs realised (and corrected on the spot) PTTES Builder applied both the old and new patterns effectively during the studies. This was an indication that the new patterns had been successfully added to the tool. Having established that the functionality of PTTES Builder had not been affected by the addition of new patterns, the next analysis involved investigating the ease at which the developers were able to add the patterns to PTTES Builder.

Using the data gathered during the study and the metrics described in Section 10.2.2, the graph in Figure 10-1 was produced (Table 10-1 is a reference table for the patterns). The graph shows the effort involved in adding each pattern to the tool, time (in days) was used as a measure of effort. It must be noted that although SC CAN was one of the patterns added to PTTES Builder during the study, it was not included in the analysis. This was due to the fact that the PTTES collection at the time only had one PIE for SC CAN that supported the Infineon C515C microcontroller. As a consequence, additional effort was put into writing additional PIEs for other platforms during the course of the study. In total it took about 4 weeks for these additional PIEs to be developed and added to the tool. It was also realised that SC CAN was the most complex pattern (in terms of LOC). To therefore ensure a fair comparison of the effort required to add each pattern to the tool, SC CAN was excluded from the analysis.

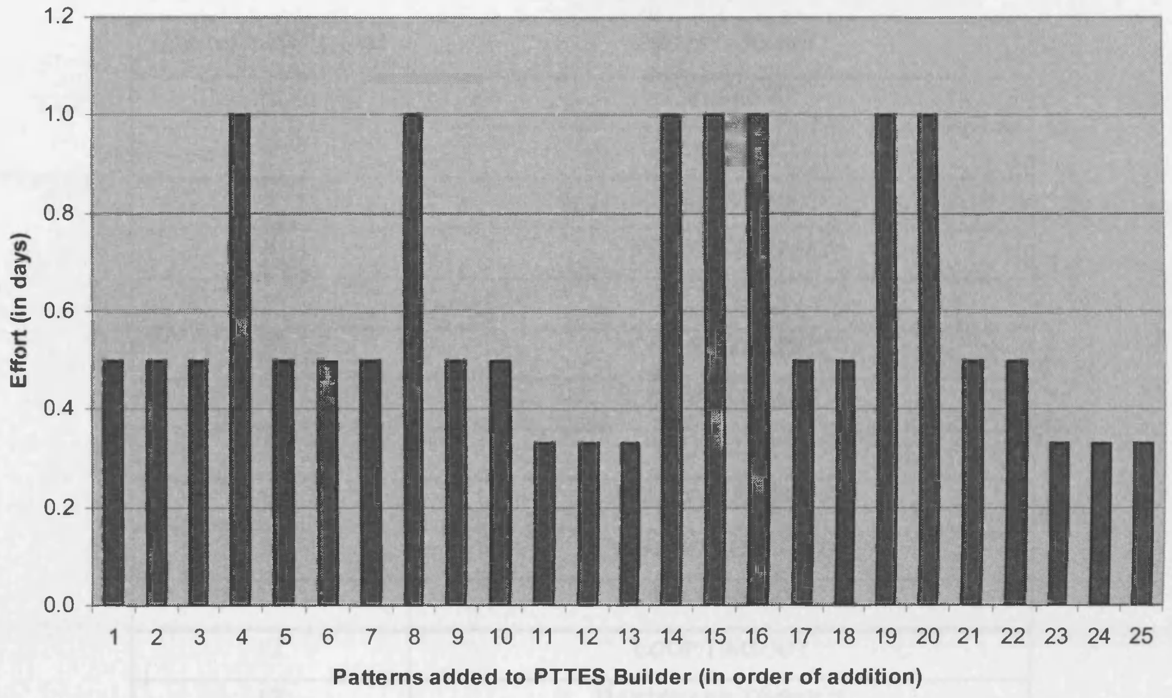


Figure 10-1 Effort involved in adding new patterns to PTTES Builder. The patterns are listed in Table 10-1

Upon an analysis of Figure 10-1, it is realised that the maximum effort involved in adding a pattern was one day and the minimum effort approximately one third of a day. To conduct a detailed analysis of these results each pattern's level of complexity was also taken into account. This was derived using each software pattern's PIE for the AT89S53 microcontroller to work out its source lines of code (SLOC). Though there has not been a consensus on how to define software complexity to date, SLOC remains a common means of estimating software complexity (Zuse, 1991).

SWITCH INTERFACE, SC UART, HYBRID SCHEDULER, 255-TICK SCHEDULER, ON-TASK SCHEDULER, STABLE SCHEDULER and SC INTERRUPT required the most effort to add to the tool out of the 25 patterns analysed. But upon considering the level of complexity of these patterns (see Table 10-2); SC UART, HYBRID SCHEDULER, 255-TICK SCHEDULER, ON-TASK SCHEDULER, STABLE SCHEDULER and SC INTERRUPT were the only patterns to have a high level of complexity. However, looking at Table 10-2 LCD CHARACTER PANEL would have been expected to have also required a considerable amount of effort as it ranked between SC UART and STABLE SCHEDULER in the complexity table. But upon further analysis this discrepancy was attributed to the fact that LCD CHARACTER PANEL has a high SLOC value (or high complexity level) as a result of its code library used for the character mapping. The same was attributed to I2C. Despite having a relatively low complexity level SWITCH INTERFACE required more effort than may have initially been expected because it consists of a hardware and software implementation which therefore resulted in more work.

Table 10-1 The patterns added to PTTES Builder (in the order in which they were added)

Pattern Number	Pattern Name
1	RESET
2	LCD CHARACTER PANEL
3	KEYPAD
4	SWITCH INTERFACE
5	SPI
6	ONE SHOT ADC
7	SEQUENTIAL ADC
8	SC UART
9	SOFTWARE DELAY
10	CERAMIC RESONATOR
11	WATCHDOG
12	LOOP TIMEOUT
13	HARDWARE TIMEOUT
14	HYBRID SCHEDULER
15	255-TICK SCHEDULER
16	ONE-TASK SCHEDULER
17	I2C
18	MX LED
19	STABLE SCHEDULER
20	SC INTERRUPT
21	SW PRM
22	SW PWM
23	HARDWARE PRM
24	HARDWARE PWM
25	3-LEVEL PWM

The rest of the patterns required an effort ranging from a third to half a days work. The effort required for these patterns is justifiable when their complexity level is considered relatively to the more complex patterns.

Table 10-2 List of patterns in an ascending order of complexity where SLOC is used to measure the complexity

Pattern Name	Complexity (in SLOC)
Ceramic Oscillator	0
Reset	0
Software Delay	10
Watchdog	24
Hardware PRM	35
Loop Timeout	40
Sequential ADC	44
One Shot ADC	50
Keypad	60
SPI	60
SW PRM	69
MX LED	73
SW PWM	73
Hardware PWM	74
Switch Interface	80
255-Tick Scheduler	94
One Task Scheduler	98
3-Level PWM	120
Hybrid Scheduler	180
I2C	204
Hardware Timeout	260
SC UART	275
LCD	295
Stable Scheduler	363
SC Interrupt	713
SC CAN	1243

10.4 Chapter conclusions

From the results of the study it was concluded that the same amount of effort is required to add patterns of the same complexity level to PTTES Builder. Furthermore, the more complex a pattern the more effort is required to add it to the tool. It can also be concluded that, based on the average time the users took to add a new pattern to the tool, the exercise was manageable once those responsible had grasped an understanding of the tool's design.

From the interviews and the daily notes written by the developers it was also observed that when adding a pattern the majority of the work involved the development of the pattern dialog window. Although each pattern was more than likely to have a unique dialog window for its configuration, it was strongly felt that this was something that could be improved upon in the tool design.

Although this study did not have a set of results to use as a benchmark, the results of the study gave a good indication that PTTES Builder is indeed extensible. The study also provided a means of realising areas of the tool that could be improved upon such as the implementation of pattern dialog boxes in the UI.

PART V: TO CONCLUDE

This part concludes the thesis.

11 Thesis Conclusions

This chapter summarises the work presented in this thesis and concludes by discussing the contributions made. The limitations and future areas of work are then discussed.

11.1 Summary

This thesis has addressed a range of issues concerning the development of reliable time-triggered embedded systems, design patterns and pattern-based code generation.

It has been argued that static schedulers such as the TTC scheduler provide a more deterministic and predictable architecture for embedded systems (if developed correctly) in comparison to dynamic schedulers. It may therefore be appropriate that during design a developer first establishes whether a TTC scheduler meets the design requirements. By so doing, the design of an application's software architecture is based on high reliability requirements. However, implementing TTC architectures comes with challenges such as the need to know task durations and any potential task overruns at design time in order to guarantee reliability.

An account of design patterns and their benefits to software development has been given. It has been argued that patterns facilitate the reuse of best practice design solutions. A collection of patterns referred to as the PTTES collection was presented and it was argued that this is a pattern language capable of supporting the development of applications with TTC architectures. However, it was observed that the 'manual' implementation of the PTTES collection and other software patterns in general does have a number of disadvantages. In particular, the transformation of patterns from design to implementation has no structured process and it typically involves users adapting example code to their design specifications. This is an ad hoc mundane, tedious and as a result error prone process that is not suitable for the development of software for systems with high reliability requirements.

The importance of code generation and CASE tools in modern day software development has also been discussed. The thesis has gone on to argue that pattern-based code generation can offer an alternative method to model-based code generation. In fact, it has been argued that pattern-based code generation has the potential to produce code of high quality and efficiently. Previous work on pattern-based CASE tools has also been reviewed and it has been argued that these tools have merely supported the application of individual patterns and not complete application code generation from appropriate groups of patterns. Furthermore, the majority of these tools have focused on the Gamma collection.

The challenges of generating application code from groups of patterns have been highlighted and addressed in this thesis. In particular, it has been argued that PIEs can be used to uphold the 'one pattern, many implementations' characteristic of patterns in a pattern-based CASE tool. It has also been argued that in order for a CASE tool to support the generation of coherent code from a set of patterns, the patterns must be members of a pattern language. PTTES Builder, a prototype CASE tool based on the PTTES collection, has been implemented in order to test these hypotheses in the context of TT embedded systems.

Two studies were conducted to assess the viability of the approach described above. The first study was aimed at establishing the effect of this approach on user effort, code quality and reliability. The study concluded that PTTES Builder contributed to the development process by reducing the effort of the developers and in particular when implementing patterns with a high level of complexity. There were also some initial indications that the use of PTTES Builder is likely to lead to improved code quality. The objective of the second study was to investigate the extensibility of the approach i.e. whether it could cope with the evolution of the PTTES collection. The study concluded that the approach is extensible and therefore can support the evolution of a pattern collection.

11.2 An analysis of the contributions

This thesis makes a novel contribution to pattern-based code generation and illustrates the capabilities of this approach to the development of reliable time-triggered embedded systems. The individual contributions are now examined in more detail, highlighting the benefits.

PTTES Builder is a tool that supports pattern-based code generation for embedded systems with a time-triggered architecture. The approach used to implement PTTES Builder is the primary contribution of the work presented in this thesis. It addresses the 'one pattern, many implementations' relationship of patterns that previous pattern-based code generation CASE tools have not addressed. Furthermore, the approach is based on a domain-specific pattern language thus facilitating the generation of integrated applications from appropriately selected patterns. This is a significant extension of previous work in this area that primarily focused on the transformation of individual patterns to code.

The primary goal of implementing PTTES Builder was to use it as a means of demonstrating the feasibility and consequences of adopting the pattern-based code generation approach described in Chapter 6. PTTES Builder is therefore a proof of concept that illustrates the benefits of this approach in the development of embedded systems with a time-triggered architecture using a pattern language in this domain. PTTES Builder clearly achieves its objectives and demonstrates how pattern-based code generation, using the approach described, can be used to generate robust code efficiently.

Through the use of a pattern language: patterns for time-triggered embedded systems (referred to in this text as the PTTES collection), PTTES Builder addresses some of the challenges that come with developing reliable embedded applications. In particular, PTTES Builder incorporates a wizard to assist a user in the creation of an appropriate hardware and software architecture. Using this wizard and a division of the pattern library into 'patterns' and 'pattern implementation examples', the tool is able to present a user with only those design options which are applicable to the design context. Furthermore, the wizard enforces certain mechanisms to ensure that important design decisions are followed. For instance, the tool keeps an inventory of certain hardware resources for the selected microcontroller (e.g. port pins) as they are used by other patterns in the design. This helps to avoid basic design errors (where, for example, two tasks have assumed exclusive access to a UART or particular port pin). This is a useful feature when working with multiple patterns in such a system design. Also, the scheduler, derived from the software architecture, facilitates the integration of the rest of the system's patterns. The TT schedulers described in the PTTES collection have a natural way of integrating tasks through the use of the 'Add Task' function.

In an empirical study described in Chapter 9, the effectiveness of the PTTES Builder approach was compared with an equivalent 'manual' approach. The results obtained demonstrate that time-triggered embedded systems can be created using this approach. There is also some evidence that the use of the tool is likely to lead to improved code reliability and quality. It was also observed that the contribution of the tool was most significant when implementing patterns with a high level of complexity. In another study described in Chapter 10 it has been demonstrated that the approach implemented by PTTES Builder can sustain the evolution of its underlying pattern collection. These results are encouraging and have demonstrated that the approach implemented in PTTES Builder is a novel and practical approach to pattern-based software development.

11.3 Limitations and future work

In this section a number of limitations of the PTTES Builder approach are discussed some of which may be addressed by future work in this area. Other extensions to this work are also highlighted.

The pattern-based code generation approach described in this thesis and implemented by PTTES Builder has been implemented using the PTTES collection. This is a limitation and therefore future research should attempt to investigate the applicability of this approach in other software domains in order to assess its applicability. Over the years the patterns community has seen the creation of more domain specific pattern collections such as the Core J2EE Patterns that provide a range of solutions for Java enterprise applications (Alur et al., 2001). Coupled with this, is the availability of application frameworks like Apache Struts, a free open-source framework for creating Java web applications (The Apache Software Foundation, 2006). The

Struts framework is designed to help developers create web applications that have an architecture based on MODEL VIEW CONTROL (MVC) described in the Gamma collection (Gamma et al., 1995). Struts provide the glue that links the various elements of a Java enterprise application into a coherent whole. Future work can therefore investigate the applicability of the pattern-based code generation approach, described in this thesis, in the J2EE context where the Struts architecture can be utilised in the same way as TTC SCHEDULER is applied in PTTES Builder. This exercise can be the basis of further empirical studies to establish the extent to which the code generation approach described in this work can be applied.

Section 5.2 discussed the current use of UML as a basis for model-based code generation. As the SysML (discussed in Section 5.2) becomes widely accepted as a UML extension for embedded systems design, future versions of the tool should explore the possibility of having a graphical modeling user interface based on SysML. This work would involve exploring the representation of the PTTES collection in SysML.

The version of PTTES Builder described in this thesis is based on the waterfall model, beginning with the selection of hardware and software architectures (as outlined in Section 8.2.1). If the resources are exhausted, a user needs to repeat the design process (in a new project) with a different processor. This is a limitation that future work can consider, a user could be given the option of completing the software design and then – on the basis of the resource requirements – selecting a suitable hardware platform.

Kurian and Pont (2006) have been exploring techniques which can be used to support the maintenance of embedded systems developed using the PTTES collection. The focus has been on techniques that will allow users to exchange patterns in such a project – with minimal or no human intervention after a project has gone into production. In doing so the aim is to identify the implementation of the pattern which a user wishes to change in the system code. Having done so, a user can remove the relevant code and then substitute a suitable implementation of the replacement pattern. Early work in this area is described in (Kurian and Pont, in press a; Kurian and Pont, 2006) and the techniques described will be incorporated in a future version of PTTES Builder.

The automation of the code generation process also presents the opportunity to enforce more general (low level) coding standards. To illustrate this, ongoing work on PTTES Builder in the ESL has incorporated support for the MISRA C coding guidelines (MISRA, 2004). These guidelines were originally written to help improve the quality of code written for automotive applications; however, they are now much more widely employed in other safety-related and safety-critical systems (Paternotte, 2002). To support these guidelines, a MISRA C module has been implemented that pre-processes and parses PTTES Builder's generated code into an Abstract Syntax Tree (AST) (Mwelwa et al., 2006). The AST is then traversed, checking for

MISRA C compliance. To aid in the compliance checking, symbol tables are created for each scope during the AST traversal. It must be noted that while some rules are checked during the pre-processing stage, the majority of rules can only be checked once a complete AST has been generated. For example, Rule 5.5 requires that – for example – the same variable name should not be re-used in different modules (MISRA, 2004, p.27): clearly, it is impossible to determine if this rule has been broken until all of the system source code is checked.

As was discussed in Chapter 2, task jitter is a concern in many embedded systems. Time-triggered Hybrid (TTH) software architectures are a good choice in a wide range of systems (e.g. data acquisition and control designs) in which jitter is an important consideration. Ongoing work in the ESL has previously described two techniques ('planned pre-emption' and 'delayed resource locking') which can reduce this jitter level (Maaita and Pont, submitted 2005). These techniques can be incorporated in future versions of the tool. Jitter can also occur due to variations in message transmission times in distributed TT designs, in this case techniques such as 'software bit stuffing' can be incorporated in the tool to reduce jitter levels (Nahas et al., 2005).

The focus of the work presented in this thesis has been on the application of software design patterns for the development of TT embedded applications. However, the potential benefits of using patterns in hardware design have recently been explored by a number of researchers (Rincon et al., 2005; Damasevieius et al., 2003; Yoshida, 2001). Future versions of the tool can consider the support for hardware design patterns such as the design of a system's hardware in relation to its software architecture.

Bibliography

Aaen, I., Siltanen, A., Srensen, C. and Tahvanainen, V.-P., *"A tale of two countries: CASE experiences and expectations,"* The Impact of Computer Supported Technologies on Information Systems Development, IFIP Transactions, North-Holland, In: Kendall, K.E., Lyytinen, K. and DeGross, J.I. (Eds.), pp. 61-93, 1992.

Albert, A., "Comparison of event-triggered and time-triggered concepts with regard to distributed control systems," *Proceedings of Embedded World,* Nurnberg, Germany, pp. 235-252, 2004.

Albin-Amiot, H. and Guéhéneuc, Y.-G., "Meta-modeling design patterns : application to pattern detection and code synthesis," *ECOOP'01 : Workshop on Automating Object-Oriented Software Development Methods,* Eötvös Loránd University, Budapest, Hungary, 2001.

Alexander, C., *"Notes on the Synthesis of Form,"* Harvard University Press, 1964. [ISBN: 0-674-62751-2].

Alexander, C., *"The Timeless Way of Building,"* Oxford University Press, 1979. [ISBN: 0-19-502402-8].

Alexander, C., "The origins of pattern theory the future of the theory and the generation of a living world," *Keynote Address: Object-Oriented Programs, Systems, Languages and Applications (OOPSLA '96),* San Jose, California, USA, 1996.

Alexander, C., "Christopher Alexander," http://www.patternlanguage.com/leveltwo/ca.htm, Accessed in 2005.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fisksdahl-King, I. and Angel, S., *"A Pattern Language,"* Oxford University Press, 1977. [ISBN: 0-19-501919-9].

Alexander, C., Silverstein, M., Angel, S., Ishikawa, S. and Abrams, D., *"The Oregon Experiment,"* Oxford University Press, 1975. [ISBN: 0-19-501824-9].

Allen, F.E., "The history of language processor technology in IBM," *IBM Journal of Research and Development,* Vol. 25, (5), pp. 535-548, 1981.

Allworth, S.T., *"An Introduction to Real-Time Software Design,"* Macmillan, 1981.

Alur, D., Crupi, J. and Malks, D., *"Core J2EE Patterns: Best Practices and Design Strategies,"* Prentice Hall / Sun Microsystems Press, 2001. [ISBN: 0130648841].

Antonakos, J.L., *"The 68000 Microprocessor: Hardware and Software Principles and Applications,"* Macmillan Publishing Company, 1993.

Apneseth, C., "Embedded system technology in ABB," *ABB Review,* 2/2006.

Araxis, "Araxis Merge," http://www.araxis.com/merge/index.html, Accessed in 2005.

Arnout, K., "From Patterns to Components," Doctoral Thesis, Swiss Institute of Technology, Zurich, 2004.

Athaide, K., Mearns, D. and Mwelwa, C., "PTTES Builder," *University of Leicester, Leicester,* Reference No. KA/DM - 2005/3, 2005.

Auer, A., Kemppainen, P., Okkonen, A. and Seppanen, V., "Automated code generation of embedded real-time systems," *Microprocessing and Microprogramming,* Vol. 24, pp. 51-56, 1988.

Ayala, K.J., *"The 8051 Microcontroller: Architecture, Programming and Applications,"* West Publishing Company, 1991. [ISBN: 0-314-77278-2].

Ayavoo, D., Pont, M.J., Fang, J., Short, M. and Parker, S., "A 'Hardware-in-the Loop' testbed representing the operation of a cruise-control system in a passenger car," *Proceedings of the 2nd UK Embedded Forum, 20th October,* Birmingham, UK, 2005.

Ayavoo, D., Pont, M.J. and Parker, S., *"Using simulation to support the design of distributed embedded control systems: A case study,"* Proceedings of the 1st UK Embedded Forum, In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.), pp. 54-65, University of Newcastle upon Tyne, Birmingham, UK, 2004. [ISBN: 0-7017-0180-3].

Ayavoo, D., Pont, M.J. and Parker, S., *"Observing the development of a reliable embedded system,"* Proceedings of the 10th Ada-Europe International Conference on Reliable Software Technologies, Lecture Notes in Computer Science, In: Vardanega, T. and Wellings, A. (Eds.), pp. 167-179, Springer-Verlag, York, UK, 2005. [ISBN: 3-540-26286-5].

Ayavoo, D., Pont, M.J. and Parker, S., "Does a 'simulation first' approach reduce the effort involved in the development of distributed embedded control systems?," *Proceedings of the 6th UKACC International Control Conference,* Glasgow, Scotland, 2006.

Backus, J., *"Programming in America in the 1950s - some personal impressions,"* A History of Computing in the Twentieth Century, In: Metropolis, N., Howlett, J. and Rota, G.-C. (Eds.), Academic Press, 1980.

Baker, T.P. and Shaw, A.C., "The cyclic executive model and Ada," *Real-Time Systems,* Vol. 1, (1), pp. 7-25, 1989.

Balarin, F., Chiodo, M., Giusto, P., Hsieh, H., Jurecska, A., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A., Sentovich, E., Suzuki, K. and Tabbara, B., *"Hardware-Software Co-Design of Embedded Systems: The Polis Approach,"* Kluwer Academic Publishers, 1997.

Bannatyne, R., "Time triggered protocol-fault tolerant serial communications for real-time embedded systems," *Wescon '98. Conference Proceedings,* Anaheim, CA, USA, pp. 86-91, 1998.

Bannatyne, R., "Microcontrollers for the Automobile," Micro Control Journal, http://www.mcjournal.com/, Accessed in 2004.

Barnett, R.H., *"The 8051 Family of Microcontrollers,"* Prentice Hall, 1995. [ISBN: 0-02-306281-9].

Baron, R.J. and Higbie, L., *"Computer Architecture,"* Addison-Wesley Publishing Company, 1992.

Basili, V., Briand, L. and Melo, W., "How reuse influences productivity in object-oriented systems," *Communications of the ACM,* Vol. 39, (10), pp. 104-116, 1996.

Basili, V. and Weiss, D., "A methodology for collecting valid software engineering data," *IEEE*

Transactions on Software Engineering, Vol. 10, pp. 728-738, 1984.

Bass, L., Clements, P. and Kazman, P., *"Software architecture in practice,"* Addison Wesley, 1998.

Bate, I., "Scheduling and Timing Analysis of Safety Critical Hard Real-time Systems," Doctoral Thesis, Real-Time Systems Research Group, University of York, York, 1999.

Bate, I., *"Introduction to scheduling and timing analysis,"* The Use of Ada in Real-Time System, In: (Eds.), IEE Conference Publication 00/034, 2000.

Bautista, R. and Pont, M.J., "Is fuzzy logic a practical choice in resource-constrained embedded control systems implemented using general-purpose microcontrollers?," *Proceedings of the 9th IEEE International Workshop on Advanced Motion Control,* Istanbul, Turkey, pp. 692-697, 2006.

Beck, K., Crocker, R., Meszaros, G., Coplien, J.O., Dominick, L., Paulisch, F. and Vlissides, J., "Industrial experience with design patterns," *18th International Conference on Software Engineering (ICSE), March 25 - 29,* Berlin, GERMANY, IEEE Computer, 1996.

Bell, R., "Code Generation from Object Models," Embedded.com, http://www.embedded.com/98/9803fe3.htm, Accessed in 1998.

Bereisa, J., "Applications of microcomputers in automotive electronics," *IEEE Transactions on Industrial Electronics,* Vol. IE-30, (2), pp. 87, 1983.

Blanc, S., Gracia, J. and Gil, P., "Experiences during the experimental validation of the time-triggered architecture," *Proceedings of Design, Automation and Test in Europe (DATE 2004),* 2004.

Bolton, W., *"Microprocessor Systems,"* Longman, 2000. [ISBN: 0 582 41881 X].

Booch, G., *"Object-Oriented Analysis and Design with Applications,"* The Benjamin/Cummings Publishing Company, Inc., 1994. [ISBN: 0-8053-5340-2].

Borland, "Together Technologies," http://www.borland.com/us/products/together/, Accessed in 2005.

Bouyssounouse, B. and Sifakis, J., *"Current Design Practice and Needs in Selected Industrial Sectors,"* Embedded Systems Design: The ARTIST Roadmap for Research and Development, In: Bouyssounouse, B. and Sifakis, J. (Eds.), pp. 15-38, Springer-Verlag GmbH, 2005.

Budinsky, F., Finnie, M., Vlissides, J. and Yu, P., "Automatic code generation from design patterns," *IBM Systems,* Vol. 35, (2), pp. 151-171, 1996.

Bulka, A., "Design pattern automation," *3rd Australasian Conference on Pattern Languages of Programs (KoalaPLoP),* The Country Place, Melbourne, Australia., 2002.

Buntine, W., "Will Domain-Specific Code Synthesis Become a Silver Bullet?," *IEEE Intelligent Systems,* Vol. 13, (2), pp. 9-15, 1998.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M., *"Pattern-Oriented Software Architecture, Volume 1: A System of Patterns,"* Wiley Computer Publishing, 1996.

Buttazzo, G.C., *"Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications,"* Kluwer Academic Publishers, 1997. [ISBN: 0792399943].

Buttazzo, G.C., "Rate monotonic vs. EDF: Judgment day," *Real-Time Systems,* Vol. 29, pp. 5-26, 2005.

Camposano, R. and Wilberg, J., "Embedded system design," *Design Automation for Embedded Systems,* Vol. 1, (1-2), pp. 5, 1996.

Computer History Museum, "Computer History Museum - Apollo Guidance Computer," http://www.computerhistory.org/exhibits/highlights/apollo.shtml, Accessed in 2005.

Cooling, J., *"Software Engineering for Real-Time Systems,"* Pearson Education, 2003. [ISBN: 0 201 59620 2].

Coplien, J., "Jim Coplien," http://c2.com/cgi/wiki?JimCoplien, Accessed in 2003.

Coplien, J.O. and Harrison, N.B., *"Organisational Patterns of Agile Software Development,"* Prentice Hall, 2004. [ISBN: 0-13-146740-9].

Cottet, F. and David, L., "A solution to the time jitter removal in deadline based scheduling of real-time applications," *5th IEEE Real-Time Technology and Applications Symposium - WIP,* Vancouver, Canada, pp. 33-38, 1999.

Crnkovic, I., "Component-based Software Engineering - New Challenges in Software Development," *Journal of Computing and Information Technology,* Vol. 11, (3), pp. 151-162, 2003.

Cuatto, T., Passerone, C., Sansoe, C., Gregoretti, F., Jurecska, A. and Sangiovanni-Vincentelli, A., "A case study in embedded systems design: An engine control unit," *Design Automation for Embedded Systems,* Vol. 6, pp. 71-88, 2000.

Cunningham, W. and Beck, K., "Using pattern languages for object-oriented programs," *Presented to the OOPSLA'87 workshop on the Specification and Design for Object-Oriented Programming,* Orlando, Florida, USA, 1987.

Damasevieius, R., Majauskas, G. and Stuikys, V., "Application of design patterns for hardware design," *Proceedings of the 40th Conference on Design Automation, Session: Design Analysis Techniques,* Anaheim, CA, USA, pp. 48 - 53, 2003. [ISBN: 1-58113-688-9].

Debardelaben, J.A., Madisetti, V.K. and Gadient, A.J., "Incorporating cost modeling in embeddeds-system design," *IEEE Design and Test of Computers,* Vol. 14, (3), pp. 24-35, 1997.

Dilger, E., Fuhrer, T. and Muller, B., "Distributed fault-tolerant and safety-critical applications in vehicles - a time-triggered approach," *Computer Safety, Reliability and Security: 17th International Conference, SAFECOMP'98,* Heidelberg, Germany, Springer Berlin / Heidelberg, 1998.

Douglass, B.P., *"Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns,"* Addison Wesley, 1999. [ISBN: 0-201-49837-5].

Eakin, E., "Architecture's irascible reformer," *New York Times,* July 12th, 2003.

Edwards, C., "Gap closes in microcontrollers," *IET Electronics Systems and Software Magazine,* April/May 2006.

Edwards, T., Pont, M.J., Short, M., Scotson, P. and Crumpler, S., *"An initial comparison of synchronous and asynchronous network architectures for use in embedded control systems with duplicate processor nodes,"* Proceedings of the Second UK Embedded Forum, In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), pp. 290-303, University of Newcastle upon Tyne, Birmingham, UK, 2004.

Eggermont, L.D.J. (Ed.) *"Embedded Systems Roadmap 2002,"* Technology Foundation (STW), 2002. [ISBN: 90-73461-30-8].

Eiffel Software, "Eiffel Programming Language," http://www.eiffel.com/products/, Accessed in 2006.

Ernest, R., "Co-design of embedded systems: status and trends," *IEEE Design and Test of Computers,* Vol. Vol. 15, (2), pp. 45-54, 1998.

EU eSafety Working Group, "eSafety: the use of information and communication technology (ICT) for road safety," *Communication from the Commission to the Council and the European Parliament on Information and Communications Technologies for Safe and Intelligent Vehicles, Brussels,* Reference No. COM (2003) 542 Final, 2003.

Fenton, N., "Software measurement: A necessary scientific basis," *IEEE Transactions - Software Engineering,* Vol. 20, (3), pp. 199-206, 1994.

Flis, T.J., "The use of microprocessors for electronic engine control," *IEEE Transactions on Industrial Electronics,* Vol. IE-30, (2), pp. 75-87, 1983.

Florijn, G., Meijers, M. and Winsen, P., "Tool support for object-oriented patterns," *ECOOP'97,* Finland, 1997.

Fowler, M., "Patterns," *IEEE Software,* Vol. 20, (2), pp. 57, 2003.

France, R.B., Kim, D.-K., Ghosh, S. and Song, E., "A UML-based pattern specification technique," *IEEE Transactions on Software Engineering,* Vol. 30, (3), pp. 193-206, 2004.

Fredriksson, J., Åkerholm, M., Sandström, K. and Dobrin, R., "Attaining flexible real-time systems by bringing together component technologies and real-time systems theory," *Proceedings of the 29th Conference on EUROMICRO,* Washington, DC, USA, IEEE Computer Society, 2003. [ISBN: 0-7695-1996-2].

Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *"Design Patterns: Elements of Reusable Object-Oriented Software,"* Addison-Wesley, 1995.

Garlan, D., Allen, R. and Ockerbloom, J., "Architectural mismatch, or why it's hard to build systems out of existing parts," *In Proceedings of the 17th International Conference on Software Engineering,* Seattle, USA, 1995.

Garofalakis, M.N., Rastogi, R. and Shim, K., "SPIRIT: Sequential pattern mining with regular expression constraints," *The VLDB Journal,* pp. 223-234, 1999.

Ghezzi, C., Jazayeri, M. and Mandrioli, D., *"Fundamentals of Software Engineering,"* Prentice Hall, 2002. [ISBN: 0133056996].

Giloi, W.K., "Konrad Zuse's Plankalkul: The first high-level "non von Neumann" programming language," *IEEE Annals of the History of Computing,* Vol. 19, (2), pp. 17-24, 1997.

Graaf, B., Lormans, M. and Toetenel, H., "Embedded software engineering: The state of the practice," *IEEE Software,* Vol. 20, (6), pp. 61-69, 2003.

Grone, B. and Tabeling, P., "A system of patterns for concurrent request processing servers," *2nd Nordic Conference on Pattern Languages of Programming (VikingPLoP),* Bergen, Norway, 2003.

Guennec, A., Sunye, G. and Jezequel, J., "Precise modeling of design patterns," *Proceedings of the 3rd International Conference on the Unified Modelling Language (UML 2000),* York, UK, LNCS, pp. 482-496, 2000.

Gupta, R.K. and Micheli, G.D., "Specification and analysis of timing constraints for embedded systems," *IEEE Transactions on CAD /ICAS,* Vol. 16, (3), pp. 240-256, 1997.

Harold, E.R., *"Processing XML with Java: A Guide to SAX, DOM, JDOM, JAXP, and TrAX,"* Addison Wesley, 2002.

Harold, E.R. and Means, W.S., *"XML in a Nutshell,"* O'Reilly, 2002.

Hatton, L., *"Safer C: Developing Software for High-integrity and Safety-critical Systems,"* McGraw-Hill Book Company Europe, 1994. [ISBN: 0-07-707640-0].

Hedenetz, B. and Belschner, R., "Brake-by-wire without mechanical backup by using a TTP-communication network," *SAE World Congress,* Detroit Michigan, Warrendale, PA, USA, SAE Press, 1998.

Heering, J. and Mernik, M., "Domain-specific languages for software engineering," *Proceedings of the 35th Hawaii International Conference on System Sciences (HICSS-35),* 2002.

Heiner, G. and Thurner, T., "Time-triggered architecture for safety-related distributed real-time systems in transportation systems," *28th Annual Symposium on Fault Tolerant Computing,* Munich, Germany, IEEE Computer Society Press, pp. 402-407, 1998.

Heister, F., Riegel, J.P., Schütze, M., Schulz, S. and Zimmermann, G., "Pattern-Based Code Generation for Well-Defined Application Domains," *EuroPLoP '97,* Kloster Irsee, Germany, 1997.

Henderson, P., Howard, Y.M. and Walters, R.J., "A Tool for Evaluation of the Software Development Process," *The Journal of Systems and Software,* Vol. 59, pp. 355-362, 2001.

Henzinger, T.A., Kirsch, C.M., Sanvido, M.A.A. and Pree, W., "From control models to real-time code using Giotto," *IEEE Control Systems Magazine,* Vol. 23, (1), pp. 50-64, 2003.

Herrington, J., *"Code Generation in Action,"* Manning Publications Co., 2003. [ISBN: 1-930110-97-9].

Heuring, V.P. and Jordan, H.F., *"Computer Systems Design and Architecture,"* Benjamin/Cummings, 1997.

Hills, C., "ACCU Reviews: Patterns for Time-Triggered Embedded Systems," http://www.accu.org/bookreviews/public/reviews/p/p003031.htm, Accessed in 2001.

Hirschfeld, R. and Lämmel, R., "Reflective designs," *IEE Proceedings on Software,* Vol. 152, (1), pp. 38-51, 2005.

Hoffnagle, G.F. and Beregi, W.E., "Automating the Software Development Process," *IBM Systems Journal,* Vol. 24, (2), pp. 102-120, 1985.

Hong, S., "Scheduling algorithm of data sampling times in the integrated communication and control systems," *IEEE Transactions on Control Systems Technology,* Vol. 3, (2), 1995.

Host, M., Regnell, B. and Wohlin, C., "Using students as subjects - A comparative study of students and professionals in lead-time impact assessment," *Empirical Software Engineering,* Vol. 5, pp. 201-214, 2000.

Hsiung, P.-A., Lee, T.-Y., See, W.-B., Fu, J.-M. and Chen, S.-J., "VERTAF: an object-oriented application framework for embedded real-time," *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing,* Washington, D.C., USA, IEEE Computer Society, pp. 322-329, 2002.

Hughes, Z., Pont, M.J. and Ong, R., *"The PH Processor: A soft embedded core for use in university research and teaching,"* Proceedings of the 2nd UK Embedded Forum, 20th October, In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.), Birmingham, UK, 2005.

Hutton, R., "Mercedes calls back 1.3m cars," *The Sunday Times*, April 3rd, 2005.

IBM, "Great Moments in Microprocessor History," http://www-128.ibm.com/developerworks/library/pa-microhist.html?ca=dgr-mw08MicroHistory, Accessed in 2006.

IBM, "IBM Patterns for e-Business," http://www-128.ibm.com/developerworks/patterns/, Accessed in 2006.

IEEE Std. 610.12-1990, "Glossary of Software Engineering Terminology," *Software Engineering Standards Collection,* Los Alamitos, Calif., IEEE CS Press, 1993. [ISBN: 1048-06T].

Intel, "Intel Museum - Moore's Law," http://www.intel.com/museum/archives/history_docs/mooreslaw.htm, Accessed in 2005.

Jacobson, I., Booch, G. and Rumbaugh, J., *"The Unified Software Development Process,"* Addison Wesley, 1999. [ISBN: 0-201-57169-2].

Jenko, M., Medjeral, N. and Butala, P., "Component-based software as a framework for concurrent design of programs and platforms - an industrial kitchen appliance embedded system," *Microprocessors and Microsystems,* Vol. 25, pp. 287-296, 2001.

Jerri, A.J., "The Shannon sampling theorem: its various extensions and applications a tutorial review," *Proceedings of the IEEE,* IEEE, pp. 1565-1596, 1977.

Kalinsky, D., "Context Switch," Embedded Systems Design, http://www.embedded.com/story/OEG20010222S0038, Accessed in 2001.

Karlsson, A., "X-by-Wire Systems and Time-Triggered Protocols," MSc Thesis, Uppaal Research Group, Department of Information Technology, Uppsala University, Uppsala, Sweden, 2002.

Kerth, N., "Amazon Reviews: Patterns for Time-Triggered Embedded Systems," http://www.amazon.com/gp/product/0201331381/102-6313247-9522544?v=glance&n=283155&%5Fencoding=UTF8&v=glance, Accessed in 2001.

Key, S., Pont, M. and Edwards, S., *"Implementing low -cost TTCS systems using assembly language,"* Proceedings of the 8th European Conference on Pattern Languages of Programs (EuroPLoP 2003), In: Henney, K. and Schutz, D. (Eds.), pp. 667-690, Universitätsverlag Konstanz, Irsee, Germany, 2003. [ISBN: 3-87940-788-6].

Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., Emam, K.E. and Rosenberg, J., "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions - Software Engineering,* Vol. 28, (1), pp. Pp. 721-734, 2002.

Koffman, E.B., *"Turbo Pascal,"* Addison-Wesley, 1998. [ISBN: 0201350866].

Kopetz, H., "The time-triggered architecture," *In Proceedings of the 1 st International Symposium on Object-Oriented Realtime Distributed Computing,* pp. 22-29, 1988.

Kopetz, H., *"Real-time Systems: Design Principles for Distributed Embedded Applications,"* Kluwer Academic, 1997.

Krishnamurthi, S. and Felleisen, M., "Toward a formal theory of extensible software," *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering,* Lake Buena Vista, Florida, United States, ACM Press, New York, USA, pp. 88 - 98, 1998.

Kukkala, P., Riihimaki, J., Hannikainen, M., Hamalainen, T.D. and Kronlof, K., "UML 2.0 profile for embedded system design," *Design, Automation and Test in Europe (DATE '05),* pp. 710-716, 2005.

Kurian, S. and Pont, M.J., *"Building reliable embedded systems using Abstract Patterns, Patterns, and Pattern Implementation Examples,"* Proceedings of the 2nd UK Embedded Forum, 20th October, In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.), Birmingham, UK, 2005.

Kurian, S. and Pont, M.J., "Evaluating and improving pattern-based software designs for resource-constrained embedded systems," *Paper to be presented at "Safety and Reliability for Managing Risk 2006" (ESREL 2006),* Estoril, Portugal, 2006.

Kurian, S. and Pont, M.J., "Restructuring a pattern language which supports time-triggered co-operative software architectures in resource-constrained embedded systems," *Paper to be presented at the 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006),* Germany, 2006.

Kurian, S. and Pont, M.J., "Maintenance and evolution of resource-constrained embedded systems created using design patterns," *Journal of Systems and Software*, in press a.

Lea, D., "Christopher Alexander: An introduction for object-oriented designers," *Source ACM SIGSOFT Software Engineering Notes,* Vol. 19, (1), pp. 39-46, 1994.

Lee, W., Yoon, M. and Sunwoo, M., "A cost- and time-effective hardware-in-the-loop simulation platform for automotive engine control systems," *Proceedings of the I MECH E Part D Journal of Automobile Engineering,* Vol. 217, (1), pp. 41-52, 2003.

Lethbridge, T.C., Sim, S.E. and Singer, J., "Studying software engineers: data collection techniques for software field studies," *Empirical Software Engineering,* Vol. 10, pp. 311-

341, 2005.

Levenez, E., "Computer Languages History," http://www.levenez.com/lang/, Accessed in 2006.

Lippiatt, A.G., *"The Architecture of Small Computer Systems,"* Prentice/Hall International, 1981. [ISBN: 0-13-044750-1].

Liu, C.L. and Layland, J.W., "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the Association for Computing Machinery,* Vol. 20, (1), pp. 46-61, 1973.

Liu, J., Chou, P.H., Bagherzadeh, N. and Kurdahi, F., "Power-aware scheduling under timing constraints for mission-critical embedded systems," *Proceedings of the 38th Conference on Design Automation,* Las Vegas, Nevada, USA, pp. 840-845, 2001. [ISBN: 1-58113-297-2].

Locke, C.D., "Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives," *Real-Time Systems,* Vol. 4, (1), pp. 37-53, 1992.

Maaita, A. and Pont, M.J., "Techniques for jitter reduction in time-triggered embedded systems in which limited pre-emption is required," *Control Engineering Practice,* submitted 2005.

MacDonald, S., Szafron, D., Schaeffer, J., Anvik, J., Bromling, S. and Tan, K., "Generative design patterns," *Proceedings 17th IEEE International Conference on Automated Software Engineering (ASE'02),* Edinburgh, Scotland, IEEE Computer Society, 2002.

Marley, J., "Evolving microprocessors which better meet the needs of automotive electronics," *Proceedings of the IEEE,* Vol. 66, (2), pp. 142-150, 1978.

Marquardt, K., "Neglected Architecture," *2nd Nordic Conference on Pattern Languages of Programming (VikingPLoP),* Bergen, Norway, 2003.

Marsh, P., "Models of control," *IEE Electronics Systems and Software,* Vol. 1, (6), pp. 16-19, 2003.

Mart, P., Fuertes, J.M., Vill?, R. and Fohler, G., "On Real-Time Control Tasks Schedulability," *European Control Conference (ECC01),* Porto, Portugal, pp. 2227-2232, 2001.

Martin, G., Lavagno, L. and Louis-Guerin, J., "Embedded UML: a merger of real-time UML and co-design," *Proceedings of the 9th International Symposium on Hardware/Software Codesign,* Copenhagen, Denmark, pp. 23 - 28, 2001.

Martin, G. and Müller, W. (Eds.), *"UML for SOC Design,"* Springer, 2005. [ISBN: 0-387-25744-6].

Martin, R.A. and Shafer, L.A., "Providing a framework for effective software quality measurement: Making a science of risk assessment," *Massachusetts, USA,* The 6th Annual International Symposium of International Council on Systems Engineering (INCOSE), Systems Engineering: Practices and Tools, 2006.

Martinez, K., Hart, J. and Ong, H.L.R., "Environmental sensor networks," *IEEE Computer,* Vol. 37, (8), pp. 50-56, 2004.

MathWorks, "The MathWorks," http://www.mathworks.com/, Accessed in 2005.

McCabe, T., "A software complexity measure," *IEEE Transactions on Software Engineering,* Vol. 2, pp. 308-320, 1976.

McGinnity, T.M. and Maguire, L.P., "A CASE-tool oriented approach for embedded systems design," *Microprocessors and Microsystems,* Vol. 24, pp. 493-499, 2001.

Meyer, B., *"Object-Oriented Software Construction,"* Prentice Hall, 1997.

Meyer, B., "The grand challenge of trusted components," *25th International Conference on Software Engineering,* IEEE Computer Society, pp. 660-667, 2003.

Meyer, B. and Arnout, K., "Componentization: The Visitor Example," *IEEE Computer,* Vol. 39, (7), pp. 23-30, 2006.

Microsoft, "Microsoft .Net," http://www.microsoft.com/net/default.mspx, Accessed in 2006.

Microsoft, "Microsoft Patterns & Practices," http://msdn.microsoft.com/practices/, Accessed in 2006.

Mili, H., Ah-Ki, E., Godin, R. and Mcheick, H., "An experiment in software component retrieval," *Information and Software Technology,* Vol. 45, pp. 633-649, 2003.

Milicev, D., "Automatic model transformations using extended UML object diagrams in modeling environments," *IEEE Transactions on Software Engineering,* Vol. 28, (4), pp. 413-431, 2002.

MISRA, "Guidelines for the use of the C language in critical systems," *The Motor Industry Software Reliability Association,* Reference No. MISRA-C:2004, 2004.

Mooney, V.J. and Micheli, G.D., "Hardware/software co-design of run-time schedulers for real-time systems," *Design Automation for Embedded Systems,* Vol. 6, pp. 89 - 144, 2000.

Morton, T.D., *"Embedded Microcontrollers,"* Prentice Hall, 2001. [ISBN: 0-13-907577-1].

Moser, S. and Nierstrasz, O., "The effect of object oriented frameworks on developer productivity," *IEEE Computer,* Vol. 29, (9), pp. 45-51, 1996.

Mullerburg, M., "Software intensive embedded systems," *Information and Software Technology,* Vol. 41, pp. 979-984, 1999.

Mwelwa, C., Athaide, K., Mearns, D., Pont, M.J. and Ward, D., *"Rapid software development for reliable embedded systems using a pattern-based code generation tool,"* In-vehicle software and hardware systems, In: Society of Automotive Engineers (Eds.), Paper presented at the Society of Automotive Engineers (SAE) World Congress, Detroit, Michigan, USA, 2006. [ISBN: 0-7680-1763-7].

Mwelwa, C. and Pont, M.J., "Two simple patterns to support the development of reliable embedded systems," *2nd Nordic Conference on Pattern Languages of Programming (VikingPLoP),* Bergen, Norway, 2003.

Mwelwa, C., Pont, M.J. and Ward, D., "Towards a CASE tool to support the development of reliable embedded systems using design patterns," *Proceedings of the 1st International Workshop on Quality of Service in Component-Based Software Engineering (CBSE),* Toulouse, France, CEPADUES-EDITIONS, pp. 67-80, 2003. [ISBN: 2-85428-617-0].

Mwelwa, C., Pont, M.J. and Ward, D., "Using patterns to support the development and maintenance of software for reliable embedded systems: A case study," *Proceedings of the IEE/ACM Postgraduate Seminar on "Systems-on-Chip" Design, Test and Technology,* Loughborough, UK, IEE, 2004(a). [ISBN: 0 86341 460 5].

Mwelwa, C., Pont, M.J. and Ward, D., *"Code generation supported by a pattern-based design methodology,"* Proceedings of the 1st UK Embedded Forum, In: Pont, M.J. (Eds.), pp. 36-55, University of Newcastle upon Tyne, Birmingham, UK, 2004(b).

Mwelwa, C., Pont, M.J. and Ward, D., *"Developing reliable embedded systems using a pattern-based code generation tool: A case study,"* Proceedings of the 2nd UK Embedded Forum, In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.), pp. 177-193, Birmingham, UK, 2005. [ISBN: 0-7017-0191-9].

Nahas, M., Short, M.J. and Pont, M.J., "Exploring the impact of software bit stuffing on the behaviour of a distributed embedded control system implemented using CAN," *Proceedings of the 10th international CAN Conference,* Rome, Italy, pp. 10-1 to 10-7, 2005.

Nakata, A., Tanimoto, T., Sasaki, S. and Higashino, T., "A global timed bisimulation preserving abstraction for parametric time-interval automata," *International Journal of Foundations of Computer Science,* Vol. 17, (4), pp. 833-850, 2006.

NASA, "The Apollo Program (1963 - 1972)," http://nssdc.gsfc.nasa.gov/planetary/lunar/apollo.html, Accessed in 2005.

Nissanke, N., *"Realtime Systems,"* Prentice Hall, 1997.

Noble, J. and Weir, C., *"Small Memory Software,"* Addison Wesley, 2001.

Object Management Group, "Model Driven Architecture," www.omg.org/mda/, Accessed in 2006.

Object Management Group, "Object Management Group," http://www.omg.org, Accessed in 2006.

Ogata, K., *"Modern Control Engineering,"* Prentice-Hall, 2002.

O'Halloran, C., "Issues for the automatic generation of safety critical software," *The 15th IEEE International Conference on Automated Software Engineering,* Grenoble, France, 2000. [ISBN: 0-7695-0710-7].

O'Reilly, "XML from the Inside Out," www.xml.com, Accessed in 2004.

Orlikowski, W.J., "CASE tools as organizational change: Investigating incremental and radical changes in systems development," *Management Information Systems,* Vol. 117, (3), pp. 309-341, 1993.

Pagel, B.-U. and Winter, M., "Towards pattern-based tools," *EuropLop '96,* Kloster Irsee, Germany, 1996.

Parikh, C.R., Pont, M.J., Li, Y.H., Jones, N.B. and Twiddle, J.A., "Towards a flexible application framework for data fusion using real-time design patterns," *Proceedings of 6th European Congress on Intelligent Techniques & Soft Computing (EUFIT),* Aachen, Germany, pp. 1131-1135, 1998.

Paternotte, S., "MISRA C in safety-critical systems: how COTS compiler technologies enforce best-practice programming," *COTS Journal,* Vol. 4, (2), pp. 20-26, 2002.

Pawlicki, J., "Formalisation of embedded system development: History and present," *Annual Quality Congress,* Kansas City, USA, pp. 581-588, 2003.

Pelechano, V., PAstor, O. and Insfran, E., "Automated code generation of dynamic specialisations: An approach based on design patterns and formal techniques," *Data & Knowledge Engineering,* Vol. 40, pp. 315-353, 2002.

Phatrapornnant, T. and Pont, M., "The application of dynamic voltage scaling in embedded systems employing a TTCS software architecture: a case study," *Proceedings of the IEE / ACM Postgraduate Seminar on System-On-Chip Design, Test and Technology,* Loughborough, UK, pp. 3-8, 2004.

Phatrapornnant, T. and Pont, M.J., "Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling," *IEEE Transactions on Computers (Special Issue on Design and Test of Systems-On-a-Chip),* Vol. 55, (2), pp. 113-124, 2006.

Pont, M.J., "Control system design using real-time design patterns," *Proceedings of Control '98,* Swansea, UK, pp. 1078-1083, 1998b.

Pont, M.J., "Patterns for embedded systems," *Invited presentation to IEE East Midland Centre,* Lincolnshire, UK, 1999.

Pont, M.J., *"Designing and implementing reliable embedded systems using patterns,"* Proceedings of the 4th European Conference on Pattern Languages of Programming and Computing (EuroPLoP 1999), In: Devos, M. (Eds.), Universittsverlag Konstanz, 2000a. [ISBN: 3-87940-774-6].

Pont, M.J., "Can patterns increase the reliability of embedded hardware-software co-designs?," *IEE Colloquium on Hardware-Software Co-Design,* Savoy Place, London, IEE Colloquium Digests, 2000b.

Pont, M.J., *"Patterns for Time-Triggered Embedded Systems,"* Addison-Wesley, 2001.

Pont, M.J., "Supporting the development of time-triggered co-operatively scheduled (TTCS) embedded software using design patterns," *Informatica,* Vol. 27, pp. 81-88, 2003.

Pont, M.J. and Banner, M.P., "Designing embedded systems using patterns: A case study," *Journal of Systems and Software,* Vol. 71, (3), pp. 201-213, 2004.

Pont, M.J., Li, Y., Parikh, C.R. and Wong, C.P., "The design of embedded systems using software patterns," *Proceedings of Condition Monitoring,* Swansea, UK, pp. 221-236, 1998a.

Pont, M.J., Li, Y.H., Parikh, C.R. and Wong, C.P., "The design of embedded systems using software patterns," *Proceedings of Condition Monitoring 1999,* Swansea, UK, pp. 221-236, 1999.

Pont, M.J. and Mwelwa, C., "Developing reliable embedded systems using 8051 and ARM processors: Towards a new pattern language," *Proceedings of the 2nd Nordic Conference on Pattern Languages of Programming (VikingPLoP),* Bergen, Norway, 2003b.

Pont, M.J., Mwelwa, C., Bonthonneau, L., Ayavoo, D., Athaide, K., Mearns, D., Kurian, S. and Ward, D., "Pattern-based development of time-triggered embedded systems using software tools: Challenges and solutions," *Journal of Systems and Software*, submitted 2006.

Pont, M.J., Norman, A.J., Mwelwa, C. and Edwards, T., *"Prototyping time-triggered embedded systems using PC hardware,"* Proceedings of the 8th European Conference on Pattern Languages of Programs (EuroPLoP 2003), In: Henney, K. and Schutz, D. (Eds.), pp. 691-716, Universitätsverlag Konstanz, Irsee, Germany, 2003a. [ISBN: 3-87940-788-6].

Pont, M.J. and Ong, H.L.R., "Using watchdog timers to improve the reliability of TTCS embedded systems," *Proceedings of the 1st Nordic Conference on Pattern Languages of Programs,* pp. 159-200, 2002.

Pop, P., "Scheduling and Communication Synthesis for Distributed Real-Time Systems," Doctoral Thesis, Institute of Technology, University of Linkopings, Linkopings, 2000.

Pop, T., Eles, P. and Peng, Z., "Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems," *Proceedings of the 10th International Symposium on Hardware/Software Codesign,* pp. 187-192, 2002. [ISBN: 1-58113-542-4].

Prechelt, L. and Unger, B., "An experiment measuring the effects of personal software process (PSP) training," *IEEE Transactions on Software Engineering,* Vol. 27, (5), pp. 465-472, 2001.

Prechelt, L., Unger, B., Philippsen, M. and Tichy, W.F., "Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance," *IEEE Transactions on Software Engineering,* Vol. 28, (6), pp. 595-606, 2002.

Purushothaman, R. and Perry, D.E., "Toward understanding the rhetoric of small source code changes," *IEEE Transactions on Software Engineering,* Vol. 31, (6), pp. 511-526, 2005.

Reason, J., *"Managing the Risks of Organizational Accidents,"* Ashgate, 1997.

Riehle, D. and Zullighoven, H., "Understanding and using patterns in software development," *Theory and Practice of Object Systems,* Vol. 2, (1), pp. 3-13, 1996.

Rincon, F., Moya, F., Barba, J. and Lopez, J.C., "Model Reuse through Hardware Design Patterns," *Design, Automation and Test in Europe (DATE'05),* pp. 324-329, 2005.

Rising, L., *"Pattern Writting,"* The Patterns Handbook : Techniques, Strategies, and Applications, In: Rising, L. (Eds.), Cambridge University Press, 1998.

Rising, L. (Ed.) *"Design Patterns in Communications Software,"* Oxford University Press, New York, USA, 2001. [ISBN: 0521790409].

Rogers, G.F., *"Framework Based Software Development in C++,"* Prentice Hall PTR, 1997.

Rosenberg, L. and Hyatt, L., *"Software Quality Metrics for Object Oriented Development,"* In: SATC, N. (Eds.), 1997.

Saasa, O. and Carlsson, J., *"Aid and Poverty Reduction in Zambia: Mission Unaccomplished,"* The Nordic Africa Institute, 2002. [ISBN: 9171064893].

Saeki, A.M., Iguchi, K., Wen-yin, K. and Shinohara, M., "A meta-model for representing software specification & design methods," *Proceedings of the IFIP WG8.1 Working Conference on Information System Development Process,* North Holland, pp. 149-166, 1993.

Salingaros, N.A., "Some Notes on Christopher Alexander," http://www.math.utsa.edu/sphere/salingar/Chris.text.html, Accessed in 2006.

Sammet, J.E., "History of IBM's Technical Contributions to High Level Programming Languages," *IBM Journal of Research and Development,* Vol. 25, (5), pp. 520-534, 1981.

Sanderson, D., "How scooter of the future went into an embarrassing reverse," *The Times,* 15th September 2006.

Schatz, B., Hain, T., Houdek, F., Prenninger, W., Rappl, M., Romberg, J., Slotosch, O., Strecker, M. and Wisspeintner, A., "CASE tools for embedded systems," *Technical University of Munich, Munich,* Reference No. TUM-I0309, 2003.

Schmidt, D.C., "Using design patterns to develop reusable object-oriented communication software," *Communications of the ACM,* Vol. 38, (10), pp. 65-74, 1995.

Schneidewind, N.F., "Measuring and evaluating maintenance process using reliability, risk and test metrics," *IEEE Transactions on Software Engineering,* Vol. 25, (6), pp. 769-781, 1999.

Shapiro, S., "Splitting the difference: The historical necessity of synthesis in software engineering," *IEEE Annals of the History of Computing,* Vol. 19, (1), pp. 20-54, 1997.

Shaw, A.C., *"Real-Time Systems and Software,"* Wiley, 2001.

Short, M., Pont, M.J. and Huang, Q., "Safety and reliability of distributed embedded systems: Simulation of vehicle longitudinal dynamics," *Embedded Systems Laboratory, University of Leicester, Leicester,* Reference No. ESL04/01, 2004a.

Short, M., Pont, M.J. and Huang, Q., "Safety and reliability of distributed embedded systems: Simulation of motorway traffic flows," *Embedded Systems Laboratory, University of Leicester, Leicester,* Reference No. ESL04/02, 2004b.

Short, M., Pont, M.J. and Huang, Q., "Safety and reliability of distributed embedded systems: Development of a hardware-in-the-loop test facility for automotive ACC implementations," *Embedded Systems Laboratory, University of Leicester, Leicester,* Reference No. ESL04/03, 2004c.

SIA, "Semiconductor Industry Association," SIA, http://www.sia-online.org/home.cfm, Accessed in 2002.

Sickle, T.V., *"Reusable Software Components: Object-Oriented Embedded Systems Programming in C,"* Prentice Hall, 1996. [ISBN: 0136136885].

Smith, J., Kokar, M. and Baclawski, K., "Formal verication of UML diagrams: A first step towards code generation," *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 99),* 1999.

Solingen, R.V. and Stalenhoef, P., "Effort measurement of support to software products," *Proceeding of the International Workshop on Empirical Studies of Software*

Maintenance, Bari, Italy, 1997.

Sparks, S., Benner, K. and Faris, C., "Managing object-oriented framework reuse," *IEEE Computer,* Vol. 29, (4), pp. 52-62, 1996.

Storey, N., *"Safety-Critical Systems,"* Addison-Wesley, 1996. [ISBN: 0-201-42787-7].

Stothert, A. and MacLeod, I., "Effect of timing jitter on distributed computer control system performance," *Proceedings of DCCS'98 - 15th IFAC Workshop on Distributed Computer Control Systems,* 1998.

Sunye, G., Guennec, A.L. and Jezequel, J., "Design patterns application in UML," *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000),* Sophia Antipolis and Cannes, France, LNCS 1850, pp. 44-62, 2000.

SysML Partners, "SysML Open Source Project," http://www.sysml.org/, Accessed in 2005.

Szemethy, T., "Case study: Model transformations for time-triggered languages," *Electronic Notes in Theoretical Computer Science,* Vol. 152, pp. 175-190, 2006.

Tanenbaum, A.S., *"Distributed Operating Systems,"* Prentice Hall, 1994. [ISBN: 0-13-219908-4].

The Apache Software Foundation, "The Apache Struts Project," http://struts.apache.org/, Accessed in 2006.

The Hillside Group, "The Patterns Home Page," http://www.hillside.net/, Accessed in 2005.

Torchiano, M., "Documenting pattern use in java programs," *Proceedings of IEEE International Conference on Software Maintenance (ICSM 2002),* Montreal, Canada, pp. 230-233, 2002.

Torngren, M., "Fundamentals of implementing real-time control applications in distributed computer systems," *Real-Time Systems,* Vol. 14, pp. 219-250, 1998.

Torngren, M. and Redell, O., "A modelling framework to support the design and analysis of distributed real-time control systems," *Microprocessors and Microsystems,* Vol. 24, pp. 81-93, 2000.

TTA-Group, "The Cross-Industry Consortium for Time-Triggered Systems," http://www.ttagroup.org/index.htm, Accessed in 2006.

Vanderperren, Y. and Dehaene, W., "UML 2 and SysML: an approach to deal with complexity in SoC/NoC design," *Design, Automation and Test in Europe (DATE '05),* pp. 716-718, 2005.

Viljamaa, A., "Pattern-based Framework Annotation and Adaptation - A Systematic Approach," *University of Helsinki, Helsinki,* Reference No. C-2001-52, 2001.

Vokac, M., Tichy, W., Sjoberg, D.I.K., Arisholm, E. and Aldrin, M., "A controlled experiment comparing the maintainability of programs designed with and without design patterns - A replication in a real programming environment," *Empirical Software Engineering,* Vol. 9, pp. 149-195, 2004.

Voros, N., Sanchez, L., Alonso, A., Birbas, A., Birbas, M. and Jerraya, A., "Hardware/software codesign of complex embedded systems: An approach using efficient process models, multiple formalism specification and validation via cosimulation," *Design Automation for Embedded Systems,* Vol. 8, pp. 5-49, 2003.

Wang, S. and Shin, K.G., "An architecture for embedded software integration using reusable components," *Proceedings of the 2000 International Conference on Compilers, Architecture and Synthesis for Embedded Systems,* San Jose, California, USA, ACM Press, pp. 110 - 118, 2000. [ISBN: 1-58113-338-3].

Ward, N.J., *"The static analysis of a safety-critical avionics control system,"* Air Transport Safety: Proceedings of the Safety and Reliability Society Spring Conference, In: Corbyn, D.E. and Bray, N.P. (Eds.), SaRS, 1991.

Warnes, L., *"Electronic and Electrical Engineering,"* MacMillan Press, 1998. [ISBN: 0-333-74311-3].

Wartnaby, C.E., Bennett, S.M. and Ellims, M., "Auto-generated production code development for Ford/Think Fuel Cell Vehicle Programme," *Presented at the Society of Automotive Engineers (SAE) World Congress,* Detroit, Michigan, USA, 2003. [ISBN: SAE-2003-03AE-60].

Whalen, M.W. and Heimdahl, M.P.E., "On the requirements of high-integrity code generation," *Proceedings of the 4th High Assurance in Systems Engineering Workshop,* Washington DC, 1999.

Wild, F., "Instantiating code patterns - Patterns applied to software development," *Dr Dobb's Journal,* Vol. 21, (6), pp. 72-76, 1996.

Wong, C.P. and Pont, M.J., *"An overview of an evolutionary algorithm pattern language,"* Advances in Soft Computing: Soft Computing Techniques and Applications, In: John, R. and Birkenhead, R. (Eds.), pp. 129-134, Springer-Verlag, Heidelberg, 2000.

Xu, J., "On inspection and verification of software with timing requirements," *IEEE Transactions on Software Engineering,* Vol. 29, (8), pp. 705-720, 2003.

Xu, J. and Parnas, D.L., "Scheduling processes with release times, deadlines, precedence and exclusion relations," *IEEE Transactions on Software Engineering,* Vol. 16, (3), pp. 360-369, 1990.

Xu, J. and Parnas, D.L., "Priority scheduling versus pre-run-time scheduling," *International Journal of Time-Critical Systems,* Vol. 18, (7-23), 2000.

Yang, H.L., "Adoption and implementation of CASE tools in Taiwan," *Information & Management,* Vol. 35, pp. 89-112, 1999.

Yoshida, N., "Design patterns applied to object-oriented SoC design," *10th Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI),* Nara, Japan, 2001.

Zimmermann, W., "Reusable software libraries," *IEE Proceedings on Software,* Vol. 152, (1), pp. 1-1, 2005.

Zuse, H., *" Software Complexity Measures and Methods,"* Walter de Gruyter, 1991.

Appendix A The PTTES Collection

Table A-1 The 72 patterns in the original PTTES collection (Pont, 2001)

STANDARD 8051	SMALL 8051	EXTENDED 8051
CRYSTAL OSCILLATOR	CERAMIC OSCILLATOR	RC RESET
ROBUST RESET	ON-CHIP MEMORY	OFF-CHIP DATA MEMORY
OFF-CHIP CODE MEMORY	NAKED LED	NAKED LOAD
IC BUFFER	BJT DRIVER	IC DRIVER
MOSFET DRIVER	SSR DRIVER (DC)	EMR DRIVER
SSR DRIVER (AC)	SUPER LOOP	PROJECT HEADER
PORT I/O	PORT HEADER	HARDWARE DELAY
SOFTWARE DELAY	HARDWARE WATCHDOG	CO-OPERATIVE SCHEDULER
HARDWARE TIMEOUT	LOOP TIMEOUT	MULTI-STAGE TASK
MULTI-STATE TASK	HYBRID SCHEDULER	PC LINK (RS-232)
SWITCH INTERFACE (SOFTWARE)	SWITCH INTERFACE (HARDWARE)	ON-OFF SWITCH
MULTI-STATE SWITCH	KEYPAD INTERFACE	MX LED DISPLAY
LCD CHARACTER PANEL	I2C PERIPHERAL	SPI PERIPHERAL
SCI SCHEDULER (TICK)	SCI SCHEDULER (DATA)	SCU SCHEDULER (LOCAL)
SCU SCHEDULER (RS-232)	SCU SCHEDULER (RS-485)	SCC SCHEDULER
DATA UNION	LONG TASK	DOMINO TASK
HARDWARE PULSE-COUNT	SOFTWARE PULSE-COUNT	HARDWARE PRM
SOFTWARE PRM	ONE-SHOT ADC	ADC PRE-AMP
SEQUENTIAL ADC	A-A FILTER	CURRENT SENSOR
HARDWARE PWM	PWM SMOOTHER	3-LEVEL PWM
SOFTWARE PWM	DAC OUTPUT	DAC SMOOTHER
DAC DRIVER	PID CONTROLLER	255-TICK SCHEDULER
ONE-TASK SCHEDULER	ONE-YEAR SCHEDULER	STABLE SCHEDULER

Table A-2 Recent additions to the original PTTES collection (Kurian and Pont, 2006; Mwelwa and Pont, 2003; Key et al., 2003; Pont et al., 2003a)

HEARTBEAT LED	ERROR LED	PORT WRAPPER (PC)
SANDWICH DELAY	TTC-SL SCHEDULER	TTC-ISR SCHEDULER
TTCO SCHEDULER (ASSEMBLY LANGUAGE)†	TTCO SCHEDULER (DOS)†	

† PIEs for TTC Scheduler also known as Co-operative Scheduler in the original PTTES collection (Pont, 2001).

Appendix B PTTES User Survey Questionnaire

Table B-1 Questionnaire used in PTTES user survey

1.) What is your current course of study and current year (if applicable)?

2.) Give a brief description of an embedded system you have recently developed.

3.) Did you use patterns to develop your system (Y/N)?

4.) Describe three patterns you have used for this system?

a) For each pattern, explain how you applied it.

b) For each pattern, what other sources of information did you refer to when applying it?

c) For each pattern, Where there any features of this pattern that were unclear? If so why?

5.) Did use of any of these patterns interfere with any other patterns you were using?

6.) Did the code work first time? If not, why?

7.) How do you commence the design of an embedded system (number applicable choices in appropriate order)?

a) Search the web for any similar work

b) Look at PTTES for any similar work

c) Sketch a software or hardware design

d) Go straight into coding

e) Build a basic microcontroller circuit

8.) What aspect of an embedded project do you deal with first?

a) Hardware design b) Software design

9.) Would you rather write your own code from scratch if you had enough time as opposed to relying on PTTES (Y/N)?

10.) Do you find PTTES beneficial in your work (Y/N)?

a) If YES, in what way? b) If NO, why not?

11.) How do you find the task of adapting code from PTTES for your application (tick appropriate answer)?

a) Easy b) Fair c) Hard

12.) How do you find the task of finding an appropriate software component in PTTES (tick appropriate answer)?

a) Fairly straight forward b) Complicated

13.) If complicated was the answer to Q.12, why (select appropriate answer)?

a) Poor nomenclature b) Poor documentation

14.) Is there anything you feel could enhance the use of PTTES?

15.) Do you use any CAD or CASE tool (apart from the compiler) to aid you with your design process (Y/N)?

16.) If YES was the answer to Q.15, which tool do you use and what is it used for?

17.) If NO was the answer to Q.15, why?

a) Not enough time

b) I do not know how to use the tools available

c) I do not understand UML

d) I rely on electronic text books for my hardware design

e) Would rather sketch my design on paper

Appendix C Two Patterns to Support Embedded Systems Development

This appendix presents two patterns: HEARTBEAT LED and ERROR LED (Mwelwa and Pont, 2003) presented at the Viking PLoP 2003 conference during the course of the work described in this thesis.

HEARTBEAT LED

Context

- You are developing (or maintaining) an embedded application based on a microcontroller or microprocessor.
- You are programming in C (or a similar language).
- Your application has an architecture based on some form of scheduler.

Problem

How can you tell, at a glance, if your system is "alive"?

Design constraints

Many embedded systems have little or no user interface. There is not generally a screen on which you can display error messages or warnings to the user. If you are working on a system prototype, or performing maintenance in the field, how can you tell that the system is "alive" - that it has power and (at least) the scheduler is running?

You could, of course, hook up a debugging link (e.g. a JTAG link), or a simpler serial link (based on RS-232), but this takes time and including suitable ports on your production system may not be practical or cost effective. Often a very simple, low-cost solution is required.

Solution

Every time we implement an embedded system, the first task we include is one that flashes a "heartbeat" LED. Wherever possible, this LED stays with the system, right into production.

We tend to use a 50% duty cycle and a frequency of 0.5 Hz (that is, the LED runs continuously, on for one second, off for one second, and so on) but this is – of course – up to you.

Use of this simple technique provides the following key benefit:

- The development team, the maintenance team and, where appropriate, the users, can tell at a glance that the system has power, and that the scheduler is operating normally.

In addition, during development, there are two less significant (but still useful) side benefits:

- After a little practice, the developer can tell "intuitively" - by watching the LED - whether the scheduler is running at the correct rate: if it is not, it may be that the timers have not been initialized correctly, or that an incorrect crystal frequency has been assumed.
- By adding the "Heartbeat" task to the scheduler array after all other tasks have been included, the developer can tell immediately if the task array is large enough to match the needs of the application (if the array is not large enough, the LED will never flash).

Reliability and safety implications

Use of this simple technique may help to improve system reliability since it provides those developing the system with an indication of its health throughout the development lifecycle.

Hardware requirements

HEARTBEAT LED has minimal hardware requirements. The only requirements are a port pin connected to an appropriate LED (with an appropriate resistor if required).

Cost implications

As noted above, the hardware requirements are very limited. The time taken to implement this pattern is also likely to be minimal. Overall, the costs are very low.

Overall strengths and weaknesses

☺ HEARTBEAT LED provides a simple, low-cost way of determining whether your system is "alive".

☹ Uses a port pin and associated LED hardware.

Fig C-1 Heartbeat LED

```
/*------------------------------------------------------------------*-
   Heartbeat_LED.C
   Simple 'Heartbeat LED' PIE for an Infineon C515C microcontroller.
   If everything is OK, flashes at 0.5 Hz
-*------------------------------------------------------------------*/
#include "Main.H"
#include "Port.H"
#include "Heartbeat_LED.H"

// ------ Private variable definitions -----------------------------
static bit Heartbeat_led_state_G;

/*------------------------------------------------------------------*-
  HEARTBEAT_LED_Init()
  Prepare for HEARTBEAT_Update() task.
-*------------------------------------------------------------------*/
void HEARTBEAT_LED_Init(void)
   {
   Heartbeat_led_state_G = 0;
   }

/*------------------------------------------------------------------*-
  HEARTBEAT_LED_Update()

  Flashes an LED on a specified port pin.
  Must schedule at twice the required flash rate: thus, for 0.5 Hz
  flash (on for 1 second, off for 1 second) must schedule at 1 Hz.
-*------------------------------------------------------------------*/
void HEARTBEAT_LED_Update(void)
   {
   // Change the LED from OFF to ON (or vice versa)
   if (Heartbeat_led_state_G == 1)
      {
      Heartbeat_led_state_G = 0;
      Heartbeat_led_pin = 0;
      }
   else
      {
      Heartbeat_led_state_G = 1;
      Heartbeat_led_pin = 1;
      }
   }
/*------------------------------------------------------------------*-
  ---- END OF FILE -------------------------------------------------
-*------------------------------------------------------------------*/
```

Fig C-2 Heartbeat LED PIE for the 8051 platform

ERROR LED

Context

- You have implemented HEARTBEAT LED and you now require a means of reporting errors.

Problem

If your embedded system is not working correctly, how can you tell what is wrong?

Design constraints

See HEARTBEAT LED for the design constraints.

HEARTBEAT LED can provide a very cost-effective way of telling whether your system is "alive". If the system is functioning, but has detected some errors, HEARTBEAT LED may not be of great help.

How can you report errors, without significantly increasing the system (or development) costs?

Solution

To implement ERROR LED a single LED is used to report error codes to the developer or (if appropriate) the user. In most cases, we like to base the ERROR LED on HEARTBEAT LED so that, if there are no errors, we see the usual (comforting) 0.5 Hz signal. If there is a problem, the display changes, and – by observing the different pulse rates – we can often identify the cause.

Implementation

We use a (global) error variable, and maintain a list of error codes (in Main.H). In the event of an error, we adjust the output of ERROR LED accordingly.

Reliability and safety implications

Most forms of error reporting – like ERROR LED – provide a means of improving system reliability.

Hardware requirements

See HEARTBEAT LED hardware requirements.

Cost implications

Implementing a basic implementation of ERROR LED will cost you little more than implementing HEARTBEAT LED. However, it takes time to include error reporting in your program code, and this may add to the development costs.

Maintenance

ERROR LED can be very valuable during system maintenance as it can be used to debug reported bugs.

Portability

Highly portable – can be implemented on a wide range of hardware platforms.

Related patterns and alternative solutions

See HEARTBEAT LED for the related patterns.

As an alternative solution one could easily substitute a buzzer for the LED, and thereby draw the attention of developers (or users) to errors using various sounds or different pulse frequencies.

Overall strengths and weaknesses

☺ ERROR LED provides a low-cost, non-invasive, means of error reporting.

☹ Uses a port pin and associated LED hardware.

☹ Adding error reporting takes time and hence may increase development costs.

Fig C-3 Error LED

```
/*------------------------------------------------------------------*-
   Error_LED.C
   Simple 'Error LED' task for a Philips LPC2106 ARM microcontroller.
   If everything is OK, flashes at 0.5 Hz. If there is an error code active, this is
displayed.
-*------------------------------------------------------------------*/
#include "Main.H"
#include "Port.H"
#include "Error_LED.H"
// see Scheduler for definition
extern int Error_code_G;
/*------------------------------------------------------------------*-
  Prepare for ERROR_LED_Update() function.
-*------------------------------------------------------------------*/
void ERROR_LED_Init(void)
   {
   // Set up Heartbeat_pin as GPIO
   PINSEL0 &= ~Heartbeat_pin;
   // Set Heartbeat_pin to output mode
   IODIR |= Heartbeat_pin;
   }
/*------------------------------------------------------------------*-
  ERROR_LED_Update()
  Flashes at 0.5 Hz if error code is zero. Otherwise, displays error code.
  Must schedule every second (soft deadline).
-*------------------------------------------------------------------*/
void ERROR_LED_Update(void)
   {
   static int LED_state = 0;
   static int Error_state = 0;
   if (Error_code_G == 0)
      {
      // No errors recorded
      // - just flash at 0.5 Hz
      // Change the LED from OFF to ON (or vice versa)
      if (LED_state == 1)
         {
         LED_state = 0;
          IOCLR = Error_pin;  // Set to 0
         } else {
         LED_state = 1;
          IOSET = Error_pin;  // Set to 1
         }
      return;
      }
   // If we are here, there is an error code ...
   Error_state++;
   if (Error_state < Error_code_G*2)
      {
      LED_state = 0;
      IOCLR = Error_pin;  // Set to 0
      } else {
      if (Error_state < Error_code_G*4)
         {
         // Change the LED from OFF to ON (or vice versa)
         if (LED_state == 1)
            {
            LED_state = 0;
            IOCLR = Error_pin;  // Set to 0
            }
         else {
            LED_state = 1;
             IOSET = Error_pin;  // Set to 1
            }
         } else {
         Error_state = 0;
         }
      }
   }
/*------------------------------------------------------------------*-
  ---- END OF FILE -------------------------------------------------
-*------------------------------------------------------------------*/
```

Fig C-4 An Error LED PIE for the ARM platform

Appendix D Abstracts of Associated Publications

Pont, M.J., Mwelwa, C., Bonthonneau, L., Ayavoo, D., Athaide, K., Mearns, D., Kurian, S. and Ward, D., "Pattern-based development of time-triggered embedded systems using software tools: Challenges and solutions," Journal of Systems and Software, submitted 2006.

Abstract: *In this paper, we identify four key design challenges which must be addressed if we wish to make effective use of tool support when generating embedded systems from a pattern-based design. We describe a prototype tool – PTTES Builder – which is intended to address these design challenges. We then go on to present the results from an empirical study in which the effectiveness of tool-based pattern development was compared with an equivalent "manual" approach. The results obtained from this study suggest that – in almost all cases – the use of the PTTES Builder tool reduced the development effort. The paper concludes by making a number of suggestions for future work in this area.*

Mwelwa, C., Athaide, K., Mearns, D., Pont, M.J. and Ward, D., "Rapid software development for reliable embedded systems using a pattern-based code generation tool," In-vehicle software and hardware systems, In: Society of Automotive Engineers (Eds.), Paper presented at the Society of Automotive Engineers (SAE) World Congress, Detroit, Michigan, USA, 2006. [ISBN: 0-7680-1763-7].

Abstract: *Automated code generation has developed over the last half century from techniques based on assembly language through high-level programming languages to those based on modeling languages (such as UML). We have previously argued that the use of design patterns to support automated code generation represents a logical next step in this process. To support this claim, a pattern-based code generation tool has been developed. In this paper, we describe the tool and explore its effectiveness by means of an automotive case study.*

Mwelwa, C., Pont, M.J. and Ward, D., "Developing reliable embedded systems using a pattern-based code generation tool: A case study," Proceedings of the 2nd UK Embedded Forum, In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.), pp. 177-193, Birmingham, UK, 2005. [ISBN: 0-7017-0191-9].

Abstract: *Automated code generation has developed over the last half century from techniques based on assembly language through high-level programming languages to those based on modelling languages (such as UML). We have previously argued that the use of design patterns to support automated code generation represents a logical next step in this evolutionary process. To support this claim, a prototype pattern-based code generation tool has been developed in the Embedded Systems Laboratory. In this paper, we describe the tool and illustrate its effectiveness by applying it in a non-trivial case study.*

Mwelwa, C., Pont, M.J. and Ward, D., "Code generation supported by a pattern-based design methodology," Proceedings of the 1st UK Embedded Forum, In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.), pp. 36-55, University of Newcastle upon Tyne, Birmingham, UK, 2004b. [ISBN: 0-7017-0180-3].

Abstract: *Automatic code generation from high-level models (such as those based on UML) are becoming increasingly common. Various researchers have sought to carry out a similar code production process beginning with a pattern-based representation: such efforts have not proved overwhelmingly successful. In this paper, we consider some of the challenges involved in creating code from patterns, and argue that the one-pattern-to-many-implementations relationship – which is fundamental to a pattern-based design – makes it very difficult to create general-purpose code-generation tools. We go on to propose a solution using components as an intermediate representation. We illustrate our discussions using a prototype tool that uses this form of representation to support code generation using design patterns in the development of high-reliability embedded systems.*

Mwelwa, C., Pont, M.J. and Ward, D., "Using patterns to support the development and maintenance of software for reliable embedded systems: A case study," Proceedings of the IEE/ACM Postgraduate Seminar on "Systems-on-Chip" Design, Test and Technology, Loughborough, UK, IEE, 2004a. [ISBN: 0-86341-460-5].

Abstract: *One of the challenges of SoC design is the integration of the hardware and software components into a reliable system. In previous papers, we have argued that the use of a pattern-based design can help to support such a development process. In the present paper we present a simple case study, which illustrates how – with appropriate tool support – pattern-based design has the potential to support both the development and maintenance of software for reliable embedded systems. We also discuss the extension of these techniques to more general SoC development.*

Mwelwa, C. and Pont, M.J., "Two simple patterns to support the development of reliable embedded systems," 2nd Nordic Conference on Pattern Languages of Programming (VikingPLoP), Bergen, Norway, 2003.

Abstract: *As the title suggests, this paper is concerned with the development of software for embedded systems. Typical application areas for this type of software range from passenger cars and aircraft through to common domestic equipment, such as washing machines and microwave ovens.*

We have previously described a 'pattern language' consisting of more than eighty patterns. This language is intended to support the development of reliable embedded systems using low-cost embedded hardware with severe memory constraints. Typical implementations will employ embedded microcontrollers with a few kilobytes of available RAM.

Over the last few years, we have had the chance to observe many people use this collection when developing a range of different systems: these observations have included industrial projects and various university research projects. In this paper, we present two patterns that have resulted from these observations: HEARTBEAT LED and ERROR LED.

The two patterns are related. HEARTBEAT LED provides a simple, low-cost mechanism for providing feedback on the overall health of your system: if the LED is flashing, the core of the system is running correctly. ERROR LED goes one step further and provides a mechanism for error reporting.

Pont, M.J. and Mwelwa, C., "Developing reliable embedded systems using 8051 and ARM processors: Towards a new pattern language," Proceedings of the 2nd Nordic Conference on Pattern Languages of Programming (VikingPLoP), Bergen, Norway, 2003b.

Abstract: *We have previously described a "language" consisting of more than eighty patterns, which will be referred to here as the "PRES Collection". This language is intended to support the development of reliable embedded systems using small resource-constrained microcontrollers, including – for example – devices from the 8051 family with a few hundred bytes of available memory.*

The first complete set of these patterns was completed around three years ago and they have since been used in a range of industrial systems, numerous university research projects, as well as in undergraduate and postgraduate teaching on many university courses (e.g. see Pont and Banner, 2004; Pont, 2003). We have also begun to develop a tool to support the development of embedded systems using these patterns (Mwelwa et al., 2003).

As our experience with the collection has grown, we have began to add a number of new patterns and revised some of the existing ones (e.g. see Key et al., 2003; Pont et al., 2003a; Pont and Ong, 2002). Inevitably, by definition, a language consists of an inter-related set of patterns: as a result, it is unlikely that it will ever be possible to refine or extend such a system without causing some side effects. However, as we have worked with this collection, we have felt that there were ways in which the overall architecture could be improved in order to reduce the impact of future changes.

The paper briefly describes some of the main alteration we have made when re-factoring our original pattern collection. It then goes on to describe one of the new patterns that has resulted from this process.

Mwelwa, C., Pont, M.J. and Ward, D., "Towards a CASE tool to support the development of reliable embedded systems using design patterns," Proceedings of the 1st International Workshop on Quality of Service in Component-Based Software Engineering (CBSE), Toulouse, France, CEPADUES-EDITIONS, pp. 67-80, 2003. [ISBN: 2-85428-617-0].

Abstract: *As design complexity grows, it is becoming more difficult to implement reliable embedded systems. We have previously argued that component-based design (using design patterns) can help to alleviate such problems. In this paper we discuss the development of a CASE tool that is intended to support the development of embedded systems using patterns.*