A case for pattern-based software engineering

Thesis submitted for the degree of

Doctor of Philosophy

at the University of Leicester

Susan Kurian
(B. Tech, MSc. Engineering)

Department of Engineering, University of Leicester

Leicester, United Kingdom

December 2008

Abstract

Embedded software development is characterized by design issues involving time and resource constraints. An application- specific user interface complicates the process of developing such software using PC-based development environments. Reusing established best-practices is a useful method of dealing with such complexities. Design patterns are well-documented, time-tested solutions to classic design problems and capture significant domain knowledge. This thesis is concerned with the use of one such pattern collection suitable for building embedded systems with a time-triggered architecture.

Traditionally, a practitioner wishing to incorporate design patterns into the software being developed would read the documentation and apply the suggested solution manually. More recently, code generators designed to automate the process of converting a pattern solution to source code, have been developed. In either approach, the example solution offered as part of the pattern documentation plays a key role in obtaining source code from the design pattern documentation. However patterns contain a lot of other information which can contribute to the evaluation and application of the design pattern in a project.

The research described here suggests a framework for the use of patterns for developing software. It recognises the fact that example implementations of patterns are well-used entities. The research focuses on the use of the remaining information, particularly pattern relationships available within the document, to support design space exploration activities. This process is illustrated using a simple cruise control system.

In a bid to standardize the process of using design-specific information captured in the pattern documentation, this thesis describes an approach to formalise the pattern language. It suggests an approach based on the use of context-free grammars, to represent the natural language information held in the pattern documentation. It illustrates the use of the suggested approach using an elevator-based case study.

Copyright notice

© Susan Kurian 2009.

This thesis is copyright material and no quotation from it may be published without proper acknowledgement.

Financial sponsors

I am grateful to the following organisations for funding this research project.

1. The UK Government (Department for Education and Skills), for the ORSAS award that was a significant financial support

2. The University of Leicester (Department of Engineering), for the departmental scholarship(s) that complemented the ORSAS award

3. TTE Systems Ltd., for funding this work as industrial sponsors

Acknowledgments

I would like to take this opportunity to express my heartfelt gratitude to my project supervisor, Prof. Michael J. Pont. I have much to learn from the professional manner in which he supervised this project and the enthusiasm with which he encouraged me through the course of this research. I am highly indebted to him for giving me the opportunity to present and defend my work at various events and for being able experience the challenges of commercialising technology.

I would like to thank Dr. Michael Short, Dr. Mohammed El-Ramly, Dr. Reiko Heckel and Dr. Royan Ong for their constructive feedback over the course of the project.

I would like to thank my lovely parents, K. K. Kurian and Mary Kurian, for their love and diligent care during the formative years of my life. I truly missed having either of them around over the last few months. I am ever-indebted to my affectionate little sister, Sarah, for her infectious nonchalance to trivial issues that so easily distress me.

A special thanks to my lovely teachers at National Public School, Indiranagar and College of Engineering, Trivandrum for leading me through the portals of knowledge. No words can capture the significant influence that all of them have had on me.

I have been blessed with great friendships. I am grateful to my dear friend Shainy for her endearing company all through this research, Vidhya for her occasional long phone conversations always timed to lift me out of one of my low-lows and Rosch for just being there in the shadows. A great bunch of colleagues at work, Ayman, Kam, Ricardo, Amir, Adi, Pete, Zemian, Devraj, Keith, Huiyan, Azura, Imran and Farah. A special thanks to Dan Slipper for helping with my final case study. Hearty thanks to an equally superb group of sub-wardens who have always been around to support and encourage me. I will miss them all – Tim, Denise, Anup, Malcolm, Rowan, Paul, Jenny and Estelle. Special thanks to Colin and Lois, Jim and Wendy, for being super wardens and for going out of the way to make me feel at home. I am indebted to Welfare for having been given the opportunity to serve in pastoral care. I seemed to have received more of it than I gave away. Thanks to Salil and Anjali for always being there and the members of St. Marys SOC, Leicester for taking care of my spiritual needs.

A very special thanks to my lovely parents-in-laws, P. V. Uthuppu and Valsa Uthuppu and the extended family – Roy Varghese, Ambili Roy, Deepa and Sobha for their valuable prayers and sincere wishes that saw me through the write-up of this thesis. Finally, not in the least, a special thanks to my dear Basil, for accommodating the uncertainties of the last few months. Your support was invaluable and I hope to return the favour someday!

This thesis is dedicated to my dear friends who have propped me up at various stages of this project.

- *by offering valuable words of encouragement through the endeavour*
- *and most importantly, for providing a reliable support network during all the trying times*

– AND –

*my dear Basil for being a **very** patient and understanding husband during a difficult period of our life*

List of related publications

A number of papers were published during the course of the work described in this thesis. These are listed below (in reverse chronological order). Please note that the contents of some of these papers have been adapted for presentation in this thesis: where applicable, a footnote at the beginning of a chapter indicates that material from one or more papers has been included. For those papers with contents that have not been included in this thesis, a copy of the paper itself is included in the appendices.

Directly-related publications

Kurian, S. and Pont, M.J. (2007) "Maintenance and evolution of resource-constrained embedded systems created using design patterns", *Journal of Systems and Software*, **80** (1): 32-41.

Kurian, S. and Pont, M.J. (2006) "Evaluating and improving pattern-based software designs for resource-constrained embedded systems". In: C. Guedes Soares & E. Zio (Eds), "Safety and Reliability for Managing Risk: Proceedings of the 15th European Safety and Reliabilty Conference (ESREL 2006), Estoril, Portugal, 18-22 September 2006", Vol. 2, pp.1417-1423. Published by Taylor and Francis, London. ISBN: 0-415-41620-5 (for complete 3-volume set of proceedings). ISBN: 978-0-415-42314-4 (for Volume 2).

Kurian, S. and Pont, M.J. (2005) "Building reliable embedded systems using Abstract Patterns, Patterns, and Pattern Implementation Examples". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), *Proceedings of the Second UK Embedded Forum* (Birmingham, UK, October 2005), pp.36-59. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

Kurian, S. and Pont, M.J. (2005) "Mining for pattern implementation examples". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), *Proceedings of the Second UK Embedded Forum* (Birmingham, UK, October 2005), pp.194-201. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

Pont, M.J., Kurian, S., Maaita, A. and Ong, R. (2005) "Restructuring a pattern language for reliable embedded systems" ESL Technical Report 2005-01.

Associated publications

Pont, M.J., Kurian, S. and Bautista-Quintero, R. (2010) "Meeting Real-Time Constraints Using "Sandwich Delays"", In: J. Noble and R. Johnson (Eds.) TPLOP I, LNCS 5770, pp.94-102. © Springer-Verlag Berlin Heidelberg 2009 [ISBN: 978-3642108310]

Pont, M.J., Kurian, S., Wang, H. and Phatrapornnant, T. (2008) "Selecting an appropriate scheduler for use with time-triggered embedded systems" In: Hvatum, Lise and Schummer, Till (eds.) Proceedings of the 12th European Conference on Pattern Languages of Programs (Irsee, Germany, July 2007). Published by Universitätsverlag Konstanz [ISBN: 978-3-87940-819]

Wang, H., Pont, M.J. and Kurian, S. (2008) "Patterns which help to avoid conflicts over shared resources in time-triggered embedded systems which employ a pre-emptive scheduler" In: Hvatum, Lise and Schummer, Till (eds.) Proceedings of the 12th European Conference on Pattern Languages of Programs (Irsee, Germany, July 2007). Published by Universitätsverlag Konstanz [ISBN: 978-3-87940-819]

Pont, M.J., Kurian, S. and Bautista-Quintero, R. (2007) "Meeting real-time constraints using 'Sandwich Delays'". In: Hvatum, Lise and Zdun, Uwe (eds.) Proceedings of the 11th European Conference on Pattern Languages of Programs (Irsee, Germany, July 2006). Published by Universitätsverlag Konstanz[ISBN: 3-8794-0813-0][1]

Poster publications

Kurian, S. and Pont M. J. (2006) "Application of Design Patterns in the Maintenance of Software for Reliable Embedded Systems". Poster presented at the Science, Engineering and Technology for Britain (SET for Britain), House of Commons, London, December 12th 2006.

Kurian, S. and Pont M. J. (2006) "Application of Design Patterns in the Maintenance of Software for Reliable Embedded Systems". Poster presented at the IET Postgraduate Workshop on Embedded Systems on 11 October 2006 (NEC, Birmingham, in conjunction with the Embedded Systems Show).

Kurian, S. and Pont, M. J. (2006) "Supporting the Maintenance and Evolution of Software for Embedded Systems Using Design Patterns and CASE Tools", Poster presentation at at the Regional Poster Presentation Competition held at the (University of Warwick), July 3rd, 2006. One of the top ten posters representing the University of Leicester at this event.

Kurian, S. and Pont, M. J. (2006) "Supporting the Maintenance and Evolution of Software for Embedded Systems Using Design Patterns and CASE Tools", Poster presentation at "Festival of Postgraduate Research" (University of Leicester), June 13th, 2006. The poster was highly commended.

[1] One of three papers selected for the "Master Track" at this conference.

Table of Contents

List of tables

List of figures

Part A: Introduction

1. Software engineering for embedded systems

1.1. Introduction

Man's eagerness to explore and expand the boundaries of knowledge has inspired him to devise and develop interesting tools and techniques to assist him with this exploration. The new knowledge in turn has furthered these technological advances. Computers or computing as we know it today is the outcome of one such line of exploration spawned by the need to mechanise the thought process. This chapter gives a historical sketch of early computing devices and emphasises the importance of software in making truly powerful and 'smart' machines. It also discusses the ubiquitous nature of computers in today's world and thus introduces the concept of embedded systems. It presents embedded systems as specialised computer systems. The chapter goes on to discuss the concept of software engineering for embedded systems and introduces the reader to a few software development issues which are unique to embedded software. Thus, it sets the stage for the research published in this thesis which primarily focuses on techniques to develop high-reliability embedded software. The concluding sections present an overview of contributions made by this thesis and describe the organisation of the material presented here.

1.2. Calculators and the Analytical Engine

Computers had their origins in mechanical computing devices. The tally stick and abacus were probably the earliest tools used by man for computational purposes. As new mathematical concepts were discovered, more complicated mathematical tools came to be devised and used. Following John Napier's discovery of the use of logarithms in computing (1612), a host of computational devices came to be invented. Napier's bones, Pascal's gear-driven calculating machine - 'Pascalene' and many other such calculating machines and slide-rules were a part of this era. The Thomas Arithmometer, based on Leibniz's stepped-drum principle was demonstrated to the French Academy of Science in 1820 and went on to become the first mass-produced calculator (Williams 1983). These early tools were primarily calculating devices.

Around this time, Charles Babbage began work on the Difference Engine – a machine intended to perform logarithmic and trigonometric computations. By 1834 his focus had shifted to the Analytical Engine, considered by many to be the first modern computer. It had a mechanical design and was to be powered by a steam engine. The design relied on the use of punch cards to provide inputs to the analytical engine which was expected to produce one (a maximum of two) output (Cohen 1988). Lady Augusta Ada Lovelace noted about the Analytical Engine thus:-

> *"Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent." (attributed to Ada Lovelace and quoted by Toole. (Toole 1991))*

The potential of "programming" this machine to perform complex operations such as composing music truly distinguished the Analytical Engine from earlier calculating devices. History credits Lady Ada Augusta as being the first software programmer for her unique grasp of the abstract and her observation that a suitable "program" could elicit a desired behaviour out of the Analytical Engine – essentially distinguishing a computer from a calculator. Fuegi and Francis make the following observation -

> *"She (Lady Augusta Ada Lovelace) became the first person to have crossed the intellectual threshold between conceptualising computing as only for calculation on the one hand, and on the other hand computing as we know it today: with wider applications made possible by symbolic substitution." (Fuegi and Francis 2003)*

It is this power of 'symbolic substitution' that truly characterised the Analytical Engine as the first known computer and by envisioning it as a 'programmable' machine the concept of software came into existence.

1.3. General Purpose Computing

Advances through the Industrial and Information Age resulted in the electronic computer (or desktop PC) as we know it today. In 1936, Alan Turing proposed the creation of a general purpose computer, capable of solving any mathematical problem presented to it in a suitable symbolic form. His reasoning set the stage for the concept of software as we know it today.

These conceptual advances soon led to the development of programmable electronic devices like the Colossus and ENIAC which used vacuum tubes or electronic valves. These were seen by many as the first modern computers. The invention of the transistor and the use of magnetic storage aided technological advances in computing (Burks 2002). The development of compilers and programming languages that supported English-like commands saw a significant growth in computer usage over the next few decades. There were an estimated 240 computers in 1955 and by 1974 that number rose to 165000 (Norberg 1984)

Around the late 1930s and early 1940s designs for programmable calculators came to be implemented. Zuse's Z2 machine was an electrical, digital automatic computing machine which worked in binary. The Atanasoff-Berry Computer was the first electronic computing device and was designed to solve systems of linear equations (Williams 2006).

As early as 1952, Grace Hopper made useful observations regarding the future of computer-aided societies. Credited to be a pioneering computer scientist and early compiler developers (B-0 compiler) (Head 2001), her early observations regarding software and its uses for the future seem matter-of-fact today. She believed that software would eventually be more expensive than hardware. She anticipated the growth of artificial intelligence and foresaw a period when mathematicians would no longer need to know instruction codes of the machines they programmed. She drew parallels between software creation and the production line model of automobile manufacture (Hopper 1988).

Since these early technological developments, computers have become an integral part of modern-day living. Computer science has emerged as a discipline in its own right. Software engineering, like other engineering fields, addresses the issues of processes and quality for developing software.

1.4. Embedded systems and computer systems?

The concept of compilers and process-initiatives to govern creation of quality software were therefore ideas long conceived. Early computers had standard input and output mechanisms and the user – often specially trained to handle the computer – was clearly aware of using a computer. At the heart of a modern computer system is a microprocessor suitably programmed to realise the desired behaviour of the computer.

As computers began to be used more ubiquitously, there was a growing need to hide the complexity of these machines from an ordinary user. In other words, there was a need to 'embed' computers in 'smart' appliances thus creating embedded systems. The "Apollo Guidance Computer" is an example of one such embedded software system that had to be reliable and provide an easy interface to the users (astronauts on the mission) given that the system would be operating in space (Hall 2000).

Today, microcontrollers are embedded in a large number of common household equipment like microwave ovens and washing machines. Embedded systems are also seen in modern cars with sophisticated electronic systems (Mak et al. 2003; Niz and Rajkumar 2003; Furukawa and Kawamura 2006; McCaffery et al. 2008). These embedded computers support automatic features in cars and aircraft systems (Carlow 1984; Damm et al. 1989).

Take for example a modern car. The sophisticated electronics systems support a range of technological advances ranging from in-car entertainment systems, satellite navigation modules, drive-by-wire systems and the like. At the heart of all these systems is a micro-controller (or a set of inter-connected micro-controllers) which needs to be suitably programmed as per the requirements of the embedded application. Thus the embedded hardware and software together make up an embedded system.

In short, general-purpose computing systems like PCs and laptops, embedded systems are designed to be used for specific computing needs. They are further characterised by custom user interfaces, built according to the needs of each specific application. Embedded systems are effectively specialised computer systems which hide the complexity of a computer behind a simple and application-specific user-interface. The fact that the user is unaware of using a computer in most cases makes embedded systems development and maintenance a relatively challenging task. Besides these differences, many embedded systems impose a heavy requirement on reliability.

1.5. Embedded software development

Subtle differences between general purpose desktop computing and application-specific embedded systems give embedded software development and architecture a unique identity independent of usual desktop applications.

Take for instance the embedded software development process. Developing desktop applications is fairly simple in hardware terms. Quite often, the application is developed on a desktop or laptop which is similar to the target hardware (similar to PCs) that executes the application. Embedded systems, however, need to be developed through a cross-development process. The embedded software is developed on a desktop computer and the executable code is transferred onto the target embedded controller. An embedded developer relies on a host of software tools to achieve this (Earnshaw et al. 1997). Debugging the embedded application is not straight-forward either. The lack of a "traditional" user interface (similar to most PCs) necessitates the use of special debugging software capable of analysing the state of the microcontroller and memory through program execution. Eventually techniques to debug embedded systems relies on the availability of a simulation of the microcontroller or some mechanism that supports an analysis of the program while it is executed on the hardware (Hand 1991; Koehnemann and Lindquist 1993).

A cross-development environment, aided by the use of suitable software tools for the different development activities, significantly simplifies the process of creating embedded software. The need to hide the complexity of a computer system and simultaneously ensure smooth operation of the embedded system places a premium on the robustness of the application design. Such functional and related performance requirements influence the architectural considerations to be taken while designing embedded systems (Obermaisser 2004).

The research presented in this thesis focuses on techniques to support the development of high-reliability embedded systems (eg: control electronics in an automobile or aircraft). The ANSI definition of software reliability is the probability of failure-free operation of software for a specified period of time in a specified environment (IEEE 1990). Software failure occurs when it no longer gives the desired result of execution and varies in the levels of their severity. Catastrophic failures can be life-threatening. This is especially true in the case of embedded systems used in safety critical applications, such as anti-lock braking systems and flight control system. High-reliability embedded systems are very sensitive to faults and can endanger the lives of people affected by the embedded system. The critical nature of the software places an additional emphasis on the development process.

The growing need to reduce cycle-times and increase productivity as well as product quality (Wetherbe and Frolick 2000; Clincy 2003) also applies to embedded software development.

The research presented here explores the potential of using design patterns to manage complexity and productivity simultaneously. Design patterns present well-documented solutions to classic design problems and hence provide an effective mechanism to capture domain expertise. By identifying processes and development practices that use design information effectively, practitioners can benefit from the wealth of domain knowledge captured in a pattern collection.

It has been previously argued (Pont 2001; Pont and Banner 2004) that use of appropriate "design patterns" can assist in the creation of reliable embedded systems. Embedded software architectures can either be event-triggered or time-triggered. These competing architectures are discussed in greater detail in Chapter 3. Research in the Embedded Systems Laboratory (ESL) has focussed on identifying design patterns that can be used while implementing time-triggered embedded designs. Research in this area has resulted in the assembly of a collection of more than seventy patterns, most of which are catalogued in the work "Patterns for Time-Triggered Embedded Systems" (Pont 2001): together these patterns will be referred to as the "PTTES collection" in this thesis.

Originally, using design patterns in software development was conceived to be a rather straightforward process. Practitioners who wished to build software systems using design patterns manually referred to the pattern collections and applied them while constructing software. More recent research in this field has explored the use of tools to incorporate design patterns in the process of engineering software. This has resulted in the creation of tools to support design pattern based software development (Budinsky et al. 1996; Florijn et al. 1997; Martin et al. 1997; Peckham and Lloyd 2003).

So, how are design patterns currently used in a tool-driven development process? Pattern documentation is usually in human-readable form (rather informal manner of representing information). What are the challenges of incorporating this "informally documented" information in a more formal development process? The work in this thesis addresses these issues.

1.6. Overview of thesis contributions

This thesis is concerned with embedded systems which employ time-triggered software architecture and for which there are both severe resource constraints and a requirement for

highly-predictable behaviour. The thesis explores the potential benefits of using "design patterns" during the development of such systems.

The thesis makes the following key contributions:

a. It identifies the lack of a standard approach to utilise the wealth of information documented in a design pattern and attempts to understand methods for pattern-based software engineering (PBSE)

b. It recognizes the need to restructure the pattern language to incorporate growing domain knowledge and support better utilization of the pattern information and suggests a new tiered architecture to organize the information in the restructured pattern language

c. It proposes a novel use of pattern relationships to perform design space exploration activities and demonstrates the use of related patterns to obtain design alternatives using suitable case studies

d. It proposes a new aspect of formalizing the enriched, pattern language in order to develop and support standard processes of using the pattern information.

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

1.7. Thesis structure

This thesis is organised as described. Chapter 2 discusses embedded software engineering in detail. It highlights the differences in the process of developing embedded applications and the unique characteristics of software architectures for such systems. It proposes the use of design patterns to manage these complexities.

Chapter 3 introduces the concept of design patterns. It briefly discusses pattern collections for embedded systems and provides a detailed introduction to the Patterns for Time-Triggered Embedded Systems (PTTES) - the pattern collection which is the focus of this research work. The chapter also discusses the process of applying the PTTES patterns when building an embedded application. The remaining work presented in this thesis is based around the workflow detailed through this process.

Chapter 4 encourages the use of patterns beyond mere code generation. It describes cost-intensive software engineering practices that stand to benefit from the effective use of domain expertise captured in a pattern collection. It describes existing tool support for the PTTES language and discusses the need to extend this support through the design phase.

Chapter 5 describes early research in this project aimed at restructuring the pattern language to support appending information to the design patterns. It discusses the need to restructure the PTTES language in order to use the patterns more effectively in a tool-assisted pattern-based development method.

Chapter 6 details early experiments in this research project that aim to understand the potential of design patterns with regards to generating multiple designs. It describes a design evaluation process that can be used to obtain alternate designs from an original design. It describes a simple cruise-control system and illustrates the behaviour of alternative systems obtained by making suitable pattern replacements to the original design.

Chapter 7 discusses the need to formalise pattern languages in order to use them in a formal tool-assisted development process. It describes approaches to formalising pattern languages and suggests a technique to formalise the PTTES collection. It illustrates the use of BNF to formalise the PTTES language.

Chapter 8 presents a case study where the suggested approach is used to derive embedded systems and explores the potential of the chosen formal representation.

Chapter 9 presents a detailed discussion of the work presented in this thesis. It sets the stage for a discussion on the scope for future work in this project.

Chapter 10 is a concluding chapter which summarises the arguments and discussions presented thus far. It also discusses the potential for future work in the area of tool support for pattern-based software engineering.

1.8. Conclusion

This chapter traced the historical developments that led to modern computing as we know it today. It emphasised the importance of software and ‘programming’ to elicit desired

behaviour from the hardware constituting a computer. It also introduced embedded systems as specialised computer system that attempt to hide the complexity of a computer behind an application-specific design. It discussed the challenges that this requirement places on the process of developing embedded software as also the importance of architectural considerations to be made while designing such systems. The next section presents the literature reviewed to understand the problem addressed by the work presented here. Chapter 2 describes and compares two important architectural frameworks against which software systems are designed. Chapter 3 introduces the concept of design pattern collections as repositories of domain-specific information. It emphasises the origins of patterns as mechanisms to manage design complexity and describes a pattern collection aimed at easing the design and creation of predictable embedded software systems. Finally Chapter 4 describes existing tool support for pattern-based software development and looks at practices that can benefit from the availability of domain information readily through mechanisms like pattern documents.

Part B: Literature Review

2. Software architectures for embedded systems

2.1. Introduction

Chapter 1 introduced the concept of an embedded system. This chapter discusses embedded software development in detail. It describes the process of developing embedded software and also architectures for the same, based on the observation that embedded software development techniques are truly distinct from those for desktop applications.

Unlike general-purpose computing systems like PCs and laptops, embedded systems are designed to be used for specific computing needs. They are further characterised by custom user interfaces, built according to the needs of each specific application. The process of developing software for embedded systems differs significantly from that of developing general desktop applications. For instance, high reliability embedded systems, as discussed in the earlier chapter, have stringent real-time requirements. A real-time system is one in which the correctness of the system lies both in the logical correctness of the system as well as the requirements to meet task deadlines (Stankovic and Ramamritham 1989; Kopetz 2000). Also, embedded systems have resource-constraints (memory and processing power). Thus designers need to incorporate constraint requirements characteristic to the real-time and embedded applications in addition to the functional requirements of the system (Kopetz 2000; Graf et al. 2006).

The functional requirements of an embedded system can be decomposed into a small number of independent software entities, also called tasks. This task set provides a suitable software abstraction of the embedded system being designed (Barr 1999). The required system behaviour is accomplished by using scheduling mechanism to execute these tasks in a certain order. However, the design of the tasks and schedulers are affected by the constraints on a real-time embedded system.

Section 2.2 discusses some of the important constraints that affect the design of tasks in an

embedded system. Section 2.3 presents two different frameworks used when designing the architecture of real-time software systems – the time-triggered approach and event-triggered approach. Section 2.4 presents current research involving scheduling strategies suitable for each of the two paradigms. A discussion comparing the two architectural paradigms and the scheduling strategies suitable for each follows in Section 2.5. This section also identifies the main drawbacks of the preferred framework for designing reliable embedded systems and indicates the scope of the research presented here, within this premise.

2.2. Designing tasks for embedded systems

The tasks designed to meet the functional requirements of an embedded system need to conform to one or more constraints on the system. Buttazo (1997) identifies these constraints as one of three kinds: timing (or temporal) constraints, precedence constraints and resource constraints. This section gives a brief insight into each of these constraints and defines the important parameters associated with design and description of tasks in an embedded systems

2.2.1 Timing constraints

Real-time embedded systems need to conform to stringent timing requirements. Table 2.1 lists the timing parameters used to specify these requirements. Some of the important timing parameters used by researchers (Buttazo 1997; Torngren 1998) to characterise an embedded task are depicted in Figure 2.1.

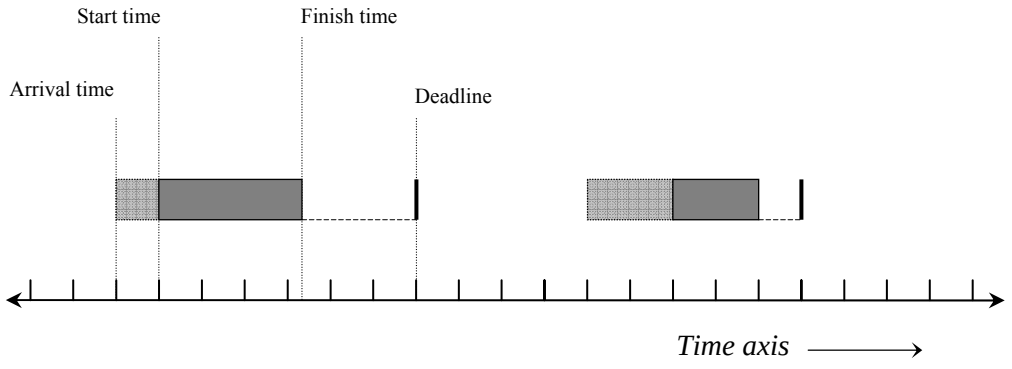


Figure 2.1: Timeline showing important times associated with an embedded task. Adapted from (Buttazo 1997)

Arrival time	The time at which the task becomes ready for execution, and is also called the request time or release time.
Start time	The time at which a task begins execution
Finish time	The time at which a task completes execution
Computation time	The time required by the processor to complete execution without being interrupted.
Worst Case Execution Time	Also known as WCET, it is the longest time that any particular instance of a task takes to complete execution (Wu and Yao, 2004)
Deadline	The time before which a system is expected to complete execution in order to avoid damage to the system. The parameter associated with the consequence of missing a deadline is termed **criticalness** (ie, hard or soft). Thus, a task can have a hard deadline or soft deadline. In case of a hard deadline, the task has to compulsorily complete execution by the deadline. If the task has a soft deadline, it has a preferred time of completion. The system has some value if the task execution is delayed beyond its soft deadline (which is not the case with a hard deadline).
Lateness	Represents the delay in completion of the task with respect to its deadline. Thus Lateness = finish time - deadline. A task that completes before its deadline has a negative value of lateness.
Tardiness	Tardiness (or exceeding time) is the time a task stays active after its deadline. Therefore Tardiness = Maximum(0, Lateness)
Laxity	Also known as slack time is the maximum time a task can be delayed on its activation prior to missing its deadline. Therefore Laxity = Deadline – Arrival time – Computation time.
Value	Represents the relative importance of the task with respect to other tasks in the system.

Table 2.1: Timing parameters associated with tasks in a real-time system

Tasks can be further differentiated on the basis of the nature of their arrival characteristics. Task arrivals may be **periodic** or **aperiodic**. Periodic tasks have a constant time interval between invocations. Figure 2.2 shows the schematic representation of a periodic task.

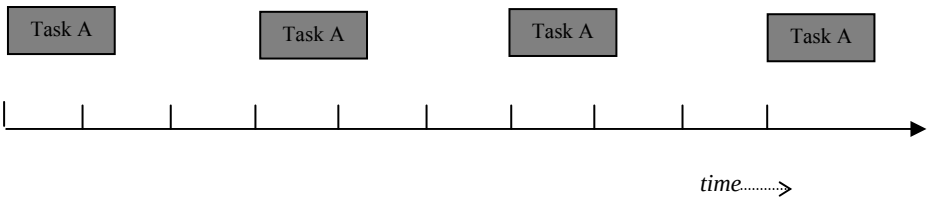


Figure 2.2: Periodic task execution

Some tasks are characterised by a statistical distribution of their inter-arrival time which is taken as the random variable. Sometimes it is possible to predict the arrival of a task within a scheduling window of say *λ time units,* though the task itself is not periodic. The scheduling window for a task is the time available from the moment it begins execution to completion so that execution is completed before the task's deadline. Figure 2.3 shows the execution schematics. Please note that the time interval marked 'λ', includes these task executions.

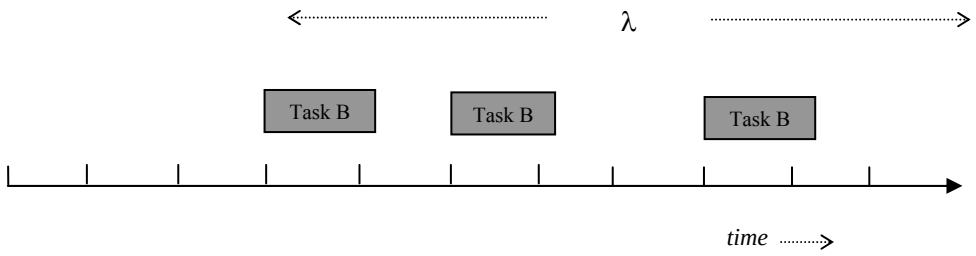


Figure 2.3: Predictable but non-periodic task execution

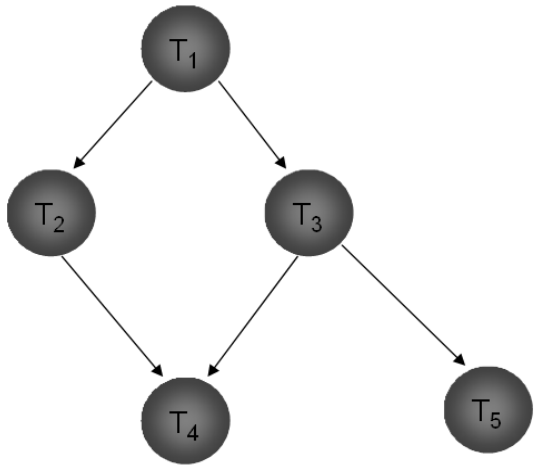
Some aperiodic tasks are completely unpredictable (Kopetz 1991). Since it is impossible to predict the arrival time of the next instance of such tasks they can be scheduled as soon as they arrive and hence the deadline co-insides with the time taken to complete execution of the task. This results in the task having a scheduling window of λ = nil. Non-periodic tasks with hard deadlines are called **sporadic** tasks.

2.2.2 Precedence constraints

In some applications, computational activities cannot be performed in any arbitrary order. For instance Torngren (1998) attempts to model top-level functions of a control application by decomposing them into elementary functions. An elementary function constitutes a

transformation of input data to output data. This is achieved by a simple sequential model which involves - read input channels, perform computations and then write the computed data to output channel(s).

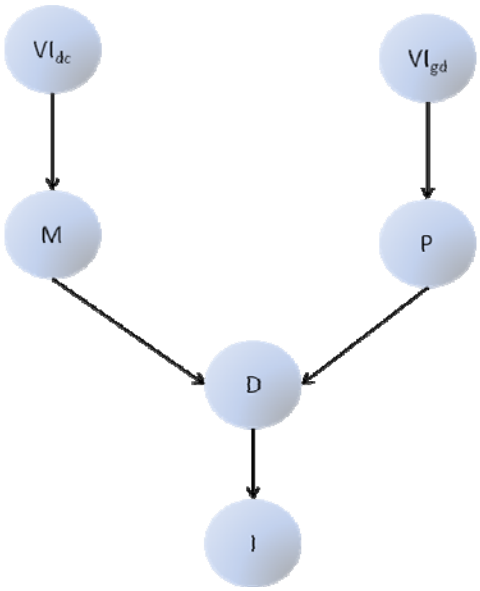
Precedence relations are usually depicted through the use of acyclic graphs (Buttazo 1997). Task designs should ensure that precedence requirements are satisfied when scheduled using a suitable scheduling strategy. Also, the temporal behaviour of any task is affected by those of the preceding and dependent tasks. For instance Figure 2.4 describes one such precedence relationship on an arbitrary set of five tasks.



- Task T_1 (beginning task) is has no predecessors and is executed first
- On completion of T_1, T_2 or T_3 starts execution.
- Task T_4 waits for the completion of both tasks T_2 and T_3
- Task T_5 begins execution only after T_4 completes execution

Figure 2.4: Task precedence captured using precedence graphs. Adapted from (Buttazo 1997)

Consider the design of a grid-connected inverter. An inverter converts DC power to an AC power. In order to do this, the DC voltage and current is sampled. An equivalent AC current and voltage is computed. This is accomplished using a suitable algorithm which computes maximum power-point. A PWM mechanism is used to obtain the AC power output. When connected to the local power-grid, the inverter control software needs to ensure that the AC power is in-phase with the grid. In order to accomplish this, the AC voltage and current on the grid is sampled. The phase of the grid power is computed using a Phase Lock Loop (PLL) and a current controller is designed to keep the output AC power in phase with the grid power. A precedence graph which captures this application logic is presented in Figure 2.5.



VI_{dc} – Sample DC power
VI_{gd} – Sample grid power
M – Maximum power-point algorithm
P – Compute phase of grid power
D –Current controller
I –Actuate inverter

Figure 2.5: Precedence in the tasks of a grid controlled inverter

2.2.3 Resource constraints

Some resources (data structures, variables, memory, peripherals, etc.) in an embedded system tend to be shared amongst tasks in the system. To maintain consistency, tasks should be prevented from simultaneously accessing shared resources. Exclusive resources require mutual exclusion amongst competing tasks. The code fragment having mutual exclusion constraints is called a critical section. Sequential access to exclusive resources is ensured through the use of synchronisation mechanisms like semaphores or locks. When using a 'lock' mechanism, the shared resource is first checked to see if it is already in use by other tasks that tend to use the resource. If this is the case, the resource will be locked for access by the task currently using it. Once the lock is relinquished, a new task will first need to lock the resource prior to using it. When use of the resource is complete, it is the responsibility of this task to release the resource for use by other tasks in the system. Instructions to set a lock and release a lock need to be atomic to ensure the proper functioning of the lock mechanism.

2.3. Architectural considerations

Tasks in a real-time embedded system are designed against suitable architectural frameworks to satisfy the constraints imposed by real-time embedded systems. Two different paradigms are currently used while architecting real-time embedded systems – event-triggered framework and time-triggered architecture. Comparative studies describing each of the

frameworks and the benefits and draw-backs of each have been documented by Kopetz and Obermaisser. Scheler and Schröder- Preikschat state that the choice of architecture is a non-functional requirement of a real-time system and hence suggest a framework-independent technique to specify the design of a real-time system. This enables the designer to focus on obtaining a design independent of the framework and postponing the decision of choosing a framework to the later stages of software development (Kopetz 1991; Obermaisser 2004; Scheler and Schröder-Preikschat 2006).

Changes in the state of a real-time entity generate events in the system. A trigger is a control signal that initiates an action (or task) in the embedded computer system (Kopetz 1994). For event-triggered systems, this control signal is obtained from events in the real-time entity. Events indicate a change in the state of a system. These events are generated from either the embedded computer system (for example, task termination) or from the environment in which the embedded computer system operates (like, data obtained from some sensor component). Similarly, a time-trigger is a control signal generated at a particular instant of time and obtained from a global synchronised clock. Since an instant of time is identified by a change in the state of a global clock, these triggers are a subset of event-triggers in real-time embedded systems (Obermaisser 2004).

Section 2.3.1 describes the event-triggered paradigm. In this framework, the software is designed to respond to a set of events generated by the embedded system. The other paradigm relies on the use of only the timer events in software design. In this framework, the system is only sensitive to a subset of all possible events that can be generated in the system. Hence the tasks designed for a time-triggered system need to detect the occurrence of other events in the system. Section 2.3.2 briefly describes the time-triggered architectural framework.

2.3.1 Event-triggered architecture

System activities in a purely event-triggered system are initiated by the occurrence of events in a real-time entity or object. The behaviour of the system is realised by tasks designed to respond to the events being handled. The task design directly follows the desired system behaviour. Any additional design features incorporated into the task is needed to manage inter-task interaction (when one task needs to be pre-empted to service another event). The signalling of an event occurrence is usually achieved by using an interrupt mechanism. The

occurrence of events may either be predictable (statistically or otherwise). However occurrence of some events cannot be predicted deterministically (referred to by Kopetz (1991) as chance events).

Event-triggered systems usually use buffers to manage flow control. Events are buffered prior to servicing. Occurrence of a large number of unpredictable events in a short span of time may require some events to be discarded to restrict the flow of events. When a system employs an event-triggered communication strategy, only the sender is aware of a message transmission at any instant of time. Error detection is thus based on time-outs of acknowledgements returned to the sender. Explicit congestion control techniques need to be employed to handle the occurrence of co-incident events.

The system can be made more tolerant to faults by replicating critical component. In the case of software this is accomplished by executing critical software components in parallel. This approach is called active replication. If all the replicas begin executing with the same initial-state, they enter the same states and produce identical outputs for a given input sequence, i.e. they behave deterministically. This capability of the replicas implemented as part of a redundancy mechanism is termed replica-determinism.

The lack of *a priori* knowledge of the run-time behaviour of an event-triggered system necessitates the use of dynamic scheduling strategies. State synchronisation is not guaranteed when using asynchronous event-triggered systems having dynamic non-pre-emptive scheduling strategies. Fault-tolerance is therefore implemented using techniques like the leader-follower method. In this approach redundant components execute the same code but one of the replicated copies is designated as the leader. The decisions affecting replica-determinism are propagated from the leader to the followers through synchronised messages (Barrett et al. 1990).

2.3.2 Time-triggered architecture

Time-triggered architectural designs rely on the occurrence of a single timed event to initiate the activities in a system (Locke 1992; Kopetz 1995; Maier et al. 2002; Obermaisser 2004; Pont and Banner 2004). This event is usually a periodic timer overflow. The period of time marked by the timer interrupt is referred to as a tick interval. An important design consideration to be made when developing such systems involves the duration of the tick

interval (Pont et al. 2003).

Since tasks are no longer executed as a response to a general event in the system, their design within a time-triggered framework must incorporate some mechanism to detect (the non-timer) events. This is usually achieved by polling the event sources periodically (Pont 2001). Subsequently, the time period between two consequent task calls is another important design consideration. A large time-period can affect the responsive of the system being designed. Similarly, frequent calls to a task polling an event-source can load the system and unnecessarily utilise processor time (Pont 2001).

Flow control in a time-triggered system is implicitly achieved during system design. By choosing appropriate rates at which a controlled object is observed and serviced (if necessary) the receiver is synchronised with the sender. The communication controller for a time-triggered system uses the concept of a state-machine (Kopetz 1998). The state of a node is periodically written to a state message over-writing the older message. A receiver may or may not read this state value, but any read guarantees that the latest state is obtained. Communication errors are detected by the receiver when an anticipated message (based on the expected arrival time against the global clock) is not received.

Scheduling strategies for time-triggered systems are based on static pre-determined schedules (Locke 1992). The schedules need to incorporate all task dependencies to provide the implicit synchronisation required at run-time. Time-triggered systems are expected to offer better state-synchronisation because they are implemented around a synchronised global clock (Obermaisser 2004).

By designing systems to respond to a single timed-event, time-triggered architectures offer temporal predictability (Kopetz 1998). This predictability is useful when using replicated software for fault-tolerance. For example, the Triple-Redundancy approach involves parallel execution of three different versions of the redundant software component. The output of the parallel execution is presented to a ‘voter’ software component for analysis and action. Fault-free operation would mean that all three parallel lines of execution produce the same outputs for further action by the voter. Hence, as far as redundancy-based fault-tolerance implementation is concerned, replica determinism is more a consequence of the design rather than a feature that needs to be implemented or considered.

The task designs in the time-triggered are not a straight-forward implementation of the desired behaviour of the system, instead, they need to be aware of other tasks in the system and detect the occurrence of non-timer events in the system. In spite of the design challenges involved in building systems that respond to only timer events, this architecture is used considerably in the design of hard real-time systems (Kopetz 1995; Maier et al. 2002; Pont and Banner 2004).

2.4. Scheduler design

The tasks in an embedded system are executed to elicit the desired behaviour of the system. A scheduling policy decides the order in which the tasks are executed in the system. There are various scheduling strategies that focus on computer systems with hard real-time requirements (Liu and Layland 1973; Baker and Shaw 1988; Xu and Parnas 1991; Audsley et al. 1995) and research also includes studies comparing their perceived strengths and inadequacies (Locke 1992; Audsley et al. 1993; Buttazzo 2005). The scheduling strategies differ in the manner in which schedules/task queues are constructed as well as the manner in which priorities are assigned to tasks in the system. This section briefly describes the main scheduling strategies adopted by developers of real-time embedded systems.

2.4.1 Building the schedule

The tasks to be executed in the system are held in a ready queue also called a schedule. The schedule can either be built prior to system execution (the pre-runtime approach) or during system execution (runtime approach). This section describes these two scheduling approaches.

2.4.1.1 Pre-run time scheduling algorithms

The pre-run time scheduling approach, static scheduling approach or off-line scheduling approach as it is variously referred to, involves developing a complete task schedule (order in which tasks execute) at compile time. The availability of a pre-run time schedule enables the designer to analyse the schedule and ascertain that the implemented system meets required constraints (Xu and Parnas 1991).

The cyclic executive described by Baker and Shaw (1988), is an example of a static scheduling strategy. Baker and Shaw (1998, p120) define cyclic executives thus:

"A cyclic executive is a control structure or program for explicitly interleaving the execution of several periodic processes on a single CPU; the interleaving time is done in a deterministic fashion so that execution time is predictable."

A cyclic schedule is used to define the process of interleaving such that all periodic tasks execute within their deadlines. The schedule is obtained by first identifying a major schedule, which describes the set of tasks to be executed over a fixed period of time called the major cycle. The length of the major cycle is equal to the least common multiple of the periods of all the tasks being scheduled. Each major cycle is further subdivide into a minor cycle with a corresponding minor schedule. Each minor schedule or frame is a list of processes that need to be executed during the period allocated to that frame. One fundamental design requirement of a cyclic schedule is that no frame should be longer than the shortest task period required by the tasks in the system. The cyclic executive finds extensive use in the design of safety-critical embedded systems used in the automobile and avionics sector (Carlow 1984; Damm et al. 1989).

Though the cyclic schedule presents a simple application model for scheduling tasks, implementing systems on a cyclic scheduler can be challenging (Locke 1992). The approach is considered to be rigid and inflexible. An input/output task needs to be executed faster than its period to ensure no significant data-loss. Implementing tasks to handle sporadic events on a polled basis can be extremely expensive (Bates 1998). Sha and Goodenough (Sha and Goodenough 1988) argue that practitioners sacrifice program structure to fit tasks in the correct slot while attempting to implement schedulable and responsive systems using a cyclic schedule. Maintaining cyclic executives also involves re-analyzing the schedule for every change made; in order to ensure that timing requirements are met by the new schedule.

However, systems using an off-line scheduling approach incorporate a certain degree of determinism because their schedules are known prior to the actual execution of the system. Since the pre-runtime schedule is usually maintained as a table of procedure calls this scheduling paradigm is also referred to as the **static table driven approach** (Ramamritham and Stankovic 1994) to scheduling.

2.4.1.2 Runtime scheduling algorithms

The pre-runtime scheduling strategies described previously indicate task execution in the near future. For this reason they are also referred to as clairvoyant algorithms. As opposed to

offline scheduling mechanisms, online strategies involve creating and maintaining a schedule during runtime (Baruah et al. 1992). Hence these approaches are also referred to as runtime scheduling algorithms.

In an online scheduling approach, resources need to be allocated to tasks with no prior knowledge of the arrival characteristics of other tasks in the future. The objective of the scheduling strategy is to employ a dynamic decision-making strategy to construct a schedule in real-time (Porter 2004) so that maximum number of tasks meet their deadlines.

Online strategies use priority-based techniques to create and maintain the schedule dynamically. The tasks can be assigned priorities statically or dynamically. Section 2.4.2.1 describes fixed-priority approaches to scheduling tasks. Two approaches to dynamically determine the task priorities are described in Section 2.4.2.2. They are the earliest-deadline first approach and the least laxity first approach. Both approaches evaluate the deadlines of the task set prior to assigning priorities.

2.4.2 Managing task priorities

Scheduling policies are also affected by the manner in which task priorities are assigned. A scheduling event may be timed to occur periodically (as in time-triggered system) or may occur due to other factors (the occurrence of an event, completion of task execution, the availability of a task in the ready queue, to name a few). Tasks may be assigned fixed-priorities prior to the execution of the system or assigned priorities dynamically as and when a scheduling event occurs. In either case, the dispatcher, responsible for executing a task at any instant of time uses this priority information to execute the task with highest priority.

2.4.2.1 Fixed-priority approaches

Audsley and colleagues (Audsley et al. 1995) give a historical perspective of fixed priority schedulers or rate-monotonic schedulers as they were called by Liu and Layland (1973). The application model involves use of a pre-emptible, fixed-priority executive to execute periodic tasks ordered monotonically (i.e. the most frequent task is assigned greatest priority) (Locke 1992). The task set is deemed schedulable as long as it satisfies the condition on the utilisation factor which is related to the number of tasks to be scheduled. This utilisation factor is defined in equation 2.1)

$$\sum C_i/T_i \quad < \quad n(2^{1/n} - 1) \qquad \text{equation 2.1}$$

where

C_i = computation time of the task i
T_i = total processor time available for task i
n = total number of tasks scheduled to be processed

Research in the area has shown that as long as the total utilisation of the processor is below 85% (Lehoczky and Sha 1986), all tasks in a randomly generated task set will meet their deadline when scheduled using the rate-monotonic approach. A variant of the rate-monotonic approach is the deadline-monotonic approach. In the deadline-monotonic approach tasks with the shortest deadlines are given the highest priority (Leung and Whitehead 1982). This scheduling strategy is especially relevant in cases when the task deadline is earlier than the period assigned to the execution of the task.

The rate-monotonic approach (or static priority-driven approach (Ramamritham and Stankovic 1994)) has many advantages over cyclic executives. The structures of the application tasks more accurately reflect the application requirements since tasks no longer need to be broken up to meet frame length limitations. Tasks can be executed at their natural period of activation, unlike tasks with cyclic executives which have harmonic frequencies based on the duration of the minor cycle. The ability to predict the schedulability of the system against total processor utilisation (as opposed to frame length in cyclic executives) is seen by many (Sha and Goodenough 1988; Audsley et al. 1993) as the biggest advantage of the rate-monotonic approach.

2.4.2.2 Dynamic-priority approaches

A scheduling policy using the dynamic priority approach supports a mechanism of assigning task priorities while the tasks are being scheduled for execution. The scheduler executes the task with the highest priority. Examples of dynamic priority driven approaches include the Earliest Deadline First (EDF) scheduler and the Least Laxity First (LLF) approach to scheduling.

In the EDF approach, any task set ready to be scheduled for execution is assigned priorities based on the deadline of each task. The task with the earliest deadline is assigned highest priority and hence executed first. Priority assignment takes place at fixed time instances –

when a task has completed execution, or there is a need to run a dispatch routine based on some scheduler event in the system (Liu and Layland 1973; Buttazzo 2005). If there are no tasks to be dispatched the processor enters idle mode. Liu and Layland (1973) prove that the earliest deadline first approach is only feasible provided –

$$C_1/T_1 + C_2/T_2 + \ldots\ldots + C_n/T_n < 1 \qquad \text{equation 2.2}$$

where

C_i = computation time of the task i
T_i = total processor time available for task i
n = total number of tasks scheduled to be processed

Another approach to dynamically assigning priorities to a task-set is the Least Laxity First approach (LLF). Also known as slack time is the maximum time a task can be delayed on its activation prior to missing its deadline (see Table 2.1). Therefore –

$$\text{Laxity} = \text{Deadline} - \text{Arrival time} - \text{Computation time} \qquad \text{equation 2.3}$$

Also referred to as the Least Slack Time First (LSTF) approach to priority-assignment, in this approach the task that needs to be executed most immediately is assigned the highest priority (Oh and Yang 1998; Zhang et al. 2007). The dispatch mechanism, like before, executes tasks in the order of their priority, with the highest priority task (having least slack time) executed first.

2.5. Discussion

Early safety-critical systems relied on the cyclic executive (Baker and Shaw, 1988) to provide a robust scheduling technique for hard real-time systems. Cyclic executives support pre-runtime scheduling. Since the execution schedule is pre-determined, cyclic schedules are deterministic (Bates 1998). A deterministic, compile-time schedule, which ensures that none of the individual tasks are interrupted, lacks support for pre-emption. The absence of unplanned context switches keeps overheads low. The determinism provided by this scheduling algorithm minimises jitter and enables tasks that can afford very less jitter to be appropriately re-factored over the major cycle (Locke 1992).

Proponents of priority based scheduling strategies argue that cyclic executives are fragile, inflexible and difficult to maintain (Locke 1992; Audsley et al. 1995). Any small changes to

an existing system would require the new schedule to be re-evaluated to ensure that timing guarantees are met whilst frame overruns are avoided. Tasks with an execution time greater than the period of the most frequent cyclic task (long tasks) need to be suitably split into a set of smaller tasks which can be fit into multiple frames. Splitting tasks across frames introduces new issues like resource sharing and consistency (Locke 1992). Another common cause for concern is that cyclic executives require the frequencies of the scheduled tasks to share a harmonic. This requirement introduces new costs in the system when tasks are called more frequently than necessary. In contrast a priority based scheduling algorithm is considered to be predictable (based on the utilisation factor of the task set). It does not require tasks to use the harmonic frequency relationships among periodic tasks. It is also argued that the structure of tasks in a rate-monotonic system reflects the application requirements more accurately (Locke 1992).

Based on these arguments it would seem that cyclic executives as scheduling strategies are better replaced by priority-based scheduling strategies. However an analysis of scheduling strategies from an architectural perspective provides an entirely different insight. For instance, of the two fundamentally different paradigms employed in designing the architecture of real-time system, the time-triggered approach is favoured for safety critical systems (Kopetz 1994; Rushby 2001). Event-triggered communication systems are considered to be flexible. Proponents also argue that these systems have better resource utilisation and are hence cost-effective. However, time-triggered embedded systems, by nature of their design, are known to be predictable, deterministic and composable. It is also easier to implement replica determinism in time-triggered systems (Kopetz 1991; Obermaisser 2004; Scheler and Schröder-Preikschat 2006). Embedded systems developed around the time-triggered framework rely on static scheduling strategies like the cyclic executive described by Baker and Shaw (1988). Fixed-priority executives may be predictable (Bates 1998), but they are not necessarily deterministic and hence prone to jitter (Locke 1992). Though task priorities are statically assigned, scheduling strategies that support pre-emption have dynamic schedules. In fact, Baruah and his colleague (Baruah and Goossens 2003) describe the rate monotonic scheduling algorithm as a "very popular runtime scheduling algorithm". Section 2.4.1.2 described the stringent restrictions made by Liu and Layland while arriving at the utilisation factor as a measure of predictability. However some of these assumptions (all tasks need to be periodic, task deadlines need to be equal to their period) have been considered impractical (Locke 1992).

Xu and Parnas (1991) state that the perceived disadvantages of a static scheduling approach are of secondary importance when the primary objective of using the strategy is to satisfy time constraints. In this regard, they state that for satisfying the timing requirements of hard real-time systems, pre-runtime scheduling (or static scheduling) is often the only practical means of providing predictability in complex systems. They also argue that, when employing a run-time scheduling approach:

> *".... no matter how clever the scheduling algorithm is, there is always a possibility that a newly arrived process possesses characteristics that will make that process either miss its deadline, or cause other processes to miss their deadlines. This is true even if the processor capacity was sufficient for the task at hand" ((Xu and Parnas 1991), pg 134)*

Locke (1992) observes that, the disadvantages of using static scheduling algorithms like the cyclic executive increase life-cycle costs. The need to constantly re-evaluate the schedule while incorporating any changes coupled with the need to manually fragment long tasks to fit the minor cycle were perceived to be the hidden costs of using a static scheduling strategy.

Given all these considerations, a practitioner can find the task of designing and implementing an embedded system quite challenging. Though extensive research projects have explored the benefits and drawbacks of the various approaches, this vast knowledge can be difficult to manage on a project time-scale. Though the embedded application being developed is comparatively small compared to some of the common desktop applications in everyday use, the practitioners involved in realising such system have to grapple with enormous design complexity. Graaf et al, (2006) observe that many companies used methods, tools and technologies of general software engineering processes while developing embedded software. They identify a gap in the availability of domain-specific tools which cater to the specific requirements of an embedded systems developer. -.

In spite of the differences in the nature of desktop applications and embedded systems, the underlying problems of design complexity can possibly benefit from the use of design patterns – a concept well used in the design and development of general-purpose computing systems. Besides the complexity of the process employed in developing embedded systems, design of such system relies on the available of highly skilled professionals for the same. Design patterns provide an effective mechanism to capture domain-specific information. Design patterns gained a footing in desktop software development, when they were adopted

as a means to manage the growing design complexity of software systems.

The focus of this project is on the use of one such pattern collection aimed at easing the process of designing time-triggered embedded systems. It understands the use of these patterns in the process of embedded software development and attempts to specify the nature of existing tool support. It then proceeds to identify the existence of standard practices or the possibility of these in order to define a formal process of applying these design patterns to a software development project. By specifying a process through which these design patterns can be used in software development, the project attempts to understand the underlying possibilities for eventually designing tools that can be used with standard methods of using these patterns. Providing tool support for the effective application of these design patterns will hopefully address some of the issues surrounding life-cycle costs involved in using static scheduling strategies and the time-triggered framework for developing real-time embedded systems.

2.6. Conclusion

This chapter discussed the complexities involved in designing high-reliability embedded systems. It described the event-triggered and time-triggered architectural frameworks. It described the relevance of time-triggered designs for obtaining predictable and deterministic system behaviour. However, designing tasks around a time-triggered framework can be quite challenging and expensive. Best practices that address classic design issues in this problem space need to be used effectively to avoid re-inventing the wheel. The concept of design patterns originated from a need to capture domain expertise. The next chapter discusses the concept of patterns in greater detail and introduces a pattern collection intended to be used in the development of embedded systems with a time-triggered architecture.

3. Design patterns: From buildings to embedded systems

3.1. Introduction

Chapter 2 described the challenges involved in developing embedded software. It suggested the use of design patterns to deal with the associated complexities of the development process as also the architectural characteristics. This chapter describes design patterns in greater detail. It provides a historical insight to the origins of this concept and its relevance in the software industry. It discusses the use of design patterns in embedded software development and describes the PTTES collection in detail.

3.2. Alexander and his architectural design patterns

Design patterns find their origins in the works of Christopher Alexander who was a qualified architect with undergraduate degrees in mathematics and architecture. Alexander's doctoral research (Alexander 1964) in architecture attempted to introduce formal mathematical methods into architecture – an early implementation of computer-aided design.

3.2.1 Handling complex architectural designs

The construction boom that followed World War II saw architects and builders grappling with large and complex designs for town plans and buildings (Gartman 2004; Lange 2006). There was a growing need to obtain successful designs for these new structures in the quickest possible manner. Alexander, a British mathematician educated at Cambridge University addressed this problem through his doctoral research in architecture at Harvard University. Using his background in mathematics Alexander developed a computer program which attempted to analyse and suggest new environments based on some logical programmatic analysis. The requirements of the project were captured through detailed diagrams and a computer program processed the inter-relationship between these design elements to concur on a suitable design for the project (Alexander 1964). This research introduced mathematics into architectural design methodologies and was well received by

contemporary architects. In 1972, the American Institute of Architects (AIA) awarded him a Gold Medal for this research – the first in the series awarded by the institute (Kohn 2002).

3.2.2 'Collective experience' for aesthetic design

Alexander's doctoral research focussed on the use of computers to generate new environments from project specifications. However his interests in fashioning new environments did not end there. Alexander was not entirely satisfied with using formal methods to develop new designs (Kohn 2002). He was in search of a methodology that produced beautiful designs – designs that created "living" structures. This desire to understand why certain places worked spatially and psychologically, resulted in his theory of design patterns (Alexander et al. 1975; Alexander et al. 1977; Alexander 1979). Through this theory he acknowledged the importance of collective experience in building beautiful structures and blended this concept with the application of formal logic to architecture.

3.2.3 Architectural design patterns

Alexander introduced design patterns as a means of documenting time-tested solutions to common technical problems that architects and builders encountered in their profession. He argued that most common design problems were repetitive in nature. Once documented design patterns could be referred to while tackling these common design problems. Two of his important works – 'A Pattern Language: Towns, Buildings and Construction' (Alexander et al. 1977) and 'The Timeless Way of Building' (Alexander 1979) were the first pattern catalogues. *A Pattern Language* is by far his most famous work.

Though Alexander did not try and give any formal definition to design patterns, a very common understanding of design patterns is stated as follows –

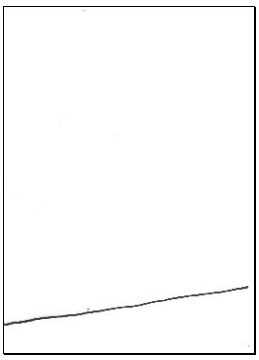
> *"A pattern is a three-part rule that expresses a certain relationship between a certain context, a problem, and a solution" (Alexander 1979).*

The only emphasis indicated through the statement being that design pattern documented a problem, defined the problem within a context and proposed a solution to this problem within the context described in the documentation. The context of the problem is documented as part of the pattern and is best used to ascertain if a design pattern is applicable in a particular situation.

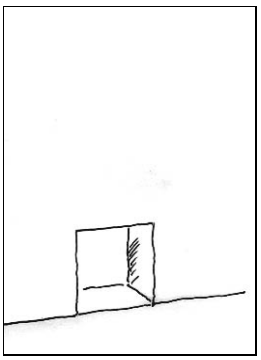
3.2.4 Practical application of architectural design patterns

Design patterns were intended to be used in a very straightforward manner. An architect involved in design – be it town planning or building construction, would ideally refer to the pattern collection to identify the patterns of interest in a project. She was then expected to adapt the suggested solution in a suitable manner taking into consideration all of the design constraints and options available. Alexander documented 253 such patterns in *A Pattern Language* (Alexander et al. 1977; Alexander 1979).

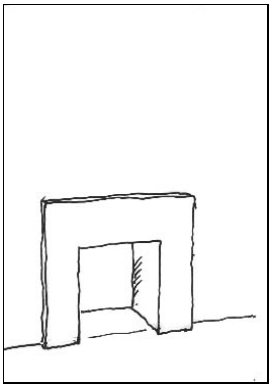
Each design obtained by using design patterns effectively used documented domain expertise that was captured in a suitable format that contained the pattern description. Figure 3.1 illustrates this concept using one such architectural pattern that can be referred to while building a fireplace. The architect was expected to follow through steps 1 to 6 in a recipe-like manner to build or design a fireplace. As indicated by the process illustrated by Figure 3.1, design patterns effectively directed the practitioner wishing to implement/design an architectural component.



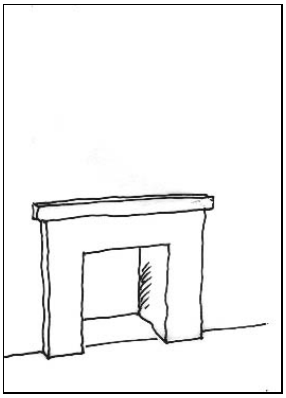
Existing Context



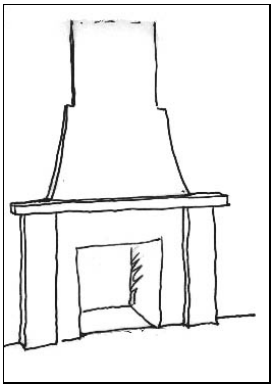
Step 1:
Define inner rectangle



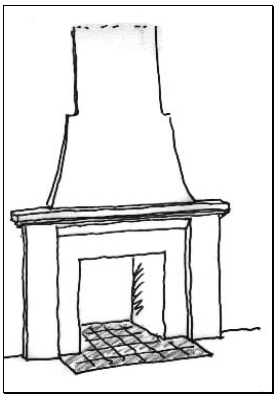
Step 2:
Define main rectangle



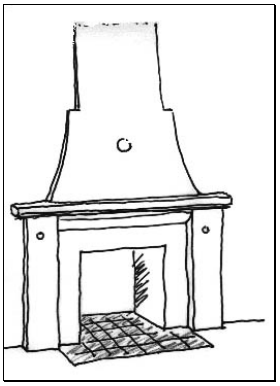
Step 3:
Get the height of mantel shelf



Step 4:
Add the periphery



Step 5:
Get colour and material



Step 6:
Add finishing touches

Photograph of the fireplace

Figure 3.1: Building a fireplace (webpage: A Pattern Language 2001)

3.2.5 Design individuality

A pattern intentionally captures the core of the solution to a classic design problem. The process of adapting/applying the pattern supports customisation at the various steps leading to the final implementation. Provisions to make design choices at each step ensure that implementations differ based on the chosen design options made for each project. Alexander emphasised the importance of uniqueness when he wrote that –

> *"Each pattern 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 could use this solution a million times over without doing it the same way twice" (Alexander et al. 1977)*

Following through the steps in depicted in the earlier figure (Figure 3.1), it is easy to understand the possibility of obtaining unique fireplaces by incorporating subtle differences in the choice of say colour/material or size of the various dimensions suggested through the solution. Thus the use of design patterns was expected to support design and implementation individuality and was not expected to rigorously enforce "the solution", but to support a solution relevant to the specifics of the problem at hand.

3.2.6 A Pattern Language – communicating through pattern names

Alexander also believed that by documenting design patterns, one could effectively build a more technical vocabulary. He argued that design patterns formed the elements of a pattern language (Alexander et al. 1977). Pattern names were intended to concisely capture the ideas behind the problem and solution being addressed. Consequently practitioners were expected to be effectively using the pattern language to communicate design ideas without having to go into the details of the design problem being discussed. For e.g.: By referring to a pattern like 'Street Window', the architects or builders could easily understand the design problems involved in constructing a window facing the street. They could also analyse the different design options available based on the actual situation of having to apply this design pattern.

3.2.7 Critical review of Alexander's works

Alexander's concepts has faced much criticism by contemporary architects. His critics have labelled him as utopian, and a man of contradictions. He has been accused of being rigid, messianic and a reactionary[2] (Alexander and Eisenman 1983; Teyssot 1983; Saunders 2002).

Alexander emphasises the "timeless" way of building by basing architectural designs on living structures that have existed over centuries. Architects who disagree with this theory of his accuse him of making assumptions that new ideas can never match up to ideas that evolved over centuries. Besides, Alexander uses design patterns to achieve certain goals – comfort, legibility, ease, sociability, peacefulness and pleasure to name a few. Peter Eisenman criticises Alexander and labels these goals as rather bourgeois and believes that design prescriptions encourage complacency, passivity and parochialism (Alexander and Eisenman 1983). Finally in a field where buildings speak of professional achievement, Alexander is seen as a person who produces more words rather than buildings (Kohn 2002).

One possible explanation of this conflict of interest between Alexander and his critics probably arises from the fact that architects identify themselves better as puzzle-makers than problem-solvers (Archea 1985). Unlike most other disciplines that involve a building process (example: space planners, engineers, programmers) who are problem-solvers, architects are puzzle-makers. Where problem-solvers state the desired effects as explicit performance criteria before initiating the decision process, architects or puzzle-makers supposedly do not seek explicit information before designing. Instead they see the design process as a means of understanding what they want to accomplish and how to realise the design. It is this particular act of designing that John Archea refers to as puzzle making (Archea 1985).

Though Alexander's work with design patterns probably did not impress a lot of his contemporary architect colleagues, his pattern language has proved useful and popular amongst other architecture practitioners. A website (webpage: A Pattern Language 2001) dedicated to his works on architectural design patterns is used heavily by laypeople designing their own houses and builders and contractors. This practice might also be influenced by Alexander's beliefs that best architecture is not art, and is produced by ordinary people trying to make a good life (Saunders 2002).

---

[2] Alexander and Eisenman 1983, is a reference to a transcript of the legendary debate which took place at the Graduate School of Design, Harvard University, on November 17th 1982

3.2.8 Appeal of design patterns

The architectural community were not very enthusiastic about design patterns; however this concept of design reuse gained a lot of attention in many other disciplines. Today pattern collections have been compiled for diverse disciplines including security patterns (Hafiz 2005), telecommunication patterns (Adams et al. 2001; Meszaros 2001), concurrency and networking designs (Buschmann et al. 1996), embedded systems (Pont and Banner 2004; Bellebia and Douin 2006), patterns for collaborative application designs (Guerrero and Fuller 1999; Tsai et al. 2005; Schummer and Lukosch 2006; Louren and Cunha 2007), patterns for testing software (Tsai et al. 2005; Louren and Cunha 2007), business process patterns (Ramachandran et al. 2006; Germain and Robillard 2008), pedagogical patterns (Fincher and Utting 2002; Carle et al. 2007) to name a few. The rest of this chapter provides a historical account of the emergence of design patterns in software engineering and describes a pattern collection for embedded software development in greater detail.

3.3. Software design patterns

Software practitioners were quick to associate with the primary issue that design patterns were aimed to address, i.e. an elegant approach to handle complex design issues. Maintaining pattern catalogues was seen to be a very effective way of capturing domain expertise. The following sections describe the growth of pattern usage in building software.

3.3.1 Potential benefits of designing with patterns

Design patterns were primarily intended to provide a suitable mechanism for capturing domain expertise to support design reuse. The early 90s saw a prolific growth in software. Design patterns were first introduced into software community by Cunningham and Beck in 1987 (Cunningham and Beck 1987) and envisioned to have tremendous potential to manage the growing complexity of software designs. They were seen to provide a mechanism to capture best practices and avoid re-inventing the wheel. (Cline 1996; Geyer-Schulz and Hahsler 2002). In addition to this, pattern information was intended to be captured in a manner that enhanced the technical vocabulary of practitioners of the domain for which the pattern collection was intended to be used. Alexander's observation that design patterns constituted the elements of a pattern language made the relationship between design patterns and communication rather explicit (Alexander, 1977).

Advocates of pattern usage argue that design patterns have many benefits. Design patterns are seen to support effective design re-use (Cline 1996). When used reactively they are considered useful to capture domain expertise. The extensive documentation is especially useful to new designers who are yet to gain familiarity with the system. Design patterns used proactively can result in robust designs after a careful analysis of trade-offs in using certain design elements. These are just some of the benefits discussed by researchers involved with studies evaluating the use of design patterns in software development activities (Schmidt 1995; Beck et al. 1996; Agerbo and Cornils 1998; Prechelt et al. 2001; Vokac 2004; Ampatzoglou and Chatzigeorgiou 2007).

3.3.2 Software design pattern collections

Alexander's concepts were initially extended to software engineering when Ward Cunningham and Kent Beck introduced the first software pattern collection (Cunningham and Beck 1987). The collection consisted of five design patterns which were intended to be used by Smalltalk programmers. The Patterns were intended to enhance the technical vocabulary of developers and designiers wishing to create graphical user interfaces using the programming language – Smalltalk.

Erich Gamma and his colleagues compiled one of the most common pattern collections used today. Their pattern collection focused on the design of object-oriented software. This pattern collection has 23 patterns that are primarily used in building object-oriented desktop applications. These patterns are catalogued in Design Patterns: Elements of Reusable Object-Oriented Software (Gamma et al. 1995).

Naturally, each software pattern collection focussed on a particular domain. Pattern catalogues comprise of well-documented body of literature that can be referred to and utilised when solving design problems. The wealth of information is presented using a suitable structure or pattern format. Most pattern documentation formats are slight variations of the Alexandrian form (Coplien 1998).

One such pattern collection, intended to be used by embedded developers is the "Patterns for Time-Triggered Embedded Systems" (Pont 2001). The patterns in this catalogue form the focus of the research documented in this thesis. Section 3.8 discusses this collection in greater detail.

3.4. Pattern collections and pattern languages

The terms ‘pattern collections’ and ‘pattern languages’ are often used interchangeably. When a set of patterns are referred to as a pattern collection there is often an indication of a reference to their collective identity. Patterns also have an individual identity. A set of patterns are often related to each other. The use of Pattern-A may affect (or maybe affected by) the use of Pattern-B. Pattern names and also other elements of a pattern’s documentation (such as information regarding related patterns, etc.) enrich the practitioner’s vocabulary by lending domain expertise effortlessly to a user. By lending their individual characteristics to enrich natural language, patterns create pattern languages which are semantically richer than pattern collections (Zimmer 1995). To achieve this, pattern documentation is deliberately maintained in a human readable form by the use of natural language

3.5. Building pattern catalogues

Experienced software architects and designers engage in significant discussion as they identify best practises in tackling certain common design patterns. The process of examining source code to identify design patterns is traditionally referred to as ‘Pattern Mining’. Pattern authors identify classic design problems and solutions and then document these as design patterns.

The pattern languages are critically reviewed by fellow authors at Pattern Languages of Programming (PLoP) events. This process is referred to as shepherding. The shepherding process begins when a paper is initially submitted to a PLoP conference (Harrison 2006). The review process is more intensive and face-to-face during the conference. These reviews take place as *Writers Workshops* (Coplien and Woolf 2000). This feedback allows the participants to improve their patterns to make them more useful or more publishable.

More recently, the patterns community have begun inviting patterns-based publications for the new patterns journal. The LNCS Transactions on Pattern Languages of Programming is a journal that accepts reviews, survey articles, criticisms of patterns and pattern languages and similar research papers. The journal aims to present validated material by relying on the shepherding process for an initial review of the papers submitted for publication. The papers are expected to be reviewed by domain experts and pattern experts prior to their publication.

3.6. Perceived inadequacies

Design patterns have their shortfalls too. The primary concerns regarding the effectiveness of using design patterns stems from the fact that patterns are not standard. Different authors write patterns differently and the success of a pattern is heavily dependent on the name of the pattern and the nature of the information captured in the documentation (Schmidt 1995; Cline 1996). Pattern documentation is held in a human readable format. Pattern information should be sufficiently detailed to be useful. If there is too much detail the core of the solution can be lost in the documentation. Similarly, keeping a pattern small might compromise the quality of the information available and will necessitate the use of other information sources to understand a design problem (Agerbo and Cornils 1998; Vokac 2004).

Effective use of patterns is only possible if the whole team is sufficiently familiar with the pattern collection of interest (Schmidt 1995; Cline 1996; Unger and Tichy 2000). This is important when patterns are used to enhance the vocabulary of practitioners and technically intensive communications need to be supported. At times it might not be necessary to use a pattern if the solution is obvious. Marshall Cline (1995) observes that patterns can be over-hyped and this is probably why Ekstrom (2000) questions if patterns should always be considered. Unger and Walter (2000) discuss this case further in their study and conclude that in some situations using design patterns can be useful, harmful or neutral depending on the circumstance of use.

Finally most pattern collections tend to be associated with a programming language – C, C++ or Java. This fact can sometimes lead to confusion when evaluating the need to use a particular pattern collection. None of the disadvantages discussed in this section however, directly relate to the primary motivation of using a design pattern – the ability to use a time-tested solution to solve increasing complex design problems.

3.7. Patterns and embedded software development

Chapter 2 described the two different approaches to designing embedded software – the event-triggered design and the time-triggered design. As discussed in Section 1.5 the embedded systems domain is very different from desktop software systems. Embedded systems are usually characterized by resource constraints. They are heterogeneous and unlike traditional software platforms which are largely standardised, embedded systems run on

specific hardware with special-purpose operating systems, programming languages network protocols, etc. All of this makes embedded systems development rather challenging.

Embedded systems are usually built and tested in simulated environments. During production, the hardware and software can often be produced together. This further complicates the process of embedded software development. In addition to this, the embedded systems used in safety-critical applications, require a certain degree of dynamic adaptability during execution. A classic example is that of the Apollo Guidance Computer (AGC) – one of the earliest embedded systems. The actual AGC system was expected to work in space and out of reach of the developers of the system. More over there was not enough time to provide extensive training to the astronauts using the system (Hall 2000). Embedded design patterns try and address such issues faced by designers/developers. The patterns attempt to capture domain specific information for the benefit of practitioners involved in the creation of embedded systems. This section introduces three such pattern collections.

Mark Bottomley documents a set of patterns to build a simple embedded system (Bottomley, 1999). This research defines the framework for a simple embedded system - *The Carousel*. All other patterns can be used in conjunction with this framework to build the simple embedded system. The Carousel is in fact the super loop architecture which is very famously used by designers of embedded system to build simple embedded applications devoid of the need of any operating system or complex control software. Bottomley (1999) presents his work as answers to a set of common design questions that designers encounter while building simple design systems.

The next pattern collection is even more informal. Event Helix is a privately held corporation based in Maryland USA. The company's primary interests are in the development of tools that aid the design and development of real-time embedded software amongst other domains. Event Helix, provides for a collection of embedded design patterns and an embedded systems pattern catalogue on their website (webpage: EventHelix.com Inc. 2008). The rather informal approach to documenting these design patterns, makes it difficult to analyse their use in the industry or in research.

The PTTES collection contains more than 70 design patterns that can be used to build high reliability embedded systems. The focus of the design patterns documented in this collection

lies on building embedded systems using time-triggered software architectures. These design patterns are suitable for safety-critical systems. The time-triggered approach is considered to be inflexible and such systems are considered to have relatively static run-time behaviour. The focus of this collection is in designing tasks and schedulers that overcome these two major drawbacks while at the same time guaranteeing high-reliability in the software being built.

3.8. The patterns in PTTES

The domain in this case is high-reliability embedded systems. The pattern authors (Pont 2001; Ong and Pont 2002) use time-triggered architectures to ensure that the embedded software is highly reliable and suitable for use in embedded systems like automotive control software. When using the time-triggered approach, embedded systems are designed so that a single timer interrupt is enabled. The timer is set to overflow periodically. The interrupt generated creates tick intervals in which tasks of the system are executed as required by the application design. One of important reasons for using a single interrupt is to increase the predictability of the system. (Section 2.5).

Each pattern document includes a section detailing the reliability implications of using the pattern. To incorporate this information, the patterns are documented in a suitable format which is not very different from the Alexandrian format (Coplien 1998). The next section details out the pattern layout used in PTTES.

3.9. Understanding the PTTES collection

The PTTES collection contains more than 70 design patterns that can be used to build high reliability embedded systems. Early research at the Embedded Systems Lab resulted in the creation of a pattern language documented in the book – Patterns for Time-Triggered Embedded Systems (PTTES) (Pont 2001). Subsequent research has added patterns to this collection which contains 70 or more design patterns today (Pont 2001; Pont and Ong 2003; Melwa and Pont 2004; Pont and Banner 2004; Pont et al. 2007; Pont et al. 2008; Wang et al. 2008). The focus of the design patterns documented in this collection lies on building embedded systems using time-triggered software architectures.

3.10. The PTTES language

The patterns of the PTTES collection can be categorised based on the nature of the problem and solution documented by each. Patterns associated with a select set of classical embedded software architectural problems are listed below:

1. Scheduler designs
2. User Interface designs
3. Condition monitoring design problems
4. Control algorithm designs
5. Communication protocol implementation designs
6. Timeout mechanisms

3.10.1 The pattern format

The patterns in PTTES are recorded using a modified Alexandrian form. Information regarding each design pattern is noted under different sections of the pattern documentation. Each pattern of the PTTES catalogue has information organised in the following layout –

- Pattern Name - The name of the pattern. Each pattern name is an element in the pattern language
- Context – This section describes the context of the problem whose design solution is documented
- Problem – The statement and description of the problem for which the design pattern is documented
- Background – Some additional information which is required to understand the solution for the problem
- Solution – The core of the solution to the problem being discussed is presented here
- Related patterns and alternative solutions – This section contains information regarding related patterns that maybe affected by the application of this pattern or alternative patterns that can be used for similar problems
- Reliability and safety issues – This is an important section that gives reliability information specific to the pattern. Such information is particularly useful considering the fact that the pattern collection is targeted for use in building high-reliability embedded systems
- Overall strengths and weaknesses – The designer is informed of the strengths and weaknesses of using a particular pattern. This section is especially important when

considering alternative solutions etc.

- Examples – This section has example source code, illustrating the use of a design pattern in a particular design problem. Developers can suitably modify this code when using a design pattern in their designs
- Further reading – This section has information that can be referred to while using the design pattern

3.10.2 Characteristics of the catalogued information

Though much of the documentation focuses on using the design patterns for the 8051 family of micro-controllers these patterns have been successfully used to build similar systems on other processor families. A part of the research reported here involves re-structuring the pattern collection (Chapter 5). One of the primary motivations behind this restructuring is to better design information related to new micro-controllers into the current pattern documentation.

3.11. Discussion

Earlier sections in this chapter introduced the concept of design patterns and discussed the PTTES collection in great detail. In order to understand the scope and motivation behind the work presented in this thesis it is vital to understand the traditional approach to designing an embedded system using the patterns in PTTES. The workflow presented as part of this discussion lays the foundation for the development approaches and restructuring strategies presented in this report.

As such, it is (sometimes implicitly) assumed that a developer will browse a catalogue, choose appropriate design patterns and – possibly using some code examples or hardware schematics as a starting point – assemble a system.

To understand how the PTTES collection is used in software development let us consider the example of creating a simple embedded application which displays the voltage applied at a port-pin. The system developer requires a suitable hardware framework to implement this system. In addition to the controller hardware, she also requires ADC hardware to convert the analogue voltage to the appropriate digital value to be used in further processing. Finally the developer also requires suitable hardware to connect the micro-controller to the PC and

transfer the digital voltage value to be displayed suitably on the PC. A developer using the PTTES collection would ideally do the following:-

i. She would first need to setup the basic hardware framework on which the software can be loaded. For this, she would need to construct an oscillator circuit by referring to CRYSTAL OSCILLATOR. The oscillation cycles provide an instruction level timeline to execute machine code.

ii. In addition to incorporating a suitable oscillator (12 MHz crystal oscillator as an example in this case), the developer proceeds to build a reset circuit in order to implement a hardware reset mechanism (RC RESET).

iii. With this basic hardware framework in place, she proceeds to build a software foundation, conceptually similar to a simple operating system. The scheduler in a time-triggered system can be viewed as a simple embedded operating system because it has the responsibility of dispatching the tasks in system. The scheduler achieves this by creating periodic ticks and executing tasks in the appropriate tick intervals. The PTTES collection describes design strategies for single and multi-processor systems as also the associated merits and concerns related to the use of each of these strategies. Let us assume that the developer/designer chooses to use a CO-OPERATIVE SCHEDULER (also known as the TTC SCHEDULER in the restructured language) in this design. Most pattern collections (including PTTES) detail examples in their documentation. The developer suitably modifies these examples prior to including the pattern in their design. For example using the CO-OPERATIVE SCHEDULER example would require suitable, application-specific modifications to the tick interval and timer used

iv. Once the basic scheduling mechanism is in place, the developer would ideally proceed with testing the timing/scheduling mechanism prior to adding the tasks that define the behaviour of the system. This can be visualised by periodically flashing an LED ON and OFF (based on the HEARTBEAT LED pattern). The example associated with the HEARTBEAT LED pattern requires minor modification such as assigning an available port pin, as also the frequency at which the LED needs to flash ON and OFF.

v. With the hardware and the basic software framework in place, the next stage is to design and implement tasks needed to realise the system behaviour. To display the voltage on the PC the practitioner only needs to use the PC LINK (RS-232) pattern. The pattern describes techniques and challenges that need to be considered while using a serial port and cable. As before the example code provides a suitable starting

point to incorporate the pattern into the projects. The example described as part of this pattern uses an RS-232 connection to display elapsed time on a HyperTerminal. Adapting the example for this project requires a few elaborate changes to the actual characters displayed on the HyperTerminal. However, much of the initialisation code and the task design strategy remains the same and hence the developer truly benefits from using the modified example code as opposed to implementing the task from scratch.

vi. The SEQUENTIAL ADC pattern describes techniques to interface an ADC chip with an 8051 micro-controller. By making application-specific modifications to the example code (and retaining most of the initialisation routines and over all task design structure) associated with this pattern the developer can effectively implement a simple ADC task which periodically obtains the digital value of a signal (voltage) applied to one of the port pins of the 8051.

vii. Much of the process up until now benefited from modifying and using the examples associated with each pattern. The logic (source code) to link the task(s) created from the SEQUENTIAL ADC and PC LINK (RS-232) patterns is user generated. In this case it involves initialising a global value to be used by these tasks. While the task derived from the SEQUENTIAL ADC pattern provides the most recent voltage value, the user-defined task interprets this digital value as equivalent to a certain voltage. It also incorporates logic to convert the numerical voltage value into suitable ASCII values prior to buffering it for use by the task derived from the PC LINK (RS-232) task, which finally transmits these characters to be written on the PC.

This example illustrated the process of using the PTTES patterns for developing a simple embedded system. Though steps i-vii, detail out a specific embedded application (displaying the voltage at a port-pin), it provides an insight to the process of using the PTTES patterns. An outline of the process can be summarised as follows:-

1. Establish/setup the hardware platform (microcontroller and oscillator/reset circuits)
2. Program a scheduler onto the hardware
3. To check the rudimentary hardware/software setup, attempt to design a simple task to periodically flash an LED on an output pin
4. Once the timing characteristics of the system is established proceed with designing the other tasks in the embedded system
5. Add these tasks to the system in a step-wise manner

Practitioners are encouraged to use the examples provided with each pattern. By adapting a

code example to the particular problem at hand, the need to start coding from scratch can be avoided. The practice of adapting a pattern example or a template to obtain pattern-based code is not unique to this collection. In fact, Gamma and colleagues acknowledge this process of adapting the solution provided as part of a design pattern to the specific problem at hand and refer to it as crystallisation (Gamma et al. 1995; Baudry et al. 2003).

3.12. Conclusion

This chapter described the concept of design patterns in detail. It traced the origins of patterns (in the brick and mortar industry) as a suitable mechanism to capture domain expertise. It discussed the adoption, by the software engineering community, of patterns to manage the growing complexity of software design issues. It introduced the PTTES collections as a set of design patterns intended to be used while building embedded systems with a time-triggered architectural framework. The chapter ended with a discussion of the process of applying the PTTES patterns. It illustrated the manual approach to incorporating patterns while building a simple embedded application. The next chapter describes tools designed to assist with the use of patterns in software development. It describes existing tool support for the PTTES collection and describes software development practices that benefit from the use of rich domain-specific design information captured in a pattern collection.

4. Exploring the boundaries of PBSE

4.1. Introduction

Chapter 3 introduced the concept of design patterns. It described the PTTES collection, a set of patterns intended to be used for developing embedded systems with a time-triggered architecture (see Section 2.3.2). It also illustrated the process of manually including the patterns in a software development project. This chapter begins with a brief introduction to other software engineering approaches that emphasize the need for software re-use. It proceeds to discuss the importance of design patterns in this research project. It describes existing research focussed on providing tool support for the application of patterns in software development (Budinsky et al. 1996; Florijn et al. 1997; Cinneide 2000; Sherif et al. 2000; Andy 2003; Bulka 2003; Peckham and Lloyd 2003). Though the object-oriented design patterns form the focus of these research activities, this chapter also describes existing tool support (ie an automatic code generator) for the PTTES collection (Mwelwa et al. 2007). The latter half of the chapter discusses the scope for extending this tool support beyond mere code generation. It introduces the concept of design space exploration. To put the motivations of the research presented here into perspective, it describes software development strategies that rely on the availability of multiple designs for the same set of requirements. This chapter concludes with a discussion on pattern usage in software development with a particular emphasis on their use in pattern-based tools.

4.2. Accomplishing software reuse

Earlier chapters (especially Chapter 2) discussed some of the difficult design and development issues relevant to software development in general and embedded software in particular. On the one hand, the prolific creation and use of software indicates a creation of domain and design knowledge. On the other hand, the need to manage this growth of software necessitates the adoption of new techniques and methods that effectively use this vast domain and design knowledge. Software reuse involves the use of existing software to

build new software. In this manner, practitioners try to avoid 're-inventing the wheel' when encountering similar design problems in system implementation (Krueger 1992).

The process of re-use can be as simple and ad-hoc as copying code segments from an old project into a new one. More sophisticated practices involve the creation and use of code libraries. A code library usually consists of a set of pre-compiled functions. These functions are included in a new project as and when required. When coding with this perspective, the developer instinctively distinguishes between application logic and the other software elements (library code) around which this application logic is built. In many ways there is an attempt to 'componentize' software.

Component based software engineering (CBSE) is one such paradigm that uses prefabricated software components to create new software (Griss 1994; Ning et al. 1994; Pour 1998). The software component itself is viewed as a black-box, implemented against strict interface rules. In this manner, it is possible to create new software using these commercial, off-the-shelf software products as the building blocks of the desired system. Components are deployed independently and intended to be used for composing third-party software. Thus, there is also a need to suitable encapsulate their implementation (black box implementations). These components are used in the perspective of a framework (Cai et al. 2000; Lucrédio et al. 2003). Hence frameworks provide a skeletal application structure that can be suitable customized to the specifics of an application and its requirements. Examples of common standardized component technologies include CORBA (Common Object Request Broker Architecture), COM (Common Object Model) & DCOM (Distributed COM), JavaBeans and Enterprise JavaBeans (Lucrédio et al. 2003). For instance, the Object Management Group (OMG) defines the standards for CORBA so that components written in different languages can communicate with each other. In order to do these CORBA standards include specifications of an Interface Definition Language (IDL) which objects can use when exposing themselves to the outside world (webpage: CORBA 1997). Also, CORBA standards specify mappings from IDL to common object implementation languages such as C++ and Java.

All of the frameworks specified previously focus on supporting the use of 'black-box' components and well-defined interfaces to reuse source code. Software re-use does not have to be restricted to mere code reuse. Other artefacts created during the process of software development also carry valuable domain-specific information that can be effectively re-used

with a concerted effort. For example, requirements documents and design documents are used to understand and design the behaviour of the system being developed. The ever-increasing focus on adopting mature, repeatable processes in software development implies that organizations invest a lot of their time into documenting design and processes with an aim to improve the repeatability of the software engineering process to ensure the creation of a mature software product. These documents often capture information in a highly abstract (where source code is the most concrete realisation of a design), human-readable format and just like source-code can be vitally employed to re-use design ideas. However, corporate software artefacts (like requirement/design documents) are often subjected to patent/copyright and privacy rules and this is where design patterns come of relevance. Design patterns attempt to capture this domain-specific, design knowledge in a human-readable form and publish this knowledge so that it is accessible by all. The research project described in this thesis attempted to understand the current process of using design patterns for design re-use. An understanding of the current process was seen to aid the process of exploring techniques that can further enhance the design pattern usage in a software development project.

Capturing information in a human-readable format is very useful when disseminating knowledge. However, using this information in a tool-assisted, development process is not that straight-forward. Software components and code libraries lie at the same level of abstraction as source code. With well-defined interfaces, re-using them to develop new source code is easier when tool support is considered. Unfortunately the same cannot be said for design patterns. The next few sections describe the process of using design patterns in tool-assisted software development and support a discussion on current practices and the scope for further research in this area.

4.3. Tools for creating software

Computer Aided Software Engineering (CASE) recognises the benefits of using tools in the software development phase - to increase productivity, improve product quality and ease maintenance tasks (Low and Leenanuraksa 1999). Tool support is used to automate the analysis, design, development and maintenance of software. The most evident benefits of using tools are reduced development times and a decrease in human errors while creating or maintaining software (Pressman 2001; Johnson and Wilkinson 2003).

Mwelwa (2006) cites compilers as one of the earliest examples of CASE tools being used by software practitioners. He is supported by John Backus (the co-inventor of FORTRAN), who observes that the compiler arose out of a need to rectify the imbalance in economics as programming and debugging costs exceeded those of running the program ((Mwelwa 2006), pp.42). The design and development of compilers (for languages like FORTRAN, Pascal and Basic), meant that a software program could be written in a high-level language and compilers and assemblers could subsequently be used to generate the corresponding binary code. Thus the use of compilers increased software quality by replacing an error-prone and tedious process of coding in (seemingly cryptic) assembly language by supporting the use of more readable high-level programming languages.

Soon Integrated Development Environments (IDEs) with custom editors and tool-chains defined the face of modern CASE (Boekhoudt 2003). This growing acceptance of tool-assisted software development encouraged patterns researchers to explore techniques that enhanced the traditionally manual approach of pattern usage with suitable tools (Bulka 2003; Peckham and Lloyd 2003). Research has explored strategies for creating software from design patterns (Coplien 1995; Budinsky et al. 1996; Florijn et al. 1997; Martin et al. 1997; Sherif et al. 2000) as well as use of design patterns in software maintenance phases (Cinneide 2000). This section provides a brief insight into these research activities.

4.4. Tools for pattern-based software development

The creation of object-oriented patterns by Gamma and colleagues (Gamma et al. 1995) was soon followed by research into tool design to support the application of these patterns in software development. Primarily intended to be automatic code generators, some tools also incorporated mechanisms to validate pattern usage. Two such research projects discussed in this section, give an insight to the approach and motivations behind supporting tools for pattern-based software development.

4.4.1 Code generation at IBM Research Labs

Budinsky and colleagues (Budinsky et al. 1996) pioneered the earliest automatic code generator for the object-oriented patterns. Their tool incorporated a mechanism to query the user for application-specific names and design trade-offs prior to using that information to create class definitions and declarations to implement a pattern. The tool also provided a

hyper-text rendition of the book *Design Patterns* for further reference if needed.

The tool interface displays sections of the pattern text (Intent, Motivation, etc.) in different tabs of a single window (as separate pages). In addition to the pattern documentation, this window is further augmented with a code-generation page. Some parts of the code-generation page contain pattern-specific information while others are common to all code-generation pages. For any single pattern, this page can be used to "Generate Declarations", "Choose implementation trade-offs", "Generate Implementations", and "Choose generate options" or instantiate "An Example". By choosing "An Example" as a project goal, the user can have all the project-specific information automatically populated to have an example of the pattern created. The tool architecture is depicted in Figure 4.1.

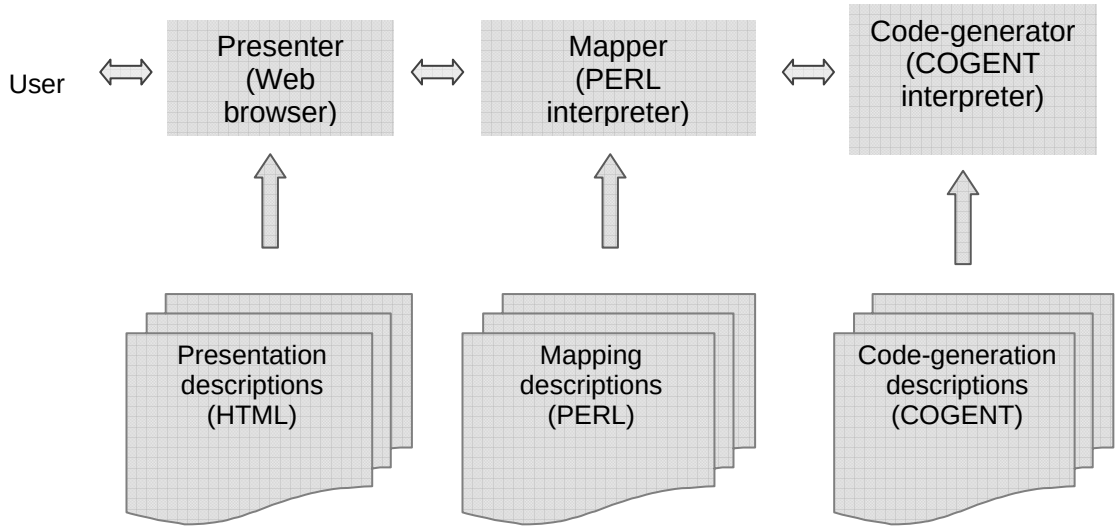


Figure 4.1: Overview of IBM's automatic code generator for OO patterns. Adapted from (Budinsky et al. 1996)

The tool itself consists of the following modules:

- a presenter/browser unit
- a mapper unit
- a code-generation unit

The presenter implements a user interface specified by presentation descriptions usually available in HTML. The code-generation unit is responsible for creating the code implementing a pattern. It comprises of a COGENT (COde GENeration Template) interpreter which interprets code generation descriptions and the parameters specified by the user to generate the application specific source code from selected patterns. The COGENT

interpreter is based on a code generation specification language developed at IBM. An intermediate layer – the mapper unit, relies on mapping descriptions relating the user interface elements to the code to be generated. The mapper unit is implemented using Perl. A practitioner wishing to use the tool would choose the patterns of interest. The wizard would guide them through a process of customising the design solution for the needs of their application. The source code created by the code generator would then be integrated into the working project.

The Eclipse Project founded by IBM to support an open development platform, aids the use of a standard framework to deploy plug-ins offering various functionality. The plug-ins constitute the engine of a complex Integrated Development Environment (IDE). By concentrating on providing a standard user interface and the capability of expanding the engine of their IDE through the use of various plug-ins, the Eclipse Foundation is slowly gaining ground as a Universal IDE. Thus teams interested in experimenting with developing software tools can fully utilize the framework provided by Eclipse and concentrate on the engine of their tool. Commercial plug-ins (such as CodePro™ and PatternBox™) that provide support for automatic code generation from patterns also rely on generating source code templates using an Eclipse front-end. This source code can be integrated into working projects using the same IDE.

4.4.2 The Utrecht University project

The code generated using the approach detailed in Section 4.4.1 was expected to be copied and pasted into the actual project that utilised the pattern(s). Although, this approach provided support for generating code from design patterns, the process of incorporating the generated code was considered error-prone and inadequate by researchers at Utrecht University. To address this inadequacy, Florijn and colleagues (Florijn et al. 1997) from Utrecht University, chose to implement a tool which also enabled automatic creation of the user code needed to complete a pattern-based software application. Their tool provided three different views.

Users were encouraged to add patterns to a project by working with a ‘patterns view’. This view presented a highly abstract picture of the actual software project. Finer details of the application were made available through a ‘design view’. Using the design view, a user

could manipulate the classes and methods of an application in progress. Finally, the user was also presented with a 'code view'. Using this view, it was possible to add code details to the methods and classes to further customise the solution as desired.

The tool architecture was based on a framework of fragments where classes, methods, associations and other such design elements constitute the fragments in a system. Fragments have roles that contain references to other fragments. The program being developed was visualized as a graph of inter-related fragments of different types. The solution documented in the pattern was essentially represented using a generic design structure. Suitable mappings from the design elements (fragments) to the programming language formed the corner stone of the automatic code generator. In its simplest form, the mappings converted the design structure into skeletal code. Figure 4.2 depicts the environment for automatically generating code from the object-oriented patterns.

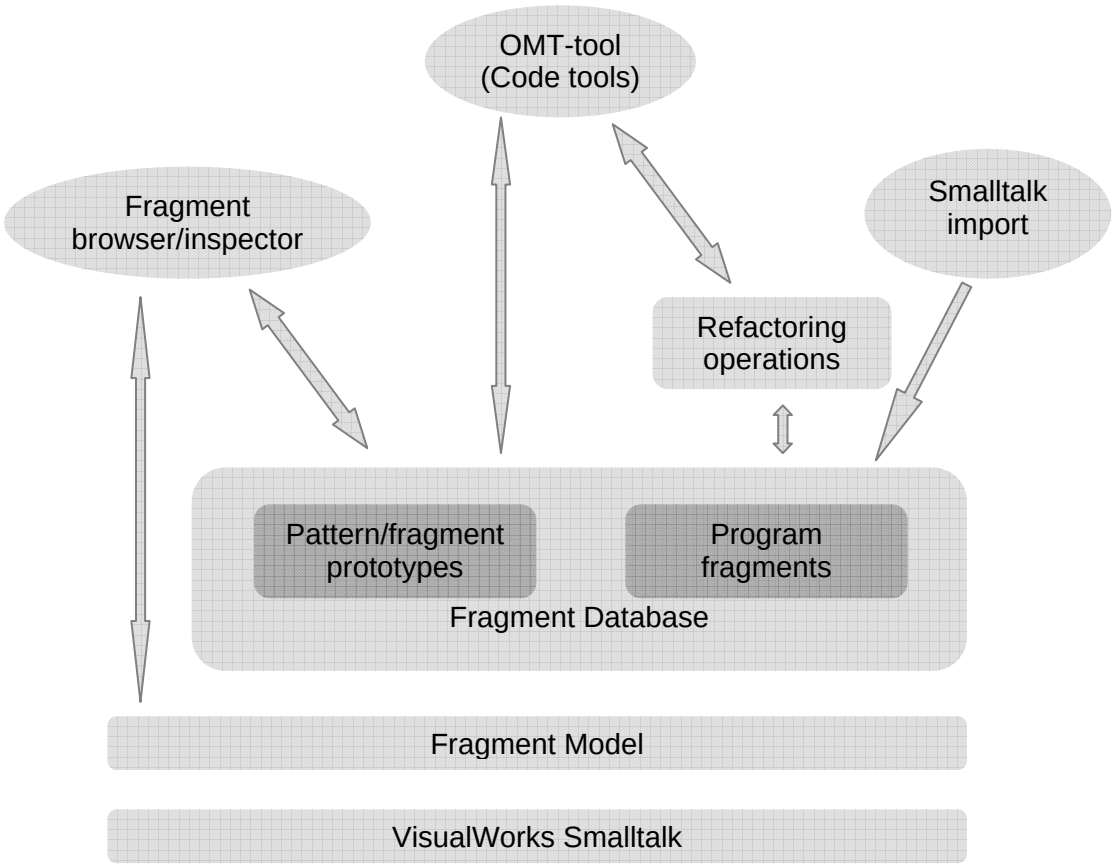


Figure 4.2: Overview of automatic code generation environment designed by Florijn and colleagues for OO patterns (Florijn et al. 1997)

The tool is composed of a fragment model and fragment database. The fragment database

consists of the pattern repository. The patterns repository contains prototype fragment configurations. These fragments are incorporated into the program using a suitable editor program. The editor binds the pattern fragments into the program fragments. A fragment browser is then used to inspect and use the fragment database. In addition to the fragment browser, a fragment inspector can be used to see the see the details of each selected fragment. The Object Modelling Technique (OMT) tool is used to work with the design view of the project. On the code level, class browsers are used to view and manipulate the source code in a project. The tool supports the import of Smalltalk programs into the fragment database.

4.5. Tool support for the PTTES collection

The earlier section discussed solutions to the problem of automatic code generation for the OO design pattern collection using two different research projects as examples. Recent research published from the Embedded Systems Laboratory has explored ways in which the process of applying design patterns from the PTTES collection may be automated (Mwelwa et al. 2007).

4.5.1 Workflow supported by PTTES Builder

The tool promotes a workflow very similar to that observed while manually adding patterns to a project. For instance, as in most IDEs, project creation begins with creating a project file and folder for the source code. This step is followed by a sequence of wizards to gather pattern-specific information to setup the hardware platform. This is followed by a sequence of wizard pages that prompt the developer to choose an appropriate scheduling strategy. The tasks to be used in the system are then implemented by choosing appropriate patterns from the collection. The system requirements give an indication of the required patterns. The design patterns are chosen manually and the wizard pages guide the user to customize the patterns as needed. The final customized source code is automatically generated and available to be used in the project.

The patterns are added to the project using the tool interface. The tool ensures that the patterns are added systematically while building the application. The patterns are chosen from a list and the tool prompts the user for information which is used to customize each pattern implementation (based on the specific requirements of the system). Thus the process of creating pattern-based software is based directly on the manual approach to applying these

patterns. Just as in the manual approach, the implementation examples documented in a pattern, make a significant contribution to the code generation mechanism. Finally, each pattern requires a custom wizard to gather the application-specific information needed to customize the implementation example. Each pattern is intended to solve a particular design problem and this uniqueness is captured in the fact that each pattern has a corresponding wizard.

4.5.2 Tool design

Mwelwa addresses the problem of automatic code generation in a manner similar to the approach adopted by Budinsky et al. (1996). An overview of this tool is given in Figure 4.3.

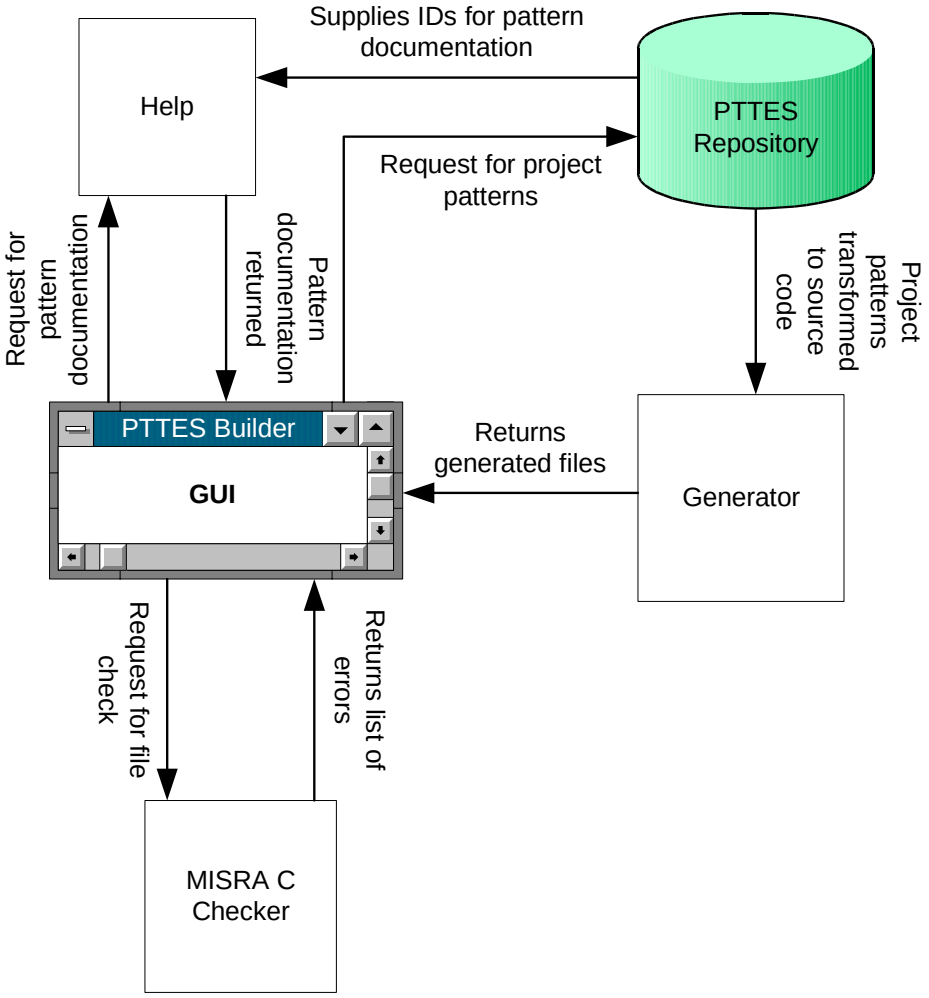


Figure 4.3: The structure of the PTTES Builder CASE tool. Adapted from (Mwelwa et al. 2007)

Mwelwa's tool stores code templates of the implementation examples in the PTTES collection using the XSLT format. The template identifies implementation elements that are

application dependent and resorts to the use of pattern-specific wizards to gather this information for each project. The project specific information is stored in an XML format to necessitate code generation from the templates of example source code. Thus, at the heart of the code generator is a DOM parser which uses the XML project file and XSLT template files to generate source code for each project. PTTES Builder and the RapidiTTy product family The PTTES Builder focussed on providing automatic code-generation support for the use of the time-triggered patterns on 8051 platforms. More recently research based on tool support for this pattern collection has been commercialised by TTE Systems Ltd (webpage: TTE Systems Limited 2008). The company is a spin-out commercialising research from the Embedded Systems Laboratory of the University of Leicester. Their core product – the RapidiTTy™ suite of development tools is designed to aid practitioners developing embedded software with the time-triggered architecture.

The RapidiTTy™ basket offers tools for developing embedded software on multiple targets ranging from COTS microcontrollers to custom FPGA soft cores. The tool as it stands now offers extensive debug support and detailed timing analysis. The TTE Builder™ engine used in RapidiTTy™ MCU is intended to aid the designer in configuring, customising and integrating code libraries when creating embedded applications.

4.6. Discussion

This section presents a discussion on the code generation approaches described in this chapter. It discusses the potential of using design patterns beyond code generation, in the design stage. It describes software engineering practices that benefit from the availability of multiple designs in a single development cycle.

4.6.1 Patterns and automatic code generation

Automatic code generators for the patterns documented by Gamma and colleagues are targeted at creating object-oriented software. Thus, most tools designed to support the object-oriented patterns work with classes, methods and associations to realise the code implementations of these patterns.

These tools also distinguish between the natures of pattern information documented in a collection. Budinsky and colleagues(Budinsky et al. 1996) use section pages to segregate the

pattern information. Similarly Florijn and colleagues (Florijn et al. 1997) support different views (such as "patterns view", "design view" and "code view") in their patterns-based tool. Eden identifies the need to distinguish between abstractions when designing tool support for the object-oriented pattern collection (Eden 2000).

Another generic feature of code-generation tools tends to be the identification of constant and variable elements of a pattern solution. The constant factors provide the template (or stub) framework from which pattern-based is generated. The variable elements of a design are queried from the user and incorporated in the code generation process.

Tool support for PBSE has primarily relied on supporting automatic code generation. Both the pattern-based tools (see (Mwelwa et al. 2007) – Section 4.5 and (Budinsky et al. 1996) – Section 4.4) described in this chapter have code generation mechanisms that rely on storing the pattern/pattern examples as modifiable templates. The tools gather application-specific information through a system of wizards and create source code for each project. In addition, both tools support access to the original pattern documentation, which is usually stored as hypertext accessible through a suitable browser mechanism.

Implementation examples are but one aspect of the pattern documentation. Tool support for pattern-based software development stands to potentially benefit from using more of the pattern information actively in the development process. Since patterns capture design information, the possibility of pattern-based design space exploration presents an interesting research problem.

4.6.2 Patterns and design space exploration

As observed earlier, design patterns were originally intended to be used as a reference source by practitioners wishing to incorporate best practices in their designs. Once identified, the solutions were usually manually adapted to obtain pattern-based software for the project at hand. Research into tool-support for pattern-based software development looked at techniques by which this manual process could be automated. Section 4.4 and Section 4.5 described automatic code generators which provide tool support for adapting a pattern.

Like in the manual process (described in Section 3.11), the pattern implementation example or a source code template representing the solution is key to most code generation attempts.

In addition to the code generation unit, tools supporting pattern-based development also include mechanisms to browse the pattern documentation in order to reference it. The implementation example plays an active role in obtaining software from a pattern, while the pattern information, though present in the tool is expected to be used on a need basis (as a manual reference).

However pattern languages capture domain expertise which can be used in stages before code generation. Since patterns contain extensive design information, how can they be used in say the design phase? Why has tool support for pattern-based software development invariably been developed around the implementation example? These are some of the questions that this research project addresses and attempts to reason.

A preliminary analysis of the pattern documentation suggests that the nature of the information captured in an implementation example is distinct from the rest of the sections. While most of the pattern documentation relies on text and figures to describe the solution, the implementation example uses programming languages for the same. Thus the example implementation can be seen as the most concrete application of the design pattern in source code terms. This is possibly a reason for their active use in tools.

The research presented in this thesis focuses on the design aspects of software and as a first step attempts to understand the use of pattern information in design space exploration. Design space exploration is the process of analysing several "functionally equivalent" implementation alternatives (Mohanty et al. 2002). Since the pattern philosophy recognises the existence of multiple solutions for a single problem and encourages documenting and capturing the relationships between these solutions, pattern collections/languages have the potential to provide a wealth of information to support design space exploration activities.

To explain the relevance of working with implementation alternatives, it is important to discuss software development approaches that rely on the use of multiple designs. Two such approaches, namely ‘prototyping’ and ‘support for design diversity’ are discussed in further detail to appreciate the need for design alternatives when developing embedded applications.

4.6.2.1 Prototyping and Rapid Application Development

Prototyping involves the use of development resources to obtain a subset of the working versions of the various aspects of the desired final system. The prototype may or may not be part of the final production code. Various researchers such as Boehm, Budde, Andriole and

Kordon have studied the process of prototyping, with the intention of identifying the nature of software development when prototypes are used (Boehm et al. 1984; Budde and Zullighoven 1990; Davis 1992; Andriole 1994; Kordon and Luqi 2002). The effort put into prototyping is often used for developing a part of the whole system and there is no emphasis on either the process or the code maintainability. In fact, researchers like Davis and Marcio view prototyping as a mechanism which helps programmers to better understand and capture effectively the system requirements to be used in a more intensive development process that usually follows the prototype cycle (Davis 1992; Marcio et al. 2006).

Enthusiasts see prototyping as fundamental to the success of operations supporting software products especially in situations where there is a heavy constraint on time and development resources. Bernstein indicates that for every $1.00 invested in prototyping there is a $1.40 return within the life cycle of system development (Bernstein 1996). Boehm and Andriole have independently documented experiments which show that prototyping reduces the program size and programmer effort (Boehm et al. 1984; Andriole 1994). This approach forms the foundation for the Spiral development method (Boehm 2000; Boehm et al. 2005). There are studies that illustrate the use of prototypes when developing embedded software. (Barry 1989; Thompson et al. 1999; Chung et al. 2007). Research projects include development of frameworks for prototyping systems (Tyszberowicz and Yehudai 1992; Tessier et al. 2003) and techniques to bridge the stages of simulation and system prototyping (Jones and Cavallaro 2003).

Pattern collections are intended to contain extensive domain-knowledge. In addition to documenting the solutions to specific design problems, patterns also capture information regarding other patterns that may be of relevance in a particular circumstance. This can either be through references to other patterns in the text of the pattern or through explicit mention of related patterns and alternatives in the corresponding section. Thus patterns seem to have potential for use in prototyping environments. For instance, developers may choose to use simpler solutions to implement an initial prototype and the pattern documentation can be used to explore more sophisticated solutions to enhance this initial solution, should the need arise. In other words, patterns may be used to obtain an initial design and explore the possibility of alternative designs based on the initial one.

4.6.2.2 Supporting design diversity

Prototyping typically involves the use of multiple system designs in a single development cycle. Another classic use of multiple designs is in the construction of redundant software systems. Safety-critical systems have long used redundancy techniques to guarantee systems reliability (Briere and Traverse 1993; Rohr 1995; Yeh 1998). NASA funded research in this area resulted in the development of fault-tolerant space applications like the Self-Testing-And-Repair (STAR) computer (Rohr 1995). Redundancy-based fault-tolerance have also been used by Airbus (Briere and Traverse 1993) and Boeing (Yeh 1998). In a redundancy-based approach to implementing fault-tolerance, system designers deal with critical component failure by providing suitable "backup" copies of the component.

Hardware redundancy is often achieved through the use of multiple copies of the same hardware component (Su and DuCasse 1980; Fuhrman et al. 1995; Neves and Saotome 2008). Implementing software redundancy is less straightforward (Romanovsky 2007). It cannot be achieved by using multiple copies of the same software executable. This is especially the case for software that fails due to the presence of bugs that are a direct outcome of flawed software design (Leveson, 1995). When supporting software redundancy in embedded applications, there is an impetus on the availability of diverse design solutions for the same applications or parts of it. The differences in the redundant software components can be based on implementation and/or design diversity or diversity incorporated in the development process (Hilford et al. 1997).

Thus design diversity relies on the ability to derive multiple designs from the same set of specification (Kelly et al. 1991; Torres 2000; Littlewood et al. 2001). Different software components can be implemented from each of these multiple designs and then used to enhance the fault-tolerant nature of the system being designed. Originally two techniques were used to implement fault-tolerance through design diversity (Pullum 2001). These approaches – the recovery blocks approach and n-version programming are described below.

a) Recovery blocks

The recovery-blocks approach to implementing software fault-tolerance uses alternative software designs in a manner similar to dynamic redundancy approach – *standby sparing* (Randell 1975). It is based on a checkpoint-restart mechanism. Checkpoints are established before the execution of a software version. The results of the software execution are verified

using a suitably designed test. If the results fail the test, the system is restored to the state of the last checkpoint and the alternative version of software is executed. The alternative software versions may offer results of a lower quality to ensure the success of the test. If all the versions fail the test the system must communicate this as a failure to the interacting world (Wilfredo 2000). The flowchart shown in Figure 4.4 depicts the recovery-block approach to realising software redundancy.

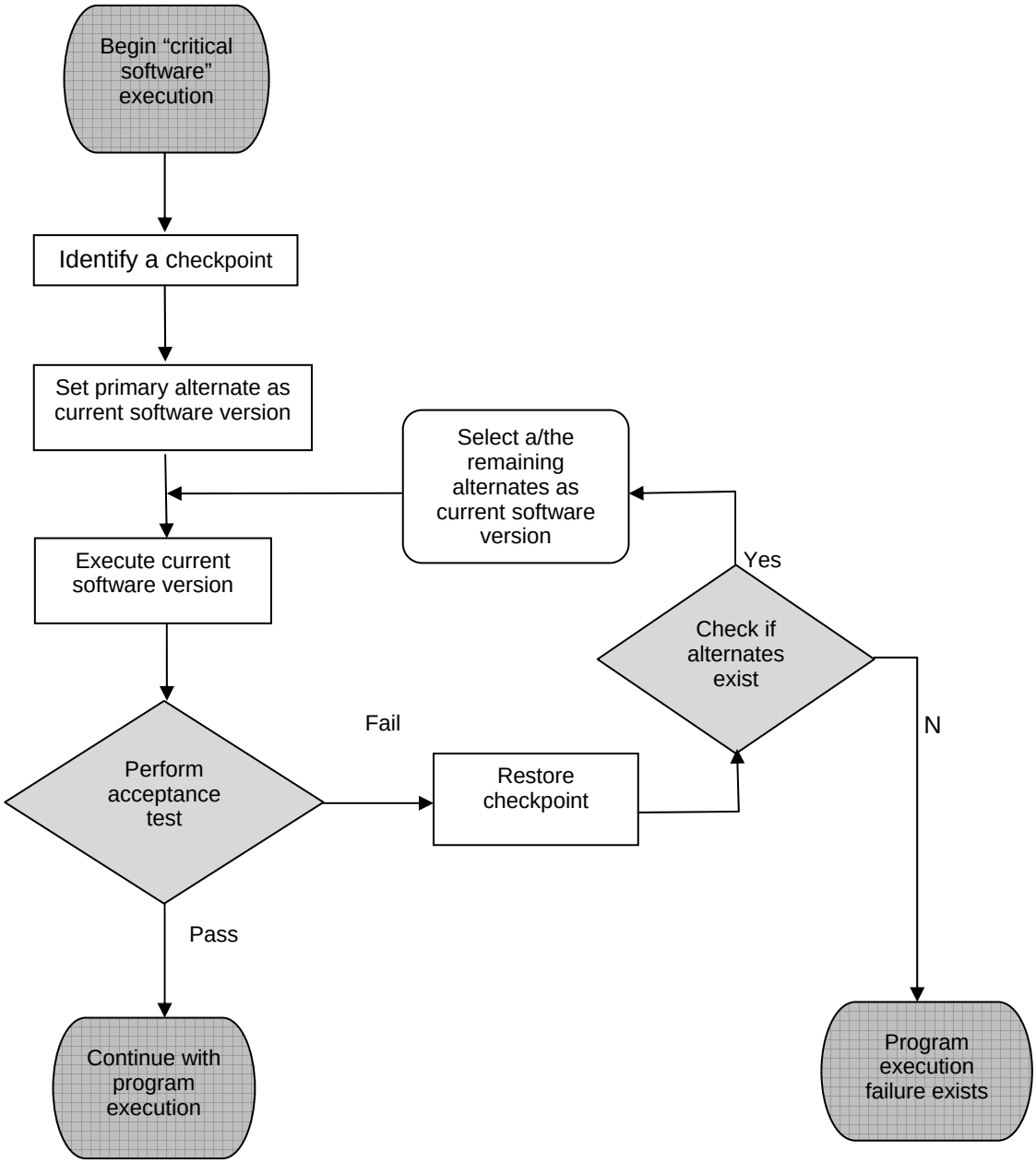


Figure 4.4: Flowchart depicting recovery-blocks technique. Adapted from (Kelly et al. 1991)

The objective of this approach is to detect design flaws at run-time using a suitably designed

'acceptance test' (Randell et al. 1978; Tyrrell 1996). The acceptance test is performed on the results obtained by executing one of the design alternatives, called the Primary Alternate. If the test fails, this approach implements the recovery by rollback, wherein a previous correct state is restored and the alternative software, called Secondary Alternate is then executed. This technique can be seen as a backward error-recovery approach.

b) N-version programming

Another technique to implement fault-tolerance using multiple software versions is the N-version approach (Avizienis 1995; Fuhrman et al. 1995) as depicted in Figure 4.5. The origins of this concept can be found as early as the works of Charles Babbage where he states that:

"*When the formula to be computed is complicated, it may be algebraically arranged for computation in two or more totally distinct ways, and two or more sets of cards [software programs] may be made. If the same constants are now employed with each set, and if under these circumstances the results agree, we may then be quite secure of the accuracy of them all.*" (Babbage 1974)

The fault-tolerant design is based on the n-fold modular redundancy technique for realising hardware fault-tolerance (Avizienis 1995). A majority voting technique is used for the decision of output correctness (Wilfredo 2000).

With respect to software redundancy realised using n-version programming, the application requires the availability of multiple (n) 'n' software versions satisfying the same set of requirements. These multiple versions are executed (usually) in parallel and the outputs of execution of each software version are sent to a voter. The voter is an important element of the fault-detection system. The voter selects the outputs that need to be sent through to the next stage of executions. (Avizienis 1995; Chen and Avizienis 1995; Lyu 2007).

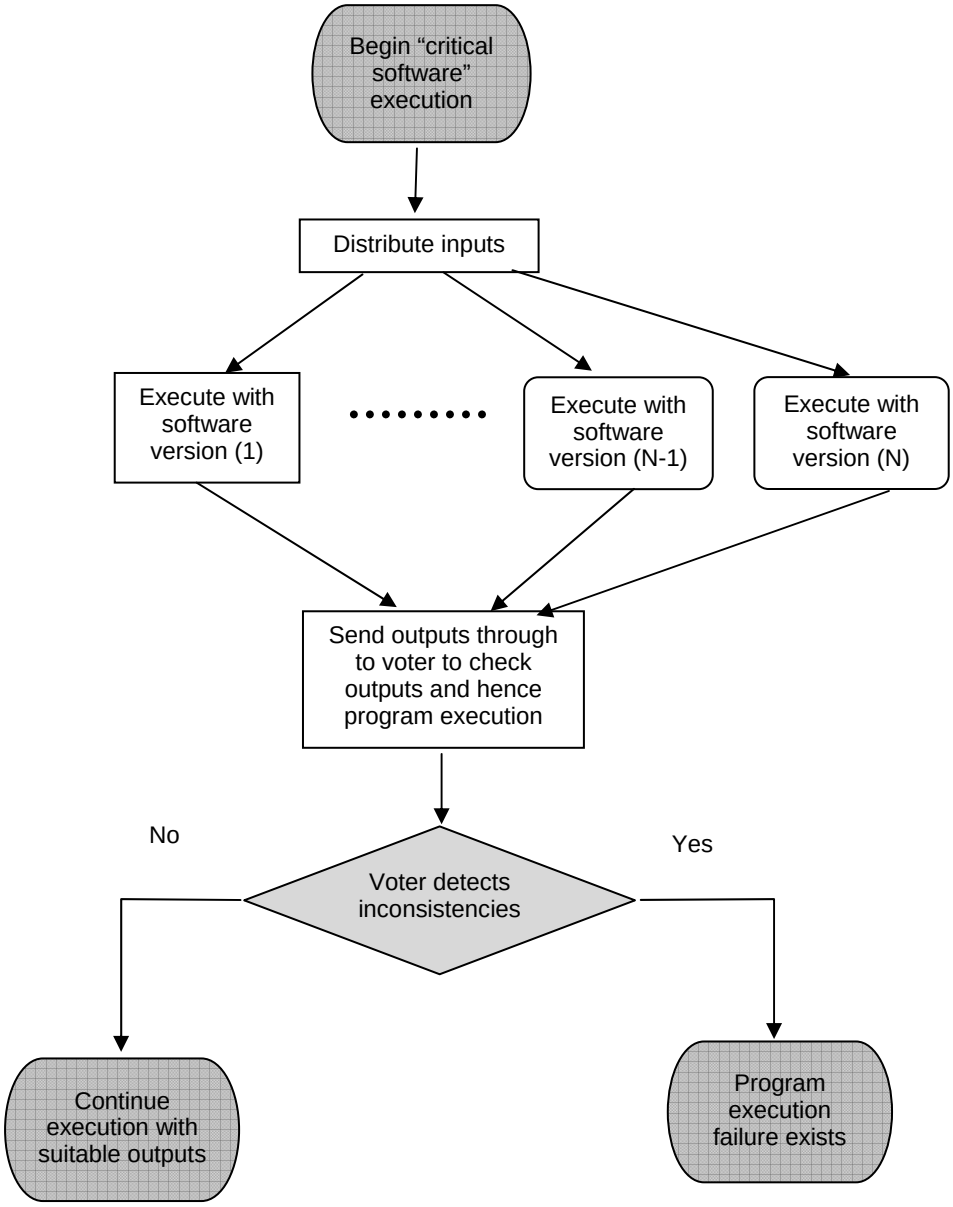


Figure 4.5: Flowchart depicting the N-version technique. Adapted from (Kelly et al. 1991)

Both n-version programming and the recovery block approach rely on the availability of diverse design/implementation to implement software fault-tolerance. In the recovery block approach the software versions are executed on a need-basis (i.e. if the primary alternate fails the acceptance test) and the technique can be seen as a backward-recovery approach which requires state information to be stored in order to support rollback(Kelly et al. 1991). N-version programming on the other hand can be viewed as a forward error-recovery technique which requires that the multiple versions of the software are executed simultaneously. This approach can be seen as a resource-hungry technique requiring n-times the resources needed

by the execution of one implementation (Fuhrman et al. 1995; Wilfredo 2000). The error-detection/recovery is implemented by designing a suitable voter mechanism that processes the multiple outputs obtained from each version of the software.

These differences aside, the similarities of these approaches arise from the need to have multiple versions of the same software. Anderson and Avizienis, independently observe that implementing software fault-tolerance is expensive The development of multiple versions and acceptance tests/voter mechanisms can significantly increase development costs (Anderson et al. 1985; Avizienis 1995). . The possibility of the different solutions sharing common failure nodes suggests that either technique cannot offer guarantees (Knight and Leveson 1986; Brilliant et al. 1990). Similarly techniques like n-version programming can increase the resource requirements needed for simultaneous execution of multiple versions. Both these approaches can only reduce failure due to faults that occur as a result of faulty software design (when implemented using design diversity). Faults that originate from inaccurate requirements are not expected to be handled in either approach. Thus, in spite of its relevance in embedded software creation, design diversity is an expense that can often be disregarded due to the personnel and resource costs associated with it. This research project looks at pattern collections and languages as repositories of domain knowledge. If the vast amount of design expertise contained within this documentation can be used more effectively to support the availability and evaluation of multiple designs it can possibly offset some of the personnel costs involved in working with multiple designs. This idea forms the basic motivation for the research presented here. Exploring techniques that generate alternative designs, the domain-knowledge captured in a pattern collection can potentially be used to reduce production costs incurred by the use of multiple teams. By studying the possibility of such techniques, current tools for pattern-based designs can be improved to provide support in phases beyond code-generation. One interesting question is whether patterns can be used effectively in such a tool, and - if so - how should they be used? The research presented here attempts to address this problem by exploring techniques to obtain design alternatives.

4.7. Conclusion

This chapter discussed the labour-intensive process of creating diverse software. This process is also expensive. Since pattern collections capture a wealth of design information, with different solutions categorised as individual patterns, they have the potential of being used to support design space exploration activities.

This chapter discussed existing tool support to aid the process of using patterns in software development. It described existing tool support for the PTTES collection. It notes that the code generation process extensively uses the example source code made available with a pattern making that information element a very active contributor to the code creation mechanism. The chapter discusses the potential of using other pattern information in the software development process. It began with a brief introduction of design space exploration and a description of development strategies that benefit from the availability of multiple designs for a single set of requirements. It discussed the potential of the domain knowledge in design patterns as an effective means of providing design alternatives for consideration in such cases. The next chapter (Chapter 5) discusses the need to restructure the pattern documentation in order to distinguish the nature of the information captured in a pattern, as also the scope of the use of this information in the software development process. The chapter also provides a brief description of a set of scheduler design patterns identified through this restructuring approach which forms a set of empirical studies described in Chapter 6.

Part C: Research Work

5. Understanding the nature of pattern information

5.1. Introduction

Chapter 3 introduced the concept of design patterns. It described the PTTES collection in detail and introduced the concept of pattern-based software engineering. It ended with a discussion detailing the approach of using patterns for creating a simple embedded application. Chapter 4 described existing tool support for the use of design patterns in software development. The discussion in both these chapters emphasised the importance of the pattern examples (made available with each pattern documentation) during the creation phase. Chapter 4 also suggested the potential of using other pattern information in design space exploration activities. This chapter discusses the need to restructure the pattern language to effectively use pattern information in the software development process[3]. It suggests a restructuring approach and illustrates the idea on a set of design patterns suitable for designing schedulers for single processor embedded systems giving a brief insight into these schedulers.

5.2. The nature of pattern information

As mentioned earlier, design patterns capture time-tested solutions to classic design problems in a human-readable format. In fact, many practitioners like to consider pattern documentation as solutions to classic design problems made available in a recipe-like manner. The previous chapter discussed current practices in pattern usage. These observations emphasised the practical importance of pseudo code/source code examples in the actual software development process. Since the primary motivation of using design patterns is to ultimately create code, source code examples illustrating the application of a pattern seem to be the most concrete elements of the pattern documentation in this regard. For this very reason, it is also easy to appreciate why source code examples tend to be most

[3] The research described in this section has been previously presented and published at The UK Embedded Forum and EuroPLoP (Kurian and Pont 2005a; Pont et al. 2008)

easily adopted in the software engineering process. Unlike the implementation examples, much of the other information is abstract. This information is important for a human practitioner as it provides a better understanding of the solution being discussed. However, this kind of information cannot be used in a tool environment which relies on machine understanding of the design patterns.

Thus the information held in design patterns tends to be of varying nature. While some of this is held in natural language and more suited in the early stages of software development where the practitioners understanding of the problem and abstract solution is more important, some other bits of the information are more important at the actual stage of obtaining an implementation from the design pattern documentation.

The current pattern documentation does not support a distinction of this nature in the pattern documentation. The research presented in this chapter attempts to restructure the pattern language in order to distinguish between the nature of the information held in pattern documentation and its likely use in the software engineering process. With this distinction in place, it will hopefully be easier to appreciate the potential of pattern usage in software engineering.

The restructuring approach presented next, attempts to distinguish the pattern documentation as containing information targeted at three different stages of pattern usage. Information that provides a very high-level, human understanding of the problem and its associated solution options is documented in a Generic Pattern. The actual documentation providing recipe-like solution information is classified as the Design Pattern and seen to be more detailed solution options that can be derived from the solution presented in a Generic Pattern. Finally, the implementation examples are categorized as Pattern Implementation Examples (PIEs) and contain information primary held as source code or pseudo code examples.

5.3. Restructuring the pattern language

This section describes the restructured pattern language. As observed earlier, the solution example associated with a design pattern is a well used element of the pattern documentation. Although the use of other information documented in a design pattern tends to be more intuitive, a first step in identifying its use is to distinguish the information on the basis of understanding the problem/solution and implementing the suggested solution. The

implementation example is most useful when the practitioner is closer to applying the design pattern to a software project – i.e. when (s)he is in the process of creating code. The rest of the information is more useful in earlier stages of software development. An algorithmic understanding of the solution is more useful during the design phase. Similarly, some of the pattern information is important to understand the problem domain better and understand the implications of opting for a particular solution. In short, organizing this information according to its use in the progressive stages of software development should potentially improve the practitioners understanding of the design solution being discussed. The newly restructured language has a three tiered architecture, the elements of which are discussed below.

5.3.1 Pattern Implementation Example (PIE)

The actual implementation of the design pattern in the system depends on certain software and hardware characteristics of the embedded system. This information is best illustrated through the use of implementation examples associated with each Design Pattern. Thus the Pattern Implementation Example or PIE constitutes the third level of the restructured language.

As the name might suggest, PIEs are intended to illustrate how a particular pattern can be implemented. This is important (in the embedded systems field) because there are great differences in system environments, caused by variations in the hardware platform (e.g. 8-bit, 16-bit, 32-bit, 64-bit), and programming language (e.g. assembly language, C, C++). The possible implementations are not sufficiently different to be classified as distinct patterns: however, they do contain useful information, often relevant to just the platform or the chosen language of implementation.

Any PIE has a lot of implementation specific information. This also includes extensive source code examples to illustrate important design considerations. Since the PIE contains information in the form of source code, this layer is closest in abstraction to the software developed using patterns/pattern information. Thus much (if not all) of the information in a PIE lies on the same level of abstraction as source code and can be seen as the most concrete documentation of the design solution presented in a family of patterns headed by a Generic Pattern. The PIEs used in the creation of an application give an insight and/or indication to the patterns used in the project. This is by virtue of the fact that PIEs are related to patterns

which are in-turn related to generic patterns in the new re-structured language.

Two important reasons for introducing a new PIE layer are stated below: -

a. Some “low level” programming patterns have been labelled as “idioms”. It was considered whether it was necessary to introduce yet another new term (i.e. PIE) into this area, or whether the term idiom could be used here. There are many possible ways of implementing any idiom, while each PIE is associated with a single (or small number of) specific implementations and so it was felt that the new term PIE was the best way to identify this kind of pattern documentation
b. Another alternative to the use of PIEs was to simply extend each pattern with a large numbers of examples. However, this would make the pattern bulky, and difficult to use. In addition, new devices appear with great frequency in the embedded sector. By having distinct PIEs, it is now possible to add new implementation descriptions when these are available and deemed useful to enhance pattern documentation, without revising the entire pattern each time the patterns are revised.

The concept of a PIE is especially important in embedded systems where new hardware platforms are introduced into the market frequently. PIEs and Design Patterns can be used effectively in speeding up the developers understanding of a new hardware platform. Since each Design Pattern has a set of PIEs associated with it, there is a one-to-many relationship between Design Patterns and PIEs. This is depicted in Figure 5.1.

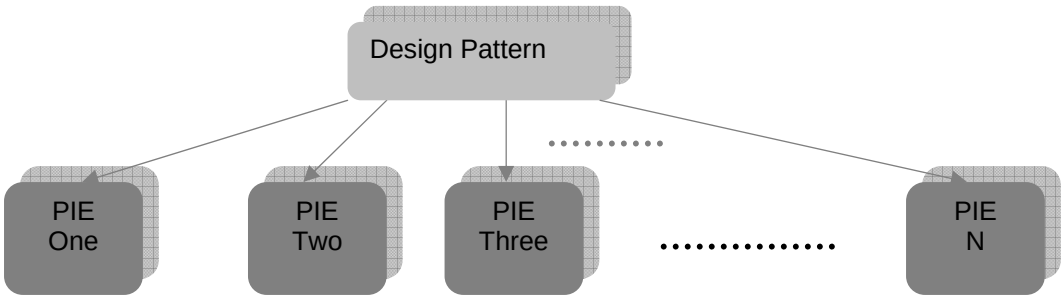


Figure 5.1: PIEs at a lower abstraction level compared to design patterns

5.3.2 Design patterns

The Design Pattern can be seen to provide a design solution for a sharper problem description derived from the problem described in the Generic Pattern. In many ways the information in

a Design Pattern describes the solution that needs to be implemented once the major architectural issues surrounding the design problem are resolved.

Thus Design Patterns form the second level of the restructured pattern language. While the Generic Pattern helps to evaluate the suitability of a set of Design Patterns for a specific problem, the Design Patterns related to a Generic Pattern of interest provide the recipe-like instructions that constitute a solution that will help realise an implementation. The information held in this kind of pattern can be of natural language descriptions or pseudo code/source code illustrations. Each Design Pattern refers to a set of supported implementation examples referred to henceforth as a Pattern Implementation Example (PIE). Though any given Design Pattern is associated to a set of known PIEs the information held in a Design Pattern should ideally be complete enough to derive new information given the need to arrive at one for a new platform or language.

5.3.3 Generic Patterns - the concept

The most abstract information held (in natural language) in the PTTES pattern documentation often relates to architectural issues that the patterns address. The restructuring approach relies on identifying this information and documenting it separately as a Generic Pattern. The information held in a Generic Pattern is intended to be an entry point to the understanding of the solution suggested for a problem at hand. The documentation consists of high-level design considerations that need to be made while attempting to solve the problem addressed by the pattern. Ideally, the Generic Pattern should contain information which can be used to evaluate if a pattern can be used in a particular problem and context. The problem description at this stage is not very specific. The solution documented as part of the Generic Pattern should give an indication of the basket of design solutions available at the disposal of the user for a given generic problem. Thus, the documentation must also contain references to Design Patterns that detail the design solutions that follow from the options presented in the Generic Pattern. Ín other words the problem addressed by each Generic Pattern can be solved using one of many equivalent Design Patterns which constitutes the second level of abstraction. In conclusion, each Generic Pattern suggests a solution on the architectural level and links in turn to a set of Design Patterns which details the multiple solutions for a particular architecture. Thus Generic Patterns and Design Patterns share a one-to-many relationship as shown in Figure 5.2.

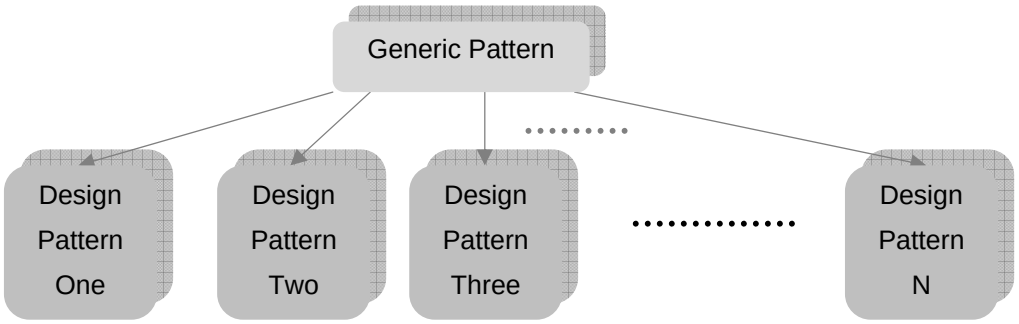


Figure 5.2: Generic patterns at a higher level of abstraction compared to design patterns

5.4. Scheduler Design Patterns

This section describes the three Design Patterns which forms the basis of the studies described in this thesis. The section begins with a description of the type of systems that form the focus of the studies presented here.

5.4.1 Context

The target platform for the embedded systems being analysed is a small microcontroller (e.g. 8051, Infineon C16x, Philips LPC2xxx, or PH Processor (Hughes et al. 2005)) which will be programmed in the C language.

The type of system, as described above, will start when power is applied, and stop when power is removed (or some error occurs). Specifically, there is no operating system to return to, and allowing the program to terminate will have unpredictable – and therefore undesirable – consequences. To avoid this, some form of (endless) "super loop" is usually employed (see Listing 5.1).

```
int main(void)
   {
   Do_X();

   while(1);

   // Should never reach here
   return 1
   }
```

Listing 5.1: Use of a "Super Loop" to avoid termination of a simple embedded application.

The application shown in Listing 5.1 has a "one shot" design: when power is applied, it will execute the function `Do_X()` once only and will then – apparently – do nothing until the system is reset. In such designs, the Super Loop is simply employed to "stop" the system.

5.4.2 TTC-SL SCHEDULER[4]

The design illustrated in Listing 5.1 is used in embedded systems. However, it is more common to use the Super Loop as the basis for the implementation of a simple "cyclic executive" (Shaw 2000), a time-triggered co-operative (TTC) architecture represented by the pattern "TTC-SL Scheduler". A possible implementation of this pattern is illustrated in Listing 5.2.

```
int main(void)
   {
   ...
   while(1)
      {
      TaskA();
      Delay_6ms();
      TaskB();
      Delay_6ms();
      TaskC();
      Delay_6ms();
      }

   // Should never reach here
   return 1
   }
```

Listing 5.2: A very simple cyclic executive (time-triggered co-operative scheduler) which executes three periodic tasks, in sequence.

If it is assumed that the tasks executed in Listing 5.2 always have a duration of 4 ms, then – through the use of the Super Loop and delay functions, a system which has a 10 ms "tick interval" can be realised as shown below (Figure 5.3).

[4] This section provides an overview of the pattern TTC-SL SCHEDULER ("Time-triggered, co-operative, Super-Loop scheduler"). The pattern is described in full in Kurian and Pont (2005a). Please note that the first published version of this pattern was called SUPER LOOP (see Pont, 2001).

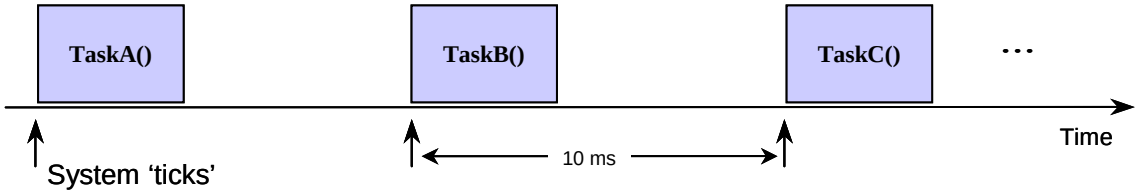


Figure 5.3: The task executions resulting from the code in Listing 5.2 (assuming all tasks are of duration 4 ms).

Note that if the duration of the tasks varies between executions then it is almost impossible to obtain a fixed tick interval with this approach, and use of a TTC-SL SCHEDULER is only appropriate in systems with soft timing constraints.

Overall, applications based on a TTC-SL SCHEDULER have extremely small resource requirements. Systems based on such a pattern (if used appropriately) **can** be both reliable and safe, because the overall architecture is extremely simple and easy to understand, and no aspect of the underlying hardware is hidden from the original developer, or from the person who subsequently has to maintain the system.

5.4.3 TTC-ISR SCHEDULER[5]

The pattern "TTC-ISR SCHEDULER" describes very simple software architecture for small embedded systems. Unlike TTC-SL Scheduler, TTC-ISR SCHEDULER is suitable for use with systems which have hard timing constraints.

The basis of a TTC-ISR SCHEDULER is an interrupt service routine (ISR) linked to the overflow of a hardware timer. For example, see Figure 5.4. The assumption is that one of the microcontroller's timers has been set to generate an interrupt once every 10 ms, and thereby call the function `Update()`. When not executing this interrupt service routine (ISR), the system is "asleep". The overall result is a system which - like that shown in Listing 5.2 – has a 10 ms "tick interval" in which three tasks are executed in sequence.

[5] This section provides an overview of the pattern TTC-ISR SCHEDULER ("Time-triggered, co-operative, ISR scheduler"). The pattern is described in full in Kurian and Pont (2005a). Please note that the first published version of this pattern was called ONE-TASK SCHEDULER (see Pont, 2001).

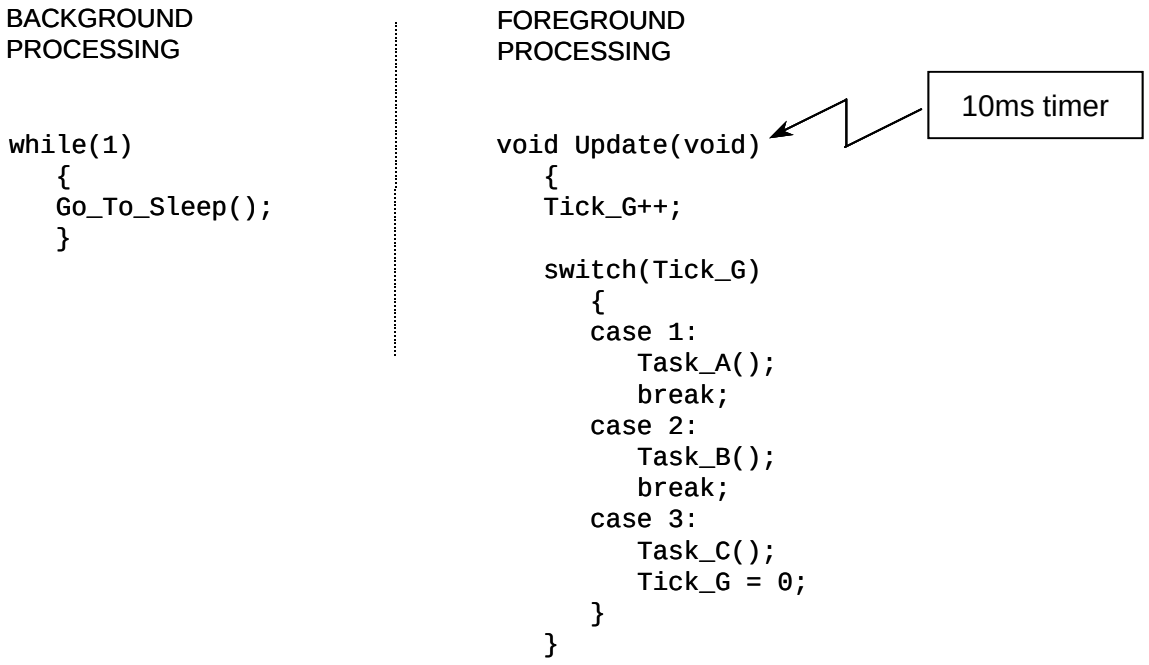


Figure 5.4: A schematic representation of a simple TTC Scheduler ("cyclic executive")

Please note that "putting the processor to sleep" means moving it into a low-power ("idle") mode. Most processors have such modes, and their use can – for example – greatly increase battery life in embedded designs. Use of idle modes is common but not essential.

Whether or not idle mode is used, the timing observed is largely independent of the software used but instead depends on the underlying timer hardware (which will usually mean the accuracy of the crystal oscillator driving the microcontroller). One consequence of this is that (for the system shown in Figure 5.4, for example), the successive function calls will take place at precisely-defined intervals (Figure 5.5), even if there are large variations in the duration of `Update()`. This is very useful behaviour, and is not easily obtained with architectures such as TTC-SL SCHEDULER.

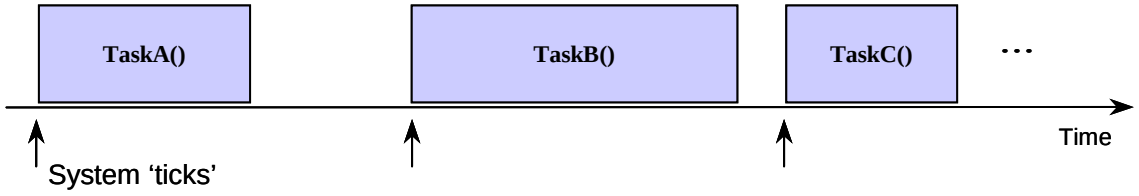


Figure 5.5: One advantage of the interrupt-driven approach is that the tasks will not normally suffer from "jitter" in their start times.

5.4.4 TTC SCHEDULER[6]

Implementation of a TTC-ISR SCHEDULER requires a significant amount of hand coding (to control the task timing), and there is no division between the "scheduler" code and the "application" code.

The pattern TTC SCHEDULER provides a more flexible alternative. TTC SCHEDULER is characterised by distinct and well-defined scheduler functions (see Listing 5.3).

```
void main(void)
   {
   // Set up the scheduler
   SCH_Init_T2();

   // Init tasks
   TaskA_Init();
   TaskB_Init();

   // Add tasks (10 ms ticks)
   // Parameters are <filename>, <offset in ticks>, <period in ticks>
   SCH_Add_Task(TaskA, 0, 3);
   SCH_Add_Task(TaskB, 1, 3);
   SCH_Add_Task(TaskC, 2, 3);

   // Start the scheduler
   SCH_Start();

   while(1)
      {
      SCH_Dispatch_Tasks();
      SCH_Go_To_Sleep();
      }
   }
```

Listing 5.3: An overview of a possible TTC Scheduler implementation: see Pont (2001) for details.

In addition to its greater ease of use, the TTC SCHEDULER also supports "one shot" tasks (tasks that are executed once, and then deleted from the task array). Periodic tasks may also be added or removed from the schedule at any time during the program execution. Neither type of "dynamic" schedule alteration is easy to achieve with a TTC-ISR SCHEDULER.

Of course, there is a price to pay. A TTC SCHEDULER implementation requires around 90 - 100 lines of code (LoC). The equivalent TTC-ISR SCHEDULER can be implemented in around 30 - 40 LoC. These code differences translate directly into memory requirements. For example, using the processor platform considered in this work (8051), a TTC SCHEDULER implementation (with two "dummy" tasks) requires 43 bytes of data memory and 675 bytes

---

[6] This section provides an overview of the pattern TTC SCHEDULER ("Time-triggered co-operative scheduler"). The pattern is described in full in Kurian and Pont (2005a). Please note that the first published version of this pattern was called CO-OPERATIVE SCHEDULER (see Pont, 2001).

of code memory. An equivalent TTC-ISR SCHEDULER implementation requires just 18 bytes of data memory and 249 bytes of code memory.

5.5. Discussion

Design patterns contain domain expertise documented in a human readable form. However, as noted in Section 3.9, patterns contain structured information. The original PTTES collection (Pont 2001) labelled everything as a "pattern". In addition to providing the time-tested solution to a classic design problem in a human readable form, it also contains implementation-specific elements usually captured as pseudo code/source code fragments. The actual design solution is often at a higher level of abstraction when compared to the source code example. This necessitated the distinction of pattern information as belonging to one of three categories:

- Generic Patterns
- Design Patterns, and,
- Pattern Implementation Examples

Where, the information held in each type of pattern documentation is used differently in the process of developing software from design patterns. The pattern documentation gets progressively implementation specific towards the PIE layer while the Generic Pattern contains very abstract descriptions of the solution to the design problem at hand.

Thus, in this new structure, the "Generic Patterns" are intended to address common architectural/high-level design decisions faced by developers of embedded systems. Such patterns do not – directly – tell the user how to construct a piece of software or hardware: instead they are intended to help a developer decide whether use of a particular design solution (perhaps a hardware component, a software algorithm, or some combination of the two) would be an appropriate way of solving a particular design challenge. The problem statements for these patterns typically begin with the phrase "Should you use a …" (or something similar). The "Design Pattern" enumerates solutions to design problems and contains algorithms and related descriptions for the same. Finally the "Pattern Implementation Examples" tends to be source code examples included to illustrate the implementation specific elements of the solution. Organised on different levels of abstraction, each lower level, pattern documentation, extensively references the patterns in the respective higher levels.

For example, the TT Scheduler described in Section 4 is a generic pattern. This pattern describes what a time-triggered co-operative (TTC) scheduler is, and discusses situations when it would be appropriate to use such an architecture in an embedded system. If you decide to use TTC architecture, then you have a number of different implementation options available: these different options have varying resource requirements and performance figures. The design patterns TTC-SL Scheduler, TTC-ISR Scheduler (Section 4) and TTC Scheduler describe some of the ways in which a TT Scheduler can be implemented. In each of these "full" patterns, the documentation refers back to the generic pattern for background information. The TTC-SL Scheduler [C, 8051] describes how the TTC-SL Scheduler can be implemented on an 8051 micro-controller using the C-language.

Figure 5.6 depicts part of the re-structured language.

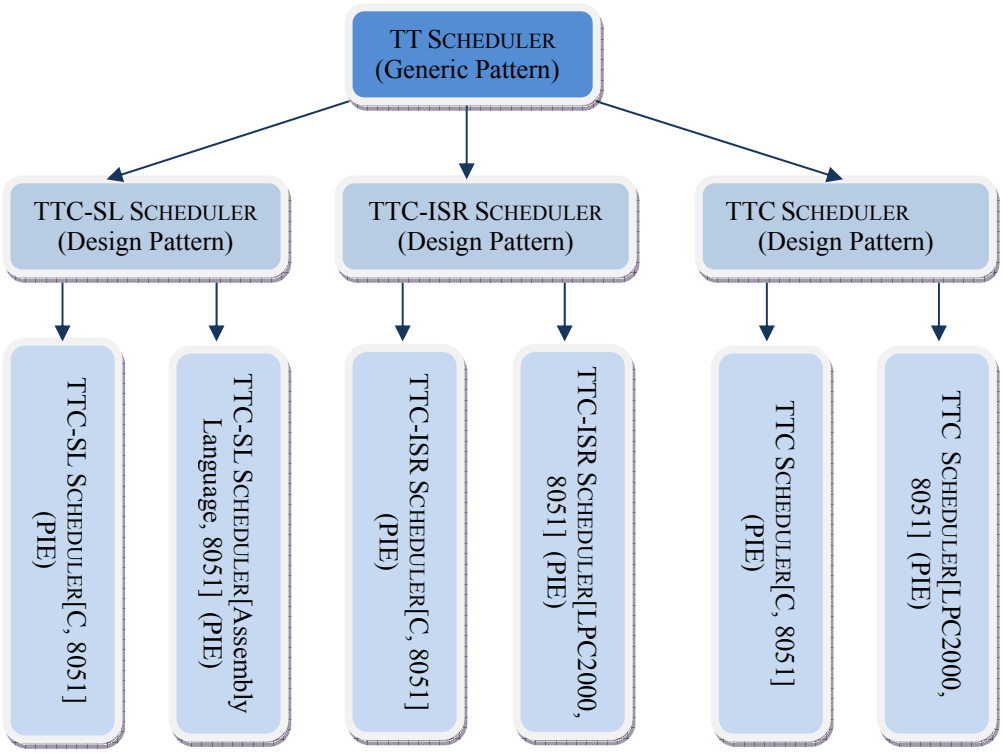


Figure 5.6: TT SCHEDULER pattern

With the inclusion of generic patterns and PIEs in the pattern collection, the language elements as discussed in the previous sections, can now be organised in a new, layered approach. The TT SCHEDULER (described in Appendix A1) is an example of a generic pattern. As depicted in the figure, TTC-SL SCHEDULER, TTC-ISR SCHEDULER and TTC SCHEDULER are patterns that belong to this generic pattern. A pattern implementation

example for the TTC-ISR SCHEDULER [C, LPC2000] (Appendix A1).

5.6. Conclusion

This chapter discussed the need to restructure the PTTES language. It suggested an approach to restructuring the language based on the nature of information associated with a pattern. It identified two new layers: a Generic Pattern (discussed the availability of possible solutions to the problem) and a PIE (that captures implementation details). The next chapter describes a set of empirical studies that primarily involve the use of PIEs to indicate the design of a system and subsequently identify systems with comparable designs.

6. Working with PIEs

6.1. Introduction

Chapter 3 introduced the PTTES collection for building embedded systems with a time-triggered architecture. The chapter discussed the importance of pattern examples for building software with design patterns. Chapter 4 emphasized the role of pattern examples in the design of an automatic code generator designed to provide tool support for the PTTES collection. Chapter 5 discussed the need to restructure the pattern collection in order to distinguish the information held in a design pattern and identify the pattern example as a unique entity called PIE, in the restructured language.

This chapter describes an empirical experiment based on the association between PIEs and patterns. The experiment evaluates design alternatives identified through exploring pattern relationships. The case study described here illustrates the possibility of creating alternatives from an initial system design, by analysing the design space specified by the patterns in the initial design. In order to add value to this kind of experiment, the design space exploration activities were applied to a more realistic embedded system – a Cruise Control System (CCS)

The study reported in this chapter describes the results while exploring the possibility of design space exploration using patterns. Similar studies conducted earlier have been published in conferences and journals[7].

6.2. Exchanging patterns

This section details an empirical experiment conducted to understand the possibility of exchanging patterns to obtain alternative solutions.

[7] The experiments discussed in this chapter have been discussed in previous conferences and journal papers (Kurian and Pont 2005b; Kurian and Pont 2007).

6.2.1 Aim

The empirical study described here aims to understand the possibility of exchanging PIEs suitably to obtain equivalent designs. Please note that the choice of the patterns that were exchanged in this study was not arbitrary but, instead, representative of the type of modification that is common in small, resource-constrained systems. Specifically, the experiment attempts to replace implementations of the pattern TTC SCHEDULER (see Section 5.4.4) with implementations of the pattern TTC-ISR SCHEDULER (see Section 5.4.3) and compare the two designs. As discussed in Section 5.4, the two patterns provide very similar system behaviour, but the TTC-ISR SCHEDULER can be implemented with significantly lower memory requirements.

In the type of small embedded system that forms the focus of this research; significant cost savings can be made if memory requirements are kept to a minimum. In such circumstances, prototyping a system using a TTC SCHEDULER and then converting the chosen design to a more (memory) efficient implementation by swapping to a TTC-ISR SCHEDULER for the final implementation is a desirable process practice. Alternatively, the product may initially be implemented using a TTC SCHEDULER: if subsequent extensions to the design are then required, the available memory on the chosen hardware platform may prove to be inadequate: changing to a TTC-ISR SCHEDULER can address this problem, without requiring expensive hardware changes.

6.2.2 Data set

The data set used consists of code submissions made by university students. The students used a selection of design patterns, documented in Pont (2001), to build their projects. They used the TTC SCHEDULER pattern to implement the scheduling sub-system. The projects were built using the sample source code provided as part of the pattern documentation.

This experiment used four such project submissions. One of the submissions involved building a simple flashing-LED system. Two other submissions used the TTC SCHEDULER to implement a switch interface, controlling LEDs. The last submission was a simple intruder-alarm system.

These submissions seemed to incorporate real world project constraints such as short turn-

around times and limited resources since they were essentially two-week exercises focussed on building a working system. They were thus considered representative of prototype systems and the scheduler code used to obtain the initial design did not undergo severe changes since the focus lay more on developing the application logic.

6.2.3 Methodology

This section describes the methodology used in this experiment.

6.2.3.1 Approach

Perl scripts were written and executed to identify the TTC SCHEDULER PIE. Also Perl scripts were used to convert between implementations of the TTC-ISR SCHEDULER pattern and the TTC SCHEDULER pattern.

6.2.3.2 Identifying the tasks

Task details were obtained by first identifying calls to the `SCH_Add_Task()` function in the source file "`main.c`". Here the first argument provides the name of the task to be scheduled. The second argument is the tick interval in which the task is to execute for the first time. The third and final argument indicates the period (in "ticks") between task executions.

Once these three parameters have been extracted for each call, the equivalent dispatcher task is written for the new system. Since the priority of the tasks is reflected (implicitly) in the order in which they are added to the scheduler queue, it is important that the TTC-ISR SCHEDULER code processes tasks in the same order.

6.2.3.3 Generating the timer files for the TTC-ISR SCHEDULER implementation

An analysis of the scheduler initialisation functions in the TTC SCHEDULER implementation gives an indication of the required timer setting for the TTC-ISR SCHEDULER. The remainder of the timer functions are statically defined in a text file. This text file forms a template: by inserting the appropriate code used to initialise the timer registers code to generate "ticks" at the required interval can be created.

The source file "`main.c`" also needed to be modified. The new `main()` function was defined in a file called "`new_main.c`".

All scheduler functions were replaced by the appropriate functions defined in one of the automatically-generated source files (either "`timeline.c`" or "`timer2.c`").

Perl scripts were also used to generate the necessary header files.

6.2.3.4 Building the new project

The process of converting between a TTC SCHEDULER and TTC-ISR SCHEDULER (as described in the previous sub-sections) was entirely automatic. Once the new source files had been produced, the generated code was compiled (manually) using the Keil C51 compiler.

The following changes were (in this trial) made manually to the Keil project file. The scheduler source files and the `main.c` file (from the TTC SCHEDULER system) were removed from the original project. These files were then replaced by the three automatically-generated source files (`new_main.c, timeline.C, timer2.c)`. The "Overlay" (linker) directive - used to support function pointers in the TTC SCHEDULER (see CO-OPERATIVE SCHEDULER (Pont 2001)) - was no longer required in the TTC-ISR SCHEDULER implementation.

Having made these changes the new project was then compiled. In the present trial, minor compilation errors need to be manually fixed. These were caused when global error variables are defined in the scheduler files, but accessed across the whole system. These global variables were used by specifying the scheduler header files that no longer belong to the project. These `#include` directives were manually removed and the global variable was defined in one of the task files.

Note that the next version of the conversion software is expected to remove the need for manual editing of the project files, and for the tracking of global variables (etc). However, these issues were not the central concern in the pilot study described here.

6.2.4 Results and analysis

The four TTC SCHEDULER implementations were successfully converted to TTC-ISR SCHEDULER implementations. There were no compilation errors. Two of the projects had minor linker errors.

The memory requirements of the TTC-ISR SCHEDULER system were found to be significantly lower than those of the TTC SCHEDULER. Table 6.1 shows a comparison of the data memory usage of the alternative architectures for each code sample.

Submission identifier	TTC SCHEDULER (BYTES)	TTC-ISR SCHEDULER (BYTES)
CODE I	903	487
CODE II	870	451
CODE III	752	321
CODE IV	2784	2370

Table 6.1: Data memory usage in both the architectures

There was also a significant decrease in the size of code memory used (Table 6.2).

Submission identifier	TTC SCHEDULER (BYTES)	TTC-ISR SCHEDULER (BYTES)
CODE I	49.4	26.3
CODE II	43.6	20.5
CODE III	32.5	14.5
CODE IV	118.2	99.1

Table 6.2: Code memory usage in both the architectures

In this study, detailed comparisons of the behaviour of the original and modified systems were not carried out. However, the behaviour of the original system and modified systems was observed and the new system was found to operate in the same way. Such an exchange is – clearly – only sensible in situations where there is at least a partial overlap in the behaviour of – and interface to - the patterns which are to be exchanged. In many cases, it should be possible to (automatically) determine from the pattern documentation whether a

particular exchange makes sense, but this has not been attempted in this study.

6.3. Case study: Cruise control system

Section 6.2 described the results from a simple assessment of the pattern-exchange technique, using code from student submissions. Such code is, of course, not entirely representative of production code. This section describes the results from a more realistic case study in which the technique was applied to code from a hardware-in-the-loop (HIL) simulation of a cruise-control system (CCS) for a passenger car.

6.3.1 Aim

An automotive CCS is intended to provide the driver of a passenger car with the option of maintaining the vehicle at a desired speed without further intervention. The system used here is adapted from a similar design presented by Ayavoo *et al.* (2004).

Such a CCS will typically have the following features:

- An ON / OFF button to enable / 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.

For the purpose of our case study a hardware-in-the-loop (HIL) simulation of the CCS was built using a single Atmel 8051 processor. In this case, the specification of the CCS was simplified such that the vehicle was assumed to be always in "cruise" mode.

In the HIL simulation, a computational model was used to represent the environment in which the CCS would operate. This model had one input (current throttle) and one output (a train of pulses representing the speed of the car), as illustrated in Figure 6.1.

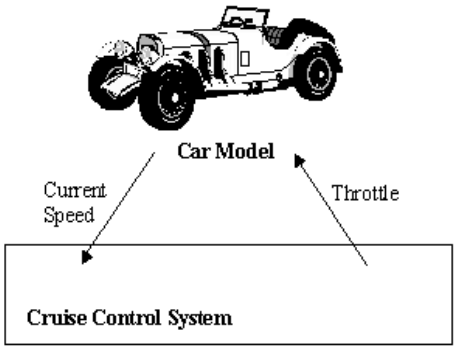


Figure 6.1: A basic cruise control system. Adapted from work done by Ayavoo and colleagues (Ayavoo et al. 2005)

The core of this simulation is a simplified physical model based on Newton's laws of motion. First the instantaneous acceleration of the vehicle is calculated (equation 6.1). Once this acceleration has been obtained, the new speed of the car is then determined (equation 6.2).

$$a = ((\theta\tau) - (v_i Fr)) / m \qquad \text{equation 6.1}$$

$$v_f{}^2 = v_i{}^2 + 2a\Delta x \qquad \text{equation 6.2}$$

where:

a	acceleration of the car
v_i	initial speed of the car
v_f	final speed of the car
m	mass of the vehicle
Δx	linear displacement of the vehicle
θ	car's throttle setting
Fr	frictional coefficient
τ	engine torque

Please note that in this case, it is assumed that the vehicle is under the influence of only two forces, the torque created by the car engine and the frictional force that acts in the opposite direction to the motion. The engine torque is assumed to be constant over the speed range and the force of this torque is proportional to the throttle settings. Hence the engine force is modelled as θ times τ for this case study These models can clearly be made more realistic, but - for the purpose of this study - this simplified model was sufficient.

The experiment aims to establish the equivalence between alternative designs of the CCS system. The first design alternative is obtained by exchanging the scheduler used to design the system. The second design alternative is obtained by using an alternative task design.

6.3.2 Methodology

The tasks in the CCS system essentially implement a sensor unit, a control unit and an actuation unit. A scheduler manages the timed execution of these tasks in order to realise the desired behaviour of the CCS. The scheduler can be viewed as a simple operating system. Figure 6.2 captures an overview of the system design.

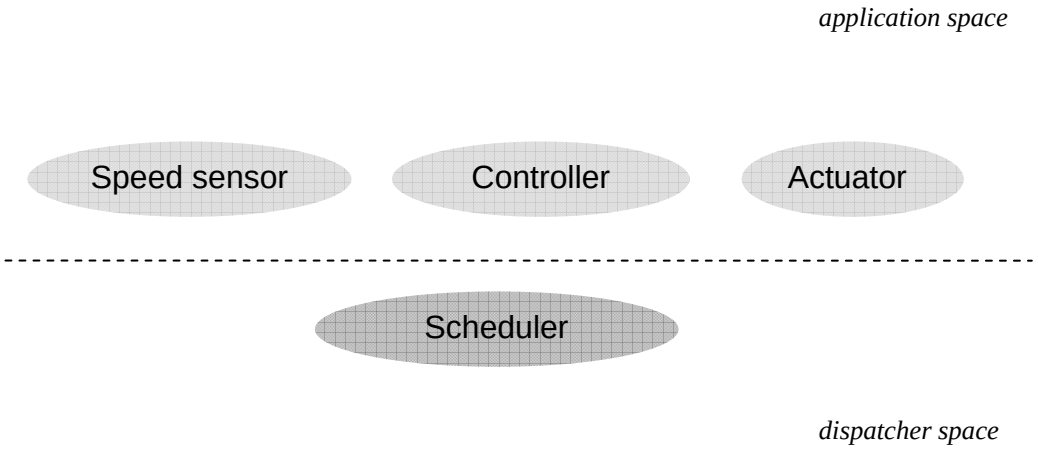


Figure 6.2: Overview of CCS design

The design is realised using four design patterns: a scheduler design pattern and the three patterns for each of the tasks in the system. The control and actuation task is designed using the PID CONTROLLER design pattern. The task designed to sense the speed of the car is implemented using the SOFTWARE PULSE COUNT design pattern. The PC LINK (RS-232) pattern is used for the sole purpose of recording speed of the vehicle for further analysis and is as such not considered for searching design alternatives. These tasks are scheduled using the framework suggested by the TTC SCHEDULER design pattern for a start. Figure 6.3 presents the CCS design from a design patterns perspective.

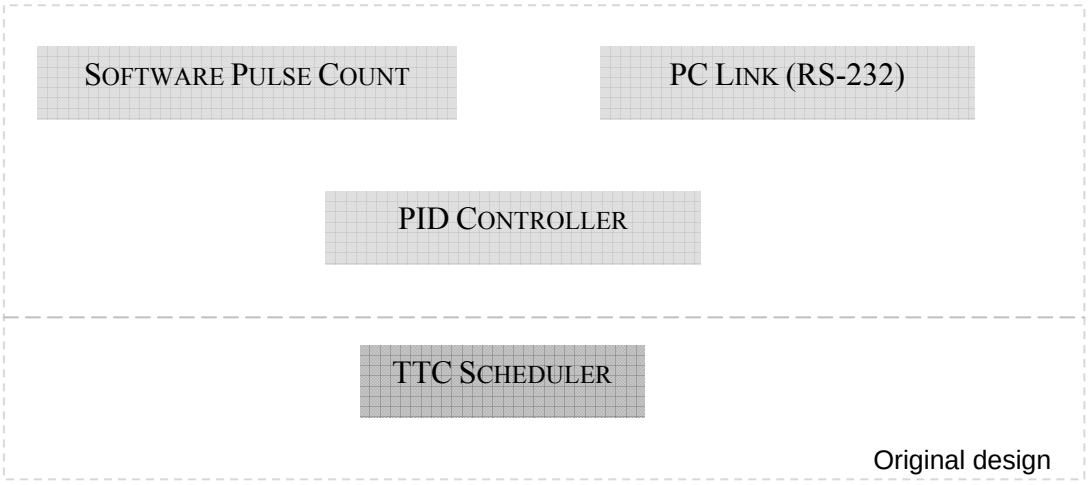


Figure 6.3: Pattern-based design overview of the original CCS system

The pattern documentation includes information regarding alternative patterns in the "Related Patterns" section of the pattern documentation (see Section 3.8). Based on this knowledge of the patterns used in the CCS design, the pattern documentation is revisited to identify options for design alternatives. This case study involved exploring alternatives for two different aspects of the system design. First, a scheduler alternative was evaluated to understand the ease of converting a system from one scheduler to another. Secondly, one of the tasks in the system was altered using a design solution derived from the pattern alternative indicated by one of the patterns used to design the CCS application.

In order to evaluate the alternative designs, the CCS was employed to obtain a certain speed behaviour from the car. The vehicle speed was programmed to be 20mph for the first 4 minutes, at which point the desired speed was set to 60 mph for the remainder of the simulation. The speed of the system, recorded against time, was obtained from the test bed and used for evaluating the alternative designs.

6.3.2.1 Experimenting with alternative scheduler designs

The CCS was initially implemented using a TTC SCHEDULER[8], based on the design by Ayavoo and colleagues (Ayavoo et al. 2005). The patterns in the original design and the alternatives to these patterns are indicated in Figure 6.4. Also, this figure indicates the set of patterns that contribute to the original design and the alternative design.

[8] TTC SCHEDULER was documented as CO-OPERATIVE SCHEDULER in PTTES

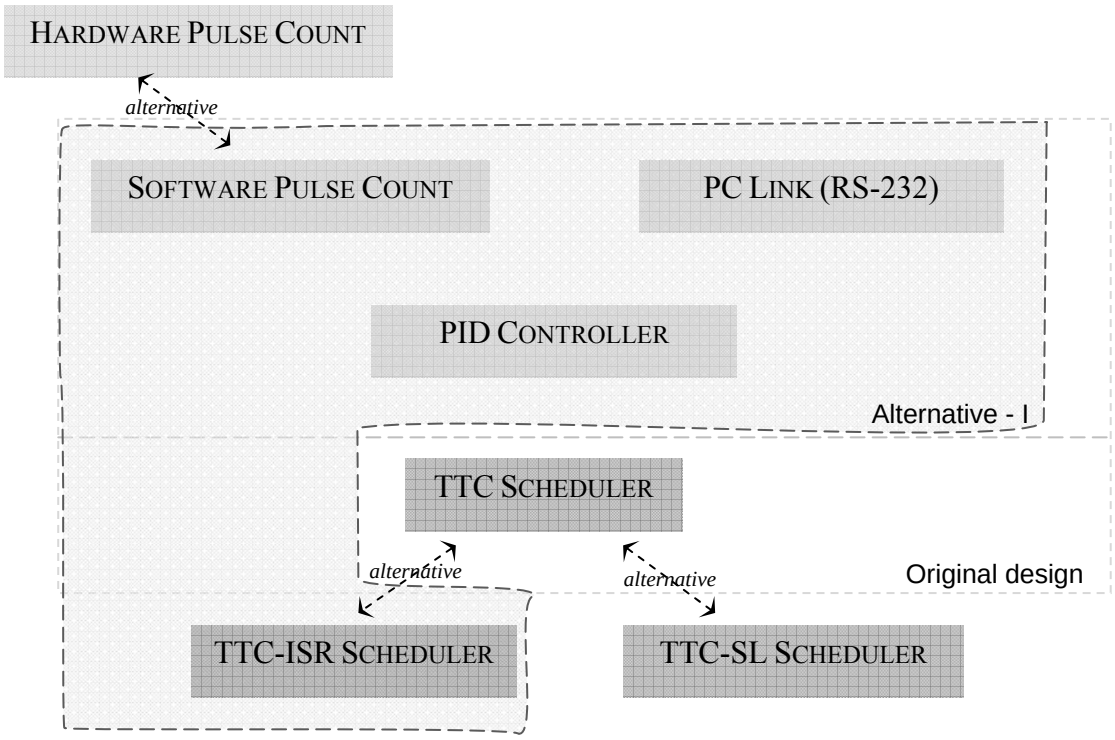


Figure 6.4: Alternative system design obtained by opting for different scheduler design

Table 6.3 lists the design patterns in both the original design and the new system design obtained by replacing a scheduler pattern with an alternative design pattern identified through the pattern documentation.

Functional behaviour realised using pattern	Patterns in the original design	Patterns in design alternative I
Task Dispatcher	TTC SCHEDULER	TTC –ISR SCHEDULER
Task implementing control algorithm	PID CONTROLLER	PID CONTROLLER
Task recording Speed	PC LINK (RS-232)	PC LINK (RS-232)
Speed Sensor	SOFTWARE PULSE COUNT	SOFTWARE PULSE COUNT

Table 6.3: Patterns in the original system and alternative system obtained by modifying the design of the scheduler entity

6.3.2.2 Analysing the tasks for design alternatives

In the second part of the case study, the CCS system was analysed to identify alternatives to design patterns used in realising the tasks of the application. Figure 6.5 captures the design patterns and alternative design solutions available for consideration. It also indicates the set of patterns used to realise the original design and the alternative design.

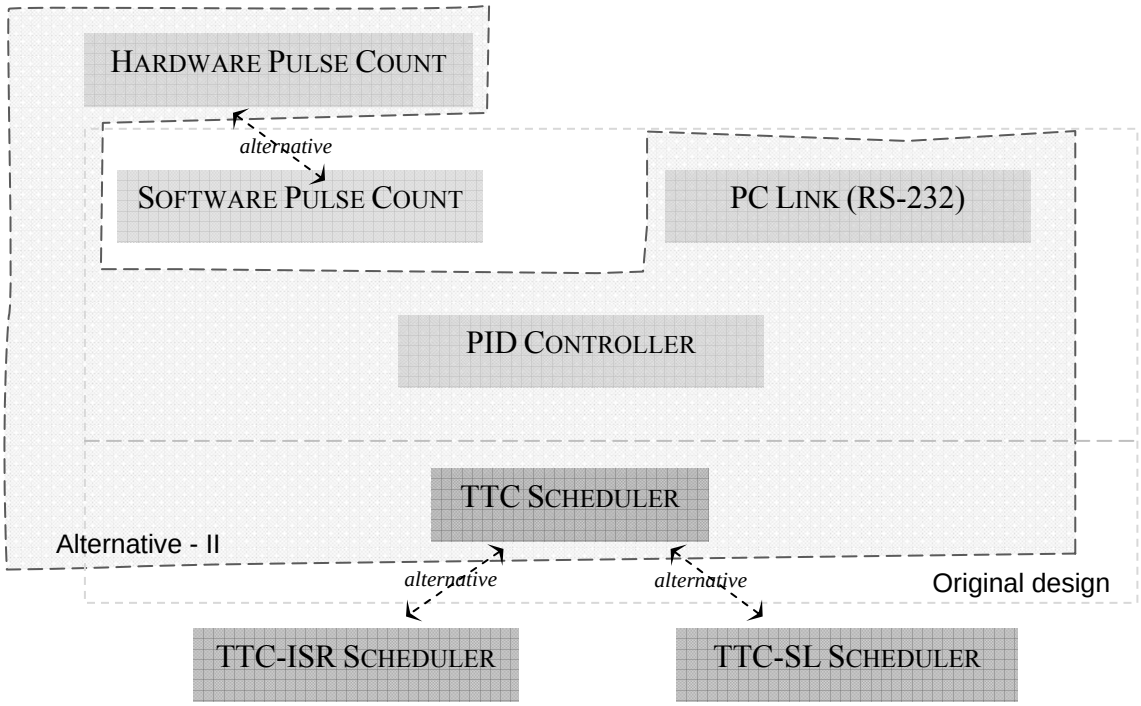


Figure 6.5: Alternative design obtained by using an alternative design for sensor task

Table 6.4 lists the design patterns in both the original design and the new system design obtained by replacing a scheduler pattern with an alternative design pattern identified through the pattern documentation.

Functional behaviour realised using pattern	Patterns in the original design	Patterns in design alternative II
Task Dispatcher	TTC SCHEDULER	TTC SCHEDULER
Task implementing control algorithm	PID CONTROLLER	PID CONTROLLER
Task recording Speed	PC LINK (RS-232)	PC LINK (RS-232)
Speed Sensor	SOFTWARE PULSE COUNT	HARDWARE PULSE COUNT

Table 6.4: Patterns in the original system and alternative system obtained by modifying the design of the sensor task

6.3.3 Results

The CCS systems developed from each of the designs were used for testing the desired behaviour of the control application. The test case required the car to begin cruising at 20 mph and accelerating to 60 mph after four minutes. The speed changes in the car were periodically logged to further analyse the results. Detailed results of each part of the case study are presented here.

6.3.3.1 Experimenting with alternative scheduler designs

In order to convert from a TTC SCHEDULER implementation to a TTC-ISR SCHEDULER, the timing of the original system is studied and mapped to the alternative system. The tick interval of the TTC SCHEDULER is retained in the alternative design. The scheduler task array created with the TTC SCHEDULER is analysed to identify timing parameters associated with the tasks in the system. The task periods are analysed and the major cycle of the new system is determined as the Least Common Multiple of the different task periods. With this information in place, the dispatcher is implemented by identifying tick intervals within the major cycle which are the desired points of execution of each of the tasks in the original system. The new design is compiled and executed to obtain a set of results for comparing both designs.

Figure 6.6 compares the response of the system to the desired speed behaviour requested of

the system. The systems implemented using different scheduling strategies, behave slightly differently to the same requested response. This may be due to the fact that in each tick interval, the TTC SCHEDULER utilises a short duration of time (at the start of the tick interval) to make the tasks expected to execute in the tick interval ready for execution. This 'house-keeping' element of the TTC SCHEDULER is required to implement a scheduler that is independent of the application logic. The TTC-ISR SCHEDULER has a simpler dispatch routine where the timing properties of the tasks (such as offset and period) are embedded into the scheduler logic.

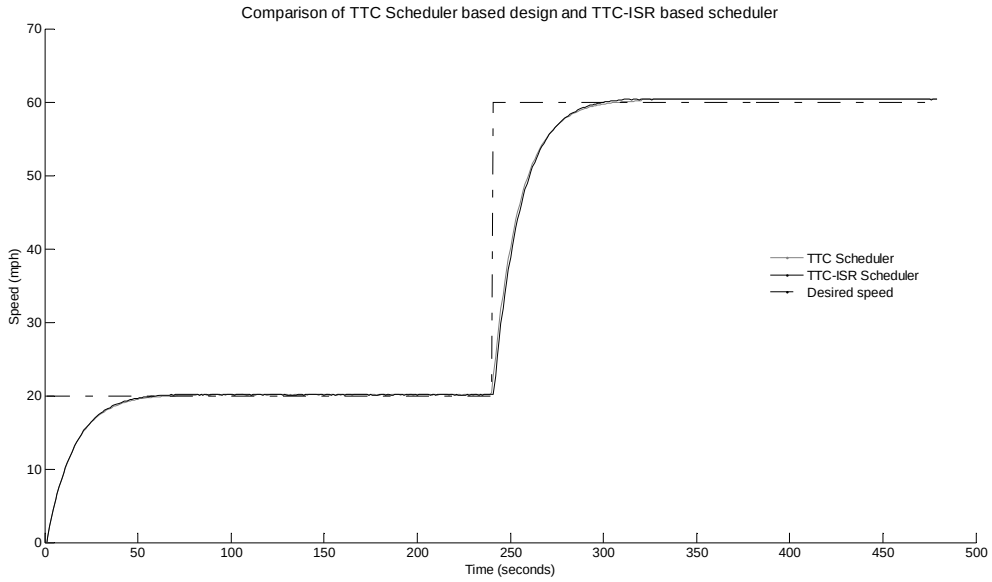


Figure 6.6: Speed comparisons - TTC SCHEDULER and TTC-ISR SCHEDULER implementations

6.3.3.2 Experimenting with alternative task designs

The second part of the case study was carried out to understand the process of obtaining a new design alternative by replacing one of the tasks in the system with an available design pattern alternative. The speed of the car is obtained as a bit stream to be processed by the CCS. This 1-bit encoded data needs to be converted to a suitable numerical value for further computation within the CCS application. When using a task design based on the SOFTWARE PULSE COUNT design pattern, two periodic tasks are needed to establish the speed of the car. One of the tasks is executed every tick interval and is needed to detect the pulses within the bit stream received from the car simulator and keep a count of the number of pulses received. A less frequent task, periodically processes this cumulated pulse count to compute the

average speed of the car. This information is used by the PID controller implemented within the CCS to determine the throttle required to minimize the error between the actual speed and the desired speed of the car.

In order to replace this task with the HARDWARE PULSE COUNT based design, it is important to have a free timer which has the capability of being implemented as a counter. The external pulses are provided as input to this counter and the counter internally keeps a track of the number of pulses encountered since it was last cleared or the last overflow. The design alternative is obtained by identifying the external pin associated with the Timer/Counter to be used. The speed pulse train from the car simulator needs to be physically connected to this input pin. The timer used to count the pulses needs to be configured in its counter mode. The frequent task used to detect pulses is no longer required. The less frequent task used to detect the cumulated pulse count essentially reads the count from the timer being used and re-initializes the timer for the next period.

Results from both the designs are shown in Figure 6.7.

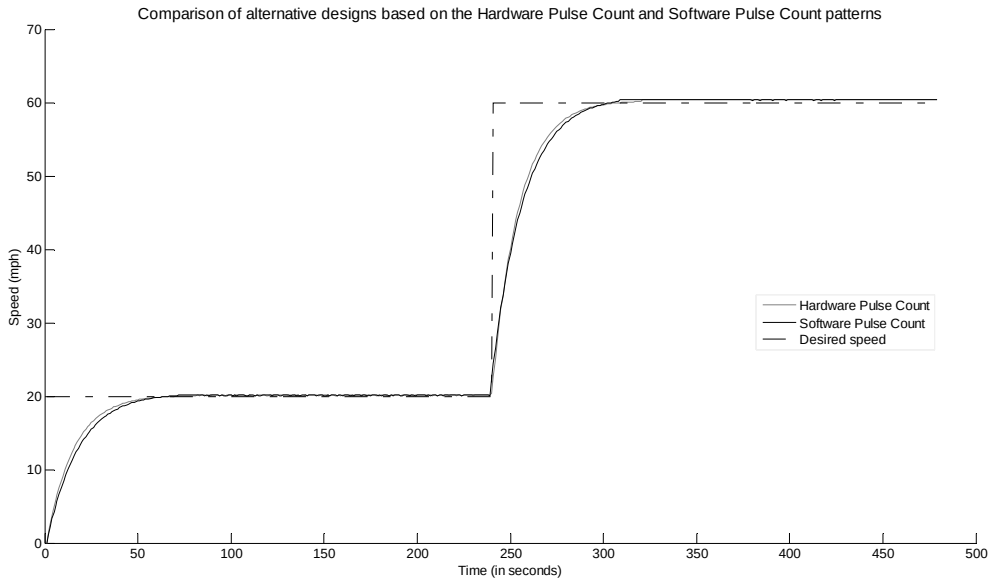


Figure 6.7: Graph showing desired speed, speed of system using different task designs

Both systems show nearly identical behaviour. Since the timer based counter is capable of detecting pulses at the rate of the processor clock frequency, there is much lesser margin for quantization errors. This is however not a significant issue in this case, because a task design

using a software pulse count, being scheduled on a one millisecond tick interval can detect pulses appearing at a rate of 500 pulses per millisecond. This translates to approximately 30,000 pulses per minute. For a car with a maximum speed of 180 km/h, this translates to a pulse to encode every 0.1 metre distance.

Such an alternative is best suited when more efficient resource availability is needed at any point in an application design. To implement the software based pulse counter requires creation of two periodic tasks. The task needed to detect pulses is a high frequency task and understandably less tolerant to execution delays. If the system redesign is necessitated on the basis of such resource contention, such a design alternative has special significance.

6.4. Discussion

The experiments and case study presented in this chapter attempted to highlight the potential of patterns for engineering processes beyond code generation. The patterns used to design a system give an indication of a design space suitable for search alternative design solutions to the same design problem at hand. By analysing the documentation associated with each of the patterns used in the system, it is possible to narrow down on possible alternative solutions to the system being designed or developed.

Just as in code generation, the PIEs played an important role in understanding the alternative solutions on offer and the best manner in which the new solutions can be adapted to the system being redesigned. This process of converting one pattern-based task design to another related pattern-based task design can also be documented within the pattern. Though this is presently unavailable within the pattern documentation, it will greatly enhance the pattern collection and individual patterns if such information is made available where suggestions of related pattern solutions or alternative pattern solutions are made available.

The focus of the experiment was to illustrate the use of information regarding pattern alternatives. The relevance of the experiments and case study presented here was primarily to understand the process of exchanging patterns to obtain alternative designs as also the need to analyse the result of such an exchange and appreciate the benefits of such a procedure. Although, the experiment may seem easy, there were certain thumb rules that needed to be followed while transforming an implementation based on design pattern X to that based on another design pattern Y. Such thumb rules are instrumental in understanding and realizing a

successful transformation.

The alternative design was not obtained in one straight-forward transformation. In the case of the scheduler exchange, the need to correctly associate the timing parameters in both design pattern alternatives was crucial to the success of the transformation. The TTC SCHEDULER provides a customisable scheduler queue. A practitioner wishing to create a schedule uses standard functions to build the schedule queue by specifying the name of the task and the period and offset of each task. However in a TTC-ISR SCHEDULER, the scheduler logic is interconnected with the application logic. Timing information is made available through a tick counter which keeps track of tick intervals. The transformation from one scheduler to another necessitated an understanding of how the individual period and offset of each task in the TTC SCHEDULER system correlated to the single tick count variable in the TTC-ISR SCHEDULER based design. In correctly associating the timing parameters in both designs would result in a system behaviour that was very obviously not correct. Similarly transforming a design based on the SOFTWARE PULSE COUNT pattern to the HARDWARE PULSE COUNT pattern, requires the availability of a free timer capable of working as a counter. The signal carrying the bit coded speed from the car simulator needed to be directed as an input to the timer configured to function in counter mode.

If the task set was not schedulable, the alternative design would not conform in behaviour with the original design. Such a case would provide other information that can be used to better define the transformation process. Scripts written to effect a pattern exchange as part of the experiment reported here provided an insight into the steps needed to swap one scheduler design implementation with that of another. In addition to that, it highlighted the different resource requirements of the scheduler designs which were studied.

Though the TTC SCHEDULER is resource intensive when compared to the TTC-ISR SCHEDULER, it is also more encapsulated. The TTC SCHEDULER design is independent of the application being developed and it is thus easier to improve the quality of the schedulers design independent of the application for which it is utilised. On the other hand the TTC-ISR SCHEDULER implementation includes some of the application logic within its implementation and it is thus difficult to ensure that the scheduler behaviour is unaffected by changes to the tasks in the system. An environment where the system is prototyped using a TTC SCHEDULER and eventually produced with a TTC-ISR SCHEDULER based design utilises the benefits that both designs have to offer.

The possibility of searching a design space supported by the wealth of domain/design knowledge captured in the pattern documentation was very useful given the need to generate and evaluate design options quickly. However, the pattern documentation should not and cannot be seen as the only source of information in this regard. It can potentially aid a streamlined and focussed approach to obtaining a design alternative. Given short turnaround times and strict project deadlines, this sort of information should potentially be beneficial to the process of exploring a set of design alternatives.

6.5. Conclusion

This chapter presented an empirical study that involved identifying PIE usage to understand system design. It also described experiments that involved swapping PIEs to obtain alternative system designs. The next chapter discusses the need to formalise pattern languages to use them more efficiently. Describes formalisation techniques adopted on the OO collection. It proceeds to suggest a formalisation technique for the PTTES collection.

7. Design patterns and Formal Representations

7.1. Introduction

The previous chapters introduced the PTTES collection and illustrated the use of these patterns in embedded software development. Chapter 3 included a discussion detailing the traditional approach to manually incorporating these patterns into a software project and Chapter 4 discussed the need to restructure the language to distinguish between the natures of information stored in the pattern documentation. It recognised the PIE as an active element of the PBSE approach. Chapter 5 discussed the need to extend tool support beyond code generation. Chapter 6 illustrated the potential of using design information to enrich PBSE. It detailed experimental results which involved the comparison of multiple designs obtained by using design alternatives suggested by pattern documentation. The need to use other design information (held rather informally as natural language descriptions) in a standard, tool-driven process is more challenging. This chapter discusses the need to formalise the pattern language to address these disparities. It provides a brief insight into some of the formalisation approaches used on the object-oriented pattern collection (Gamma et al. 1995). Through this discussion it attempts to obtain a mechanism to formalise the PTTES language.

7.2. The need for design pattern formalisation

Design patterns, as discussed in earlier chapters, provide an ideal mechanism to maintain documented domain expertise. However, much of this information is documented rather informally. Using a combination of textual information, diagrams and source code examples, the pattern documentation is intended to provide a well-tested solution to a design problem. This lack of formality can result in the pattern documentation being interpreted ambiguously and is definitely not suitable to be used directly in a more formal tool-assisted development process (Raje and Chinnasamy 2001; Mapelsden et al. 2002; France et al. 2004; Taibi and Taibi 2006). In some cases researchers seek to associate representations that capture the semantics of patterns in a language with which to analyse their behaviour (Mikkonen 1998).

The research presented in this chapter describes a formalisation approach intended to describe the PTTES language. The following sections briefly describe and analyse the approaches taken by some of the research projects that focus on formalizing the object-oriented pattern collection. Based on this understanding it suggests a technique to formalise the PTTES language.

7.3. Formalizing the object-oriented patterns

Researchers wishing to formalise the object-oriented patterns use one of two approaches. The formalisation strategies are either based on rigorous mathematical formulae (Mikkonen 1998; Eden 2000; Mak et al. 2003; Taibi and Taibi 2006) or the use of the Unified Modelling Language (Kim et al. 2002; Mapelsden et al. 2002) to represents patterns or pattern solutions. This section describes a set of representative approaches of each to understand the challenges and technique of formalisation in greater detail.

7.3.1 Formalisation based on mathematical foundations

The Language for Pattern Uniform Specification (LePUS) is a rigorous formal language used to specify design patterns. It uses Higher Order Monadic Logic and predicate calculus to specify design pattern leitmotifs. Leitmotifs are the recurring structure and behaviour of pattern solutions (Eden 2000). Design patterns are manifested as logic statements using LePUS formulae (Raje and Chinnasamy 2001). LePUS focuses on the structural aspects of pattern solutions and neglects other aspects such as intent and collaboration. The need to address these issues has resulted in the development of other formal representations based on LePUS – eLePUS (Raje and Chinnasamy 2001) and ExLePUS (Mak et al. 2003) to name a few. For instance eLePUS modifies the basic LePUS representation of patterns to support the availability of information such as pattern applicability.

The Distributed Co-operation (DisCo) approach is based on the reasoning that ambiguity in pattern application can be avoided by providing for a mechanism which supports rigorous analysis of the temporal behaviour of patterns. The formal basis of this approach lies in Temporal Logic of Actions (TLAs) and is primarily used to formalise the temporal behaviour of patterns. The Disco specification essentially defines a system (derived from a pattern). The developer can introduce classes, relations and actions to define the system. Classes, represented using formulae, define the forms of all possible objects. Relations associate these

objects with each other. Actions, seen as multi-object methods are atomic units of execution. An action representation includes a list of participants, parameters, enabling conditions and state-changes resulting from the execution of an action. Thus each DisCo specification is intended to describe the temporal behaviour of a closed system (Mikkonen 1998).

LePUS specifications, as described previously, capture the structural aspects of the design pattern with little emphasis on capturing the behavioural aspects of a pattern. DisCo on the other hand was designed to capture the temporal/behavioural aspects of a design pattern.

The Balanced Pattern Specific Language (BPSL) is an approach that considers both these aspects to obtain formal representations of design patterns (Taibi and Taibi 2006). Much like LePUS, it uses First-Order Logic (FOL) to specify the structural aspects of design patterns. The behavioural aspects are specified using TLAs in an approach similar to that used by DisCo. The BPSL approach is used to specify specific instances of the pattern solutions using the formulae used to describe a pattern with full and partial substitutions that identify that instance.

The Balanced Pattern Specification Language (BPSL) proposed by Taibi and colleagues (Taibi and Taibi 2006) is intended to specify both the structural aspects and behavioural aspects of patterns. It does this by using a suitable subset of First Order Logic (FOL) to formalise the structural aspects of a pattern solution. The structure of the design pattern solution, consists of variable symbols, connectives like Λ , quantifiers such as $\exists$ and predicate symbols that act upon the variable symbols. Variable symbols represent classes, attributes, methods, objects and untyped values (with the domain set of each of these entities designated as C, A, M, O, and V respectively). The predicate symbols are used to represent permanent relations. Examples of primary permanent relationships are derived directly from the object-oriented domain that the pattern collection targets. A set of them are captured in Table 7.1.

Relationship	Domain	Intent
Defined-in	MxC	Method is defined in a class
	AxC	Attribute is defined in a class
Reference-to-one (many)	CxC	First class defines a member whose type is a reference to one (many) instance(s) of the second class
Creation	CxC	One of the methods of the class contains an instruction that creates a new instance of another class
	MxC	Method contains an instruction that creates a new instance of a class
Inheritance	CxC	First class inherits from the seond
Invocation	MxM	First method invokes second method
	CxM	Reference to a class is an argument of a method
Argument	VxM	untyped value is argument of a Method
Instance	OxC	object is an instance of a class

Table 7.1: Primary permanent relationships(Based on information from Taibi and Taibi, 2006)

The formal representation of the behavioural aspects is achieved using Temporal Logic of Actions (TLAs). This is primarily used to formally capture the behaviour of the objects in a pattern solution. An infinite sequence of state (S_0, S_1, ...) defines the behaviours within a pattern solution. State variable values (class attributes) and temporal relations between objects constitute a state S_i in the behaviour. A temporal relation, defined as *TR(C₁[cardinality], C₂[cardinality]),* includes the name of the relation (*TR*), classes affected by the relation (C_1 and C_2) and the cardinality (i.e. the number of affected object instances) of the participating classes. The cardinality may be over all instances of the class (*[*]*) or within a closed interval of positive integers *a* and *b* (*[a..b]*).

Given this understanding of the manner in which temporal relations are represented, the behaviour of the pattern solution is captured as described. The system is expected to begin from an initial state S_0. Actions executed over time change the system state accordingly. Two consecutive states (S_i, S_{i+1}) in the behaviour constitute a transition. Within the domain of consideration, actions are considered to act non-deterministically with the restriction that the pre-condition for an action must be true for it to execute. The formal representation of an action includes a list of parameters (objects and untyped values), a precondition and a body for the action that needs to be executed when the precondition is true. By formally defining

an action as $A(o_1, o_2, p) : TR(o_1, o_2) \Lambda o_1.x \neq p \rightarrow -TR'(o_1, o_2) \Lambda o_1.x' = p$, Taibi and colleagues (Taibi and Taibi 2006) indicate that, given temporal relation *TR* between two classes C_1 and C_2, action *A* takes as parameters o_1 (object of C_1), o_2 (object of C_2) and un-typed variable *p*. The action *A* is executed when the precondition $TR(o_1, o_2) \Lambda o_1.x \neq p$ is satisfied and the action execution is essentially $-TR'(o_1, o_2) \Lambda o_1.x' = p$ which constitutes the body of the action. Un-primed and primed attributes refer to the values of the attributes before and after the execution of the action and so is the case with primed and un-primed temporal relations. Hence in semantic terms, an action is a Boolean expression that evaluates to true or false with regard to a pair of states. Temporal relations and actions are specific to the pattern being formalised.

Thus a BPSL representation for a pattern consists of four aspects. A declaration of the variables, followed by a set of permanent relations, followed by a set of temporal relations and the fourth aspect is a description of the actions. Taibi and colleagues (Taibi and Taibi 2006) illustrate the use of the above formal representation on the Observer pattern (Gamma et al. 1995). The Observer pattern alternatively known as Model-View-Controller pattern can be represented using UML as shown in Figure 7.1 and Figure 7.2.

Based on the class diagram shown in Figure 7.1, the variables in the pattern solution are described as shown in Listing 7.1.

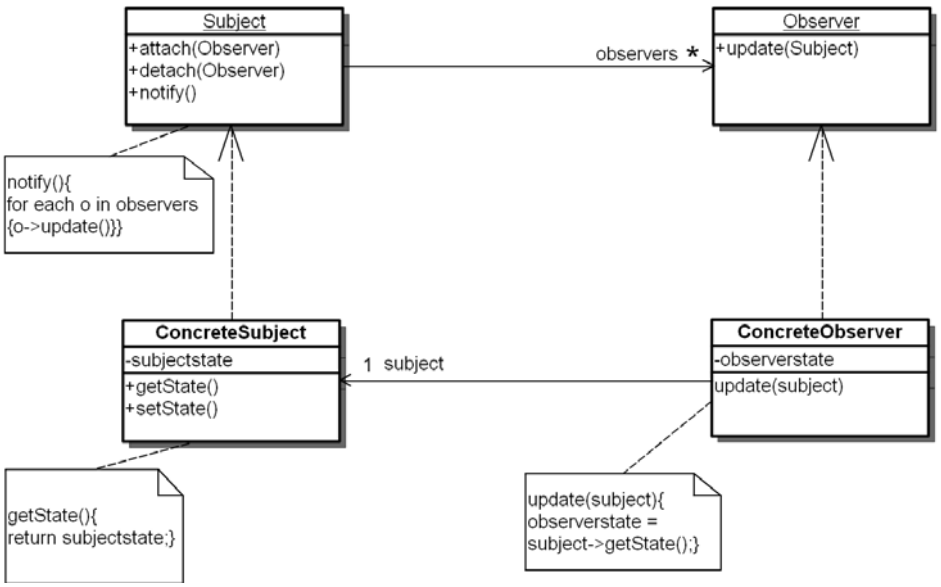


Figure 7.1: Class diagram of the Observer design pattern documented by Gamma and colleagues (Gamma et al. 1995)

$\exists$ subject, concrete-subject, observer, concrete-observer $\in C$
subject-state,observer-state $\in A;$
attach, detach, notify, get-state, set-state, update $\in M;$
subject, concrete-subject, object, concrete-object, concrete-observer $\in C;$
subject, concrete-subject, object, concrete-object, concrete-observer $\in O;$
subject, concrete-subject, object, concrete-object, concrete-observer $\in V;$

Listing 7.1: Variables identified from the class diagram

Permanent relations that take the above specified variables are described as shown in Listing 7.2.

Defined-in (subject-state, concrete-subject) Λ
Defined-in (observer-state, concrete-observer) Λ
Defined-in (attach, subject) Λ
Defined-in (detach, subject) Λ
Defined-in (notify, subject) Λ
Defined-in (set-state, concrete-subject) Λ
Defined-in (get-state, concrete-subject) Λ
Defined-in (update, observer) Λ
Reference-to-one (concrete-observer, concrete-subject) Λ
Reference-to-many (subject, observer) Λ
Inheritance (concrete-subject, subject) Λ
Inheritance (concrete-observer, observer) Λ
Invocation (set-state, notify) Λ
Invocation (notify, update) Λ
Invocation (update, get-state) Λ
Argument (observer, attach) Λ
Argument (observer, detach) Λ
Argument (subject, update) Λ
Instance (s, concrete-subject) Λ
Instance (o, concrete-observer).

Listing 7.2: Permanant relations in the formal representation of the Observer Pattern

The temporal relations represented in this pattern solution are captured in Listing 7.3.

Attached (concrete-subject [0..1], concrete-observer[*]) Λ
Updated (concrete-subject [0..1], concrete-observer[*])

Listing 7.3: Temporal relationships to capture behaviour

Finally the sequence diagram depicting the behaviour of the Observer pattern is represented using a set of actions as depicted in Listing 7.4.

Initially : $\neg$Attached(s, concrete-observer).
Attach (s,o) : $\neg$Attached(s, o) $\rightarrow$Attached'(s, o) $\vee$

Detach (s,o) : Attached(s, o) ∨ Updated(s,o)→Attached'(s, o) ∨
Notify (s,o,d) : Attached(s, o) ∨ Updated(s,o)→¬Updated'(s, concrete-observer)
s.subject-state' = d ∨
Updated*(s,o) : ¬Updated(s,o) → Updated'(s,o) Λ o.observer-state' = s.subject-state.

Listing 7.4: Representation of the behaviour captured from the sequence diagram

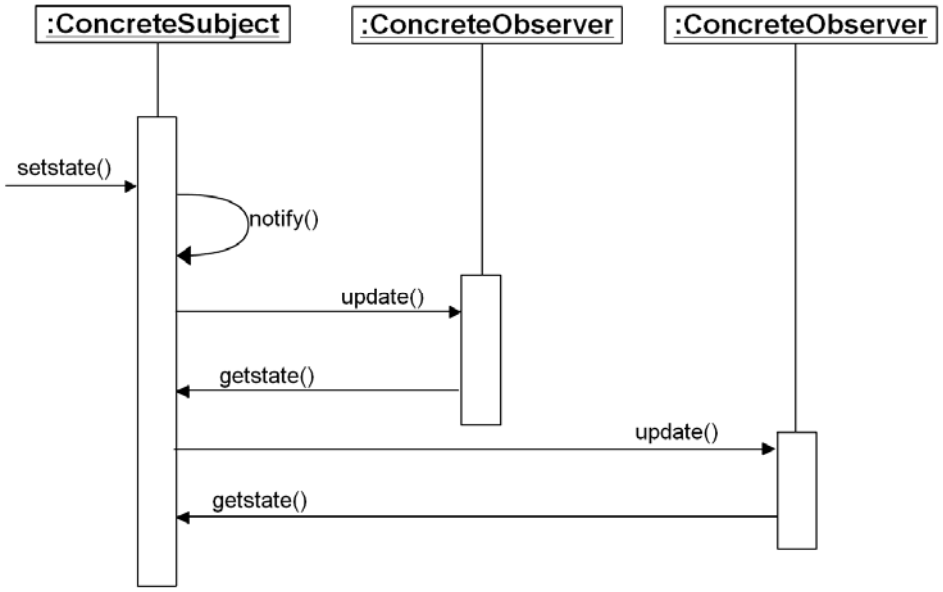


Figure 7.2: Sequence diagram indicating behaviour of the Observer design solution

The approaches described so far focussed on obtaining strong mathematical representations for the solutions documented as part of the object oriented patterns. These techniques addressed solutions to formalise both the structural aspects and the behavioural aspects of the pattern solutions. The structural aspects were formally described using propositional logic while the behavioural aspects were formally captured using TLAs. A BPSL representation seemed to offer an integrated framework with both structural and behavioural representations and was hence used to illustrate the application of these formalisation techniques. The next section describes formalisation techniques that use UML to model design patterns.

7.3.2 UML-based approaches to representing patterns

UML is a semi-formal design representation language (Mostafa et al. 2007). It is especially popular in the object-oriented industry and is used to detail design artefacts (Fowler 2003). Researchers looking to formalise design patterns have explored techniques to incorporate UML in the formalisation approach. Two such research projects have resulted in the creation

of the Design Pattern Mark-up Language (DPML) and the Role-Based Mark-up Language (RBML).

The Design Pattern Mark-up Language (DPML) defines a meta-model and notation for modelling pattern solutions and instances of pattern solutions. The language can be seen as a restrictive subset of UML. The design pattern specification model described by DPML is used to describe generalised structures of design patterns that are useful to the user. By identifying these structures in a general solution of the design pattern it is possible to obtain other diagrams in the object model. The design pattern described as before is instantiated on a need-basis while obtaining the object model of a particular project. The pattern instance is then suitably linked to the respective elements of the object model. This is the realisation phase. The object model, once realised in this manner, can be used in a normal UML tool-supported development process (Mapelsden et al. 2002). In many ways DPML attempts to ensure that valid design representations of the pattern solutions are used in the design phase rather than in the implementation phase

In the RBML approach, design patterns are represented as sub-languages of UML. The domain-specific design pattern is represented using specialised UML notation. In order to realise this RBML describes design patterns from different perspectives. It works on the meta-model level. RBML is used to specify a Role Model which in turn characterizes a family of UML diagrams. The set of UML diagrams obtained from an RBML specification are specialized versions of a particular kind of UML diagram (Kim et al. 2002; France et al. 2004). A domain-specific design pattern is then defined as a set of Role Models expressed using RBML, with each Role Model defining a sub language. It primarily uses two types of Role Models to define the subset of valid UML diagrams. These are:-

- Static Role Models (SRMs) – Characterize the static structures, i.e. models depicting classifiers (UML classes, interfaces) and relationships (associations and generalisations)
- Interaction Role Models (IRMs) – Used to characterize a family of interaction diagrams (collaboration and sequence diagrams)

RBML is thus used to model the structural and behavioural aspects of a particular domain (Kim et al. 2002).

7.4. An analysis of formalisation techniques

UML-based approaches identify valid subsets of UML diagrams to represent pattern-based design. They invariably involve some level of manipulation of the meta-model to describe the valid notations. For instance the Role-Based Meta-modelling Language (RBML) specifies patterns as a family of UML models. The specification supports a mechanism to generate UML models and check for the existence of patterns in a model. Similarly the Design Pattern Modelling Language (DPML) represents design pattern solutions using UML. DPML incorporates basic UML notations like interfaces, implementation, operations and methods, attributes, relations and constraints. Both these approaches use UML notations to represent pattern-based designs.

UML-based representations benefit from the use of standard notations (Kim et al. 2002; Mapelsden et al. 2002). This is especially useful when describing object-oriented patterns since UML is considered to be a de-facto standard in describing object-oriented design.

The UML is however a semi-formal language. Rigorously formal representations tend to have a mathematical base. For instance, LePUS addresses the incompleteness of diagrams and sample implementations by using mathematical formulae to represent the structure of a pattern solution. Though LePUS lacks support for the analysis of the behavioural aspect of patterns, subsequent formalisation techniques have addressed this shortcoming (Raje and Chinnasamy 2001; Taibi and Taibi 2006). Critics of the rigorously formal representation approach argue that using these approaches can be daunting for the average software engineer (Mapelsden et al. 2002) and requires a strong mathematical foundation for applying these techniques (France et al. 2004).

Either level of formality has its merits. Both approaches focus on formalizing the solution element of the pattern documentation. However, design patterns have much more information within the complete documentation. Though the pattern solution tends to be most directly useful in the application of a pattern, the other information elements are vital to understanding the domain of the application, the design problem at hand and the contributions made by a design pattern towards the realisation of a solution. The formalisation presented further in this thesis attempts to formally describe the pattern language rather than the individual solutions offered by the constituting patterns. Thus it attempts to describe a suitable manner of composing designs using the patterns in the pattern

language. Thus the manner in which the patterns are used (and consequently contribute to usage of the pattern language) forms the basis of the formalisation approach suggested here. The next section looks at existing standard practices in the use of the PTTES language that can aid the formalisation strategy.

7.5. A notation for formalizing the PTTES language

The formalisation approach described in this chapter acknowledges the fact that patterns in a pattern collection are intended to create a pattern language. When used with natural language, the pattern names contribute as nouns or phrases that domain-experts use to succinctly describe classic design problems or solutions in technical discussions or communications.

The technique adopted in the research presented here is based on the understanding that a pattern language is essentially an enriched natural language. Chomsky suggested Context Free Grammars (CFGs) as the basis for describing natural languages. Thus formalising the pattern language can be based on a context-free grammar (Greibach 1981). This pattern language is formally described using a context-free grammar (CFG). A CFG consists of a set of production rules that indicate how a valid sentence can be obtained starting from a start symbol (Aho et al. 1986). A CFG can be represented using a 4-tuple as G = (N, Σ, P, S) where

- N is a finite set of non-terminals
- Σ is a finite set of terminals which constitute the actual sentences formed using G
- P is the set of productions that define G and
- S is the start variable, starting from which the valid sentences of the language can be generated by applying suitable productions in a step-wise manner.

This section describes using the Backus-Naur Form (BNF) notation to represent the CFG used to describe PTTES language. BNF was originally designed as 'meta-linguistic formulae' to describe ALGOL 60 (Backus et al. 1960). The language description essentially consists of a set of production rules that can be used to generate a valid string in the language described by the BNF. Beginning with a start symbol, the production rules are used to replace this start symbol repeatedly in order to obtain the strings defined by the language. For instance, a BNF representation to describe a floating point number can be obtained using the rules of production described in Listing 7.5.

S:= ‘-’ F | F

F:= DS | DS ‘.’ DS

DS:= D | D DS

D:= ‘0’ | ‘1’ |‘2’ |‘3’ |‘4’ |‘5’ |‘6’ |‘7’ |‘8’ |‘9’

Listing 7.5: BNF production rules that describe a signed floating point number

Based on this representation, the BNF starts with a start symbol S. S can either be a negative or positive floating point number F. F consists of a digit sequence DS. F can either be an integer or a floating point number or integer depending on the use of a decimal point. The digit sequence DS consists of a series of digits D. D can be one of the 10 digits in the number system. S, F, DS and D constitute the symbols of the grammar and the digits from zero to nine enclosed within single quotes constitute the terminals of the grammar.

This representation however, includes a recursive production rule to indicate that numbers are essentially a sequence of digits. This recursion can be eliminated if an Extended BNF (EBNF) is used. Thus, the equivalent EBNF notation for describing a floating point number can be described as in Listing 7.6.

S:= ‘-’ $^{?}$ D^{+} (‘.’ D^{+})$^{?}$

D:= ‘0’ | ‘1’ |‘2’ |‘3’ |‘4’ |‘5’ |‘6’ |‘7’ |‘8’ |‘9’

Listing 7.6: EBNF production rules that describe a signed floating point number

EBNF uses certain standard operators to avoid recursive production rules. These are described in

?	Symbols or group of symbols operated upon is optional
*	Symbols or group of symbols operated upon can occur zero or more times in the production
+	Symbols or group of symbols operated upon can occur one or more times in the production

Table 7.2: Special operators in EBNF notation

When formally representing the PTTES language with a BNF notation, the design names of the design patterns constitute the terminals of the language. The rules of production need to be formulated in a manner that reflects the intuitive process of obtaining a design from a pattern collection. This can be best appreciated with an insight into the nearly formal processes that already exist in the use of patterns in the PTTES language. The next section gives an insight to these pseudo-formal elements of the PTTES language which can influence

the formalisation approach.

7.5.1 Pseudo-formalism in the PTTES language

The need to formalise the PTTES language stems from the fact that the pattern information is held in a human-readable, natural language format but is intended to be used through a more efficient and standard PBSE approach. In order to arrive at a formal representation for the pattern language it is important to appreciate the existing 'standards' in PTTES. These are listed below:-

- The pattern information is documented in a standard format. Thus the information in the pattern documentation is structured and classified (see Section 3.10)
- The traditional approach to using the patterns is fairly standard. (see Section 3.11)
- The automatic code generator also supports a standard workflow in its design (see Section 4.5.1)

7.5.2 Production rules for generating a PTTES-based system

The BNF rules of production for the PTTES language reflect the workflow described in Section 3.11. They intentionally reflect the pseudo-formal process (referred to in Section 7.5.1) of incorporating the PTTES patterns into a software project. In addition to the symbols used in the BNF representations of Listing 7.5, the representation presented here uses the following –

ε	The special symbol used to indicate an empty production. This is useful to terminate recursive production rules
< >	The angle brackets are used to distinguish symbols from terminals
“name”	A string to support a label in the production

Listing 7.7 details the set of BNF rules that provide a preliminary representation of the PTTES language. As required by the grammar, the first rule begins with a start symbol, the <system> node in this case, and progresses through a set of rules which identifies the other patterns that contribute as elements of the software design. For instance, rule 4 indicates that the software project consists of a scheduler, one or more pattern-based tasks and one or more user tasks.

Production rule	Rule #
<system> := Project_Name: *"name"* Hardware_platform: <microcontroller> Language: <programmingLanguage> Software_design: <software_project> End_Project	….(1)
<microcontroller> := AT89C55WD | LPC2106 | LPC2129	….(2)
<programmingLanguage> := C-Language | Assembly-Language	….(3)
<software_project> := <scheduler> + <pattern_based_tasks> + <user_tasks>	…..(4)
<scheduler> := <single_processor_scheduler> | <multiprocessor_scheduler>	…..(5)
<single_processor_scheduler> := ttc_scheduler | ttc_isr_scheduler | ttc_sl_scheduler | hybrid_scheduler	…..(6)
<pattern_based_tasks> := <pattern_based_task> + <pattern_based_tasks$_2$> | ε	…..(7)
<pattern_based_task> := <diagnostic_tasks> | <user_interface_tasks> | <control_tasks> | <monitoring_tasks> | <communication_tasks> | <delay_tasks>	…..(8)
<diagnostic_tasks> := <diagnostic_task> + <diagnostic_tasks$_2$> | ε	…..(9)
<diagnostic_task> := heartbeat_led | error_port	…..10)
<user_interface_tasks> := <user_input_tasks> + <user_output_tasks>	….(11)
<user_input_tasks> := <user_input_task> + <user_input_tasks$_2$> | ε	….(12)
<user_output_tasks> := <user_output_task> + <user_output_tasks$_2$> | ε	….(13)
<user_input_task> := switch_interface_sw | on_off_switch | multistate_switch | keypad_interface	….(14)
<user_output_task> := mx_led_display | lcd_character_panel | pc_link_rs232	….(15)
<control_tasks> := <control_task> + <control_tasks$_2$> | ε	….(16)
<control_task> := pid_controller	….(17)
<delay_tasks> := <delay_task> + <delay_tasks$_2$> | ε	….(18)
<delay_task> := hardware_delay | software_delay | sandwich_delay	….(19)
<communication_tasks> := <communication_task> + <communication_tasks$_2$> | ε	….(20)
<communication_task> := rs232_communication | i2c_peripheral | spi_peripheral	….(21)
<monitoring_tasks> := <monitoring_task> + <monitoring_tasks$_2$> | ε	….(22)
<monitoring_task> : = hardware_pulse_count | software_pulse_count | hardware_prm | software_prm | hardware_pwm | software_pwm | one_shot_adc | adc_preamp | sequential_adc | aa_filter |	….(23)

Production rule	Rule #
current_sensor | dac_output | dac_smoother | dac_driver | pwm_smoother | 3level_pwm	
<multiprocessor_scheduler> := scc_scheduler	….(24)
<user_tasks> := <user_task> + <user_tasks$_2$> | ε	….(25)
<user_task> := "*name*"	….(26)

Listing 7.7: Representing PTTES using a BNF like notation

The terminals of this grammar are the names of the design patterns catalogued as part of the PTTES language. The representation is intended to help a practitioner compose the design of a system by progressively working through these rules. Thus the grammar G defined by these rules provides a mechanism to specify a project design in terms of the patterns used in the design.

At this stage, only the design patterns that can be used to independently design tasks have been considered. Some of the design patterns excluded from the current representation are used in conjunction with these pattern-based tasks or user-tasks to extend the primary functionality of these tasks. A classic example is the use of patterns to design timeout loops for use within tasks implementing application functionality. At this stage, the BNF representation also excludes design patterns relevant to multiprocessor designs. The design space resulting from the use of a multiprocessor remains to be established and understood prior to incorporating rules aimed at multiprocessor designs. Also, user specified information such as project name and the names of the user tasks is not considered within this representation and hence these are identified by the special *"name"* symbol in the production rules.

7.6. Incorporating PIEs into the BNF notation

The rules described in Section 7.5 can be used to obtain the system design as a set of design patterns. In other words, the rules described earlier are useful in generating the system, by identifying a set of design patterns relevant to the application in consideration. The actual implementation of the system can then be obtained by using PIEs relevant to the design patterns identified as useful for a certain application. These rules are primarily intended to

navigate through the design space captured by the PTTES language.

However as observed earlier, the details in the pattern documentation are capable of augmenting the PBSE approach in methods more than mere system generation. PIEs hold a wealth of implementation specific information (see Section 5.3.2). Similarly information regarding pattern alternatives can be used to indicate design alternatives. This sort of information is context-sensitive. It contributes to the semantic aspects of the PTTES language. In order to support such semantics to language elements defined by the grammar G, there is a need to incorporate semantic associations with the rules of this grammar. This is where attribute grammars come in handy. Developed by Donald Knuth in 1968, attribute grammars formalise the semantics of context-free grammars (Knuth 1968).

By using a set of attributes to extend the context-sensitive nature of a CFG, the language representation can be made more general and in this case closer to the natural language being represented. In this approach, a finite set of attributes associated with the symbols in the grammar are used to augment it. These attributes are then suitably passed between elements of the parse tree to use and transfer context-sensitive information (Paakki 1995). Attributes can be simple data types or more complex data structures. Similarly attributes can be synthesised or inherited. When the value of an attribute at a node is computed from that of the children, it is referred to as a synthesised attribute. The value of an inherited attribute is computed either from the parent node or from the sibling nodes.

In order to augment the BNF grammar detailed in Listing 7.7, the design patterns that constitute the terminals of the language need to be associated with properties useful for analyzing a design. This approach takes inspiration from the concept of attribute grammars defined previously. For a start, it is important to understand the kind of properties that the pattern documentation can be expected to provide. It is also important to understand how these properties will be used for a design evaluation. Essentially, the properties associated with a design pattern are expected to gradually include context-sensitive information into the formal representation of the language. In order to be utilised effectively these properties need to be transferred from the design pattern/PIEs to the system using them. By associating the semantics of these properties with the rules of the BNF grammar specified previously, it is possible to introduce context-sensitive information into the grammar.

The PBSE approach described in this thesis is affected by implementation details such as the choice of the micro-controller used in the design and the language of implementation. This

information is known in the early stages of system production and needs to be made available in the later stages to select the appropriate PIE. In order to use this information, these properties need to be implemented in a manner similar to inherited attributes. The design space exploration techniques described in Chapter 6 benefit from resource-specific information to characterize a particular design and compare diverse design options. This necessitates the use of synthesised attributes to store resource-specific information such as memory requirements and hardware requirements. They cumulatively add to the total resource requirements of the system. Information regarding pattern alternatives is also vital to realising design diversity and is also a synthesised attribute passed to the start node. However unlike the resource attributes discussed earlier, the alternatives of the child node cannot be added to obtain the alternatives of the parent node. The combination of alternatives of the child nodes contribute to the alternatives available at a parent node. This behaviour needs to be defined in the rules that attach semantics to the BNF productions. Listing 7.7, Listing 7.8, Listing 7.9, Listing 7.10 and Listing 7.11 illustrate this approach of associating semantic information.

Rule #	Production rules	Semantic associations
1	<system> := Project_Name: <projectName> Hardware_platform:<microcontroller> Language: <programmingLanguage> Software_design: <software_project> End_Project	system.uC = microcontroller.value system.language= programmingLanguage.value software_project.uC = system.uC software_project.language = system.language system.patterns = software_project. patterns system.codeMemory = software_project.codeMemory system.dataMemory = software_project. dataMemory system.hardware = software_project.hardware system.globals= software_project.globals system.alternatives = software_project.alternatives
2	<microcontroller> := AT89C55WD | LPC2106 | LPC2129	microcontroller.value = AT89C55WD ***OR*** microcontroller.value = LPC2106 ***OR*** microcontroller.value = LPC2129
3	<programmingLanguage> := C-Language | Assembly-Language	programmingLanguage.value= C-Language ***OR*** programmingLanguage.value= Assembly-Language

Listing 7.8 Semantic associations to the BNF production rules for PTTES

Rule #	Production rules	Semantic associations
4	<software_project> := < scheduler> + <pattern_based_tasks> + <user_tasks>	scheduler.uC = software_project.uC scheduler.language = software_project.language pattern_based_tasks.uC = software_project.uC pattern_based_tasks. language = software_project. Language software_project.patterns = scheduler.patterns + pattern_based_tasks.patterns software_project.codeMemory = scheduler.codeMemory + pattern_based_tasks.codeMemory software_project. dataMemory = scheduler. dataMemory + pattern_based_tasks. dataMemory software_project. hardware = scheduler. hardware + pattern_based_tasks. Hardware software_project. globals = scheduler. globals + pattern_based_tasks. globals software_project. alternatives = scheduler. alternatives × pattern_based_tasks. alternatives
5	<scheduler> := <single_processor_scheduler> | <multiprocessor_scheduler>	single_processor_scheduler.uC = scheduler.uC single_processor_scheduler.language = scheduler.uC scheduler.patterns = single_processor_scheduler.patterns scheduler.codeMemory = single_processor_scheduler.codeMemory scheduler.dataMemory = single_processor_scheduler.dataMemory scheduler.hardware = single_processor_scheduler.hardware scheduler. globals = single_processor_scheduler. globals scheduler.alternatives = single_processor_scheduler.alternatives ***OR*** multiprocessor_scheduler.uC = scheduler.uC multiprocessor_scheduler.language = scheduler.uC scheduler.codeMemory.patterns = multiprocessor_scheduler.patterns scheduler.codeMemory = multiprocessor_scheduler.codeMemory scheduler.dataMemory = multiprocessor_scheduler.dataMemory scheduler.hardware = multiprocessor_scheduler.hardware scheduler. globals = multiprocessor_scheduler. globals scheduler.alternatives = multiprocessor_scheduler.alternatives

Listing 7.9: Semantic associations to the BNF production rules for PTTES (contd.)

Rule #	Production rules	Semantic associations
6	<single_processor_scheduler> := ttc_scheduler | ttc_isr_scheduler | ttc_sl_scheduler | hybrid_scheduler	/* <single_processor_scheduler> := ttc_scheduler */ *if* (single_processor_scheduler.uC == "8051" and single_processor_scheduler.language = "C") *then* ttc_scheduler.codeMemory = 578 ttc_scheduler.dataMemory = 24.1 *else if* (single_processor_scheduler.uC == "ARM" and single_processor_scheduler.language = "C") ttc_scheduler.codeMemory =522 ttc_scheduler.codeMemory = 24.1 *end* single_processor_scheduler.patterns = "TTC SCHEDULER" single_processor_scheduler.codeMemory = ttc_scheduler.codeMemory single_processor_scheduler.dataMemory = ttc_scheduler.dataMemory single_processor_scheduler.hardware = (timer2, timer) single_processor_scheduler.globals = (tick_interval, u_int8) single_processor_scheduler.alternatives = [ttc_isr_scheduler, ttc_sl_scheduler] ***OR*** /* <single_processor_scheduler> := ttc_sl_scheduler */ *if* (single_processor_scheduler.uC == "8051" and single_processor_scheduler.language = "C") *then* ttc_sl_scheduler.codeMemory = 57 ttc_sl_scheduler.dataMemory = 9.0 *else if* (single_processor_scheduler.uC == "ARM" and single_processor_scheduler.language = "C") ttc_sl_scheduler.codeMemory =60 ttc_sl_scheduler.codeMemory = 9.0 *end* single_processor_Scheduler.patterns = "TTC SCHEDULER" single_processor_scheduler.codeMemory = ttc_sl_scheduler.codeMemory single_processor_scheduler.dataMemory = ttc_sl_scheduler.dataMemory single_processor_scheduler.hardware = null single_processor_scheduler.globals = (tick_interval, u_int8) single_processor_scheduler.alternatives = [ttc_isr_scheduler, ttc_scheduler, ttc_sl_scheduler] ***OR ….*** */*similarly for other productions in this rule */*

Listing 7.10: Semantic associations to the BNF production rules for PTTES (contd.)

Rule #	Production rules	Semantic associations
7	<pattern_based_tasks> := <pattern_based_task> + <pattern_based_tasks$_2$> | ε	pattern_based_task.uC = pattern_based_tasks.uC pattern_based_task.language = pattern_based_tasks.language pattern_based_tasks$_2$.uC = pattern_based_tasks.uC pattern_based_tasks$_2$.language = pattern_based_tasks.language pattern_based_tasks.patterns = pattern_based_task.patterns + pattern_based_tasks$_2$. patterns pattern_based_tasks.codeMemory = pattern_based_task.codeMemory + pattern_based_tasks$_2$. codeMemory pattern_based_tasks.dataMemory = pattern_based_task.dataMemory + pattern_based_tasks$_2$. dataMemory pattern_based_tasks.hardware = pattern_based_task.hardware + pattern_based_tasks$_2$.hardware pattern_based_tasks.globals = pattern_based_task.globals + pattern_based_tasks$_2$.globals pattern_based_tasks.alternatives = pattern_based_task.alternatives × pattern_based_tasks$_2$.alternatives ***OR*** pattern_based_tasks.patterns = null pattern_based_tasks.codeMemory = 0 pattern_based_tasks.dataMemory = 0 pattern_based_tasks.hardware = null pattern_based_tasks.globals = null pattern_based_tasks.alternatives = null
	patterns(<system>)	"Patterns in system = " return (system.patterns)
	alternatives(<system>)	for each combination in (system.alternatives) return "Patterns in alternative system = " [the combination] Endfor excluding patterns(<system>)
	resources(system)	"Code memory usage =" return (system.codeMemory) "Data memory usage =" return (system.dataMemory) "Hardware required =" return (system.hardware)

Listing 7.11: Semantic associations to the BNF production rules for PTTES (contd.)

7.7. The heartbeat LED example

The previous section described the use of BNF to formally describe the PTTES language. As observed earlier in this chapter, one of the key considerations in choosing this technique was the fact that pattern documentation is originally made available in a human-readable format for practitioners wishing to use the patterns through the traditional, manual approach of applying patterns to a project. The formalisation technique suggested here keeps this enriched language as the basis on which the abstractions described above are made.

This section illustrates the use of these techniques in the creation of a simple "Heartbeat LED" example. The Heartbeat LED project requires the embedded system to flash an LED ON and OFF at a periodic rate, and is used in many systems to provide a simple visual indication of the system state ("alive" or "dead").

Following through the rules detailed in Listing 7.8 to Listing 7.11, the system begins with the <system> symbol.

```
<system> := Project_Name: <projectName>
                    Hardware_platform: <microcontroller>
                    Language: <programmingLanguage>
                    Software_design:
                    <software_project>
            End_Project
```

The project is to be called Heartbeat and needs to be implemented on an AT89C55 chip using C-language. This information is useful in obtaining the next production rule.

```
<system> := Project_Name: Heartbeat
                    Hardware_platform: AT89C55WD
                    Language: C-Language
                    Software_design:
                    <scheduler> + <pattern_based_tasks>+ +
                    <user_tasks>*
            End_Project
```

The next production rule steps into specification of the software design. Since the system is to be developed on a single processor we have the following production described below

```
<system> := Project_Name: Heartbeat
                    Hardware_platform: AT89C55WD
                    Language: C-Language
                    Software_design:
                    <single_processor_scheduler> +
                    <pattern_based_task> + <pattern_based_tasks> *
            End_Project
```

After identifying the project as a single processor design, the next step is to specify the actual scheduler that the design proposes to use. By specifying the TTC SCHEDULER as the chosen design pattern, the production takes the form shown below.

```
<system> := Project_Name: Blinky
                    Hardware_platform: AT89C55WD
                    Language: C-Language
                    Software_design:
                    ttc_scheduler + <diagnostic_tasks>
                    |<diagnostic_tasks>* + <pattern_based_tasks> *
              End_Project
```

Since, the HEARTBEAT LED is essentially a diagnostic pattern; the production rule takes the form shown below.

```
<system> := Project_Name: Blinky
                    Hardware_platform: AT89C55WD
                    Language: C-Language
                    Software_design:
                    ttc_scheduler + <diagnostic_task> |
                    <pattern_based_tasks> *
              End_Project
```

The productions are terminated when the application has been specified through these rules.

```
<system> := Project_Name: Blinky
                    Hardware_platform: AT89C55WD
                    Language: C-Language
                    Software_design:
                    ttc_scheduler + heartbeat_led
              End_Project
```

In its final state, the production contains details such as the name of the project, the implementation details, the scheduler used by the system, the patterns used in the system and user tasks if any. This simple project has a single pattern-based task and does not require any additional application logic. Hence it does not include a user task.

7.8. Discussion

The formalisation approach described here uses a set of production rules to describe the composition of patterns to obtain the pattern-based design of a system. Since pattern languages, closely resemble natural languages the formal representation described here uses a BNF notation to describe the grammar of this language.

The rules detailed here are not an exhaustive representation of the complete pattern language. Only a subset of the patterns in PTTES is incorporated in the rules. The set of patterns used in the representation can be incorporated into the software development process in a more-or-less standard manner. Context-sensitive information specific to each pattern in the pattern representation is incorporated by associating properties to the terminals and symbols of the grammar. These properties are utilised much like attribute grammars in a context-sensitive grammar definition.

Though the rules of production detailed so far are specific to the PTTES language, the basic approach of using a BNF-based approach to formally represent the pattern language is expected to be applicable to other pattern languages as well. For instance Listing 7.12, describes the production rules that can be used to represent Alexander's pattern language.

Alexander categorises his patterns as those used for towns, buildings and construction. The "construction patterns" indicate the actual process of realising a build. Each construction pattern in turn refers to patterns from the other two categories as and when needed. Thus the pattern names enrich the pattern language and seamlessly fit into the vocabulary of the practitioner. The construction patterns and documentation give an indication to the BNF rules for a new build.

The rules of production begin by establishing a philosophy of structure on which to base the building design and construction. Two patterns that can be used in this step are - structure FOLLOWS PHYSICAL SPACES, EFFICIENT STRUCTURE (to complement the previous pattern).

In addition to establishing the philosophy of the structure, the materials to be used in the new building needs to be made. The choice of materials may include considerations such as the eco-friendly nature of the building (for instance, GOOD MATERIALS).

```
new_building ::= <philosophy_of_structure> +
      <materials_to_be_used> + <philosophy_of_construction>
      + <structural_layout> + <actual_construction>

<philosophy of structure> ::= <STRUCTURE FOLLOWS PHYSICAL SPACES>
      + <EFFICIENT STRUCTURE>

<materials_to_be_used> ::= <GOOD MATERIALS>

<philosophy_of_construction> ::= <GRADUAL STIFFENING>

<structural_layout> ::= <roof_layout> +
      <floor_and_ceiling_layout> +
      <thickening_the_outer_walls> +
      <columns_at_the_corners> +
      <final_column_distribution>

<actual_construction> ::= <mark_columns>+
      <position_openings> + <main_frame> + <surfaces> +
      <indoor_details> + <outdoor_details> +
      <ornamentation>


<ROOF_LAYOUT> ::= Alternatives:
                    <CASCADE OF ROOFS>
                    <SHELTERING ROOFS>
                    <ROOF GARDEN>
```

```
<FLOOR_AND_CEILING_LAYOUT> ::= Alternatives:
                        <WINDOW_PLACE>
                        <FARMHOUSE KITCHEN>
                        <COMMON AREAS AT THE HEART>
                        <ROOF VAULT>
                        <FLOOR-CEILING VAULTS>
                        <FLOOR SURFACE>

<mark_columns>::= <ROOT FOUNDATIONS>|<GROUND FLOOR SLAB>|<BOX
       COLUMNS> |<PERIMETER BEAMS>|<WALL MEMBRANES>|<FLOOR-CEILING
       VAULTS>|<ROOF VAULTS>

<position_openings> ::= <NATURAL DOORS AND WINDOWS>|<LOW
       SILL>|<DEEP REVEALS>|<LOW DOORWAY>|<FRAMES AS THICKENED EDGES>

<main_frame> ::= <COLUMN PLACE>|<COLUMN CONNECTION>|<STAIN
       VAULT>|<DUCT SPACE>|<RADIANT HEAT>|<DORMER WINDOWS>|<ROOF
       CAPS>

<surfaces> ::= <FLOOR SURFACE>|<LAPPED OUTSIDE WALLS>| <SOFT INSIDE
       WALLS> | <WINDOWS WHICH OPEN WIDE | SOLID DOORS WITH GLASS

<indoor_details> ::= <FILTERED LIGHT>|<SMALL PANES>|<HALF-INCH
       TRIM>

<outdoor_details> ::= <SEAT SPOTS>|<FRONT DOOR BENCH>|<SITTING
       WALL>|<CANVAS ROOFS>|<RAISED FLOWERS>|<CLIMBING
       PLANTS>|<PAVING WITH CRACKS BETWEEN THE STONES>|<SOFT TILE AND
       BRICK>

<ornamentation>::= <ORNAMENT>|<WARM COLOURS>|<DIFFERENT
       CHAINS>|<POOLS OF LIGHT>|<THINGS FROM YOUR LIFE>
```

Listing 7.12: BNF rules of production for the “original” pattern language (Alexander et al. 1977)

The philosophy of construction is another element that defines the new building. The pattern GRADUAL STIFFENING, describes one such approach where the new building is initially constructed to be loose and flimsy while final adaptations to the plan are made. With the

philosophy of structure in place the structural layout needs to be identified. The structural layout encompasses details of roof, floor, ceiling layouts and those of other structures of the building. The structural layout sets the stage for the actual construction. This step can be further elaborated to detail the next level of activity. The approach described here uses a simple BNF-based approach to represent the process of using Alexander's pattern to describe the creation of a new building. Alexander's pattern language contains many more patterns and describes the creation of many other living spaces.

7.9. Conclusion

The next chapter details out a case study which uses the techniques described in Sections 7.5 and 7.6 to obtain the design of an embedded system. A preliminary design of the system detailed as a composition of a set of design patterns is obtained using the rules of production described earlier. The semantic information captured using the properties of the tokens in the grammar is used to obtain a set of design alternatives.

8. Case study

8.1. Introduction

Chapter 7 described an EBNF-based approach of formalising the PTTES language. This chapter uses a case study – an elevator system to illustrate the use of the EBNF rules described earlier. Alternative designs of the elevator system are obtained using these rules and the semantic association of specified for these rules. The embedded software systems based on each design alternative are developed using suitable PIEs corresponding to the design patterns that constitute the system design. The behaviour of the alternative implementations is then analysed to evaluate the designs and the approach.

8.2. The elevator test bed

The elevator control panel is an example of an embedded system that benefits from reliable operation. The test bed used in this case study is an Elevator 34-150 from Feedback Instruments Ltd (webpage: Feedback Instruments Ltd. 2009).

This elevator model has four floors and an elevator car that moves between these floors. The elevator control software receives input signals to direct the car up at the first floor, up/down at the second and third floor and signals to move the car down at the fourth floor. The movement can also be controlled through a similar interface (usually placed in the car) which takes floor requests as inputs. For convenience this interface is provided on the body of the model shown in Figure 8.1.

The software controls the motion of the car through a series of output signals. The car is held at a floor by a brake mechanism and motorised doors open and close the elevator door if a request for entry/exit has been placed at a floor. The door and brake control signals are most obvious. Additionally a set of LEDs and buzzers realise the complete

functioning of the system. Some of the visible input and output sensors are labelled on the elevator shown in Figure 8.1.

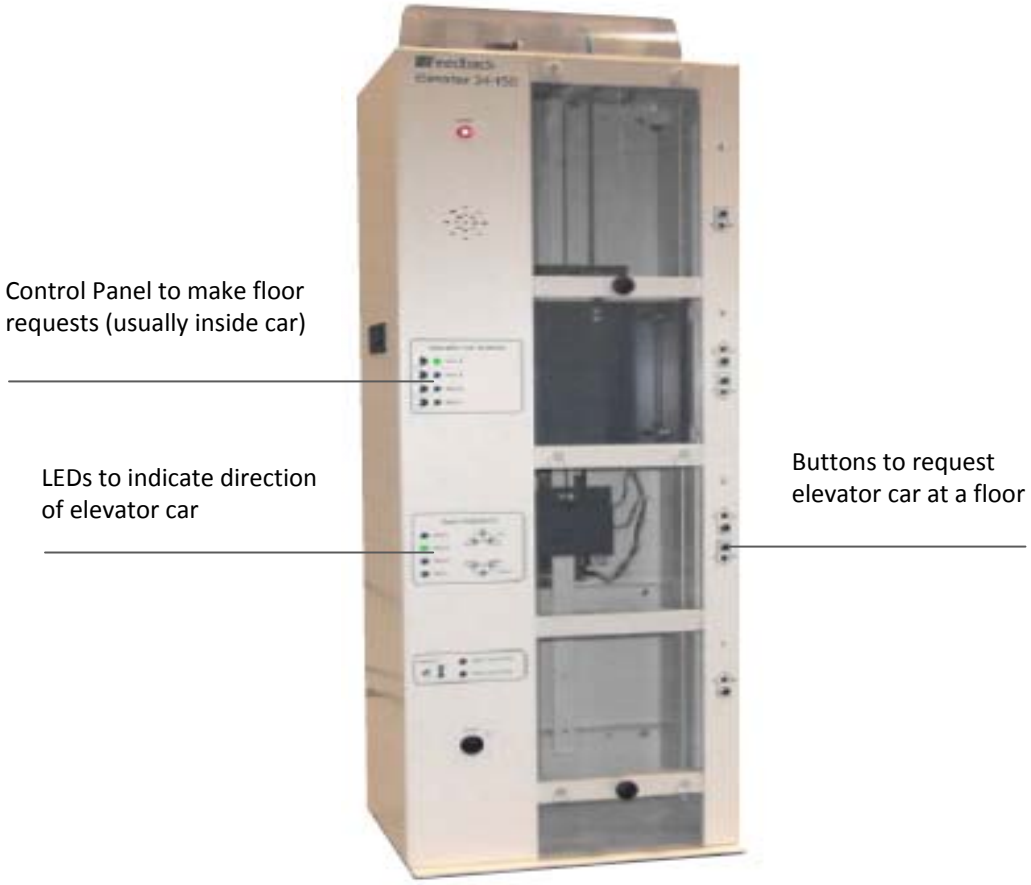


Figure 8.1: The elevator model used for the case study

8.3. Designing the embedded system

The embedded system in this case is the elevator control software which interfaces with the model shown in Figure 8.1 in order to realise the desired operational behaviour of an elevator. This section describes the design approach and illustrates the use of the EBNF rules described in Chapter 7.

8.3.1 Identifying the input and output signals to the system

The elevator control system receives input signals from the model elevator and responds accordingly. It achieves the desired elevator behaviour by sending a set of output signals that

control the functioning of the motor and brake mechanism as also the controls of the elevator car door. These input and output signals are tabulated in Table 8.1.

Input sources for the elevator control system	Output signals sent to the elevator model
Floor 1 request (inside the car)	Brake control
Floor 2 request (inside the car)	Motor control
Floor 3 request (inside the car)	Control direction of motion
Floor 4 request (inside the car)	Open car door
Up request at floor 1 (outside car)	Close car door
Up request at floor 2 (outside car)	Floor 1-4 request LED (inside car)
Down request at floor 2 (outside car)	Floor 1-4 request LED (outside car)
Up request at floor 3 (outside car)	
Down request at floor 3 (outside car)	
Down request at floor 4 (outside car)	
Elevator car door open	
Elevator car door closed	
Elevator car position(analogue)	

Table 8.1: The elevator control system – input and out signals

8.3.2 Deriving the system design

Using the BNF rules of production detailed in Section 7.5, the original system is generated as follows. Beginning with the start symbol, the project name and specifics are captured. The software design requires a suitable scheduler and relies on the use of the ONE-SHOT ADC design pattern to use the ADC component which indicates position of the elevator cab.

As described in Section 7.7, the original design of the elevator system is obtained by following the rules of production specified in Listing 7.7. The project begins as a <system> using the first production rule.

```
<system> := Project_Name: <projectName>
                    Hardware_platform: <microcontroller>
                    Language: <programmingLanguage>
                    Software_design:
                    <software_project>
            End_Project
```

To create a project called ‘Elevator’ on an ARM processor, the production takes the new form shown below. This production also indicates that the system is to be implemented using C-Language.

```
<system> := Project_Name: Elevator
                    Hardware_platform: ARM
                    Language: C-Language
                    Software_design:
                    <scheduler> + <pattern_based_tasks> +
                    <user_tasks>
            End_Project
```

The production is further developed to specify a requirement of using a single processor design. The system may contain user tasks and pattern-based tasks. The application logic is built around the pattern-based tasks. The system needs a user task to link the functioning of the pattern-based tasks when realising the final system . The production rule incorporates this by elaborating the <user_task> symbol. Thus the production takes the form shown below.

```
<system> := Project_Name: Elevator
                    Hardware_platform: ARM
                    Language: C-Language
                    Software_design:
                    <single_processor_scheduler> +
                    <pattern_based_task> + <pattern_based_tasks2> | ε
                    + ut:Elevator_State_Machine
            End_Project
```

With the TTC SCHEDULER specified as the single processor scheduler chosen to implement the design, the production rule transforms as shown below. The elevator control software also responds to button presses from inside the elevator car and from the floors serviced by the elevator. Since, the pattern collection includes a selection of patterns that explain the design of a switch read task, a pattern can be used for the user input.

```
<system> := Project_Name: Elevator
                Hardware_platform: ARM
                Language: C-Language
                Software_design:
                ttc_scheduler + <user_interface_tasks> +
                <pattern_based_task> + <pattern_based_tasks2> | ε
                + ut:Elevator_State_Machine
            End_Project
```

As described earlier, an ADC task is needed to establish the position of the cab. The PTTES language supports patterns to assist with using ADCs in the design of monitoring tasks. With this information, the production rule takes the new form shown next.

```
<system> := Project_Name: Elevator
                Hardware_platform: ARM
                Language: C-Language
                Software_design:
                ttc_scheduler + <user_interface_task> +
                <user_interface_tasks> | ε + <monitoring_tasks> |
                <pattern_based_tasks2> | ε +
                ut:Elevator_State_Machine
            End_Project
```

```
<system> := Project_Name: Elevator
                Hardware_platform: ARM
                Language: C-Language
                Software_design:
                ttc_scheduler + <user_input_task>
                <monitoring_task> + <monitoring_tasks> | ε +
                ut:Elevator_State_Machine
            End_Project
```

If the first design opts to use the ONE-SHOT ADC design pattern and the SWITCH INTERFACE patterns, the design of the system can be represented using the following production rule.

This final production, with all symbols elaborated, is essentially a representation of the first design.

<system> := Project_Name: *Elevator*
Hardware_platform: *ARM*
Language: *C-Language*
Software_design:
ttc_scheduler + *switch_interface* + *one_shot_adc*
+ *ut:Elevator_State_Machine*
End_Project

In order to evaluate this design for design alternatives, the semantic associations for each production rule used in deriving this design is re-visited. The following semantic association are useful to identify the patterns in the system.

"Patterns in system = " return (software_project.patterns)

In order to obtain the value of the patterns property of the software project, the patterns property associated with a terminal or non-terminal of the grammar is inherited by the software_project node of the <system>. In this case, the patterns are identified as follows

system.patterns = software_project.patterns
software_project.patterns = scheduler.patterns + pattern_based_tasks.patterns
scheduler.patterns = single_processor_scheduler.patterns
pattern_based_tasks.patterns = pattern_based_task.patterns + pattern_based_tasks2. patterns
pattern_based_tasks.patterns = user_input_task.patterns + monitoring_task.patterns
user_input_task.patterns ="SWITCH INTERFACE"
monitoring_task.patterns = "ONE-SHOT ADC TASK"
single_processor_scheduler.patterns = "TTC SCHEDULER"

patterns(<system>)

Patterns in system = *ttc_scheduler* + *switch_interface* + *one_shot_adc*

Similarly, a set of alternative designs can be deduced from the production rules by using similar semantic associations. The altenatives to a design are derived from the following rule:

for each combination in (system.alternatives)
return
"Patterns in alternative system = " [the combination]
endfor

excluding patterns(<system>)

system.alternatives = software_project.alternatives
software_project. alternatives = scheduler. alternatives × pattern_based_tasks. alternatives
scheduler.alternatives = single_processor_scheduler.alternatives

When the non-terminal, single_processor_scheduler produces ttc_scheduler

single_processor_scheduler.alternatives = [ttc_isr_scheduler, ttc_sl_scheduler, ttc_scheduler]

Similarly pattern_based_tasks.alternatives are obtained through the following semantic associations

pattern_based_tasks.alternatives = pattern_based_task.alternatives × pattern_based_tasks$_2$.alternatives

pattern_based_tasks.alternatives = user_input_task.alternatives × pattern_based_task.alternatives × pattern_based_tasks$_2$.alternatives

pattern_based_tasks.alternatives = user_input_task.alternatives × monitoring_task.alternatives × null

When the token - user_input_task produces *switch_interface,* the alternative patterns suggested for the SWITCH INTERFACE pattern are used. Since the Switch Interface pattern lacks alternatives, user_input_task.alternatives = [switch_interface] in this case.

Similarly, when monitoring_task produced One-Shot ADC, the Hardware Pulse Count is a suggested alternative. Thus monitoring_task.alternatives =[one_shot_adc, hardware_pulse_count]

With these three sets of patterns to consider we have the following combinations

ttc_scheduler + switch_interface + one_shot_adc
ttc_sl_scheduler + switch_interface + one_shot_adc
ttc_isr_scheduler + switch_interface + one_shot_adc
ttc_scheduler + switch_interface + hardware_pulse_count
ttc_sl_scheduler + switch_interface + hardware_pulse_count
ttc_isr_scheduler + switch_interface + hardware_pulse_count

excluding the patterns in the system, the available alternatives are depicted below.

alternatives(<system>)

Patterns in alternative system = *ttc_isr_scheduler* + *switch_interface* + one*_shot_adc*
Patterns in alternative system = *ttc_sl_scheduler* + *switch_interface* + one*_shot_adc*
Patterns in alternative system = *ttc_scheduler* + *switch_interface* + hardware*_pulse_count*
Patterns in alternative system = *ttc_isr_scheduler* + *switch_interface* + hardware*_pulse_count*
Patterns in alternative system = *ttc_sl_scheduler* + *switch_interface* + hardware*_pulse_count*

Thus the original system is designed using the TTC SCHEDULER pattern, SWITCH INTERFACE pattern and the ONE-SHOT ADC pattern. System alternatives are obtained by establishing the system properties from the details of the patterns used in the design. The combinations of the alternatives suggested for each of the design patterns that constitute this system indicate the available system variations that can be obtained from the original set of patterns. This case study uses one other alternative derived from the original system. It compares a system

designed using the TTC SCHEDULER pattern, SWITCH INTERFACE pattern and the ONE-SHOT ADC pattern with that designed using the TTC-ISR SCHEDULER pattern, SWITCH INTERFACE pattern and the ONE-SHOT ADC pattern.

8.4. Designing and executing the test case

To test the functioning of the elevator system, a sequence of floor requests is made to the elevator control system. In order to study the functioning of the system, suitable timing information is extracted from the elevator control system. Manually entering floor requests in the desired sequence exposes the system to large timing discrepancies. To obtain accurate timing information corresponding to the behaviour of these designs the test-case is designed to automate the floor requests. This is achieved by using a batch file which makes the floor requests over a period of time. The test case is designed to incorporate time delays between floor requests to simulate a real environment. The cross-compilation environment is suitably modified to support this automated test-case. The floor requests from the test-case are fed directly through to the input and output pins of the embedded development board through the parallel port. Thus the input panels in the elevator are effectively bypassed.

Depending on the floor being requested an eight bit code is sent to the microcontroller through the parallel port. The time at which the floor request is made and the time at which the elevator car arrives at the floor is logged to allow analysis of the alternative implementations. The same sequence of floor requests is made on each design alternative. The case study is designed to execute five runs on each alternative implementation and to use the average timing information for further comparison between design alternatives.

8.5. Results

The alternative systems were built using different scheduler designs – one based on the TTC SCHEDULER design pattern and the other using the TTC-ISR SCHEDULER design pattern. The test case was executed on each design and the results are plotted as shown in Figure 8.2. This data was used to ascertain if both designs exhibited comparable behaviour over the period of the simulated test case. From the figure, we notice that though implementations obtained from both designs do not exhibit identical behaviour, they exhibit comparable responses to the floor requests made in described in Section 8.4.

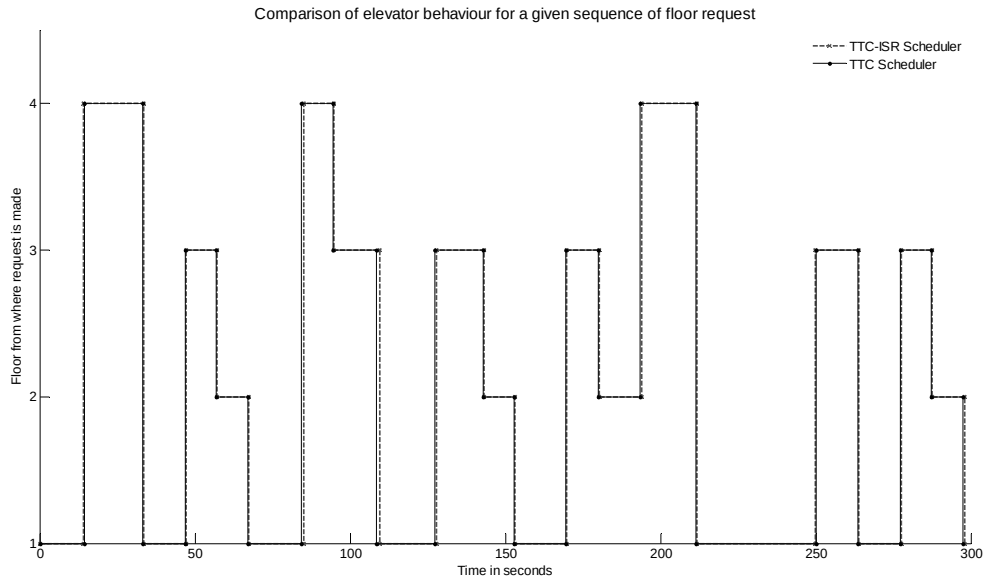


Figure 8.2: Graph comparing timing behaviour of TTC Scheduler and TTC-ISR Scheduler based designs

In order to compare both designs further, the response times from the test results obtained from each execution of the test case was further analysed. The response time was computed as the time difference between the time of request for a floor and the time of actual arrival at the floor. Table 8.2 details the response time computations for each design. Further analysis of this response time gives an indication of the behaviour of one implementation with regards to the other.

Table 8.2 compares the difference in the response times of each of the implementations. From the results tabulated, there is no clear indication of any one implementation being consistently faster than the other. However, in most requests the response of the TTC-ISR SCHEDULER based design is faster. This may be due to the fact that the dispatcher based on a TTC SCHEDULER has a more complex implementation to support the flexibility that this scheduler offers.

TTC SCHEDULER Design			
Floor	**Time request made (secs.)**	**Time of arrival at floor (secs.)**	**Response time (secs.)**
4	0	14.4	14.4
1	19.3	13.87	13.87
3	22.83	23.97	23.97
2	31.86	24.99	24.99
1	45.89	21.07	21.07
4	69.92	14.41	14.41
3	78.96	15.46	15.46
1	93	15.38	15.38
3	117.04	10.34	10.34
2	136.06	6.76	6.76
1	145.1	7.85	7.85
3	159.13	10.34	10.34
2	173.16	6.74	6.74
4	177.19	16.31	16.31
1	186.23	25.08	25.08
3	239.31	10.79	10.79
1	243.33	20.25	20.25
3	262.37	14.85	14.85
2	276.39	10.89	10.89
1	280.43	16.95	16.95

TTC-ISR SCHEDULER Design			
Floor	**Time request made (secs.)**	**Time of arrival at floor (secs.)**	**Response time (secs.)**
4	0	14.07	14.07
1	19.03	33.52	14.49
3	23.08	47.17	24.09
2	32.11	57.22	25.11
1	46.15	67.3	21.15
4	70.19	85.11	14.92
3	79.23	94.75	15.52
1	93.25	109.51	16.26
3	117.29	127.81	10.52
2	136.31	143.06	6.75
1	145.35	153.15	7.8
3	159.38	169.71	10.33
2	173.41	180.17	6.76
4	177.46	193.82	16.36
1	186.49	211.61	25.12
3	239.56	249.89	10.33
1	243.58	263.84	20.26
3	262.71	277.49	14.78
2	277.01	287.55	10.54
1	281.31	297.91	16.6

Table 8.2: Response times for each floor request (TTC Scheduler Vs TTC-ISR scheduler)

Figure 8.3 captures this difference in the response time offered by both implementations.

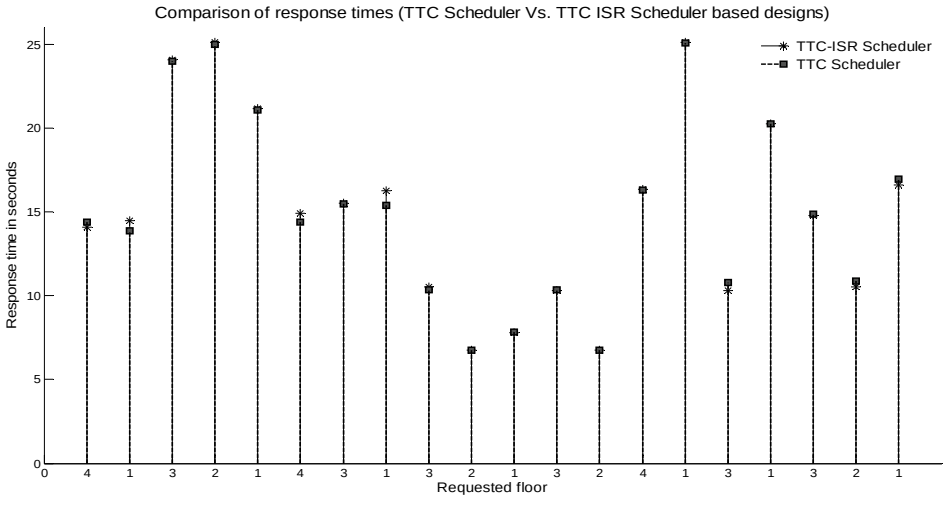


Figure 8.3: Comparison of response times with each design alternative

The difference between response times is not significant. Table 8.3 tabulates this difference between the response times of each design alternative. There is no evident trend in the manner in which the response of one design compares with that of the other.

Response time in seconds (TTC-ISR SCHEDULER)	**Response time in seconds (TTC SCHEDULER)**	**Difference in response times (seconds)**
14.07	14.4	-0.33
14.49	13.87	0.62
24.09	23.97	0.12
25.11	24.99	0.12
21.15	21.07	0.08
14.92	14.41	0.51
15.52	15.46	0.06
16.26	15.38	0.88
10.52	10.34	0.18
6.75	6.76	-0.01
7.8	7.85	-0.05
10.33	10.34	-0.01
6.76	6.74	0.02
16.36	16.31	0.05
25.12	25.08	0.04
10.33	10.79	-0.46
20.26	20.25	0.01
14.78	14.85	-0.07
10.54	10.89	-0.35
16.6	16.95	-0.35

Table 8.3: Comparison of response times (TTC SCHEDULER Vs TTC-ISR SCHEDULER)

The behaviour of the system and the response times computed from the data obtained by executing the simulated test case indicate that both implementations offer comparable though non-identical behaviour. The experiment focuses on the primary functionality of the elevator – i.e. to service floor requests in an orderly manner. The data presented in this section and the graphs representing this information indicate that both alternatives are comparable in this regard.

8.6. Discussion

This case study illustrated the use of BNF production rules to obtain the design of a very basic elevator control system. The focus at this stage has been to assess the usefulness of the BNF representation for deriving a pattern-based representation of the systems design. The process of deducing the design through the successive application of production rules was very educative. The set of rules defining the grammar were constantly re-evaluated as a representation of the design evolved gradually.

The alternatives design was obtained by attaching semantic properties to the terminals and non-terminals of the grammar. Semantic associations were introduced to define the manner in which these properties were handled by the terminals and non-terminals in a production. A set of semantic associations have been documented in this thesis to indicate the concept of attaching semantic information to production rules of a CFG. The semantic rules applicable to this case study have been detailed in the section explaining the process of using these patterns to derive an initial representation of the design.

The response times tabulated from testing both these approaches was thought to be a useful means to evaluating the two designs. However, the results from the case study clearly indicate that both designs exhibit comparable behaviour with regards to the main functionality of an elevator system. Both systems can be effectively used for obtaining the implementation of an elevator control system, since they respond to the same timed, sequence of floor requests in a similar manner.

However, future case studies will need to devise other test cases that identify the design sensitive elements in order to test the ease with which successful alternatives can be deduced from an original design. The current system does not consider safety issues important to the

design of the elevator. A test case designed to test the response to a potentially dangerous state should provide more insight to the usefulness of such a design space exploration activity.

This case study illustrated the use of a formal representation of the design pattern to compose a pattern-based design of an embedded system. The source code implementations of each design were obtained by manually adapting the PIEs of the patterns constituting the design. The primary focus of this research project has been to indicate the potential of other pattern information in a software engineering process framework.

An alternative approach to obtaining a design for an elevator control system maybe through the use of the UML to represent the elements of an elevator system and their behaviour. UML relies on capturing these aspects of the designs with a set of standard diagrams. Presented here is a simple UML representation of an elevator system. An overview of the functionality in the elevator control application and the lone actor in the system is depicted through the Use Case diagram of Figure 8.4. The goals of the system are directly obtained from the behaviour expected of the application.

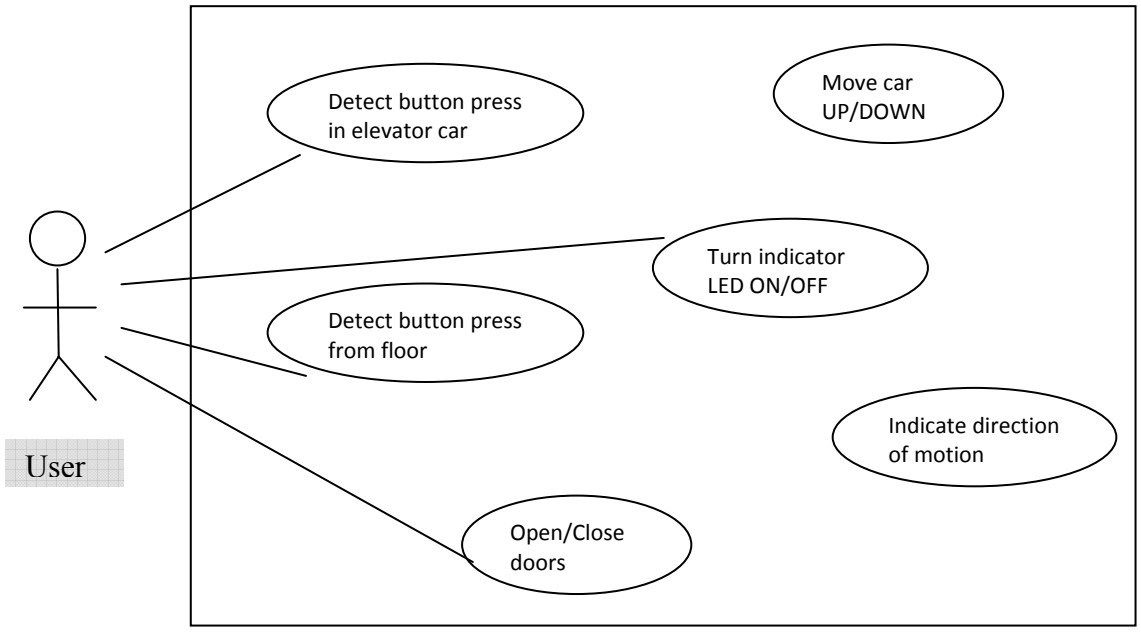


Figure 8.4: Use case diagram - elevator system

These include the ability to detect a floor request from inside the elevator car, as also the floor. The elevator controller should be able to open and close the car doors and enable the

car to move along a specified direction at a desired speed. The user should be intimated of request for the elevator car and arrival at a floor by suitably turning LEDs/Indicators ON and OFF. These aspects help identify the various classes and object required to build the application. Figure 8.5 captures the Class Diagram of an elevator system.

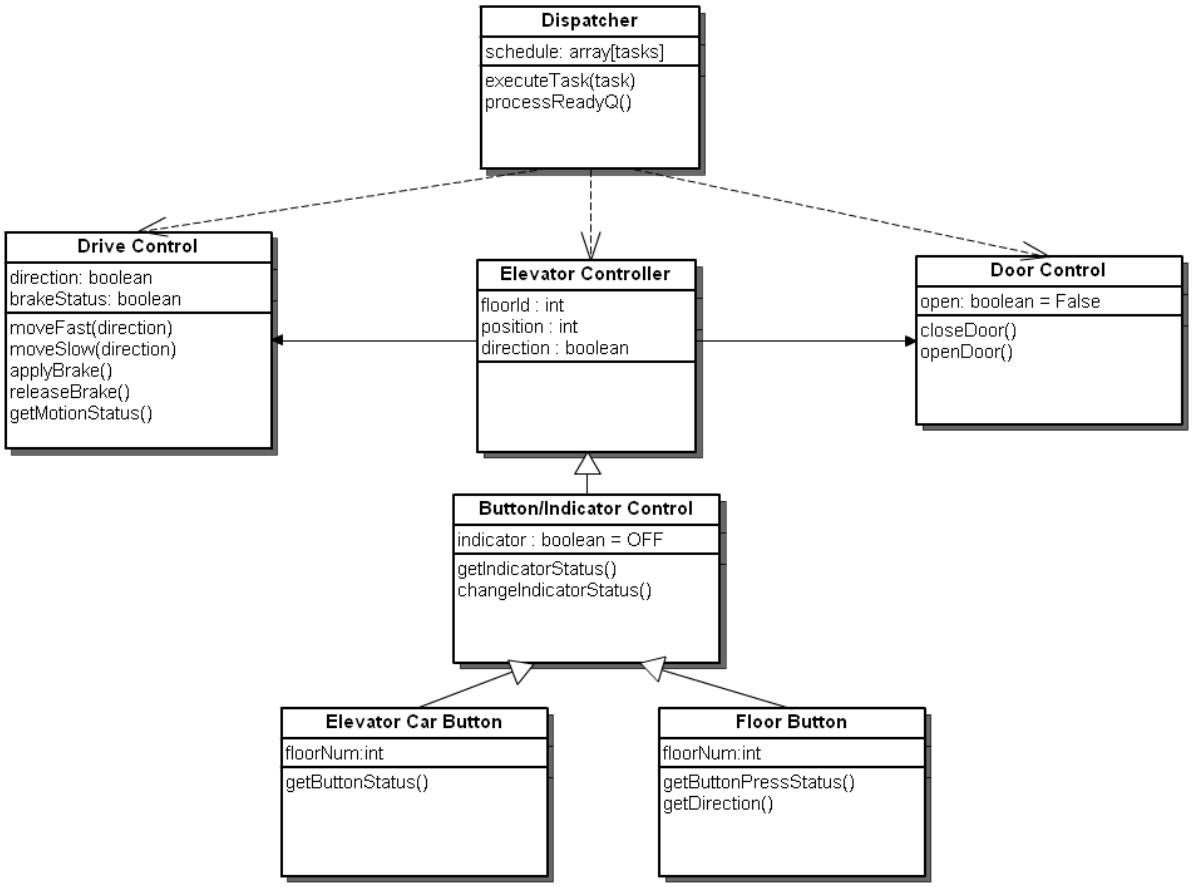


Figure 8.5: Class diagram - elevator system

The Class Diagram captures a static overview of the elements in the system. UML also provides the ability to capture the desired behaviour of the system. Sequence diagrams are one such mechanism used to represent the behaviour of the system being designed. Each sequence diagram captures a scenario representing one aspect of the desired system behaviour. It indicates the objects affected by the behaviour being depicted and the messages passed between these objects.

Figure 8.6: Sequence diagram – servicing a floor request from the elevator car

Figure 8.6, Figure 8.7 and Figure 8.8 describes three such scenarios.

Figure 8.6: Sequence diagram – servicing a floor request from the elevator car

Figure 8.6 captures the desired behaviour when a floor request is made from inside the elevator car.

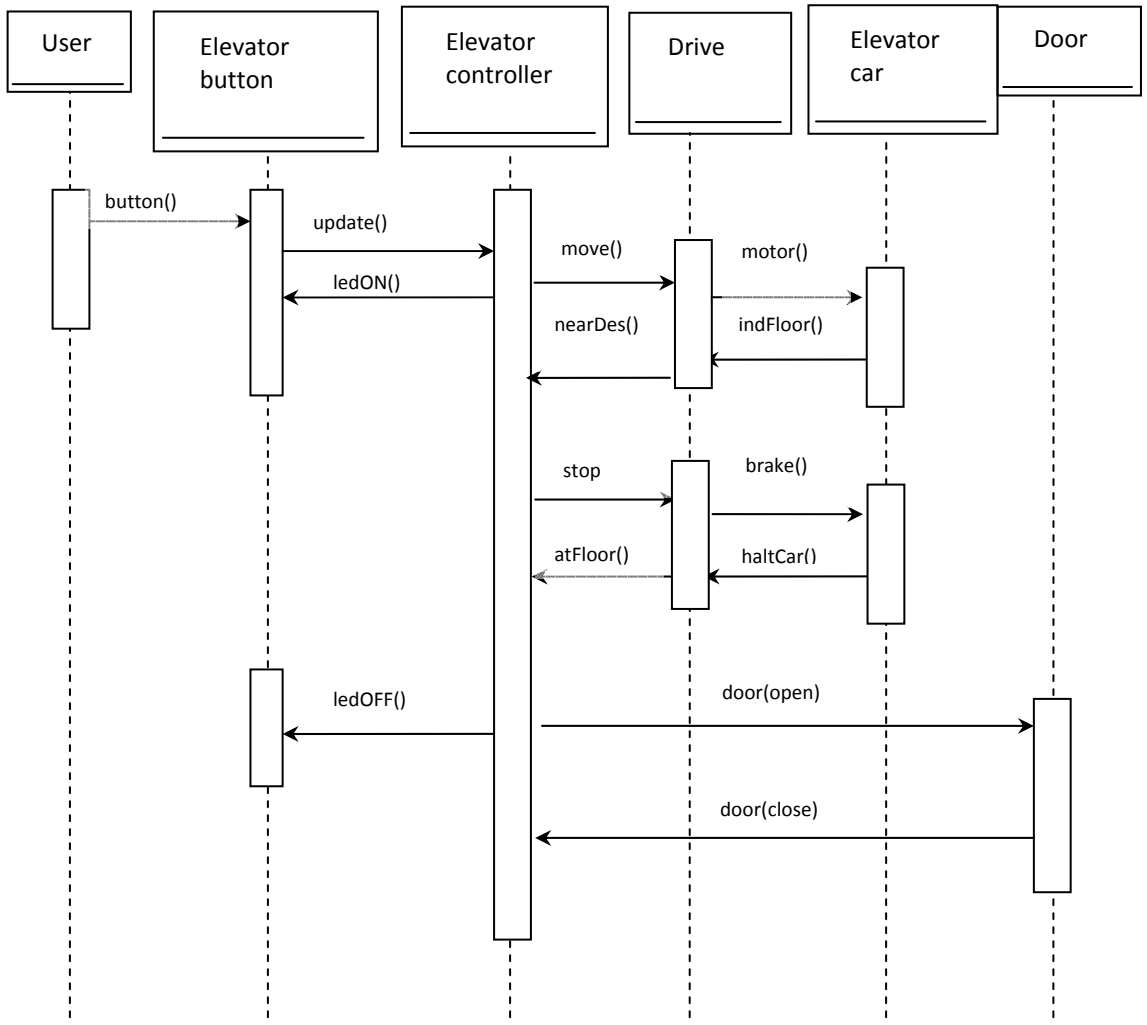


Figure 8.6: Sequence diagram – servicing a floor request from the elevator car

Similarly, Figure 8.7 depicts the expected behaviour of the system when the user makes a floor request from another floor. Both behaviours are almost the same as illustrated.

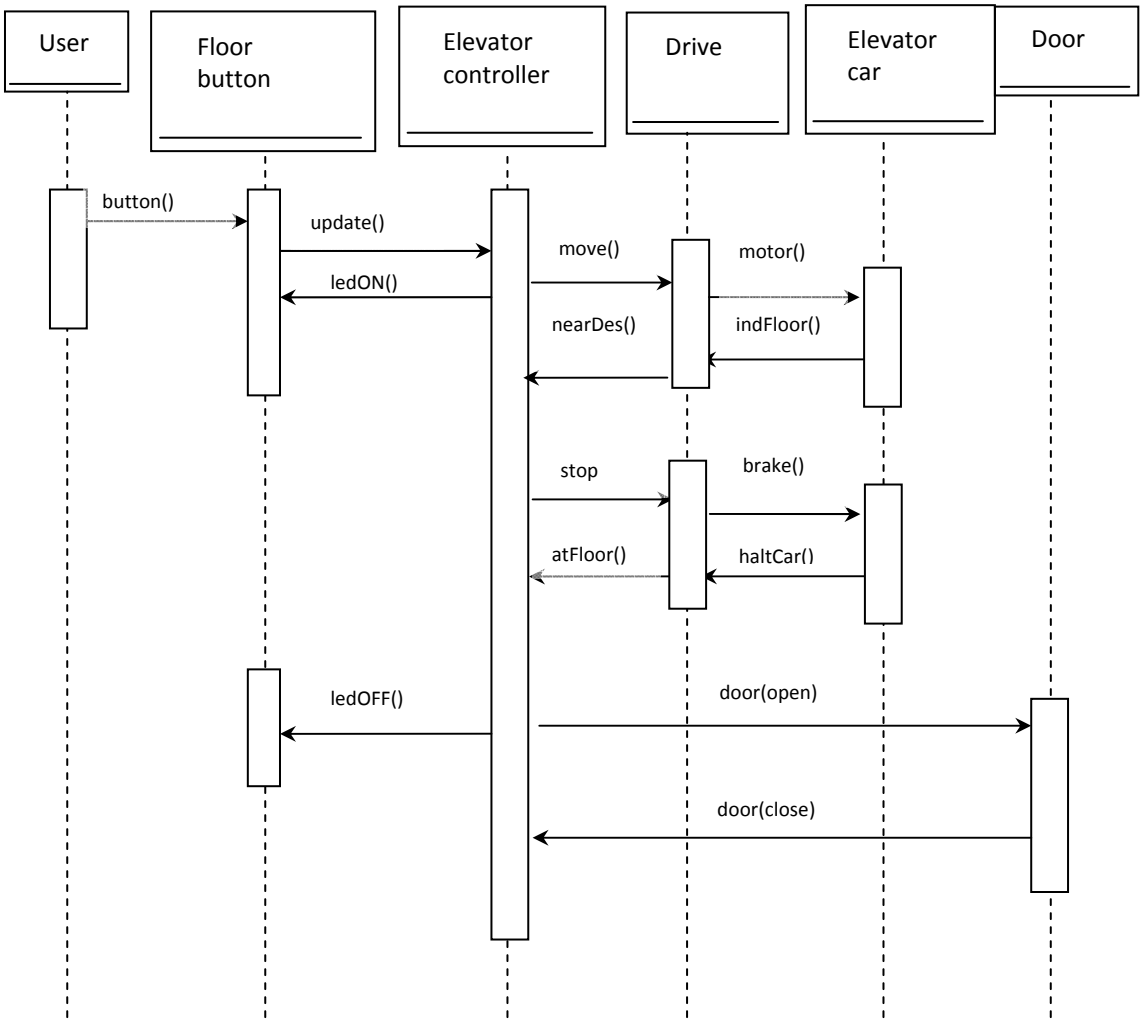


Figure 8.7: Sequence diagram – servicing a floor request from another floor

Figure 8.8 describes the manner in which the elevator cars are driven to realise a floor request. These diagrams are not exhaustive. The behaviour of the system can be further modelled using Collaboration Diagrams and State machines. The focus of the approach is to model the design of the system using a set of diagrams to pictorially represent the design of the system that needs to be implemented. UML also encourages the use of standard symbols to aid the understanding of these diagrams. Used with a well-defined set of diagrams, these models can also be used as blueprints of the design and further assist in automatically generating source code from this set of diagrams.

Though the UML-based approach described here was intended to illustrate other approaches currently used to develop applications like the elevator control system, it should not be considered as a competing technology to the research presented here. The UML-based

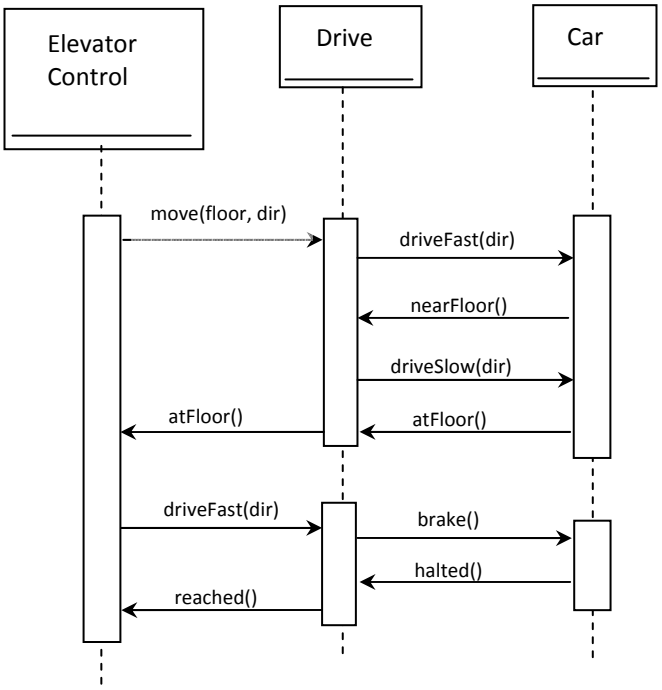


Figure 8.8: Sequence diagram – controlling the motion of the elevator car

approach is well researched and documented. By contrast pattern-based approaches to software development have up until now focussed primarily on the manner in which patterns languages can be created and utilised in the process of source code creation (often using the solutions examples as starting point). Chapter 9 includes a discussion of the relevance of each approach and the possible manner in which they could be used in tandem to aid the process of delivering quality software quickly.

8.7. Conclusion

This case study was intended to illustrate the manner in which the BNF representation presented in Chapter 7 could be used to search the design space specified by the patterns in PTTES. The rules of production were used to compose an initial pattern-based representation of the application design. The context-sensitive information, captured through semantic associations for the production rules in the representation were further utilised to understand the design alternatives available as options for the current system.

At this stage, the focus of the research has been on understanding an approach to formalise the PTTES pattern language. To evaluate the suggested approach against other approaches will require conducting a series of experiments with software development teams and

production environments if possible, where each of the teams are given the same application requirement specifications and the tools relevant to each design methodology. Such an experiment is essential to ascertain the benefits of the suggested approach. However it is seen as future work in this research project.

The penultimate section of this thesis presents a discussion on the work presented in this thesis, followed by a conclusion detailing scope for future work.

Part D: Discussion and Conclusion

9. Discussion

9.1. Introduction

The research presented in this thesis analysed the use of design patterns in software development. This also involved exploring techniques to use patterns in design space exploration. Chapter 5 described an approach to restructure the pattern language of interest with a view of better utilizing the information presented in the design pattern. Chapter 6 described empirical studies to illustrate the potential of design patterns beyond code generation. The experiments explored the possibilities of obtaining alternative designs using pattern information. It also compared the behaviour of these alternative systems. Chapter 7 proceeded to describe an approach to formalise the PTTES language and Chapter 8 presented results from a case study demonstrating the application of this formal representation to derive alternative designs. This chapter presents a discussion to put all these activities in perspective. It also includes a discussion describing the need for PBSE when approaches such as model-driven software engineering have already begun looking at working with the design level of abstraction.

9.2. PBSE for time-triggered embedded systems

Time-triggered architectures are built around a single interrupt source (usually from a timer-overflow). Advocates of the approach encourage the use of time-triggered designs when predictability is a key concern in the design and implementation process. However, designing tasks against an imposed timeline can be quite challenging. A few of the important considerations to be made while designing embedded systems with a time-triggered architecture include

- the length of a tick interval
- the time period between consecutive calls of a task
- the delay before a task can be called for the first time
- the need to ensure that all tasks complete execution prior to end of a tick period

The PTTES collection is a set of design patterns that capture some of these classic design

problems specific to developing time-triggered embedded systems and document solutions for these design problems.

However there is no prescribed manner in which patterns are expected to be used in software development and this remains the case with PTTES as well. Research into the use of pattern in software development has primarily involved techniques to use them in the creation of software. Since patterns contain extensive domain expertise there is a need and possibility of using this information beyond mere code generation. The research presented here has indicated and illustrated one such potential of the PTTES patterns – i.e. to aid design space exploration activities.

In short, most previous research in the use of design patterns for software development has looked at techniques to use the solution documented as part of the design pattern. By looking at links to related patterns and alternative patterns, this research shifst the focus towards a more effective use of the pattern documentation.

9.3. Re-structuring PTTES

Patterns are documented best-practices to common design problems. Information contained in the pattern documentation is structured and held in a mostly-standard human-readable form. This is primarily because patterns are intended to facilitate an easy understanding of the domain, the problem and the solution being discussed. Up until now, software design patterns were seen to be primarily used for generating source code. However, when envisioning new techniques to better utilise the design patterns, there was an obvious need to restructure the pattern language and better understand the nature of the different kinds of information provided as part of a pattern documentation to utilise this information more effectively in varying stages of software development.

The PIEs associated with a design pattern play provide a useful link between source code and design patterns. Both the manual approach and tool-based code generation techniques adapt PIEs to obtain the required source code. By separating this implementation specific information from the pattern documentation, yet associating a PIE with a design pattern, the restructured language gains to benefit as follows –

- include multiple PIEs to illustrate the application of the pattern in various implementation environments

- build an association between source code and design (which lies at a different level of abstraction)
- enrich the pattern language without having to rewrite the pattern documentation for each new implementation being discussed

The restructuring approach described in this thesis, further distinguished between the design-specific information (usually detailing a specific algorithm) and a more general solution to the problem at hand. Each kind of information has particular relevance in different stages of pattern-based software development. The information held in a generic pattern document provides a very general solution to the problem being considered. In many ways it gives an overview of the various algorithmic options available to realise a solution. By associating design patterns with generic patterns, the user is informed of the design options available to arrive at a more specific design of the solution to the problem. Thus the information held in a generic pattern is primarily intended to be used to understand the problem and be aware of the design options available. The actual design specific details are documented in the design pattern.

In conclusion, the restructured language identifies pattern information as belonging to one of three levels –

- A generic pattern
- A design pattern and
- A pattern implementation example

By associating the elements of each level with the other, the user is encouraged to look at different aspects of the problem and solution depending on the stage at which the problem is being discussed. The information required by a practitioner wishing to decide if a suitable pattern exists for a design problem is very different from the information provided to a developer attempting to apply a specific design solution on a certain implementation environment.

The aim of the restructuring exercise was to better understand the nature of the information documented in a pattern and its potential use in the software development process.

9.4. Patterns and design space exploration

Traditionally pattern information has primarily been intended to facilitate design re-use. Pattern collections, as mentioned earlier, contain a wealth of domain-specific information. The research

presented in this thesis described techniques to use patterns in design space exploration activities. As observed earlier, tools designed to facilitate pattern-based software development primarily focus on using code templates based on the PIEs in the software generation process. This thesis presented the hypothesis that there are other elements of the structured pattern information that can be used in the process of software development. It illustrated the use of information regarding related patterns to obtain design alternatives to the system being considered.

Case studies presented in Section 6.3and Section 8.3 demonstrated the effect of using pattern alternatives in more realistic embedded applications. The comparable behaviour of the alternative systems supports the hypothesis that pattern information can be used to explore the potential of obtaining design alternatives. By using different design patterns corresponding to the same generic pattern being considered, it is possible to support a certain level of design diversity within reasonably short development schedules.

One of the important implications of the potential of obtaining design alternative quickly is the ability to implement software redundancy through design diversity. The use of different PIEs of the same design pattern, supports implementation diversity in the system. The restructured language also contributes to both design and implementation diversity in implementation. Implementation diversity is also key to realizing software redundancy. Thus by restructuring the language and detailing techniques that utilise pattern relations, this research project has attempted to suggest a useful method of realizing software redundancy. Though design/implementation diversity has special significance for realizing software fault-tolerance, this thesis does not suggest techniques to implement fault-tolerance using design patterns.

9.5. Formalising pattern languages

Existing research in design pattern formalisation documents techniques to formally represent the solution aspect of a design pattern. The primary motivation behind such approaches is to formally describe the solution suggested by a design pattern so that unwanted ambiguities related to the application and use of the pattern can be avoided.

The main theme of this research project is to understand the process of using design patterns in software engineering. Motivated by the fact that design pattern documentations are used sparingly due to the lack of well-defined processes for the same, this project explored techniques

to utilise this pattern information more effectively.

This is however, not a simple task. The pattern documentation is held primarily in natural language form. Practitioners using these patterns develop an intuitive knowledge of the domain over a period of time. The major obstruction in incorporating this knowledge into a tool-assisted development process is the fact that information in the patterns lacks the formality needed for a process-based approach to software development.

To address these issues, the research exercise described in Chapter 7 and Chapter 8 attempted to understand and illustrate a technique to formalise the pattern language. The approach represents the pattern language (an enriched natural language) using a context-sensitive grammar. The representation should be considered as a first step in understanding the challenges of describing a recommended method of synthesising a pattern-based design. However, by using a context-free grammar to represent a pattern language, future research is directed towards design of a suitable compiler that will effectively formalise the manner in which the pattern language is used. This idea is especially interesting because current practices in software engineering try and encourage use of formal artefacts in the earlier stages of development. Modern engineering practices encourage designs that can be compiled into source code to decrease the errors that may be created by a manual process of code creation or maintenance.

The remainder of this chapter discusses the relevance of PBSE in an era where compilers are being designed for artefacts at higher levels of abstraction than source code.

9.6. Programming with design

Software practitioners have always wanted to work with higher abstractions of program representations. Assembly language programmers avoided the cumbersome process of thinking and implementing in machine language (Schmidt 2006). The creation of sophisticated programming languages distanced practitioners away from implementation details and encouraged working with representations that are closer to a human understandable form. Advances in language design (such as the use of OO languages like C++, Java etc.) and CASE tool technologies (IDEs and support for platforms such as .NET, CORBA, J2EE, etc.) encouraged the need to work with the conceptual representations of the system being developed (using objects or components) (Uhl and Ambler 2003; Schmidt 2006). This shift towards working with higher levels of abstraction, has progressed and resulted in the development of

techniques which represent and manipulate program design. Design patterns address solutions to problems at the design level of abstraction. Model-Driven Architecture (MDA), as detailed next, is another such approach which supports a framework for working with design level abstractions of code.

9.7. PBSE in a nutshell

Design patterns were adopted by the software engineering community in a bid to manage the growing complexity of design patterns. With a need to produce quality software under tight production schedules, it was imperative to use past knowledge rather than re-inventing the wheel every time a classic design problem needed attention. Design patterns are detailed documentations of best-practices. They were initially intended to be a reference point for practitioners wishing to understand a problem or its solution in greater detail. Pattern documentation was thus organised in a structured document and the details of the solution were made available in a human-readable format.

Patterns in a collection attempted to capture domain expertise in a comprehensive manner. Thus the pattern names and certain characteristic information were intended to enhance the practitioner vocabulary, thus aiding technical communication amongst developers and designers. In spite of all this care in making the expert-information available for the use of practitioners, there has never been a concerted effort in recommending a suggested manner in which the information can be utilised. In other words, patterns capture domain expertise, but it is more important for this information to be used suitably to reap the benefits of this knowledge.

The lack of a formally defined process for the use of patterns in software development leads to the use of ad-hoc procedures for the same. Tools to support pattern-based software development implicitly enforce a certain process to incorporate patterns in the development process. PBSE attempts to suggest a standard framework based on the manual approach of using patterns and the processes supported by pattern-tools. By focussing on techniques to effectively use design patterns in the development life-cycle, PBSE attempts to shift the focus of the software development from obtaining source code to obtaining a good system design. Thus PBSE encourages attention on working with information at higher levels of abstraction than source code.

9.8. PBSE and MDA

To understand how PBSE compares with a contemporary technology such as MDA it is important to understand the underlying similarities of these approaches and then understand the relevance of PBSE in the context of these similarities.

9.8.1 Understanding Model-Driven Architecting (MDA)

The MDA initiative, introduced by the Object Management Group (OMG), offers a conceptual framework for a set of standards that support model-based software engineering. Models capture expert knowledge as mapping functions that transform one model to another (Dzafic et al. 2004).

Model-driven architecture emphasises –

- The distinction of business logic from implementation specific details
- The need to model business logic and implementations of the logic separately
- Identifying translational framework to convert the models capturing business logic and implementation details to the actual source code

This is achieved using the tiered framework presented in Figure 9.1. A practitioner using the MDA approach analyses the problem at hand and designs a solution to this problem. The design is captured as a set of models. The high-level abstract details of the design are captured in the Platform-Independent Model (PIM). These models essentially capture the business logic for which the program is designed and are specified using a modelling language like UML (Fowler 2003; Feiler et al. 2007; webpage: Object Management Group 2008). In order to use UML to specify design rigorously there is a need to use a restrictive subset of it. Executable UML is one such subset of UML used to specify designs in a completely automated model-driven approach. Various other subsets of UML, like SysML (Vanderperren and Dehaene 2005), UML-RT (Küster and Stroop 2001) and RT-UML (Wehrmeister et al. 2005) are used specifically for the design and development of embedded and real time systems and to address the inadequacies of UML as a formal specification language.

The implementation specifics of the application are captured in a set of Platform Specific Models (PSM). These models have the technology dependent information. The application is obtained by parsing the PIM and PSM using a suitable model compiler. The model compiler

and the model transformation mechanism are crucial to MDA (Selic 2003; Sendall and Kozaczynski 2003). Research into MDA-based generation of embedded software systems focuses on developing UML for embedded systems domain (Kukkala et al. 2005; Ziegenbein et al. 2005) and using object technologies for designing code-generators (Nascimento et al. 2006; Liu et al. 2008).

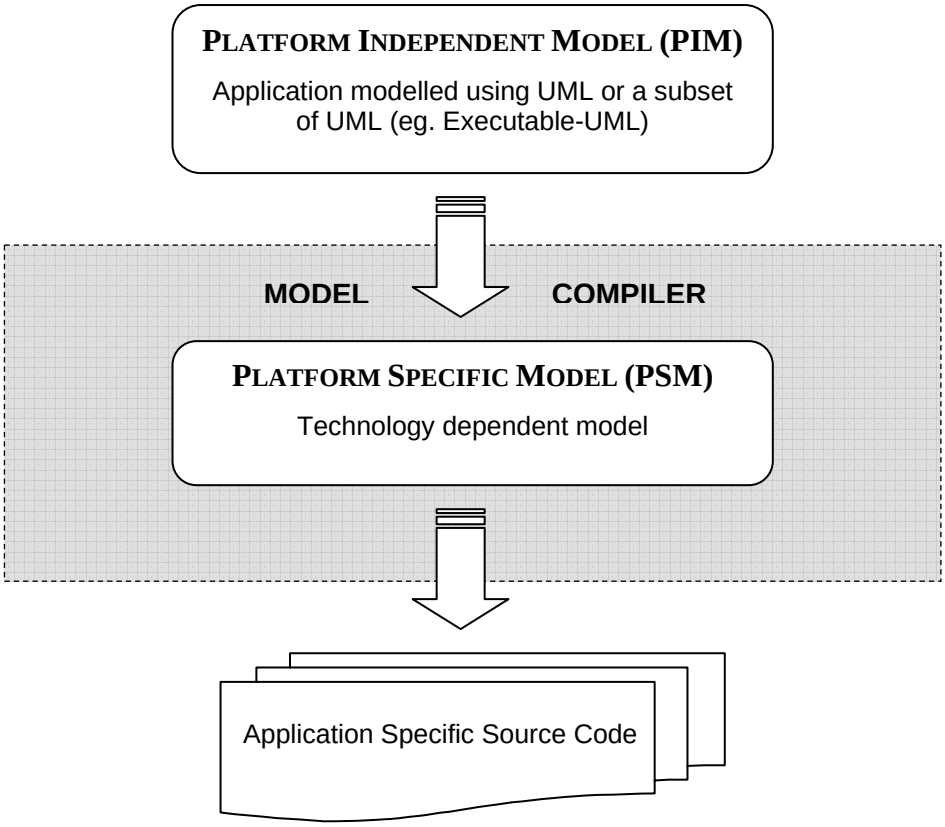


Figure 9.1: MDA using Executable UML, adapted from (Fowler 2003)

The inspiration behind MDA is the need to decouple implementation technology from a higher level understanding of the application's design. The use of model compilers is vital to automating the process of code-generation from a design model (Mellor and Balcer, 2003). Through the framework described earlier, MDA expects to provide an open, vendor-neutral approach to handle business and technology change. It achieves this by separating the business and application logic from the platform technology. By separating the business and the implementation technologies, each can evolve at its own pace. Thus the models capturing business logic always reflect the functionality of the system while the actual implementation of the system can incorporate technological changes as is required.

9.8.2 Similarities between PBSE and MDA

Both PBSE (as described in this thesis) and MDA focus on working with higher levels of abstraction (specifically design rather than code). Both approaches emphasize the use of tools to automate the generation of code from the design of a system, shifting the focus from the details of obtaining code implementations of the system being designed to that of capturing the design of a system.

While the MDA approach introduces the concept of a model compiler to support automatic code generation from abstract representations of the system, the PBSE approach advocates the use of a formal representation of the pattern language to provide tool support. The use of BNF to represent the language is intended to form the basis for the design of a compiler for the pattern language.

MDA clearly distinguishes between the business/application logic captured using the Platform Independent Models (PIMs) and the implementation specifics captured in the Platform Specific Models (PSMs). The approach to re-structure the PTTES language is based on a similar distinction where pattern information is identified to be on different levels of abstraction. The architecture/domain significant information is captured in the generic patterns and design patterns which constitute the design of the system. The actual system implementation is derived from the PIEs which contribute in a manner similar to the PSMs in MDA.

9.8.3 What more does PBSE have to offer?

Though PBSE and MDA support development practices that focus on the design phases of software development, the use of MDA does not necessitate the use of best-practices in obtaining the design representation. If the design of the application modelled using UML is incorrect or inadequate this will be reflected in the quality of the code derived from the model. Conceptually, design patterns provide time-tested solutions to classic design problems. The primary motive of PBSE is to support the use of best-practices in the design process in order to improve the quality of the resulting software product.

The PBSE approach described here acknowledges the fact that pattern languages store a wealth of domain-specific knowledge. It illustrated the use of pattern information for design-space exploration activities. This is especially useful when there is a need to support

design/implementation diversity of software systems. MDA implicitly supports implementation diversity through the use of PSMs. This is however not the focus of the MDA approach. The motivation behind a tiered architecture in the MDA has been to isolate the challenges of supporting rapid changes in implementation technology by differentiating between the implementation dependent details of the solution and the business logic.

9.9. Conclusion

The discussion presented here attempted to unify the various threads of research that took shape over the course of understanding the potential of patterns in aiding software development. The concept of design patterns originated in the brick-and-mortar industry. Though they were seen as extremely useful to manage the growing complexity of design patterns, their use in software development has been restricted to their use in source code generation.

This thesis has attempted to study techniques that can better utilise the pattern information in the software development cycle. This research project began with an attempt to associate source code to human readable pattern information. The implementation examples were seen to be useful links between the two. The next stage discussed the need to restructure this language to elaborate upon these associations. Finally the research presented here discussed the need to formalise the pattern information and use it so that the associations are defined more clearly. The concluding chapter presented next re-visits the objectives of the research as detailed in Section 1.6. It discusses the scope for future work in this area.

10. Conclusion

10.1. Introduction

This chapter summarises the contributions made by the research presented in this thesis. It revisits some of the key contributions made by the work presented here and discusses the extent to which the initial aim has been achieved. It discusses the potential for further research in the area of pattern-based software engineering. It emphasises the need for appropriate CASE support for the effective use of design patterns in software engineering.

10.2. Contributions made by this work

The work presented in this thesis identified the lack of an acknowledged process of using patterns in software development. It analysed the current, "informal" process of using patterns in the development of reliable embedded system (manual and tool-based approaches). The framework for Pattern-Based Software Engineering or PBSE as suggested in this thesis is based on this understanding of pattern usage.

Chapter 4 discussed the importance of the implementation example when creating software from design patterns. Chapter 5 discussed the need to extract the implementation example from the pattern documentation while retaining its association to the design pattern. It suggested a novel approach to restructuring the language based on the nature of information documented in a pattern. It identified two new layers: a Generic Pattern (discuss the availability of possible solutions to the problem) and a PIE (a document that captures implementation specific details of a pattern solution). Where Generic Patterns document general solutions to domain-specific problems and explain the design considerations needed to choose a design pattern associated with it, the PIEs capture the implementation specific details of a pattern. This way, the pattern language is more extensible as the addition of new PIEs can illustrate the use of design patterns on different platforms without the need to change the design pattern documentation itself.

Chapter 4 also described the potential of using patterns information to identify design alternatives. This chapter presented the hypothesis that by identifying a suitable mechanism to create multiple designs, patterns have the potential to be used in processes like prototyping and diversity based software fault-tolerance. Chapter 6 presented an empirical study aimed at understanding the scope of exchanging design patterns to derive design alternatives. The pattern documentation was subsequently used to obtain alternative designs to this system. The empirical studies described in Chapter 6, illustrate the creation of alternative systems with comparable behaviour by exchanging patterns in the system. The restructuring approach described in Chapter 5 provides a mechanism to identify multiple implementations of a single design solution. Such a restructuring approach presents the potential of implementing software diversity. However, this research has not explored this potential in great detail. Though this information has not been used in the current research work, there is potential for implementation diversity and its uses to be studied in the future.

Chapter 7 discussed the need to formalise the PTTES pattern language. It described techniques used by patterns researchers for obtaining formal representations of patterns. The research presented here suggests a novel approach to formally represent the PTTES language. This representation is based on the fact that pattern languages are intended to improve practitioner communication by enhancing natural language with terminology particular to the domain being discussed. The BNF-based approach used here describes a set of production rules to compose systems using the PTTES patterns. It acknowledges the need to use context-sensitive pattern specific information when supporting a mechanism to use the patterns. This is accomplished by using attribute grammars to specify the context-sensitive information in patterns. Chapter 8 described a case study to illustrate the approach of working with patterns and alternative designs using the formal representation of the PTTES language.

To summarise, this thesis identified the lack of a standard approach to effectively use design patterns and attempted to describe an early form of PBSE. It recognized a need to restructure the pattern language to incorporate growing domain knowledge and support better utilization of this knowledge. It proposed a novel use of pattern relationships to perform design space exploration activities and demonstrated the use of related patterns to obtain design alternatives using suitable case studies. Finally, it discussed the need to formalise the pattern language in order to support standard practices which used design patterns.

10.3. Scope for future work

Though design patterns have been used previously in software development (and - more specifically - in the development of reliable embedded systems), the work presented here has attempted to develop and describe a process for the use of such patterns. By documenting such a pattern-based design process, it is hoped that the work described here will provide a framework for a process-based understanding of pattern usage. In this way domain-expertise documented in a pattern collection has the potential to be used more extensively in a manual or tool-based development approach.

The design exploration activities described here looks at approaches to use pattern information for deriving alternative designs. The use of alternative design information to support prototyping seems straight forward. However, using the design patterns for building software fault-tolerance systems offers tremendous scope for future research. In addition to information detailing alternative systems, patterns should probably be refined to capture their behaviour when used in conjunction with related patterns. The pattern language will also need to be enhanced to discuss techniques of voter design and the check-point/recovery mechanism when the patterns are actually used in implementing fault-tolerant software.

The research presented here identified PIEs as active elements of pattern-based software development. It then proceeded to suggest techniques by which information regarding pattern alternatives can be used for design space exploration activities. However patterns have detailed information documented in various other sections. Further research in this area can benefit from defining processes by which other information in the pattern documentation can be used in pattern-based software development. Research should probably also include a comparative study of PBSE with the traditional software engineering approaches to identify other information that can be added to make pattern documentation complete and more useful in the complete life cycle. For instance, patterns do not contain any information detailing tests to check the solution being implemented. Testing is however a very important aspect of software engineering. Future research should probably include identifying test cases to verify the general aspects of the solution being documented. This can then be used to derive general and specific test cases to be used whenever an associated pattern is used in the creation of the software product.

This research project suggested an approach to formally represent the PTTES language. This formal representation is at a very early stage of development. It is however expected to provide

an initial foundation upon which a more complete formal representation can be built. The example documented in Chapter 8 illustrated the ideas behind the use of such a formal representation. Detailed case studies to evaluate the suggested techniques using real-world practitioners and development teams are crucial to understand the possibilities that PBSE has to offer.

In many ways PBSE as it is described here can itself be seen as a best practice for use of patterns in software development. Like design patterns, PBSE can be viewed as a process pattern (a time-tested process which is of potential use in a certain domain). The early sections of this thesis attempted to understand the process of manually applying patterns (as it was traditionally conceived to be used) as well as the process encouraged by tools designed to be used in pattern-based software developed. It suggested a process enhancement by describing techniques to use the information documented in the alternatives section to generate design alternatives. As other approaches of using patterns are identified, the PBSE pattern/ pattern language should be refined so that practitioners are made aware of the potentials of pattern information. A tool that is capable of encouraging such a process is far more promising because new possibilities or potential can be introduced to the end user by suitably upgrading the tool which supports PBSE.

10.4. Conclusions

In the pursuit of managing the complexities of software development, practitioners have constantly revisited other engineering streams like Mechanical and Civil Engineering. Software engineering is relatively new compared to the other branches of engineering. The design techniques and process initiatives used in the older engineering streams are used as a basis to identify techniques to manage complexities in software engineering. The abstract nature of the artefacts created and used in the software engineering process contributes to the challenges of any approach used. Thus research in Software Engineering rightly focuses on deriving well-defined representations of this abstract information so that it can be used more effectively in the process of generating software.

Part E: References

References

A Pattern Language. (2001). "Pattern Language.com." Retrieved 10, November, 2008, from http://www.patternlanguage.com/.

Adams, M., J. Coplien, R. Gamoke, R. Hanmer, F. Keeve and K. Nicodemus (2001). Fault-tolerant telecommunication system patterns. Design patterns in communications software, Cambridge University Press**:** 81-94.

Agerbo, E. and A. Cornils (1998). How to preserve the benefits of design patterns. Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. Vancouver, British Columbia, Canada, ACM.

Aho, A., V. , R. Sethi and J. Ullman, D. (1986). Compilers: principles, techniques, and tools, Addison-Wesley Longman Publishing Co., Inc.

Alexander, C. (1964). Notes on the Synthesis of Form. Cambridge, MA, Harvard University Press.

Alexander, C. (1979). The Timeless Way of Building. New York, NY, Oxford University Press.

Alexander, C. and P. Eisenman (1983). Contrasting Concepts of Harmony in Architecture: Debate between Christopher Alexander and Peter Eisenman. Lotus International, 40. **IV:** 60-68.

Alexander, C., S. Ishikawa and M. Silverstein (1977). A Pattern Language. New York, NY, Oxford University Press.

Alexander, C., M. Silverstein, S. Angel, S. Ishikawa and D. Abrams (1975). The Oregon Experiment. New York, NY, Oxford University Press.

Ampatzoglou, A. and A. Chatzigeorgiou (2007). "Evaluation of object-oriented design patterns in game development." Information and Software Technology **49**(5): 445-454.

Anderson, T., P. A. Barrett, D. N. Halliwell and M. R. Moulding (1985). "Software Fault Tolerance: An Evaluation." IEEE Transactions on Software Engineering **11**(12): 1502-1510.

Andriole, S. J. (1994). "Fast, Cheap Requirements: Prototype, or Else!" IEEE Software **11**(2): 85-87.

Andy, B. (2003). Design pattern automation. Proceedings of the 2002 conference on Pattern languages of programs - Volume 13. Melbourne, Australia, Australian Computer Society, Inc.

Archea, J. (1985). "Architecture's Unique Position Among the Disciplines : Puzzle-Making vs. Problem Solving." The Architectural Student Journal. Summer: 20-22.

Audsley, N., K. Tindell and A. Burns (1993). "The end of the line for static cyclic scheduling." Proceedings of the Fifth Euromicro Workshop on Real-Time Systems: 36-41.

Audsley, N. C., A. Burns, R. I. Davis, K. W. Tindell and A. J. Wellings (1995). "Fixed priority pre-emptive scheduling: an historical perspective." Real-Time Systems **8**(2-3): 173-198.

Avizienis, A. A. (1995). The Methodology of N-version Programming. Software Fault Tolerance. M. R. Lyu, John Wiley & Sons Ltd.

Ayavoo, D., M. J. Pont, J. Fang, M. Short and S. Parker (2005). A 'Hardware-in-the-Loop'

testbed representing the operation of a cruise-control system in a passenger car. Proceedings of the Second UK Embedded Forum A. Koelmans, Bystrov, A., Pont, M.J., Ong, R. and Brown, A. Birmingham, UK, October 2005, University of Newcastle upon Tyne 60-90.

Babbage, C. (1974). On the mathematical powers of the calculating engine Subsequently published In The Origins of Digital Computers: Selected Papers, B. Randell, Ed. Springer- Verlag, New York, 17-52.

Backus, J. W., F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. v. Wijngaarden and M. Woodger (1960). "Report on the algorithmic language ALGOL 60." Communications of the ACM **3**(5): 299-314.

Baker, T. P. and A. Shaw (1988). "The cyclic executive model and Ada." Real-Time Systems Symposium: 120-129.

Balanyi, Z. and R. Ferenc (2003). Mining Design Patterns from C++ Source Code. Proceedings of the International Conference on Software Maintenance, IEEE Computer Society.

Barr, M. (1999). Programming Embedded Systems in C and C++, O'Reilly & Associates.

Barrett, P. A., A. M. Hilborne, P. G. Bond, D. T. Seaton, P. Veríssimo, L. Rodrigues and N. A. Speirs (1990). The Delta-4 Extra Performance Architecture (XPA). Proceedings of the 20th International Symposium on Fault-Tolerant Computing Systems (FTCS-20). Newcastle upon Tyne,UK, IEEE Computer Society Press**:** 481-488.

Barry, B. M. (1989). Prototyping a real-time embedded system in Smalltalk. Conference proceedings on Object-oriented programming systems, languages and applications. New Orleans, Louisiana, United States, ACM.

Baruah, S., G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha and F. Wang (1992). "On the competitiveness of on-line real-time task scheduling." Real-Time Systems **4**(2): 125-144.

Baruah, S. K. and J. I. Goossens (2003). "Rate-Monotonic Scheduling on Uniform Multiprocessors." IEEE Transactions on Computers **52**(7): 966-970.

Bates, I. (1998). Scheduling and Timing Analysis for Safety-Critical Systems. Department of Computer Science. York. **PhD thesis**.

Baudry, B., Y. L. Traon, G. Sunyé and J.-M. Jézéquel (2003). Measuring and Improving Design Patterns Testability. Proceedings of the 9th International Symposium on Software Metrics, IEEE Computer Society Washington, DC, USA

Beck, K., R. Crocker, G. Meszaros, J. Vlissides, J. O. Coplien, L. Dominick and F. Paulisch (1996). Industrial experience with design patterns. Proceedings of the 18th international conference on Software engineering. Berlin, Germany, IEEE Computer Society.

Bellebia, D. and J. M. Douin (2006). Applying patterns to build a lightweight middleware for embedded systems. Proceedings of the 2006 conference on Pattern languages of programs. Portland, Oregon, ACM.

Bennett, K., N. Gold and A. Mohan (2005). "Cut the biggest IT cost." The Computer Bulletin **47**: 20-21.

Bernstein, L. (1996). "Forward: Importance of Software Prototyping." Journal of Systems Integration - Special Issue on Computer Aided Prototyping **6**(1): 9-14.

Boehm, B. (2000). Spiral Development: Experience, Principles and Refinements. W. J. Hansen. Pittsburgh, PA, Carnegie-Mellon University, Software Engineering Institute**:** 49.

Boehm, B., W. Brown and R. Turner (2005). Spiral development of software-intensive systems of systems. Proceedings of the 27th international conference on Software engineering. St. Louis, MO, USA, ACM.

Boehm, B. W., T. E. Gray and T. Seewaldt (1984). Prototyping vs. specifying: A multi-

project experiment. Proceedings of the 7th international conference on Software engineering. Orlando, Florida, United States, IEEE Press.
Boekhoudt, C. (2003). "The Big Bang Theory of IDEs." Queue **1**(7): 74-82.
Briere, D. and P. Traverse (1993). "AIRBUS A320/A330/A340 electrical flight controls - A family of fault-tolerant systems." Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on: 616-623.
Brilliant, S. S., J. C. Knight and N. G. Leveson (1990). "Analysis of Faults in an N-Version Software Experiment." IEEE Transactions of Software Engineering **16**(2): 238-247.
Budde, R. and H. Zullighoven (1990). Prototyping revisited. CompEuro '90. Tel-Aviv, Israel, Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering**:** 418-427.
Budinsky, F. J., M. A. Finnie, J. M. Vlissides and P. S. Yu (1996). "Automatic code generation from design patterns." IBM Systems Journal **35**(2): 151-171.
Bulka, A. (2003). Design pattern automation. Proceedings of the 2002 conference on Pattern languages of programs - Volume 13. Melbourne, Australia, Australian Computer Society, Inc.
Burks, A. W. (2002). "The invention of the universal electronic computer--how the Electronic Computer Revolution began." Future Generation Computer Systems **18**(7): 871-892.
Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad and M. Stal (1996). Pattern-Oriented Software Architecture, A System of Patterns. Chichester, England, John Wiley & Sons Ltd.
Buttazo, G. (1997). Hard real-time computing systems: Predictable scheduling algorithms and applications. Norwell, MA, Kluwer Publishers.
Buttazzo, G. C. (2005). "Rate monotonic vs. EDF: judgment day." Real-Time Systems **29**(1): 5-26.
Caglayan, A. K., P. R. Lorczak and D. E. Eckhardt (1988). An experimental investigation of software diversity in a fault-tolerant avionics application. Reliable Distributed Systems, 1988. Proceedings., Seventh Symposium on.
Carle, A., M. Clancy and J. Canny (2007). Working with pedagogical patterns in PACT: initial applications and observations. Proceedings of the 38th SIGCSE technical symposium on Computer science education. Covington, Kentucky, USA, ACM.
Carlow, G. D. (1984). "Architecture of the space shuttle primary avionics software system." Commun. ACM **27**(9): 926-936.
Cai X., M. R. Lyu, K. Wong, K. Roy (2000). Component Based Software Engineering: technologies, development frameworks and quality assurance schemes. Proceedings of the Seventh Asia-Pacific Software Engineering Conference. APSEC 2000 :372 - 379
Chen, L. and A. Avizienis (1995). "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation." Twenty-Fifth International Symposium on Fault-Tolerant Computing **27**(30): 113.
Chung, H.-S., Y.-R. Lee, J.-I. Kim, Y.-J. Jung, S.-K. Kang and D.-e. Kim (2007). The Embedded Prototyping System for Car Based on Object. Proceedings of the Sixth International Conference on Advanced Language Processing and Web Information Technology (ALPIT 2007) - Volume 00, IEEE Computer Society.
Cinneide, M. (2000). Automated refactoring to introduce design patterns. Proceedings of the 22nd international conference on Software engineering. Limerick, Ireland, ACM.
Clincy, V., A. (2003). "Software development productivity and cycle time reduction." Journal of Computing Sciences in Colleges **19**(2): 278-287.
Cline, M. P. (1996). "The pros and cons of adopting and applying design patterns in the real world." Communications of the ACM **39**(10): 47-49.
Cohen, B. I. (1988). "Babbage and Aiken." IEEE Annals of the History of Computing **10**(3):

171-193.
Coplien, J. O. (1995). A Generative Development-Process Pattern Language. J. O. Coplien and D. C. Schmidt. Reading, MA, Addison-Wesley.
Coplien, J. O. (1998). Software design patterns: common questions and answers. The patterns handbooks: techniques, strategies, and applications, Cambridge University Press**:** 311-319.
Coplien, J. O. and B. Woolf (2000). A Pattern Language for Writer's Workshops. Pattern Languages of Program Design 4. N. Harrison, B. Foote and H. Rohnert. Reading, MA, USA, Addison Wesley**:** 557-580.
CORBA. (1997). "Welcome To The OMG's CORBA Website." Retrieved 15, June, 2009, from http://www.corba.org/.
Cunningham, W. and K. Beck (1987). Using pattern languages for object-oriented programs. Proceedings of OOPSLA 1987. Orlando, Florida.
Damm, A., J. Reisinger, W. Schwabl and H. Kopetz (1989). "The real-time operating system of MARS." SIGOPS Operating Systems Review **23**(3): 141-157.
Davis, A. M. (1992). "Operational Prototyping: A New Development Approach." IEEE Softw. **9**(5): 70-78.
Dominus, M.-J. (1998). "Perl: Not Just for Web Programming." IEEE Software **15**(1): 69-74.
Dzafic, I., M. Glavic and S. Tesnjak (2004). Power system model driven architecture. Electrotechnical Conference, 2004. MELECON 2004. Proceedings of the 12th IEEE Mediterranean.
Earnshaw, R., W., L. Smith, D. and K. Welton (1997). "Challenges in Cross-Development." IEEE Micro **17**(4): 28-36.
Eden, A. H. (2000). Precise Specification of Design Patterns and Tool Support in Their Application. Department of Computer Science. Tel Aviv, Tel Aviv University. **Ph.D.**
EventHelix.com Inc. (2008). "EventHelix.com." Retrieved 10, November, 2005, from http://www.eventhelix.com/.
Feiler, P., H. , D. d. Niz, C. Raistrick and B. Lewis, A. (2007). From PIMs to PSMs. Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007), IEEE Computer Society.
Feedback Instruments Ltd. (2009). "Model Feedback Instruments Ltd, Teaching Solutions for Education and Industry" Retrieved 17-September-2009, 2009, from http://www.fbk.com/product.aspx?pid=34-150.
Fincher, S. and I. Utting (2002). Pedagogical patterns: their place in the genre. Proceedings of the 7th annual conference on Innovation and technology in computer science education. Aarhus, Denmark, ACM.
Florijn, G., M. Meijers and W. P. (1997). "Tool Support for Object-Oriented Patterns." Proc. 11th European Conf. Object Oriented Programming—ECOOP'97: pp. 472-495.
Fowler, M. (2003). UML Distilled: A Brief Guide to the Standard Object Modeling Language, Addison-Wesley Longman Publishing Co., Inc.
France, R., B., D.-K. Kim, S. Ghosh and E. Song (2004). "A UML-Based Pattern Specification Technique." IEEE Transactions on Software Engineering **30**(3): 193-206.
Fuegi, J. and J. Francis (2003). "Lovelace & Babbage and the Creation of the 1843 'Notes'." IEEE Annals of the History of Computing **25**(4): 16-26.
Fuhrman, C. P., S. Chutani and H. J. Nussbaumer (1995). Hardware/software fault tolerance with multiple task modular redundancy. Proceedings of the IEEE Symposium on Computers and Communications (ISCC'95), IEEE Computer Society.
Furukawa, Y. and S. Kawamura (2006). Automotive electronics system, software, and local area network. Proceedings of the 4th international conference on Hardware/software codesign and system synthesis. Seoul, Korea, ACM.
Gamma, E., R. Helm, R. Johnson and J. Vlissides (1995). Design Patterns: Elements od

Reusable Object-Oriented Software. New York, NY, Addison-Wesley Publishing Company.
Gartman, W. D. (2004). The Rise of Modern Architecture in Postwar America: Class and Spatial Roots of Aesthetic Change. Annual meeting of the American Sociological Association. Hilton San Francisco & Renaissance Parc 55 Hotel, San Francisco, CA.
Germain, i. and P. N. Robillard (2008). "Towards software process patterns: An empirical analysis of the behavior of student teams." Information and Software Technology **50**(11): 1088-1097.
Geyer-Schulz, A. and M. Hahsler (2002). Software Reuse with Analysis Patterns. Proceedings of the 8th AMCIS, Dallas, TX, Association for Information Systems.
Graf, S., O. Haugen, I. Ober and B. Selic (2006). "Preface of "Specification and Validation of Real Time and Embedded systems in UML"." International Journal on Software Tools for Technology Transfer **8**(2): 93-96.
Greibach, S. A. (1981). "Formal Languages: Origins and Directions." IEEE Annals of the History of Computing **3**(1): 14-41.
Griss M. L. (1994). Software reuse experience at Hewlett-Packard. Proceedings of the 16th International Conference on Software Engineering: 270
Guerrero, L. A. and D. A. Fuller (1999). Design Patterns for Collaborative Systems. Proceedings of the String Processing and Information Retrieval Symposium \& International Workshop on Groupware, IEEE Computer Society.
Hafiz, M. (2005). Security patterns and evolution of MTA architecture. Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. San Diego, CA, USA, ACM.
Hall, E. C. (2000). "From the Farm to Pioneering with Digital Control Computers: An Autobiography." IEEE Annals of the History of Computing **22**(2): 22-31.
Hand, T. (1991). Debugging embedded systems implemented in C. Proceedings of the second and third annual workshops on Forth. San Antonio, Texas, United States, ACM.
Harrison, N. (2006). The Language of Shepherding. Pattern Languages of Program Design 5. D. Manolescu, M. Voelter and J. Noble. Reading, MA, USA, Addison Wesley**:** 507 - 530.
Head, R., V. (2001). "Univac: A Philadelphia Story." IEEE Annals of the History of Computing **23**(3): 60-63.
Hilford, V., M. R. Lyu, B. Cukic, A. Jamoussi and F. Bastani, B. (1997). Diversity in the Software Development Process. Proceedings of the 3rd Workshop on Object-Oriented Real-Time Dependable Systems - (WORDS '97), IEEE Computer Society.
Hopper, G. M. (1988). "The education of a computer." IEEE Annals of the History of Computing **9**(3-4): 271-281.
Huang, H., S. Zhang, J. Cao and Y. Duan (2005). "A practical pattern recovery approach based on both structural and behavioral analysis." Journal of Systems and Software **75**(1-2): 69-87.
Hughes, Z. M., M. J. Pont and H. L. R. Ong (2005). The PH Processor: A soft embedded core for use in university research and teaching. Proceedings of the Second UK Embedded Forum. A. Koelmans, A. Bystrov, M. J. Pont, H. L. R. Ong and A. Brown. Birmingham, UK, University of Newcastle upon Tyne**:** 194-201.
IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology, IEEE.
Johnson, H. A. and L. Wilkinson (2003). "Case tools in object-oriented analysis and design." Journal of Computing Sciences in Colleges **19**(2): 306-313.
Jones, B., A. and J. Cavallaro, R. (2003). "A rapid prototyping environment for wireless communication embedded systems." EURASIP Journal on Applied Signal Processing **2003**(1): 603-614.
Keller, R. K., R. Schauer, S. Robitaille and P. Page (1999). Pattern-based reverse-engineering of design components. Proceedings of the 21st international conference on Software

engineering. Los Angeles, California, United States, ACM.
Kelly, J. P. J., T. I. McVittie and W. I. Yamamoto (1991). "Implementing Design Diversity to Achieve Fault Tolerance." IEEE Software **8**(4): 61-71.
Kim, D.-K., R. France, S. Ghosh and E. Song (2002). Using Role-Based Modeling Language (RBML) to Characterize Model Families. Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems, IEEE Computer Society.
Knight, J. C. and N. G. Leveson (1986). "An experimental evaluation of the assumption of independence in multiversion programming." IEEE Trans. Softw. Eng. **12**(1): 96-109.
Knuth, D. E. (1968). "Semantics of context-free languages." Mathematical Systems Theory **2**(2): 127-145.
Koehnemann, H. and T. Lindquist (1993). Towards target-level testing and debugging tools for embedded software. Proceedings of the conference on TRI-Ada '93. Seattle, Washington, United States, ACM.
Kohn, W. (2002). The Lost Prophet of Architecture. The Wilson Quarterly, Woodrow Wilson International Center for Scholars. **26**.
Kopetz, H. (1991). Event-Triggered Versus Time-Triggered Real-Time Systems. Proceedings of the International Workshop on Operating Systems of the 90s and Beyond, Springer-Verlag.
Kopetz, H. (1994). The Systematic Design of Large Real-Time Systems or Interface Simplicity. Revised Papers from a Workshop on Hardware and Software Architectures for Fault Tolerance, Springer-Verlag.
Kopetz, H. (1995). Why time-triggered architectures will succeed in large hard real-time systems. Proceedings of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems, IEEE Computer Society.
Kopetz, H. (1998). The Time-Triggered Model of Computation. Proceedings of the IEEE Real-Time Systems Symposium, IEEE Computer Society.
Kopetz, H. (2000). Software engineering for real-time: a roadmap. Proceedings of the Conference on The Future of Software Engineering. Limerick, Ireland, ACM.
Kordon, F. and Luqi (2002). "An Introduction to Rapid System Prototyping." IEEE Transactions on Software Engineering **28**(9): 817-821.
Krueger C. W. (1992). Software reuse. ACM Computing Surveys (CSUR) **24**(2):131-183
Kukkala, P., J. Riihimäki, M. Hännikäinen, T. D. Hämäläinen and K. Kronlöf (2005). UML 2.0 Profile for Embedded System Design. Proceedings of the conference on Design, Automation and Test in Europe - Volume 2, IEEE Computer Society. **2** 710-715 (ISBN: 0-7695-2288-2).
Kurian, S. and M. Pont, J (2007). "The maintenance and evolution of resource-constrained embedded systems created using design patterns." Journal of Systems and Software **80**(1): 32-41.
Kurian, S. and M. J. Pont (2005a). Building reliable embedded systems using Abstract Patterns, Patterns, and Pattern Implementation Examples. Proceedings of the Second UK Embedded Forum. A. Koelmans, A. Bystrov, M. J. Pont, H. L. R. Ong and A. Brown. Birmingham, UK, University of Newcastle upon Tyne**:** 36-59.
Kurian, S. and M. J. Pont (2005b). Mining for Pattern Implementation Examples. Proceedings of the Second UK Embedded Forum. A. Koelmans, A. Bystrov, M. J. Pont, H. L. R. Ong and A. Brown. Birmingham, UK, University of Newcastle upon Tyne**:** 194-201.
Küster, J. M. and J. Stroop (2001). Consistent Design of Embedded Real-Time Systems with UML-RT. Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing, IEEE Computer Society.
Lange, A. (2006). "This Year's Model: Representing Modernism to the Post-war American Corporation " Journal of Design History **19**(3): 233-248.
Lehoczky, J. P. and L. Sha (1986). Performance of real-time bus scheduling algorithms.

Proceedings of the 1986 ACM SIGMETRICS joint international conference on Computer performance modelling, measurement and evaluation. Raleigh, North Carolina, United States, ACM.
Leung, J. Y. T. and J. Whitehead (1982). "On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks." Performance Evaluation **2**(4): 237 - 250.
Littlewood, B., P. Popov and L. Strigini (2001). "Modeling software design diversity: a review." ACM Comput. Surv. **33**(2): 177-208.
Liu, C. L. and J. W. Layland (1973). "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." Journal of the ACM **20**(1): 46-61.
Liu, Y., Y. Zhang, G. Xu and Y. Zhang (2008). Rapid Development of Embedded Software Based on Matlab. Proceedings of the 2008 International Conference on Embedded Software and Systems Symposia - Volume 00, IEEE Computer Society.
Locke, C. D. (1992). "Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives." Real-Time Systems **4**(1): 37-53.
Louren, J. and G. Cunha (2007). Testing patterns for software transactional memory engines. Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging. London, United Kingdom, ACM.
Low, G. and V. Leenanuraksa (1999). Software Quality and CASE Tools. Proceedings of the Software Technology and Engineering Practice, IEEE Computer Society.
Lucrédio D., Bianchini C. P., Prado A. F.,Trevelin L. C.,Almeida E. S. (2003). "Orion – A Component Based Software Engineering Environment." The Journal of Object Technology. Special issue: TOOLS USA 2003. **3**(4): 51 - 74.
Lyu, M. R. (2007). Software Reliability Engineering: A Roadmap. 2007 Future of Software Engineering, IEEE Computer Society.
Maier, R., G. Bauer, G. Stoger and S. Poledna (2002). "Time-Triggered Architecture: A Consistent Computing Platform." IEEE Micro **22**(4): 36-45.
Mak, J. K. H., C. Choy, S. T. and D. Lun, P. K. (2003). Precise Specification to Compound Patterns with ExLePUS. Proceedings of the 27th Annual International Conference on Computer Software and Applications, IEEE Computer Society.
Mapelsden, D., J. Hosking and J. Grundy (2002). Design pattern modelling and instantiation using DPML. Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications. Sydney, Australia, Australian Computer Society, Inc.
Marcio, F. d. S. O., B. d. B. Lisane, C. Luigi and R. W. Flavio (2006). Early Embedded Software Design Space Exploration Using UML-Based Estimation. Proceedings of the Seventeenth IEEE International Workshop on Rapid System Prototyping, IEEE Computer Society.
Martin, S., R. Jan Peter and Z. Gerhard (1997). "A pattern-based application generator for building simulation." SIGSOFT Softw. Eng. Notes **22**(6): 468-482.
McCaffery, F., M. Pikkarainen and I. Richardson (2008). Ahaa --agile, hybrid assessment method for automotive, safety critical smes. Proceedings of the 30th international conference on Software engineering. Leipzig, Germany, ACM.
Melwa, C. and M. J. Pont (2004). Two Simple Patterns to Support the Development of Reliable Embedded Systems. Proceedings of The Second Nordic Conference on Pattern Languages of Programs, Bergen, Norway.
Meszaros, G. (2001). Design patterns in telecommunications system architecture. Design patterns in communications software, Cambridge University Press**:** 21-37.
Mikkonen, T. (1998). Formalizing design patterns. Proceedings of the 20th international conference on Software engineering. Kyoto, Japan, IEEE Computer Society.
Mohanty, S., V. K. Prasanna, S. Neema and J. Davis (2002). Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. Proceedings of the joint conference on Languages, compilers and tools for

embedded systems: software and compilers for embedded systems. Berlin, Germany, ACM.
Mostafa, A., M. , M. Ismail, A. , H. Bolok, E. L. and E. M. Saad (2007). Toward a Formalisation of UML2.0 Metamodel using Z Specifications. Proceedings of the Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007) - Volume 01, IEEE Computer Society.
Mwelwa, C. (2006). Development and assessment of a CASE tool to support the design and implementation of time-triggered embedded systems Embedded Systems Laboratory, Department of Engineering. Leicester, University of Leicester. **Ph. D.** .
Mwelwa, C., K. Athaide, D. Mearns, M. J. Pont and D. Ward (2007). "Rapid software development for reliable embedded systems using a pattern-based code generation tool." SAE Transactions: Journal of Passenger Cars (Electronic and Electrical Systems) **115**(7): 795-803.
Nascimento, F., A. M. do , M. Oliveira, F. da S. , M. Wehrmeister, A., C. Pereira, E. and F. Wagner, R. (2006). MDA-based approach for embedded software generation from a UML/MOF repository. Proceedings of the 19th annual symposium on Integrated circuits and systems design. Ouro Preto, MG, Brazil, ACM.
Neves, F. G. R. and O. Saotome (2008). Comparison between Redundancy Techniques for Real Time Applications. Proceedings of the Fifth International Conference on Information Technology: New Generations, IEEE Computer Society.
Ning J. Q., K. Miriyala, W. Kozaczynski (1994). "An Architecture-Driven, Business-Specific, and Component-Based Approach to Software Engineering. Proceedings of Third International Conference on Software Reuse: Advances in Software Reusability: 84 -93.
Niz, D. d. and R. Rajkumar (2003). Time weaver: a software-through-models framework for embedded real-time systems. Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems. San Diego, California, USA, ACM.
Norberg, A. L. (1984). "Another Impact of the Computer - The History of Computing." IEEE Transactions on Education **27**(4): 197-203.
Obermaisser, R. (2004). Event-Triggered and Time-Triggered Control Paradigms, Springer-Verlag TELOS.
Object Management Group. (2008). "Model Driven Architecture (MDA) Guide v1.0.1." Retrieved 27-November-2008, 2008, from http://www.omg.org/cgi-bin/doc?omg/03-06-01.
Oh, S. H. and S. M. Yang (1998). A Modified Least-Laxity-First Scheduling Algorithm for Real-Time Tasks. Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications, IEEE Computer Society.
Ong, R. H. L. and M. J. Pont (2002). "The impact of instruction pointer corruption on program flow: A computational modelling study." Microprocessors and Microsystems **25**: 409-419.
Paakki, J. (1995). "Attribute grammar paradigms—a high-level methodology in language implementation." ACM Computing Surveys. **27**(2): 196-255.
Peckham, J. and S. Lloyd, J. (2003). Integrating patterns into CASE tools. Practicing software engineering in the 21st century, IGI Publishing**:** 1-10.
Pont, M. J. (2001). Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers, Addison-Wesley/ACM Press.
Pont, M. J. (2003). "An object-oriented approach to software development in C for small embedded systems." Transactions of the Institute of Measurement and Control **25**(3): 217-238.
Pont, M. J. and M. P. Banner (2004). "Designing embedded systems using patterns: a case

study." Journal of Systems Software **71**(3): 201-213.
Pont, M. J., S. Kurian and R. Bautista-Quintero (2007). Meeting real-time constraints using 'Sandwich delays'. 11th European Conference on Pattern Languages of Programs. U. Zdun and L. Hvatum. Irsee, Germany, UVK Universitätsverlag Konstanz
Pont, M. J., S. Kurian, H. Wang and T. Phatrapornnant (2008). Selecting an appropriate scheduler for use with time-triggered embedded systems. 12th European Conference on Pattern Languages of Programs. L. Hvatum and T. Schummer. Irsee, Germany, UVK Universitätsverlag Konstanz 595-618.
Pont, M. J., A. J. Norman, Mwelwa C. and T. Edwards (2003). Prototyping time-triggered embedded systems using PC hardware. Proceedings of EuroPLoP 2003, Irsee, Germany.
Pont, M. J. and H. L. R. Ong (2003). Using watchdog timers to improve the reliability of TTCS embedded systems. Proceedings of the First Nordic Conference on Pattern Languages of Programs. P. Hruby and K. E. Soressen, Micrsoft Business Solutions**:** 159-200.
Porter, R. (2004). Mechanism design for online real-time scheduling. Proceedings of the 5th ACM conference on Electronic commerce. New York, NY, USA, ACM.
Pour,G. (1998). Component-Based Software Development Approach: New Opportunities and Challenges. Proceedings of Technology of Object-Oriented Languages. TOOLS 26:375 - 383
Prechelt, L., B. Unger, W. F. Tichy, P. Br, ssler and L. G. Votta (2001). "A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions." IEEE Trans. Softw. Eng. **27**(12): 1134-1144.
Pressman, R. S. (2001). Software Engineering: A Practitioner's Approach, McGraw-Hill Higher Education.
Pullum, L. L. (2001). Software Fault Tolerance Techniques and Implementation, Artech House.
Raje, R. R. and S. Chinnasamy (2001). eLePUS - a language for specification of software design patterns. Proceedings of the 2001 ACM symposium on Applied computing. Las Vegas, Nevada, United States, ACM.
Ramachandran, B., K. Fujiwara, M. Kano, A. Koide and J. Benayon (2006). Business process transformation patterns \& the business process transformation wizard. Proceedings of the 38th conference on Winter simulation. Monterey, California, Winter Simulation Conference.
Ramamritham, K. and J. A. Stankovic (1994). "Scheduling Algorithms and Operating Systems Support for Real-Time Systems." Proceedings of the IEEE **82**(1): 55-67.
Randell, B. (1975). System structure for software fault tolerance. Proceedings of the international conference on Reliable software. Los Angeles, California, ACM.
Randell, B., P. Lee and P. C. Treleaven (1978). "Reliability Issues in Computing System Design." ACM Comput. Surv. **10**(2): 123-165.
Rohr, J. A. (1995). "STAREX SELF-REPAIR ROUTINES: SOFTWARE RECOVERY IN THE JPL-STAR COMPUTER." Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'., Twenty-Fifth International Symposium on: 201-.
Romanovsky, A. (2007). "A looming fault tolerance software crisis?" ACM SIGSOFT Software Engineering Notes **32**(2): 1-4.
Rushby, J. M. (2001). Bus Architectures for Safety-Critical Embedded Systems. Proceedings of the First International Workshop on Embedded Software, Springer-Verlag.
Saunders, W. S. (2002). A Pattern Language: reviewed Harvard Design Magazine. **Winter-Spring 2002,** .
Scheler, F. and W. Schröder-Preikschat (2006). Time-Triggered vs. Event-Triggered: A matter of configuration? GI/ITG Workshop on Non-Functional Properties of Embedded Systems 27.03. - 29.03.2006. W. S.-P. Dulz, Wolfgang Nuremberg MMB

Workshop Proceedings, VDE Verlag**:** 107-112.
Schmidt, D., C. (1995). “Using design patterns to develop reusable object-oriented communication software.” Commun. ACM **38**(10): 65-74.
Schmidt, D. C. (2006). “Guest Editor's Introduction: Model-Driven Engineering.” Computer **39**(2): 25.
Schummer, T. and S. Lukosch (2006). “Structure-preserving transformations in pattern-driven groupware development.” International Journal of Computer Applications in Technology **25**(2/3): 155-166.
Selic, B. (2003). “The Pragmatics of Model-Driven Development.” IEEE Software **20**(5): 19-25.
Sendall, S. and W. Kozaczynski (2003). “Model Transformation: The Heart and Soul of Model-Driven Software Development.” IEEE Software **20**(5): 42-45.
Sha, L. and J. Goodenough (1988). Real-Time Scheduling Theory and ADA
Shaw, A. C. (2000). Real-Time Systems and Software, John Wiley \& Sons, Inc.
Sherif, M. Y., X. Hengyi and H. A. Hany (2000). Automating the Development of Pattern-Oriented Designs for Application Specific Software Systems. Proceedings of the 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'00), IEEE Computer Society.
Stankovic, J. A. and K. Ramamritham (1989). “The Spring kernel: a new paradigm for real-time operating systems.” SIGOPS Oper. Syst. Rev. **23**(3): 54-71.
Su, S. Y. H. and E. DuCasse (1980). “A Hardware Redundancy Reconfiguration Scheme for Tolerating Multiple Module Failures.” IEEE Transactions on Computers **29**(3): 254-258.
Taibi, T. and F. Taibi (2006). Formal Specification of Design Patterns and Their Instances. Proceedings of the IEEE International Conference on Computer Systems and Applications - Volume 00, IEEE Computer Society.
Tessier, P., S. Gerard, C. Mraidha and J.-M. Geib (2003). A Component-Based Methodology for Embedded System Prototyping. Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP'03), IEEE Computer Society.
Teyssot, G. (1983). Marginal Comments on the Debate Between Alexander and Eisenman. 40 LOTUS INTERNATIONAL. **IV:** 69-73.
Thompson, J., M. , M. Heimdahl, P. E. and S. Miller, P. (1999). Specification-based prototyping for embedded systems. Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering. Toulouse, France, Springer-Verlag.
Toole, B. A. (1991). “Ada, an analyst and a metaphysician.” ACM SIGAda Ada Letters **XI**(2): 60-71.
Torngren, M. (1998). “Fundamentals of Implementing Real-Time Control Applicationsin Distributed Computer Systems.” Real-Time Systems **14**(3): 219-250.
Torres, W. (2000). Software Fault Tolerance: A Tutorial, NASA Langley Technical Report Server.
Tsai, W.-T., L. Yu, F. Zhu and R. Paul (2005). “Rapid Embedded System Testing Using Verification Patterns.” IEEE Software **22**(4): 68-75.
TTE Systems Limited. (2008). “TTE Systems | Rapid development of reliable embedded systems.” Retrieved 10, December, 2008, from http://www.tte-systems.com/.
Tyrrell, A. M. (1996). Recovery blocks and Algorithm-Based Fault tolerance. EUROMICRO Conference, Proceedings of the 22nd EUROMICRO Conference, 1996.
Tyszberowicz, S. and A. Yehudai (1992). “OBSERV—a prototyping language and environment.” ACM Transactions on Software Engineering and Methodology (TOSEM) **1**(3): 269-309.
Uhl, A. and S. Ambler, W. (2003). “Point/Counterpoint.” IEEE Software **20**(5): 70-73.

Unger, B. and W. F. Tichy (2000). Do Design Patterns Improve Communication? An Experiment with Pair Design. Proc. Int'l Workshop Empirical Studies of Software Maintenance. G. Stark.
Vanderperren, Y. and W. Dehaene (2005). UML 2 and SysML: An Approach to Deal with Complexity in SoC/NoC Design. Proceedings of the conference on Design, Automation and Test in Europe - Volume 2, IEEE Computer Society. **2**.
Vokac, M. (2004). "Defect Frequency and Design Patterns: An Empirical Study of Industrial Code." IEEE Trans. Softw. Eng. **30**(12): 904-917.
Wall, L. (2000). Programming Perl, O'Reilly & Associates, Inc.
Wang, H., M. J. Pont and S. Kurian (2008). Patterns which help to avoid conflicts over shared resources in time-triggered embedded systems which employ a pre-emptive scheduler. 12th European Conference on Pattern Languages of Programs. L. Hvatum and T. Schummer. Irsee, Germany, UVK Universitätsverlag Konstanz 619-642.
Wehrmeister, M., A. , L. Becker, B. , F. Wagner, R. and C. Pereira, E. (2005). An Object-Oriented Platform-based Design Process for Embedded Real-Time Systems. Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, IEEE Computer Society.
Wetherbe, J., C. and M. Frolick, N. (2000). "Cycle time reduction: concepts and case studies." Communications of the AIS **3**(4es): 1.
Wilfredo, T. (2000). Software Fault Tolerance: A Tutorial, NASA Langley Technical Report Server.
Williams, M., R. (2006). What Does It Mean To Be The First Computer? Proceedings of the IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing, IEEE Computer Society.
Williams, M. R. (1983). "From Napier to Lucas: The Use of Napier's Bones in Calculating Instruments." IEEE Annals of the History of Computing **5**(3): 279-296.
Xu, J. and D. L. Parnas (1991). On satisfying timing constraints in hard-real-time systems. Proceedings of the conference on Software for citical systems. New Orleans, Louisiana, United States, ACM.
Yeh, Y. C. (1998). Design Considerations in Boeing 777 Fly-By-Wire Computers. The 3rd IEEE International Symposium on High-Assurance Systems Engineering, IEEE Computer Society.
Yoder, J. W., B. Foote, D. Riehle and M. Tilman (1998). Metadata and active object-models. OOPSLA'98 Addendum: Addendum to the 1998 Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM Press.
Zhang, W., T. Shaohua, Z. Zhaohui, F. Xiufen and Z. Haibin (2007). An Improved Least-Laxity-First Scheduling Algorithm of Variable Time Slice for Periodic Tasks. 6th IEEE International Conference on Cognitive Informatics**:** 548-553.
Ziegenbein, D., U. Freund, P. Braun, R. Sandner, A. Bauer, J. Romberg and B. Schätz (2005). AutoMoDe - Model-Based Development of Automotive Software. Proceedings of the conference on Design, Automation and Test in Europe - Volume 2, IEEE Computer Society. **2** 171-177 (ISBN: 0-7695-2288-2).
Zimmer, W. (1995). Relationships Between Design Patterns. J. O. Coplien and D. C. Schmidt. Reading, MA, Addison-Wesley**:** 345-364.

Part F: Appendices

A1.Pattern documentation examples

This appendix contains an example of an abstract pattern (TT SCHEDULER), a design pattern (TTC-ISR SCHEDULER) and a pattern implementation example (TTC-ISR SCHEDULER [C, LPC2000])

TT SCHEDULER

{generic pattern}

Context

- You are developing an embedded system.
- Your design is likely to employ a single processor.
- You are likely to employ a processor which has – compared with a desktop PC – significant resource constraints (e.g. limited memory, limited CPU performance).
- Predictable system behaviour is a key design requirement: in particular, predictable task timing is a concern.

Problem

Should you use a time-triggered (TT) scheduler as the basis of your embedded system (and, if so, which form of TT scheduler should you use)?

Background

This pattern is concerned with systems which have at their heart a TT scheduler. We will be concerned both with "time-triggered co-operative" (TTC) designs, "time-triggered rate-monotonic" (TTRM) designs and "time-triggered hybrid" (TTH) designs.

We provide some essential background material and definitions in this section.

What is a task?

Tasks are the building blocks of embedded systems. A task is simply a labelled segment of program code: in the systems we will be concerned with in this pattern, a task will generally be implemented using a function in the C programming language[9].

Working with periodic tasks

Most embedded systems will be assembled from collections of tasks. In this pattern, we will be concerned primarily with systems implemented using periodic tasks[10]. In our case, such tasks will be implemented as functions which are called – for example – every millisecond or every 100 milliseconds during some or all of the time that the system is active.

---

[9] A task implemented in this way does not need to be a "leaf" function: that is, a task may call (other) functions.

[10] The key requirement in a TT design is simply that we know in advance when a task is due to execute. One way of achieving this is to create the system using (only) periodic tasks, but other designs are possible. The present version of this pattern focuses on TT systems implemented using periodic tasks.

Jitter

The term "jitter" is used to refer to variations in the interval between events. For example, suppose a periodic task is due to start at the following times (in ms):

{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, …}

Suppose, instead, that it runs as follows:

{1.0, 2.1, 2.9, 4.0, 5.1, 6.2, …}

The specified times have no jitter (the interval between tasks is 1.0 ms). By contrast, the interval between the observed times varies between a minimum of 0.8 ms and a maximum of 1.1 ms (a worst-case variation of 20% of the sample interval).

Jitter can have a serious impact on a range of applications in which a TT architecture can be employed. For example, Cottet and David (1999) show that – during data acquisition tasks – jitter rates of 10% or more can introduce errors which are so significant that any subsequent interpretation of the sampled signal may be rendered meaningless. Similarly Jerri (1977) discusses the detrimental impact of jitter on applications such as spectrum analysis and filtering. Hong (1995) and Stothert and Macleod (1998) have discussed the degradation in performance caused by jitter in control applications. In such systems, jitter can greatly degrade the performance by varying the sampling period (Torgren, 1998; Mart et al., 2001).

Scheduling tasks (overview)

For many projects, a key challenge is to work out how to schedule these tasks so as to meet all of the timing constraints (including jitter constraints).

The scheduler we use can take two forms: pre-emptive and co-operative (or "non-pre-emptive"). The difference between these two forms is - superficially – rather small but has very large implications for our discussions in this pattern.

To illustrate this distinction, suppose that – over a particular period of time – we wish to execute four tasks (Task A, Task B, Task C, Task D) as illustrated in Figure 0.1.

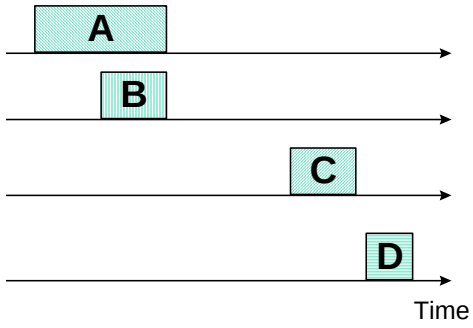


Figure 0.1: A schematic representation of four tasks (Task A, Task B, Task C, Task D) which we wish to schedule for execution in an embedded system with a single CPU.

We assume that we have a single processor. As a result, what we are attempting to achieve is shown in Figure 0.2.

Figure 0.2: Attempting the impossible: Task A and Task B are scheduled to run simultaneously.

In this case, we can run Task C and Task D as required. However, Task B is due to execute before Task A is complete. Since we cannot run more than one task on our single CPU, one of the tasks has to relinquish control of the CPU at this time.

In the simplest solution, we schedule Task A and Task B *co-operatively*. In these circumstances we (implicitly) assign a high priority to any task which is currently using the CPU: any other task must therefore wait until this task relinquishes control before it can execute. In this case, Task A will complete and then Task B will be executed (Figure 0.3).

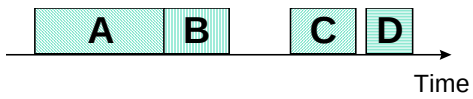


Figure 0.3: Scheduling Task A and Task B co-operatively.

Alternatively, we may choose a *pre-emptive* solution. For example, we may wish to assign a higher priority to Task B with the consequence that – when Task B is due to run – Task A will be interrupted, Task B will run, and Task A will then resume and complete (Figure 0.4).

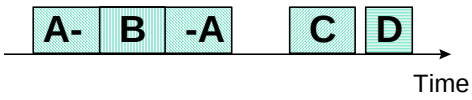


Figure 0.4: Assigning a high priority to Task B and scheduling the two tasks pre-emptively.

TTC architectures

Provided that an appropriate implementation is used, a time-triggered, co-operative (TTC) architecture is a good match for a wide range of low-cost, resource-constrained applications. TTC architectures also demonstrate very low levels of task jitter (Locke, 1992), and can maintain their low-jitter characteristics even when techniques such as dynamic voltage scaling (DVS) are employed to reduce system power consumption (Phatrapornnant and Pont, 2006).

The type of TTC scheduler implementation discussed in this paper is usually implemented using a hardware timer, which is set to generate interrupts on a periodic basis (with "tick intervals" of around 1 ms being typical). In most cases, the tasks will be executed from a "dispatcher" (function), invoked after every scheduler tick. The dispatcher examines each task in its list and executes (in priority order) any tasks which are due to run in this tick interval. The scheduler then places the processor into an "idle" (power saving) mode, where it will remain until the next tick.

Figure 0.5 shows an example of two tasks run with TTC scheduler with a tick interval of 1 ms. This "tick" is derived from a timer overflow: drift or jitter in this timing is, in large part, dependent on the associated computer hardware.

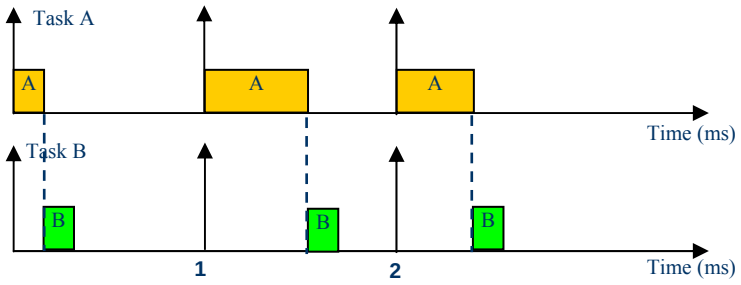


Figure 0.5: Illustrating the operation of a typical (interrupt-driven) TTC scheduler implementation.

TTRM architectures

Where a TTC architecture is not found to be suitable for use in a particular resource-constrained embedded systems, fixed-priority scheduling has been proposed as an attractive alternative (e.g. Audsley *et al.*, 1993; Bate, 1998).

"Time-triggered rate monotonic" (TTRM) is a well-known fixed-priority scheduling algorithm that was introduced by (Liu and Layland, 1973) in 1973. Technically, TTRM is a pre-emptive scheduling algorithm which is based on a fixed priority assignment. In particular, the priorities assigned to periodic tasks accord to their occurrence rate or, in other words, priorities are inversely proportional to their period, and they do not change through out of the operation (because their periods are constant).

The TTRM algorithm has been proved to be optimal amongst all fixed-priority algorithms (Liu and Layland, 1973): that is, Liu and Layland demonstrated that - if it is possible to schedule a task set using a fixed-priority algorithm and meet all of its timing constraints – then a TTRM algorithm can achieve this. Theoretically, every task can meet its deadline if the total CPU utilization is <= 69% and: all tasks are periodic and independent of each other; the deadline of every task is equal to its period; the worst-case execution time of all tasks is known; and, context switching time can be ignored (Liu and Layland, 1973; Locke, 1992; Bate, 1998; Buttazzo, 2004).

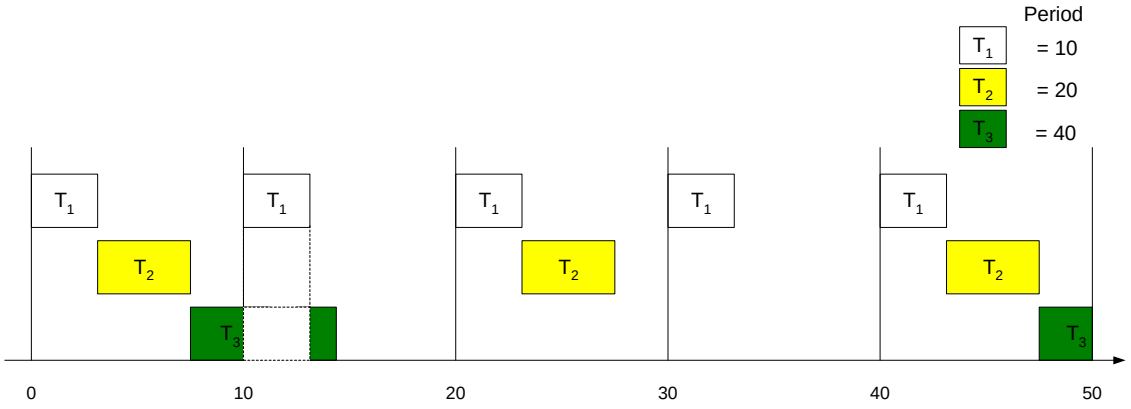


Figure 0.6: Structure of rate monotonic scheduling (adapted from Locke, 1992, Figure 3).

To illustrate the use of TTRM scheduling, Figure 0.6 shows how a set of periodic tasks can be scheduled by this algorithm. Task T_1 is executed periodically at the fastest rate, every 10 ms, and is determined to be the highest priority in this scheduling policy, while task T_2 and T_3, which are run every 20 and 40 ms respectively, have lower priority levels according to their rates. A task scheduled by the TTRM algorithm can be pre-empted by a higher priority task. As illustrated in Figure 0.6, task T_3 - which is running - is pre-empted by task T_1 is at time 10: it carries on after the completion of task T_1. Generally, the deadline of a task in TTRM scheduling is defined as the period: this assumption may have implications for task jitter levels (as we will

discuss further in "Solution").

TTH architectures

Where a TTC architecture is not found to be suitable for a particular system, use of a TTRM design may not be necessary. For example, a single, time-triggered, pre-empting task can be added to a TTC architecture, to give what we have called a "time-triggered hybrid" (TTH) scheduler (Pont, 2001; Maaita and Pont, 2005) and others have called a "multi-rate executive with interrupts" (Kalinsky, 2001): see Figure 0.7.

Use of a TTH scheduler allows the system designer to create a static schedule made up of (i) a collection of tasks which operate co-operatively and (ii) a single – short - pre-empting task[11,12]. In many of the systems employing a TTH architecture, the pre-empting task will be used for periodic data acquisition, typically through an analogue-to-digital converter or similar device. Such requirements are common in, for example, control systems (Buttazzo, 2005), and applications which involve data sampling and Fast-Fourier transforms (FFTs) or similar techniques: see, for example, the work by Schlindwein et al. (1988).

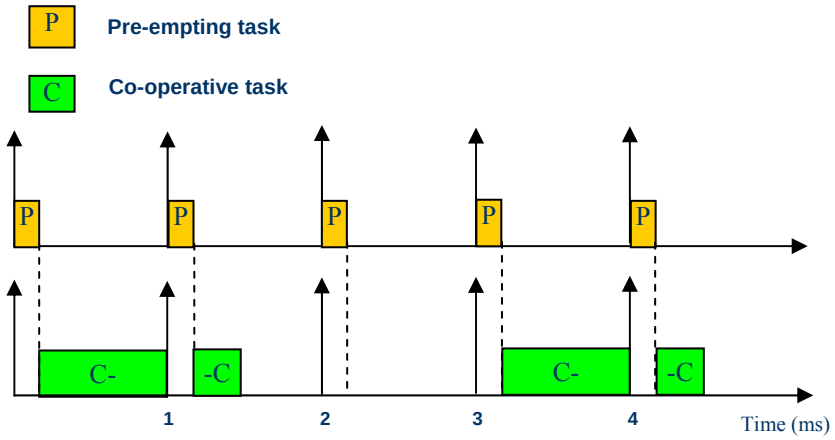


Figure 0.7: Illustrating the operation of a typical TTH scheduler implementation (adapted from Maaita and Pont, 2005, Figure 1).

Please note that it is not our intention to imply that a TTH architecture has – in terms of its scheduling behaviour – any particularly novel characteristics. Indeed, in many cases, a TTH architecture will be used with a very small number of tasks to implement a TTRM schedule. In

[11] In the TTH architecture, the co-operative tasks all have the same priority (Priority C). The – single – pre-emptive task has Priority P. Priority P > Priority C.

[12] Please note that, in the TTH architecture, both the "co-operative" task and the (single) "pre-empting" task are periodic. This is in contrast to architectures investigated in some previous studies (e.g. Sandstrom et al., 1988) which have sought to integrate time-triggered task scheduling with the response to aperiodic (event related) interrupts.

addition, it should be emphasised that we support in this architecture only a single pre-empting task (since this is all we require). As a consequence, in terms of a theoretical scheduling analysis, this type of scheduler is of limited interest. However, in a resource-constrained embedded system, it is a very attractive proposition because it allows us to create a scheduler with minimal resource requirements which is precisely matched to the needs of many practical applications.

Solution

This pattern is intended to help answer the questions:

"Should you use a time-triggered (TT) scheduler as the basis of your embedded system (and, if so, which form of TT scheduler should you use)?"

In this section, we will explain how you can determine whether a TT architecture is a good choice for your application, and – for situations where such an architecture is appropriate – we will provide an overview of different scheduler options, to help you select the most appropriate solution.

Overall, our argument will be that – to maximise the reliability of your design – you should use the simplest "appropriate architecture", and only employ the level of pre-emption that is essential to the needs of your application.

Should you use a TT architecture?

Some systems are "obvious" candidates for TT architectures. These systems include applications which involve data sampling or data playback, or other periodic activities (notably control algorithms).

Good uses for TT architectures include:

- Music players (for example, MP3 players) are required to play back music samples at a fixed – known – rate: the precise the rate will depend on the music quality: full "CD quality" sound will be played back at 44,400 samples per second. Any jitter in the playback times will result in a degradation in the music quality.
- Data acquisition and sensing systems (for example, environmental systems for temperature monitoring) usually involve making data samples on a periodic basis. Some cases (high-frequency systems) may involve making millions of samples per second: other cases (e.g. temperature monitoring at a weather station) may involve making one sample per hour. Whatever the rate, a TT architecture will usually be used to put the system "to sleep" between samples.
- Control systems (for example, cruise control in your car, temperature control in your central heating or air conditioning system, control of the hard disk in your computer, control of the industrial robots in a local factory or the toy robots you buy for your children). Such systems all involve three core – periodic – activities: measuring some aspect of the system to be controlled (e.g. the room temperature), calculating changes required to the control system (e.g. calculating what new settings are required to your air conditioning system) and applying the changes to the control system (e.g. altering the settings on the air conditioning). In almost every case, a control system will be implemented using a TT architecture.

Of course, not every system is a good match for a TT architecture. In particular, if your system must only respond to aperiodic events, a TT architecture may not be appropriate. For example, a radio transmitter used to open your garage doors may be used only a few times a week. We could use a TT architecture to poll the switch on this system every 20 ms, just in case the switch has been pressed (see Listing 0.1). However, while such a solution would work, it would be likely to use more energy (and have shorter battery life) than a simple "event triggered" design (which might, for example, operate in power-down mode, except when the "reset switch" on the unit was pressed: see Listing 5.1).

```
int main(void)
   {
   ...
   while(1)
      {
      Check_Switch();
      Control_RF_Transmitter();
      Delay_20ms();
      }

   // Should never reach here
   return 1
   }
```

Listing 0.1: A very simple implementation of a time-triggered co-operative scheduler which is being used to control a radio-frequency transmitter. .

```
// System is reset every time switch is pressed

int main(void)
   {
   Switch_On_RF_Transmitter();
   Delay_20ms();
   Switch_Off_RF_Transmitter();

   Enter_Power_Down_Mode();

   // Should never reach here
   while(1);
   return 1
   }
```

Listing 0.2: A one-shot architecture for control of an RF transmitter.

In a similar way (but very different timescale), some forms of engine management designs require responses to events which are highly aperiodic. Such designs may not be a good match for TT architectures.

We will provide some further examples of systems which might best be implemented using a TT architecture in the remainder of this pattern. In considering these possible designs, our argument will be that - if a TT architecture is appropriate for your system - then to maximise the reliability and minimise resource requirements - you should use the simplest "appropriate architecture", and only employ the level of pre-emption that is essential to the needs of your application.

When is it appropriate (and not appropriate) to use a pure TTC architecture?

Pure TTC architectures are a good match for a wide range of applications. For example, we have previously described in detail how these techniques can be in – for example - data acquisition systems, washing-machine control and monitoring of liquid flow rates (Pont, 2002), in various automotive applications (e.g. Ayavoo et al., 2004), a wireless (ECG) monitoring system (Phatrapornnant and Pont, 2006), and various control applications (e.g. Edwards et al., 2004; Key et al., 2004).

Of course, this architecture is not always appropriate. The main problem is that long tasks will have an impact on the responsiveness of the system. This concern is succinctly summarised by Allworth: *"[The] main drawback with this [co-operative] approach is that while the current process is running, the system is not responsive to changes in the environment. Therefore, system processes must be extremely brief if the real-time response [of the] system is not to be impaired."* (Allworth, 1981).

We can express this concern slightly more formally by noting that if the system must execute

one of more tasks of duration X and also respond within an interval T to external events (where T < X), a pure co-operative scheduler will not generally be suitable.

In practice, it is sometimes assumed that a TTC architecture is inappropriate because some simple design options have been overlooked. We will use two examples to try and illustrate how – with appropriate design choices – we can meet some of the challenges of TTC development.

Example: Multi-stage tasks

Suppose we wish to transfer data to a PC at a standard 9600 baud; that is, 9600 bits per second. Transmitting each byte of data, plus stop and start bits, involves the transmission of 10 bits of information (assuming a single stop bit is used). As a result, each byte takes approximately 1 ms to transmit.

Now, suppose we wish to send this information to the PC:

```
Current core temperature is 36.678 degrees
```

If we use a standard function (such as some form of `printf()`) - the task sending these 42 characters will take more than 40 milliseconds to complete. If this time is greater than the system tick interval (often 1 ms, rarely greater than 10 ms) then this is likely to present a problem (Figure 0.8).

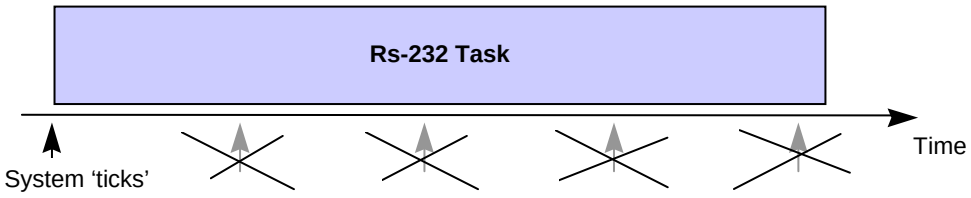


Figure 0.8: A schematic representation of the problems caused by sending a long character string on an embedded system with a simple operating system. In this case, sending the massage takes 42 ms while the OS tick interval is 10 ms.

Perhaps the most obvious way of addressing this issue is to increase the baud rate; however, this is not always possible, and - even with very high baud rates - long messages or irregular bursts of data can still cause difficulties.

A complete solution involves a change in the system architecture. Rather than sending all of the data at once, we store the data we want to send to the PC in a buffer (Figure 0.9). Every ten milliseconds (say) we check the buffer and send the next character (if there is one ready to send).

In this way, all of the required 43 characters of data will be sent to the PC within 0.5 seconds. This is often (more than) adequate. However, if necessary, we can reduce this time by checking the buffer more frequently. Note that because we do not have to wait for each character to be sent, the process of sending data from the buffer will be very fast (typically a fraction of a millisecond).

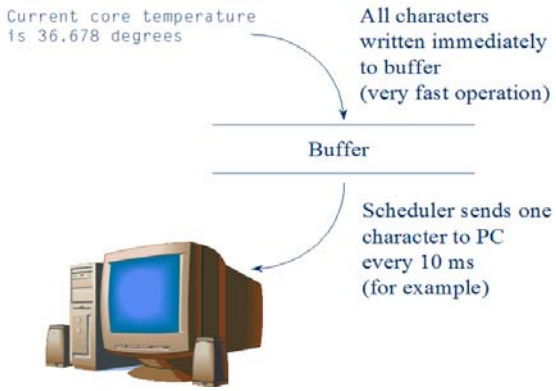


Figure 0.9: A schematic representation of the software architecture used in the RS-232 library.

This is an example of an effective solution to a widespread problem. The problem is discussed in more detail in the pattern MULTI-STAGE TASK (Pont, 2001).

Example: Rapid data acquisition

The previous example involved sending data to the outside world. To solve the design problem, we opted to send data at a rate of one character every millisecond. In many cases, this type of solution can be effective.

Consider another problem (again taken from a real design). This time suppose we need to receive data from an external source over a serial (RS-232) link. Further suppose that these data are to be transmitted as a packet, 100 ms long, at a rate of 115,200 baud and that one such packet will be sent every second for processing by our embedded system.

At this baud rate, data bytes will arrive approximately every 87 µs. To avoid losing data, we would – if we used the architecture outlined in the previous example – need to have a system tick interval of around 40 µs. This is a short tick interval, and would only produce a practical TTC architecture if a powerful processor was used.

However, a pure TTC architecture may still be possible, as follows. First, we set up an interrupt service routine (ISR), set to trigger on receipt of UART interrupts:

```
void UART_ISR(void)
   {
   // Get first char

   // Collect data for 100 ms (with timeout)
   }
```

These interrupts will be received roughly once per second, and the ISR will run for 100 ms. When the ISR ends, processing continues in the main loop:

```
void main(void)
   {

   ...

   while(1)
      {
      Process_UART_Data();
      Go_To_Sleep();
      }
   }
```

Here we have up to 0.9 seconds to process the UART data, before the next tick.

Pros and cons of TTRM

If a TTC architecture is not appropriate for your application, then a TTRM architecture may match your requirements.

Overall, it has been claimed that the main advantage of TTRM scheduling is flexibility during design or maintenance phases, and that such flexibility can reduce the total life cost of the system (Locke, 1992; Bate, 1998). The schedulability of the system can be determined based on the total CPU utilization of the task set: as a result - when new functionalities are added to the system – it is only necessary to recalculate the new utilization values. In addition, unlike a TTC design, there is no need to break up long individual tasks in order to meet the length limitations of the minor cycle. The need to employ harmonic frequency relationships among periodic tasks is also avoided. Finally, the scheduling behaviour can be predicted and analysed using a task model proposed by Liu and Layland (1973).

However, the scheduling overheads of TTRM schedulers tend to be larger than those of TTC schedulers because of the additional complexity associated with the context switches when saving and restoring task state (Locke, 1992). This is a concern in embedded systems with

limited resources.[13]

Of greater concern in this pattern is that RM scheduling seems likely to have more jitter than TTC scheduling, because the pre-emption from higher priority tasks may interrupt or block the lower priority tasks. These interferences may delay the release time of tasks, or interrupt running tasks and then prolong the output of a process residing at the end of a task: this may which result in jitter (Buttazzo, 2004).

Overall, in the type of "low jitter" application with which this pattern is concerned, use of an RM algorithm presents two main challenges.

The first challenge is that the RM algorithm is based on the assumption that task deadlines are equal to periods: this means that use of RM guarantees only that a given task will complete its execution before it is due to run again. For short tasks, this means that jitter rates may be in the region of 90% (of the sample period), and the schedule will still be "correct". In many cases, however, even jitter levels of 10% (of the sampling period) can render sampled data meaningless. Note that use of high task priorities will tend to reduce jitter levels: however – even if the tasks are wholly independent - the only safe assumption is that the highest-priority task will be guaranteed to demonstrate very low jitter levels (Locke, 1992).

The second challenge is that tasks are unlikely to be independent and that more than one task may require access to a mutually-exclusive resource (e.g. serial port, ADC and etc.). Where such critical sections are accessed through semaphores, even the highest-priority task may be blocked by a lower priority task (a process known as priority inversion) and then experience jitter or delay (Buttazzo, 2005). The priority inversion problem can be "solved" by using appropriate protocols (e.g. Priority Inheritance Protocol or Priority Ceiling Protocol, developed by Sha et al., 1990), to control access shared resources: however, such techniques were developed to address problems of deadlock and their impact on jitter is not always easy to predict.

---

[13] It has been argued that another popular pre-emptive scheduler ("Earliest Deadline First", EDF) has a lower runtime overhead than RM approaches (Buttazzo, 2005). Even though EDF always needs to update task deadlines this increased load may be offset by a reduction in the number of preemptions that occur under EDF (with a consequent reduction in context-switching time). Overall, Buttazzo (2005) suggests that the real advantage of TTRM scheduling is its simpler implementation. We would argue that TTRM also has (compared with a dynamic scheduling algorithm like EDF) more predictable behaviour and lower levels of task jitter. We say a little more about EDF in the "Related patterns" section of this pattern.

Don't forget the TTH option

Sometimes a TTC architecture cannot meet our needs, but a TTRM architecture may still be "overkill". For example, consider a wireless electrocardiogram (ECG) system (Figure 0.10).

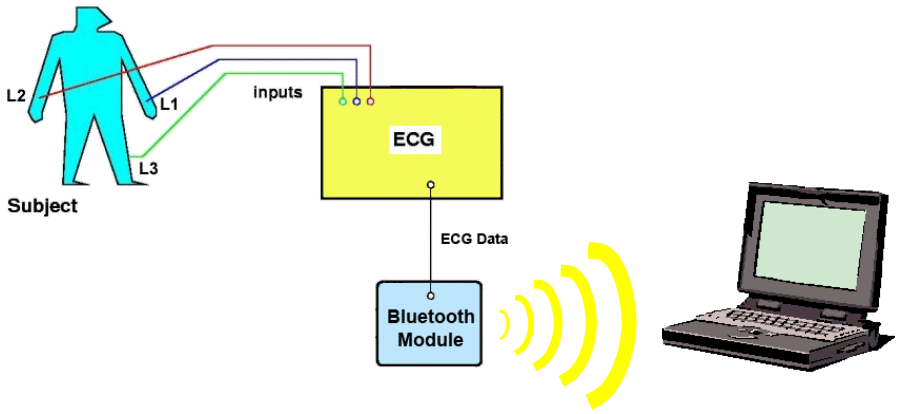


Figure 0.10: A schematic representation of a system for ECG monitoring. See Phatrapornnant and Pont (2006) for details.

An ECG is an electrical recording of the heart that is used for investigating heart disease. In a hospital environment, ECGs normally have 12 leads (standard leads, augmented limb leads and precordial leads) and can plot 250 sample-points per second (at minimum). In the portable ECG system considered here, three standard leads (Lead I, Lead II, and Lead III) were recorded at 500 Hz. The electrical signal were sampled using a (12-bit) ADC and – after compression – the data were passed to a "Bluetooth" module for transmission to a notebook PC, for analysis by a clinician (see Phatrapornnant and Pont, 2006)

In one version of this system, we are required to perform the following tasks:

- Sample the data continuously at a rate of 500 Hz. Sampling takes less than 0.1 ms.
- When we have 10 samples (that is, every 20 ms), compress and transmit the data, a process which takes a total of 6.7 ms.

In this case, we will assume that the compression task cannot be neatly decomposed into a sequence of shorter tasks, and we therefore cannot employ a pure TTC architecture. However, even if you cannot – cleanly - solve the long task / short response time problem, then you can maintain the core co-operative scheduler, and add only the limited degree of pre-emption that is required to meet the needs of your application.

For example, in the case of our ECG system, we can use a time-triggered hybrid architecture

(Figure 0.11).

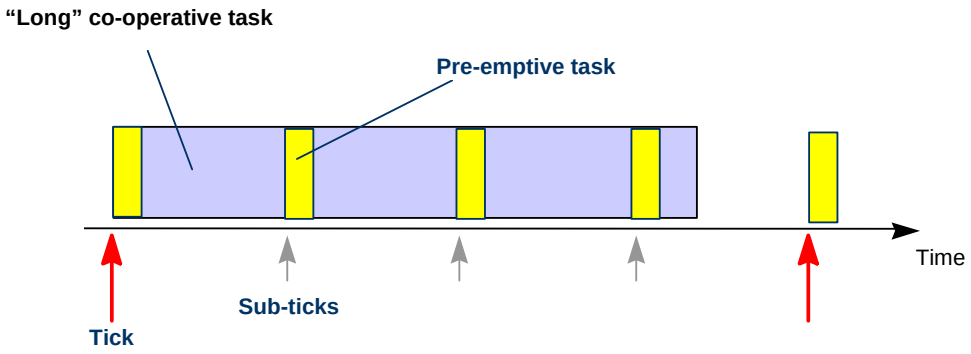


Figure 0.11: A "hybrid" software architecture. See text for details.

In this case, we allow a single pre-empting task to operate: in our ECG system, this task will be used for data acquisition. This is a time-triggered task, and such tasks will generally be implemented as a function call from the timer ISR which is used to drive the core TTC scheduler. As we have discussed in detail elsewhere (Pont, 2001: Chapter 17) this architecture is extremely easy to implement, and can operate with very high reliability. As such it is one of a number of architectures, based on a TTC scheduler, which are co-operatively based, but also provide a controlled degree of pre-emption.

Related patterns and alternative solutions

TTC-SL Scheduler

The simplest way of implementing a TTC scheduler is by means of a "Super Loop" or "endless loop" (e.g. Pont, 2001; Kurian and Pont, 2007). A possible implementation of such a scheduler is illustrated in Listing 5.2.

```
int main(void)
   {
   ...
   while(1)
      {
      TaskA();
      Delay_6ms();
      TaskB();
      Delay_6ms();
      TaskC();
      Delay_6ms();
      }

   // Should never reach here
   return 1
   }
```

Listing 0.3: A very simple cyclic executive (time-triggered co-operative scheduler) which executes three periodic tasks, in sequence.

If we assume that the tasks executed in Listing 5.2 always have a duration of 4 ms, then – through the use of the Super Loop and delay functions, we have created a system which has a 10 ms "tick interval" (Figure 5.3).

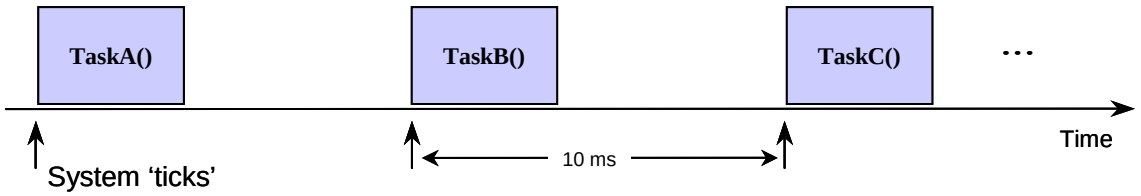


Figure 0.12: The task executions resulting from the code in Listing 5.2 (assuming all tasks are of duration 4 ms).

Applications based on a TTC-SL SCHEDULER have extremely small resource requirements. Systems based on such a pattern (if used appropriately) can be both reliable and safe, because the overall architecture is extremely simple and easy to understand, and no aspect of the underlying hardware is hidden from the original developer, or from the person who subsequently has to maintain the system.

TTC-ISR Scheduler

The pattern "TTC-ISR SCHEDULER" describes another very simple software architecture for small embedded systems. Like a TTC-SL SCHEDULER, the TTC-ISR implementation this is a "hard wired" table-based scheduler. Unlike TTC-SL SCHEDULER, TTC-ISR SCHEDULER is suitable for use with systems which have hard timing constraints. The particular implementation discussed in this section is based on that described in detail elsewhere (see: Pont, 2002).

The basis of a TTC-ISR SCHEDULER is an interrupt service routine (ISR) linked to the overflow of a hardware timer. For example, see Figure 5.4. Here we assume that one of the microcontroller's timers has been set to generate an interrupt once every 10 ms, and thereby call the function `Update()`. When not executing this interrupt service routine (ISR), the system is "asleep". The overall result is a system which - like that shown in Listing 5.2 – has a 10 ms "tick interval" in which three tasks are executed in sequence.

BACKGROUND PROCESSING

```
while(1)
   {
   Go_To_Sleep();
   }
```

FOREGROUND PROCESSING

10ms timer

```
void Update(void)
   {
   Tick_G++;

   switch(Tick_G)
      {
      case 1:
         Task_A();
         break;
      case 2:
         Task_B();
         break;
      case 3:
         Task_C();
         Tick_G = 0;
      }
   }
```

Figure 0.13: A schematic representation of a simple TTC-ISR Scheduler

Please note that "putting the processor to sleep" means moving it into a low-power ("idle") mode. Most processors have such modes, and their use can – for example – greatly increase battery life in embedded designs. Use of idle modes is common but not essential.

Whether or not idle mode is used, the timing observed is largely independent of the software used but instead depends on the underlying timer hardware (which will usually mean the accuracy of the crystal oscillator driving the microcontroller). One consequence of this is that (for the system shown in Figure 5.4, for example), the successive function calls will take place at precisely-defined intervals (Figure 5.5), even if there are large variations in the duration of `Update()`. This is very useful behaviour, and is not easily obtained with architectures such as TTC-SL SCHEDULER.

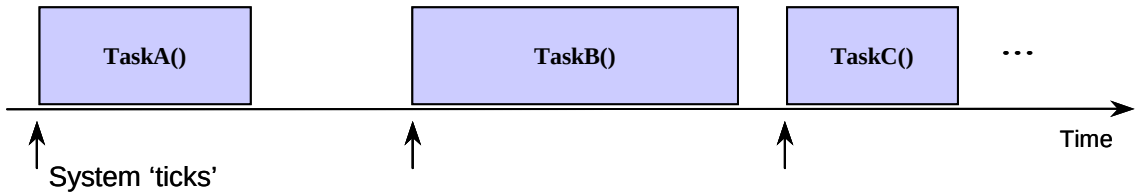


Figure 0.14: One advantage of the interrupt-driven approach is that the tasks will not normally suffer from "jitter" in their start times.

TTC Scheduler

The implementation of a TTC-ISR SCHEDULER is highly system dependent. In addition, the implementation requires a significant amount of hand coding (to control the task timing), and there is no division between the "scheduler" code and the "application" code.

The TTC scheduler implementation referred to here as a "TTC-Dispatch" scheduler provides a more flexible alternative. The particular implementation discussed in this section is based on that described in detail elsewhere (see: Pont, 2001).

The TTC scheduler implementation considered in this section is characterised by distinct and well-defined scheduler functions (see Listing 5.3).

```
void main(void)
   {
   // Set up the scheduler
   SCH_Init_T2();

   // Init tasks
   TaskA_Init();
   TaskB_Init();

   // Add tasks (10 ms ticks)
   // Parameters are <filename>, <offset in ticks>, <period in ticks>
   SCH_Add_Task(TaskA, 0, 3);
   SCH_Add_Task(TaskB, 1, 3);
   SCH_Add_Task(TaskC, 2, 3);

   // Start the scheduler
   SCH_Start();

   while(1)
      {
      SCH_Dispatch_Tasks();
      SCH_Go_To_Sleep();
      }
   }
```

Listing 0.4: An overview of a possible TTC Scheduler implementation: see Pont (2001) for details.

In this paper, we summarise the operation of a TTC Scheduler which has been fully documented (Pont, 2001). We will refer to this implementation here as "TTC-2001". Please note that this scheduler provides support for "one shot" tasks and dynamic scheduling: these features are not considered in this paper.

The TTC-2001 scheduler is driven by periodic interrupts generated from an on-chip timer. When an interrupt occurs, the processor executes an "Update" function (see Listing 0.5: "Update" ISR of TTC-2001 scheduler.). In the Update function, the scheduler checks to see if any tasks are due to run and sets appropriate flags. After these checks are complete, a Dispatch function Listing 0.6) will be called, and the identified tasks (if any) will be executed. When not executing

the Update and Dispatch functions, the system will usually enter a low-power ("idle") mode (see Pont, 2001 for further details).

```
void SCH_Update(void) interrupt INTERRUPT_Timer_6_Overflow
   {
   tByte Index;

   // Clear T6 interrupt request flag
   T6IR = 0;

   // NOTE: calculations are in *TICKS* (not milliseconds)
   for (Index = 0; Index < SCH_MAX_TASKS; Index++)
      {
      // Check if there is a task at this location
      if (SCH_tasks_G[Index].pTask)
         {
         if (--SCH_tasks_G[Index].Delay == 0)
            {
            // The task is due to run
            SCH_tasks_G[Index].RunMe += 1;  // Incr.  the 'Run Me' flag

            if (SCH_tasks_G[Index].Period)
               {
               // Schedule rperiodic tasks to run again
               SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
               }
            }
         }
      }
   }
```

Listing 0.5: "Update" ISR of TTC-2001 scheduler.

```
void SCH_Dispatch_Tasks(void)
   {
   tByte Index;

   // Dispatches (runs) the next task (if one is ready)
   for (Index = 0; Index < SCH_MAX_TASKS; Index++)
      {
      if (SCH_tasks_G[Index].RunMe > 0)
         {
         (*SCH_tasks_G[Index].pTask)();  // Run the task

         SCH_tasks_G[Index].RunMe -= 1;   // Reset / reduce RunMe flag

         // Periodic tasks will automatically run again
         // - if this is a 'one shot' task, remove it from the array
         if (SCH_tasks_G[Index].Period == 0)
            {
            SCH_Delete_Task(Index);
            }
         }
      }
   // Report system status
   SCH_Report_Status();

   // The processor enters idle mode at this point
   SCH_Go_To_Sleep();
   }
```

Listing 0.6: Dispatch function of TTC-2001 scheduler.

TTH Dispatch Scheduler

There are numerous ways in which a TTH scheduler can be implemented. One possible

implementation of a "TTH Dispatch Scheduler" is described by Pont (2001).

Implementing a TTRM scheduler

If you are determined to implement a fully pre-emptive design, then Jean Labrosse (1999) and Anthony Massa (2003) discuss – in detail – the construction of such systems.

Alternative scheduling algorithms

We have considered a range of TT scheduling algorithms in this paper. There are, of course, various other alternatives. Briefly, these include techniques which schedule tasks:

- According to their deadline, with the "earliest deadline first" (EDF). For further details, see Liu and Layland (1973).
- According to their slack - or laxity – time, with the "least Laxity first" (LLF). For further details, see Chen (2002).
- According to their worst-case execution time: usually referred to as "shortest job first" (SJF) scheduling. See Stankovic and Ramamritham (1987) for further details.

We are not aware of patterns which describe how to implement these various schedulers.

Locking mechanisms

If you use any architecture which involves pre-emption (TTH or TTRM), you need to consider ways of preventing more than one task from accessing critical resources at the same time. Huiyan and Pont (this conference) describe a number of patterns which can help you to achieve this. See also SCOPED LOCKING in Buschmann et al. (2007).

Maximising reliability of pre-emptive designs

If using pre-emptive architectures, Jai Xu and David Parnas have worked for a number of years on what they call "pre-runtime scheduling". This approach has the potential to improve the reliability of TT designs which employ pre-emption. For further details, please see: Xu (1993); Xu and Parnas (1990); Xu and Parnas (1993); Xu and Parnas (2000).

Multi-processor alternatives

Finally, we should note that all of the patterns in this paper assume the use of a single-processor solution. Various time-triggered architectures for multi-processor systems have also been described: see, for example, Kopetz (1997); Herzner et al. (2006); Pont (2001); Ayavoo et al. (2007); Short and Pont (2007).

Reliability and safety implications

For reasons discussed in detail in the previous sections of this pattern, time-triggered co-operative schedulers are generally considered to be a highly appropriate platform on which to construct a reliable (and safe) embedded system.

Overall strengths and weaknesses

☺ Use of a TT scheduler tends to result in a system with highly predictable patterns of behaviour.

☹ Inappropriate system design using this approach can result in applications which have a comparatively slow response to external events.

Further reading

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

Audsley, N., Tindell, K. and Burns, A. (1993), *"The end of the line for static cyclic scheduling?"* Proceedings of the 5th Euromicro Workshop on Real-time Systems, Finland, pp. 36-41.

Ayavoo, D., Pont, M.J. and Parker, S. (2004) "Using simulation to support the design of distributed embedded control systems: A case study". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004). Published by University of Newcastle.

Ayavoo, D., Pont, M.J., Short, M. and Parker, S. (2007) "Two novel shared-clock scheduling algorithms for use with CAN-based distributed systems", *Microprocessors and Microsystems*, **31**(5): 326-334.

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

Bate, I.J. (1998) "Scheduling and timing analysis for safety critical real-time systems", PhD thesis, University of York, UK.

Bate, I.J. (2000) "Introduction to scheduling and timing analysis", in *"The Use of Ada in Real-Time System"* (6 April, 2000). IEE Conference Publication 00/034.

Bennett, S. (1994) *"Real-Time Computer Control"* (Second Edition) Prentice-Hall.

Buschmann, F., Henney, K. and Schmidt, D.C. (2007) "Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing" (Volume 4). Wiley. ISBN: 978-0-470-05902-9

Buttazzo, G. C. (2004), *"Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications"*, 2nd ed, Springer.

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

Cottet, F. and David, L. (1999) "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.

Edwards, T., Pont, M.J., Scotson, P. and Crumpler, S. (2004) "A test-bed for evaluating and comparing designs for embedded control systems". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004). Published by University of Newcastle.

Fohler, G. (1999) "Time Triggered vs. Event Triggered - Towards Predictably Flexible Real-Time Systems", Keynote Address, Brazilian Workshop on Real-Time Systems, May 1999.

Hartwich F., Muller B., Fuhrer T., Hugel R., Bosh R. GmbH, (2002), Timing in the TTCAN Network, Proceedings 8th International CAN Conference.

Herzner, W., Kubinger, W. and Gruber, M. (2006) "Triple-T - A system of patterns for reliable communication in hard real-time systems"; in D. Manolescu, M. Völter, J. Noble (eds): Pattern Languages of Program Design 5 (PLOPD5); pp.89-126; Software Engineering/Patterns Series, Addison-Wesley, Boston. ISBN 0-321-32194-4

Hong, S.H. (1995) "Scheduling algorithm of data sampling times in the integrated communication and control systems". IEEE Transactions on Control Systems Technology, 3(2): 225-230

Jerri, A.J. (1977) "The Shannon sampling theorem: its various extensions and applications a tutorial review", Proc. of the IEEE, vol. 65, n° 11, p. 1565-1596.

Kalinsky, D., 2001. Context switch, Embedded Systems Programming, 14(1), 94-105.

Key, S. and Pont, M.J. (2004) "Implementing PID control systems using resource-limited embedded processors". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004). Published by University of Newcastle.

Kopetz, H. (1997) "Real-time systems: Design principles for distributed embedded applications", Kluwer Academic.

Kurian, S. and Pont, M.J. (2007) "Maintenance and evolution of resource-constrained embedded systems created using design patterns", *Journal of Systems and Software*, **80**(1): 32-41.

Labrosse, J. (1999) "MicroC/OS-II: The real-time kernel", CMP books. ISBN: 0-87930-543-6.

Liu, C. L. and Layland, J. W. (1973), "Scheduling algorithms for multi-programming in a hard real-time environment", *Journal of the ACM*, **20**(1): 40-61.

Locke, C.D. (1992) "Software architecture for hard real-time systems: Cyclic executives vs. Fixed priority executives", *The Journal of Real-Time Systems*, 4: 37-53.

Maaita, A. and Pont, M.J. (2005) "Using 'planned pre-emption' to reduce levels of task jitter in a time-triggered hybrid scheduler". In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), *Proceedings of the Second UK Embedded Forum* (Birmingham, UK, October 2005), pp.18-35. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

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

Massa, A.J. (2003) "Embedded Software Development with eCOS", Prentice Hall. ISBN: 0-13-035473-2.

Phatrapornnant, T. and Pont, M.J. (2006) "Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling", *IEEE Transactions on Computers*, **55**(2): 113-124.

Pont, M.J. (2001) "Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers", Addison-Wesley / ACM Press. ISBN: 0-201-331381.

Pont, M.J. (2002) "Embedded C", Addison-Wesley. ISBN: 0-201-79523-X.

Proctor, F. M. and Shackleford, W. P. (2001), "Real-time Operating System Timing Jitter and its Impact on Motor Control", proceedings of the 2001 SPIE Conference on Sensors and Controls for Intelligent Manufacturing II, Vol. 4563-02.

Schlindwein, F.S.; Smith, M.J. and Evans, D.H. (1988) "Spectral analysis of Doppler signals and computation of the normalized first moment in real time using a digital signal processor", Medical & Biological Engineering & Computing, 26, pp. 228-232.

Sha, L., Rajkumar, R. and Lehoczky, J. P. (1990), *"Priority inheritance protocols: an approach to real-timesynchronization",* IEEE Transactions on Computers, **39**(9): 1175-1185.

Shaw, A.C. (2001) "Real-time systems and software" John Wiley, New York. [ISBN 0-471-35490-2]

Short, M.J. and Pont, M.J. (2007) "Fault-tolerant time-triggered communication using CAN", *IEEE Transactions on Industrial Informatics*, **3**(2): 131-142.

Stankovic, J. and Ramamritham, "The design of the spring kernel," *Proc. of the IEEE real-Time Systems Symposium, 1987*, pp. 146-157

Stothert A. and Macleod I.M. (1998) "Effect of timing jitter on distributed computer control system performance". Proceedings of 15th IFAC Workshop DCCS'98-Distributed Computer Control Systems, Villa Olmo: Pergamon Press, 1998, pp.25-30.

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

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

Xu , J. (1993) "Multiprocessor Scheduling of Processes with Release Times, Deadlines, Precedence, and Exclusion Relations,'' IEEE Transactions on Software Engineering, 19(2), pp. 139-154.

Xu , J. and Parnas, D.L. (1990) "Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations,'' IEEE Transactions on Software Engineering, 16(3), pp. 360-369.

Xu , J. and Parnas, D.L. (1993) "On Satisfying Timing Constraints in Hard-Real-Time Systems," IEEE Transactions on Software Engineering, 19(1), pp. 70-84.

Xu , J. and Parnas, D.L. (2000) "Priority Scheduling Versus Pre-Run-Time Scheduling,'' International Journal of Time-Critical Systems, 18, 7-23, Kluwer Academic Publishers.

TTC-SL SCHEDULER

{design pattern}

Context

- You have decided that a TT SCHEDULER will provide an appropriate basis for your embedded system.

and

- Your application will have a single periodic task (or a single transaction).
- Your task / transaction has soft or firm constraints.
- There is no risk of task overruns (or occasional overruns can be tolerated).
- You need to use a minimum of CPU and memory resources.

Problem

How can you implement a TT SCHEDULER which meets the above requirements?

Background

See TT SCHEDULER for relevant background information.

Solution

A TTC-SL SCHEDULER allows us to schedule a single periodic task. To implement such a scheduler, we need to do the following:

1. Determine the task period (that is, the interval between task executions).
2. Determine the worst case execution time (WCET) of the task.
3. The required delay value is task period – WCET.
4. Choose an appropriate delay function (e.g. SOFTWARE DELAY or HARDWARE DELAY: Pont, 2001) that meets the delay requirements.
5. Implement a suitable SUPER LOOP (Pont, 2001) containing a task call and a delay call.

For example, suppose that we wish to flash an LED on and off at a frequency of 0.5 Hz (that is, on for one second, off for one second, etc). Further suppose that we have a function - `LED_Flash_Update()` – that changes the LED state every time it is called. `LED_Flash_Update()` is the task we wish to schedule. It has a WCET of approximately 0, so we require a delay of 1000 ms. Listing 7 shows a TTC-SL SCHEDULER framework which will allow us to schedule this task as required.

```
#include “Main.h”
#include “Loop_Del.h”
#include “LED_Flas.h”

void main(void)
   {
   LED_Flash_Init();

   while (1)
      {
      LED_Flash_Update();
      Loop_Delay(1000);     // Delay 1000 ms
      }
   }
```

*Listing 7: Implementation of a **TTC-SL SCHEDULER***

Related patterns and alternative solutions

We highlight some related patterns and alternative solutions in this section.

- SOFTWARE DELAY
- HARDWARE DELAY
- TTC-ISR SCHEDULER
- TTC SCHEDULER

Reliability and safety implications

In this section we consider some of the key reliability and safety implications resulting from the use of this pattern.

Running multiple tasks

TTC-SL SCHEDULERS can be used to run multiple tasks with soft timing requirements. It is important that the worst-case execution time of each task is known before hand to set up the appropriate delay values.

Use of Idle mode and task jitter

The processor does not benefit from using idle mode. There is considerable jitter in scheduling tasks when using a SUPERLOOP in conjunction with a SOFTWARE DELAY.

What happens if a task overruns?

Task overruns are undesirable and can upset the proper functioning of the system.

Overall strengths and weaknesses

☺ Simple design, easy to implement

☺ Very small resource requirements

☹ Not sufficiently reliable for precise timing

☹ Low energy efficiency, due to inefficient use of idle mode

Further reading

Pont, M.J. (2001) "Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers", Addison-Wesley / ACM Press. ISBN: 0-201-331381.

Pont, M.J. (2002) "Embedded C", Addison-Wesley. ISBN: 0-201-79523-X.

Pont, M.J. (2004) "A "Co-operative First" approach to software development for reliable embedded systems", invited presentation at the UK Embedded Systems Show, 13-14 October, 2004. Presentation available here: www.le.ac.uk/eg/embedded

TTC-SL SCHEDULER [C, C167]

{pattern implementation example}

Context

- You wish to implement a TTC-SL SCHEDULER [this paper]
- Your chosen implementation language is C[14].
- Your chosen implementation platform is the C167 family of microcontrollers.

Problem

How can you implement a TTC-SL SCHEDULER for the C167 family of microcontrollers?

Background

-

Solution

Listing 8 shows a complete implementation of a "flashing LED" scheduler, based on the example in TTC-SL SCHEDULER (Solution section).

[14] The examples in the pattern were created using the Keil C compiler, hosted in a Keil uVision 3 IDE.

```
/*-------------------------------------------------------------------*-

   LED_167.C (7 November, 2001)

  ------------------------------------------------------------------

   Simple 'Flash LED' test function for C167 scheduler.

-*-------------------------------------------------------------------*/

#include “Main.h”
#include “Port.h”
#include “LED_167.h”

// ------ SFRs ------------------------------------------------------

sfr PICON = 0xF1C4;

// ------ Private variable definitions ----------------------------

static bit LED_state_G;

/*-------------------------------------------------------------------*-

  LED_Flash_Init()

  - See below.
-*-------------------------------------------------------------------*/
void LED_Flash_Init(void)
   {
   LED_state_G = 0;

   PICON = 0x0000;

   P2   = 0xFFFF;  // set port data register
   ODP2 = 0x0000;  // set port open drain control register
   DP2  = 0xFFFF;  // set port direction register
   }

/*-------------------------------------------------------------------*-

  LED_Flash_Update()

  Flashes an LED (or pulses a buzzer, etc) on a specified port pin.

  Must schedule at twice the required flash rate: thus, for 1 Hz
  flash (on for 0.5 seconds, off for 0.5 seconds) must schedule
  at 2 Hz.

-*-------------------------------------------------------------------*/
void LED_Flash_Update(void)
   {
   // Change the LED from OFF to ON (or vice versa)
   if (LED_state_G == 1)
      {
      LED_state_G = 0;
      LED_pin0 = 0;
      }
   else
      {
      LED_state_G = 1;
      LED_pin0 = 1;
      }
   }
```

Listing 8: Implementation of a simple task – LED_Flash_Update on C167 platform

```
//-------------------------------------------------------------------
//
//   File: Delay.C (v1.00)
//   Author: M.J.Pont
//   Date: 05/12/2002
//    Description: Simple hardware delays for C167.
//
//-------------------------------------------------------------------

#include “hardware_delay_167.h”

//-------------------------------------------------------------------
//
//  Hardware_Delay()
//
//  Function to generate N millisecond delay (approx).
//
//  Uses Timer 2 in GPT1.
//
//-------------------------------------------------------------------
void Hardware_Delay(const tWord N)
{
   tWord ms;

      // Using GPT1 for hardware delay (Timer 2)
      T2CON = 0x0000;
      T3CON = 0x0000;

      // Delay value is *approximately* 1 ms per loop
      for (ms = 0; ms < N; ms++)
    {
         // 20 MHz, prescalar of 8
         T2 = 0xF63C;  //  Load timer 2 register

         T2IR = 0;          // Clear overflow flag
         T2R = 1;           // Start timer

         while (T2IR == 0); // Wait until timer overflows

         T2R = 0;           // Stop timer
    }
}
```

Listing 9: Hardware Delay implemented on C167 platform.

```
void main(void)
   {
   // Prepare for the 'Flash_LED' task
   LED_Flash_Init();

   while(1)
      {
      LED_Flash_Update();
      Hardware_Delay(1000);
      }
   }
```

Listing 10: Using a simple SUPERLOOP architecture to schedule an LED_Flash_Update task. The LED continuously flashes on for 1s and off for 1s

Further Reading

-

TTC-ISR SCHEDULER

{design pattern}

Context

- You have decided that a TT SCHEDULER will provide an appropriate basis for your embedded system.

and

- Your application will have a single periodic task (or a single transaction).
- Your task / transaction has firm or hard timing constraints.
- There is no risk of task overruns (or occasional overruns can be tolerated).
- You need to use a minimum of CPU and memory resources.

Problem

How can you implement a TT SCHEDULER which meets the above requirements?

Background

See TT SCHEDULER for relevant background information.

Solution

TTC-ISR SCHEDULER is a simple but highly effective (and therefore very popular) implementation of a TT SCHEDULER.

The basis of a TTC-ISR SCHEDULER is an interrupt service routine (ISR) linked to the overflow of a hardware timer. For example, see Figure 15. Here we assume that one of the microcontroller's timers has been set to generate an interrupt once every 10 ms, and thereby call the function `Update()`. When not executing this interrupt service routine (ISR), the system is "asleep". The overall result is a system which has a 10 ms "tick interval" (sometimes called a "major cycle") which – in this case – involves execution of a transaction consisting of a sequence of three tasks.

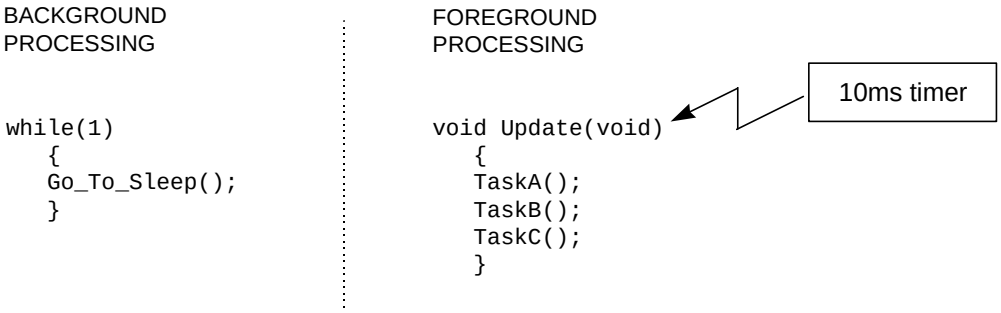


Figure 15: A schematic representation of a simple TTC SCHEDULER ("cyclic executive").

The end result of this activity is the sequence of function calls illustrated in Figure 16.

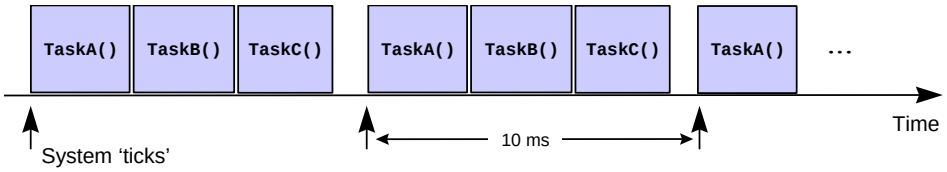


Figure 16: The sequence of task executions resulting from the architecture shown in Figure 15

Please note that "putting the processor to sleep" means moving it into a low-power ("idle") mode. Most processors have such modes, and their use can – for example – greatly increase battery life in embedded designs. Use of idle modes is common but not essential. For example, Figure 17 shows a simple implementation, with a single periodic task implemented directly using the Update function. In this case idle mode is not used.

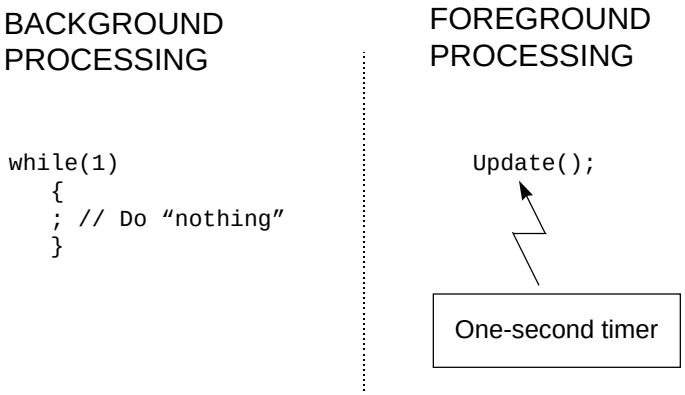


Figure 17: A schematic representation of the processes involved in using interrupts. See text for details.

Please note that – in both implementations - the timing observed is largely independent of the software used but instead depends on the underlying timer hardware (which will usually mean the accuracy of the crystal oscillator driving the microcontroller). One consequence of this is that (for the system shown in Figure 15, for example), the successive function calls will take place at precisely-defined intervals (Figure 18), even if there are large variations in the duration of `Update()`. This is very useful behaviour, and is not obtained with architectures such as

TTC-SL SCHEDULER.

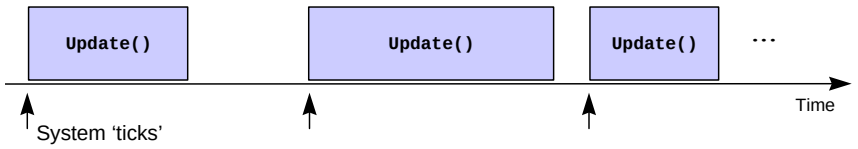


Figure 18: One advantage of the interrupt-driven approach is that the tasks will not normally suffer from "jitter" in their start times.

Hardware resource implications

We consider the hardware resource implications under three main headings: timers, memory and CPU load.

Timer

This pattern requires one hardware timer. If possible, this should be a timer, with "auto-reload" capabilities: such a timer can generate an infinite sequence of precisely-timed interrupts with minimal input from the user program.

Memory and CPU Load

The scheduler will consume no significant CPU resources: short of implementing the application using a TTC-SL SCHEDULER (with all the disadvantages of this rudimentary architecture), there is generally no more efficient way of implementing your application in a high-level language.

Reliability and safety implications

In this section we consider some of the key reliability and safety implications resulting from the use of this pattern.

Running multiple tasks

TTC-ISR SCHEDULER provides an excellent platform for executing a small number of tasks. If you need to run multiple (indirectly related) tasks, particularly tasks with different periods, then you can achieve this with a TTC-ISR SCHEDULER: however, the system will quickly become cumbersome, and may prove difficult to debug and / or maintain.

For systems with multiple tasks, please consider using a more flexible TTC approach, such as that described in CO-OPERATIVE SCHEDULER [Pont, 2001].

Safe use of idle mode

As we discussed in Solution, putting the processor to sleep means moving it into a low-power "idle" mode. Most processors have several power-saving modes: when selecting a suitable mode, make sure you choose one that (a) does not disable the timer you are using to generate the system ticks, and (b) allows the processor to enter the normal operating mode in the event of a

timer interrupt.

Note also that changing the processor mode may change the behaviour of other on-chip components (such as watchdog timers). You must ensure that any facilities required by your application remain operational in the idle mode which you choose.

Use of idle mode to reduce task jitter

In addition to saving power, use of idle mode can help to reduce task jitter.

This is the case because – on most processors – instructions take varying numbers of clock cycles to execute, and your processor can only respond to an interrupt when the currently-executing instruction has completed. For example, if your timer interrupt sometimes occurs when your processor is at the start of a 100-cycle instruction, and sometimes occurs at the start of a 2-cycle instruction, then the time taken to respond to the interrupt will vary considerably.

By contrast, if you use an idle mode, the time taken to return to the normal operating mode will be longer than the time taken to respond to interrupts if the processor is fully active – but the time will not generally vary.

Overall, using idle mode can usually reduce jitter, and reduce power consumption. The only drawback will be a very slight increase in the time taken to perform the task scheduling.

What happens if a task overruns?

With a TTC-ISR SCHEDULER, there will only be one active interrupt (the timer interrupt), and all tasks are called from the timer ISR. Because ISRs cannot interrupt themselves, there is no possibility that tasks in your system can be pre-empted. This results in highly predictable behaviour.

Although such behaviour is often highly desirable, it is important that you understand what happens if a task overruns. For example, suppose that you have a task that normally takes 1 ms to execute, and has to run every 10 ms. If –infrequently – this task takes 100 ms to execute, then the timer "ticks" that occur in this period will be ignored.

Inevitably, there are some applications for which this is not appropriate behaviour. For example, if you have a periodic task that keeps track of elapsed time (with a millisecond resolution), this task must run 60,000 times every minute, without fail, or your system will lose track of the current time.

To avoid losing ticks, you may need to separate the timer ISR and the process of task execution. Alternatively, you may need to consider using TTH SCHEDULER, or a TASK GUARDIAN.

Strengths and weaknesses

☺ An efficient environment for running a single periodic task or periodic transaction.

☹ Only appropriate for applications which can be implemented cleanly using a single task.

Related patterns and alternative solutions

Please also consider the following implementations of TT SCHEDULER:

- TTC-SL SCHEDULER
- TTC SCHEDULER
- TTC-ISR SCHEDULER can be particularly effective if used in combination with MULTI-STATE TASK.

Further reading

-

TTC-ISR SCHEDULER [C, LPC2000]

{pattern implementation example}

Context

- You wish to implement a TTC-ISR SCHEDULER [this paper]
- Your chosen implementation language is C[15].
- Your chosen implementation platform is the Philips LPC2000 family of (ARM7-based) microcontrollers.

Problem

How can you implement a TTC-ISR SCHEDULER for the Philips LPC2000 family of microcontrollers?

Background

Timers and interrupts on the LPC2000 family

The ARM core at the heart of the LPC2000 family has seven interrupt sources (see Table 1).

Interrupt	Description
Reset	Caused by a chip reset.
Undefined instruction	An attempt has been made to execute an instruction with is not recognised.
Software interrupt	The software interrupt instruction can be used for calls to an operating system (sometimes known as a "supervisor call").
Prefetch abort	Caused by an instruction fetch memory fault.
Data abort	Caused by a data fetch memory fault.
IRQ	Used for programmer-defined interrupts which are not handled in FIQ mode.
FIQ	This provides the fastest way of responding to programmer-defined interrupts. This is generally used for handling a single critical interrupt: in this paper, it will almost always be used for handling timer interrupts.

Table 1: Interrupt sources from the ARM7 core.

Behaviour is as follows (see Furber, 2000):

1. Change to the operating mode corresponding to the exception
2. Save the address of the next instruction in r14 of the new mode
3. Save the old value of the CPSR in the SPSR of the new mode

[15] The examples in the pattern were created using the GNU C compiler, hosted in a Keil uVision 3 IDE.

4. Disable IRQs by setting bit 7 of the CPSR and, if the exception is a fast interrupt, disable further fast interrupts by setting bit 6 of the CPSR

5. Set the PC to the relevant vector address (above table).

Normally the vector address will contain a branch to the relevant routine.

Return behaviour from exceptions is as follows (again from Furber, 2000):

1. Any modified user registers must be restored from the handler's stack.

2. The CPSR must be restored from the appropriate SPSR.

3. The PC must be changed back to the relevant instruction address in the user instruction stream.

Note that the FIQ mode has additional private registers to give better performance by avoiding the need to save user registers. It is therefore the logical way of handling our timer interrupt in this scheduler.

The process of handling an FIQ interrupt from the timer hardware in this way is summarised in **Error! Reference source not found.**.

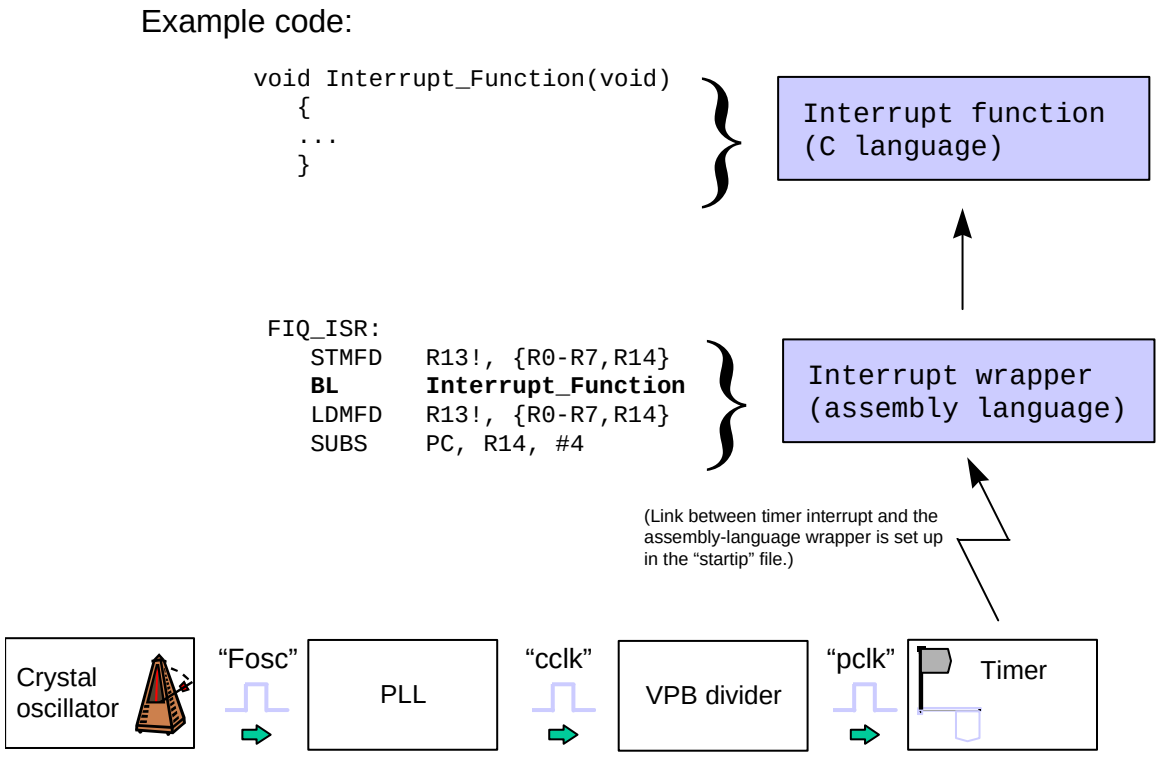


Figure 19: Interrupt handling (timers) in the LPC2000 family.

The operation of the phase-locked loop (PLL) and VLSI Peripheral Bus (VPB) divider may be clear from the code example that follows: if not, Philips (2004) provides further details.

Solution

A complete code example illustrating the implementation of a TTC-ISR SCHEDULER is given in

this pattern.

```
/*-------------------------------------------------------------------*-

   main.c (v1.00)

  ------------------------------------------------------------------

   A simple “Hello Embedded World” test program for LPC2129 family.

   (P1.16 used for LED output)

-*-------------------------------------------------------------------*/

// Device header (from Keil)
#include <lpc21xx.h>

// Oscillator / resonator frequency (in Hz)
// e.g. (10000000UL) when using 10 MHz oscillator
#define FOSC (12000000UL)

// Between 1 and 32
#define PLL_MULTIPLIER (5U)

// 1, 2, 4 or 8
#define PLL_DIVIDER (2U)

// 1, 2 or 4
#define VPB_DIVIDER (1U)

// CPU clock
#define CCLK (FOSC * PLL_MULTIPLIER)

// Peripheral clock
#define PCLK (CCLK / VPB_DIVIDER)

#define PLL_FCCO_MIN (156000000UL)
#define PLL_FCCO_MAX (320000000UL)

#define CCLK_MIN (10000000UL)
#define CCLK_MAX (60000000UL)


// Function prototypes
void LED_FLASH_ISR_Init(void);
void LED_FLASH_ISR_Change_State(void);

void System_Init(void);

int PLL_Init(void);
int VPB_Init(void);

void MAM_Init(void);
void Set_Interrupt_Mapping(void);

/*.................................................................*/

int main()
   {
   // Set up PLL, VPB divider and MAM (disabled)
   System_Init();

   // Prepare to flash LED
   LED_FLASH_ISR_Init();

   while(1) // Super Loop
      {
      // Enter idle mode
      PCON = 1;
      }

   // Should never reach here ...
   return 1;
   }
```

```
// ------ Private constants ---------------------------------------

// Interrupt mapping set through the “target” settings in the IDE
#ifndef RAM
#define MAP 0x01
#else
#define MAP 0x02
#endif


/*--------------------------------------------------------------------*-

  System_Init()

   Configures:
   - PLL
   - VPB divider
   - Memory accelerator module
   - Interrupt mapping

-*--------------------------------------------------------------------*/
void System_Init(void)
   {
   // Set up the PLL
   if (PLL_Init() != 0)
      {
      while(1);  // PLL error - stop
      }

   // Set up the VP bus
   if (VPB_Init() != 0)
      {
      while(1);  // VPB divider error - stop
      }

   // Set up the memory accelerator module
   MAM_Init();

   // Control interrupt mapping
   Set_Interrupt_Mapping();
   }
```

```
/*--------------------------------------------------------------------*-

  PLL_Init()

  Set up PLL.

-*--------------------------------------------------------------------*/

int PLL_Init(void)
   {
   unsigned int Fcco;
   unsigned int PLL_tmp;

   // Cclk will be PLL_MULTIPLIER * FOSC
   // Fcco will be PLL_MULTIPLIER * FOSC * 2 * PLL_DIVIDER

   // To allow us to check the frequencies
   Fcco = CCLK * PLL_DIVIDER * 2;

   // Check that the cclk frequency is OK
   if ((CCLK > CCLK_MAX) || (CCLK < CCLK_MIN))
      {
      return 1;  // Error
      }

   // Check that the CCO frequency is OK
   if ((Fcco > PLL_FCCO_MAX) || (Fcco < PLL_FCCO_MIN))
      {
      return 1;  // Error
      }

   // Set up PLLCFG register - the divider
   switch (PLL_DIVIDER)
      {
      case 1:
         PLL_tmp = 0;
         break;

      case 2:
         PLL_tmp = 0x20;
         break;

      case 4:
         PLL_tmp = 0x40;
         break;

      case 8:
         PLL_tmp = 0x40;
         break;

      default:
         return 1;  // Error
      }

   // Set up the PLLCFG register - now the multiplier
   PLL_tmp |= PLL_MULTIPLIER - 1;

   // Apply the calculated values
   PLLCFG |= PLL_tmp;

   PLLCON = 0x00000001;  // Enable the PLL

   PLLFEED = 0x000000AA; // Update PLL registers with feed sequence
   PLLFEED = 0x00000055;

   while (!(PLLSTAT & 0x00000400))  // Test Lock bit
      {
      PLLFEED = 0x000000AA; // Update PLL with feed sequence
      PLLFEED = 0x00000055;
      }

   PLLCON = 0x00000003;  // Connect the PLL
```

```
   PLLFEED = 0x000000AA; // Update PLL registers
   PLLFEED = 0x00000055;

   return 0;
   }


/*--------------------------------------------------------------------*-

  VPB_Init()

  Demonstrates setup of VPB divider

-*--------------------------------------------------------------------*/

int VPB_Init(void)
   {
   // Input to VPB divider is output of PLL (cclk)

   // VPB divider consists of two bits
   // 0 0 - VPB bus clock is 25% of processor clock [DEFAULT]
   // 0 1 - VPB bus clock is same as processor clock
   // 1 0 - VPB bus clock is 50% of processor clock
   // 1 1 - Reserved (no effect - previous setting retained)

   switch (VPB_DIVIDER)
      {
      case 1:
         VPBDIV &= 0xFFFFFFFC;
         VPBDIV |= 0x00000001;
         break;

      case 2:
         VPBDIV &= 0xFFFFFFFC;
         VPBDIV |= 0x00000002;
         break;

      case 4:
         VPBDIV &= 0xFFFFFFFC;
         break;

      default:
         return 1;  // Error
      }

   // OK
   return 0;
   }


/*--------------------------------------------------------------------*-

  MAM_Init()

  Set up the memory accelerator module.

  NOTE: Here we DISABLE the MAM, for maximum predictability.

  Adapt as needed for your application.

-*--------------------------------------------------------------------*/
void MAM_Init(void)
   {
   // Turn off MAM
   MAMCR = 0;
   }
```

```
/*---------------------------------------------------------------------*-

  Set_Interrupt_Mapping()

  Remaps interrupts to RAM or Flash memory, as required.

  For Flash, MAP = 0x01
  For RAM, MAP = 0x02

  Here, value is set through Keil uVision
  (depending on target built).

-*---------------------------------------------------------------------*/
void Set_Interrupt_Mapping(void)
   {
   MEMMAP = MAP;
   }


/*---------------------------------------------------------------------*-

  LED_FLASH_ISR_Init()

  Prepare for LED_FLASH_ISR_Change_State() function - see below.

-*---------------------------------------------------------------------*/
void LED_FLASH_ISR_Init(void)
   {
   // First, set up the timer
   // We require a “tick” every 1000 ms
   // (Timer is incremented PCLK times every second)
   T0MR0 = PCLK - 1;

   T0MCR = 0x03;  // Interrupt on match, and restart counter
   T0TCR = 0x01;  // Counter enable

   VICIntSelect = 0x10;   // Assign “Interrupt 4” to the FIQ category
   VICIntEnable = 0x10;   // Enable this interrupt

   // Now set the mode of the I/O pin
   // using the appropriate pin function select register

   // Here we assume that Pin 1.16 is being used.

   // First, set up P1.16 as GPIO
   // Clearing Bit 3 in PINSEL2 configures P1.16:25 as GPIO
   PINSEL2 &= ~0x0008;

   // Now set P1.16 to output mode
   // through the appropriate IODIR register
   IODIR1 = 0x00010000;
   }
```

```
/*---------------------------------------------------------------------*-

  LED_FLASH_ISR_Update()

  Changes the state of an LED (or pulses a buzzer, etc) on a
  specified port pin.

  Must call at twice the required flash rate: thus, for 1 Hz
  flash (on for 0.5 seconds, off for 0.5 seconds),
  this function must be called twice a second.

-*---------------------------------------------------------------------*/
void LED_FLASH_ISR_Update(void)
   {
   static int LED_state = 0;

   // Change the LED from OFF to ON (or vice versa)
   if (LED_state == 1)
      {
      LED_state = 0;
      IOCLR1 = 0x10000;
      }
   else
      {
      LED_state = 1;
      IOSET1 = 0x10000;
      }

   // After interrupt, reset interrupt flag (by writing “1”)
   T0IR = 0x01;
   }


/*---------------------------------------------------------------------*-

  LOOP_DELAY_Wait()

  Delay duration varies with parameter.

  Parameter is, *ROUGHLY*, the delay, in milliseconds,
  on 12.0 MHz LPC2129 (no PLL used).

  You *WILL* need to adjust the timing for your application!

-*---------------------------------------------------------------------*/
void LOOP_DELAY_Wait(const unsigned int DELAY)
   {
   unsigned int x,y,z;

   for (x = 0; x <= DELAY; x++)
      {
      for (y = 0; y <= 1000; y++)
         {
         z = z + 1;
         }
      }
   }
```

```
/*-------------------------------------------------------------------*-

  Trap_Interrupts()

  Interrupt trap - see Chapter 3 (Pont, 2001).

-*-------------------------------------------------------------------*/
void Trap_Interrupts(void)
   {
   // *** Basic behaviour ***
   // DISABLE ALL INTERRUPTS
   VICIntEnClr = 0xFFFFFFFF;
   while(1);
   }


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

Listing 11: Implementing a TTC-ISR SCHEDULER for the LPC2000 family (example).

Further reading

Philips (2004) "LPC2119 / 2129 / 2194 / 2292 / 2294 User Manual", Philips Semiconductors, 3 February, 2004.

Furber, S. (2000) "*ARM System-on-Chip Architecture*", Addison-Wesley.

TTC SCHEDULER

{design pattern}

Context

- You are developing an embedded application using the TT SCHEDULER.
- You need to schedule many tasks in your system
- You need to schedule one or more periodic (co-operative) tasks.
- You may need to schedule one or more aperiodic (co-operative) tasks.

Problem

How can you implement a TT SCHEDULER which meets the above requirements?

Background

See TT SCHEDULER for relevant background information.

Solution

We consider a TTC SCHEDULER made up of 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).

We consider each of the required components in this section.

Overview

Before looking at the individual components, we consider how the scheduler will typically appear to the user. To do this we will use a simple example: a scheduler used to flash a single LED on and off, repeatedly: on for one second, off for one second, etc.

1. We assume that the LED will be switched on and off by means of a 'task' `LED_Flash_Update()`. Thus, if the LED is initially off and we call `LED_Flash_Update()` twice, we assume that the LED will be switched on, and then switched off again.

To obtain the required flash rate, we therefore require that the scheduler calls `LED_Flash_Update()` every second *ad infinitum*.

2. We prepare the scheduler using the function `SCH_Init()`.

3. After preparing the scheduler, we add the function `LED_Flash_Update()` to the scheduler task list using he `SCH_Add_Task()` function. At the same time we specify that the LED will be turned on and off at the required rate as follows:

```
// Add the 'Flash LED' task (on for ~1000 ms, off for ~1000 ms)
// - timings are in ticks (1 ms tick interval)
// (Max interval / delay is 65535 ticks)
SCH_Add_Task(LED_Flash_Update, 0, 1000);
```

4. The timing of the `LED_Flash_Update()` function will be controlled by the function `SCH_Update()`, an interrupt service routine triggered by the overflow of a timer:

```
void SCH_Update(void) // Timer-related ISR
   {
   // Increment tick variable
   ...
   }
```

5. The 'Update' ISR does not execute the task: it calculates when a task is due to run, and sets a flag. The job of executing `LED_Flash_Update()` falls the dispatcher function (`SCH_Dispatch_Tasks()`), which runs in the main ('super' loop):

```
while(1)
   {
   SCH_Dispatch_Tasks();
   }
```

Before considering these components in detail, we should acknowledge that this is - undoubtedly - a complicated way of flashing an LED: if our intention was to develop an LED flasher application that required minimal memory and minimal code size, then this would **not** be a good solution. However, the key point is that we will be able to use **the same scheduler architecture** for building other systems, including a number of substantial and complex applications, and the effort required to understand the operation of this environment will be rapidly repaid.

It should also be emphasised that the scheduler is a 'low cost' option: it consumes a small percentage of the CPU resources (we will consider precise percentages shortly). In addition, the scheduler itself requires no more than 7 bytes of memory for each task. Since a typical application will require no more than four to six tasks, the task-memory budget (around 40 bytes) is not excessive, even on an 8-bit microcontroller.

The scheduler data structure and task array

At the heart of the scheduler is the scheduler data structure: this is a user-defined data type which collects together the information required about each task. Each task that is scheduled in the system must have a reference to some user code (a function defined by the user). It should

also have basic timing information determined by the system designer. For eg: the task interval that it is due to be first executed in and the time period at which the task is called for periodic execution. All this information is stored in a single data structure. The task period can be set to 0 to denote an aperiodic task.

The different tasks in the system that need to be scheduled are then stored as an array of this 'task' element. The scheduler processes the information in the task array, so that the tasks in any TTC system are processed as required. It is however important to store information about the maximum number of tasks that need to be scheduled at any instance. This way we can ensure that unused memory is not allocated to the task array.

A simple C-implementation of such a user defined data type is given in Listing 12.

```
typedef data struct
   {
   // Pointer to the task (must be a 'void (void)' function)
   void (code * pTask)(void);

   // Delay (ticks) until the function will (next) be run
   // - see SCH_Add_Task() for further details
   tWord Delay;

   // Interval (ticks) between subsequent runs.
   // - see SCH_Add_Task() for further details
   tWord Period;

   // Set to 1 (by scheduler) when task is due to execute
   tByte RunMe;
   } sTask;
```

Listing 12: Defining a task object using the 'struct' construct in C

Once the basic task element is defined the queue of tasks is defined as an array of task elements, as shown in Listing 13.

```
// The array of tasks
sTask SCH_tasks_G[SCH_MAX_TASKS];
```

Listing 13: Defining the task array which stores tasks to be scheduled

The initialisation function

Like most of the tasks we wish to schedule, the scheduler itself requires an initialisation function. While this performs various important operations - such as preparing the scheduler array (discussed above) and the error code variable (discussed below) - the main purpose of this function is to set up a timer that will be used to generate the regular 'ticks' that will drive the scheduler.

The designer generally needs to adapt the initialisation code to match the system requirements. In particular, we will need to ensure that:

1. The oscillator / resonator frequency assumed in the initialisation function matches thehardware.

2. The tick interval of the scheduler matches your requirements.

Listing 14 shows the code fragment that initialises the timers in an LPC2000 implementation.

```
// Set up required match register
TIMER0_MR0 = ((PCLK / 1000U) * TICK_MS) - 1;
TIMER0_MCR = 0x03;   // Interrupt on match, and automatically

// Ox10 -> 0b10000; Set bit 4 in these registers ...
VICIntSelect |= 0x10;   // Assign "Interrupt 4" to the FIQ category
VICIntEnable |= 0x10;   // Enable this interrupt
```

Listing 14: Initialising timers and interrupts in the 'Init' function

Guidance on the choice of the tick interval is provided below in the section 'Reliability and safety implications'.

The 'Update' function

The 'Update' function is the scheduler ISR. It is invoked by the overflow of the timer (set up using the 'Init' function, as discussed in the previous section).

```
{
// Note that an interrupt has occured
Tick_count_G++;

// After interrupt, reset interrupt flag (by writing "1")
TIMER0_IR = 0x01;
```

Listing 15: Registering a 'tick' in the Timer -ISR

Like most of the scheduler, the update function is not complex (Listing 14). When it determines that a task is due to run, the update function modifies the timing information of the task instance which in turn indicates that a task is due to be executed in the coming interval: the task will then be executed by the dispatcher, as we discuss below.

The 'Add Task' function

As the name suggests, the 'Add Task' function is used to add tasks to the task array, to ensure that they are called at the required time(s). The basic 'add task' function takes 3 parameters: a reference to a user function (task definition), the initial delay in tick intervals before the task can be executed for the first time and finally the period (in ticks) when the task is called repeatedly. The parameters for the 'add task' function are described in Figure 20.

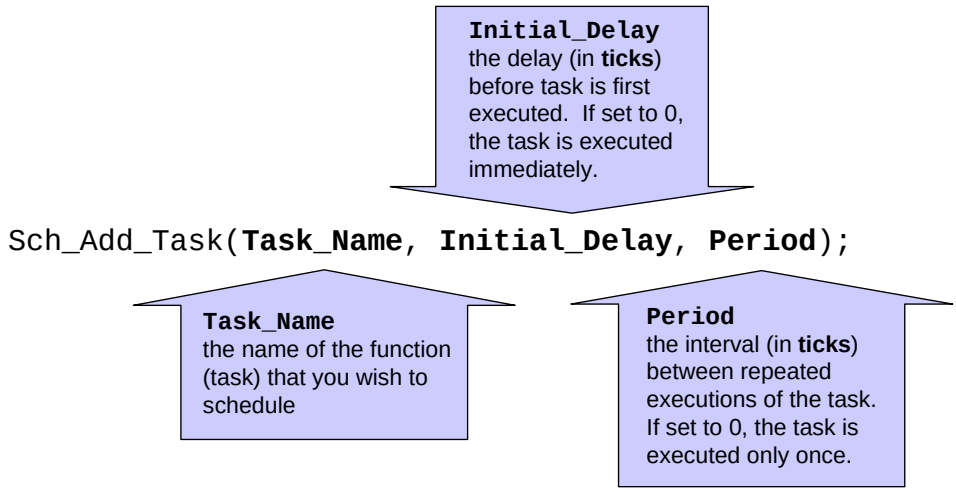


Figure 20: ***The parameters of the SCH_Add_Task() function***

Here are some examples.

This set of parameters causes the function `Do_X()` to be executed once after 1000 scheduler ticks:

```
SCH_Add_Task(Do_X,1000,0);
```

This does the same, but saves the task ID (the position in the task array) so that the task may be subsequently deleted, if necessary (see 'Delete Task' function for further information about the deleting of tasks):

```
Task_ID = SCH_Add_Task(Do_X,1000,0);
```

This causes the function `Do_X()` to be executed regularly, every 1000 scheduler ticks; the task will first be executed as soon as the scheduling is started:

```
SCH_Add_Task(Do_X,0,1000);
```

This causes the function `Do_X()` to be executed regularly, every 1000 scheduler ticks; task will be first executed at T = 300 ticks, then 1300, 2300, etc:

```
SCH_Add_Task(Do_X,300,1000);
```

The 'Dispatcher'

As we have seen above, the 'Update' function does not execute any tasks: the tasks that are due to run are invoked through the 'Dispatcher' function. Suppose we have a scheduler with a tick interval of 1ms and - for whatever reason - a scheduled task sometimes has a duration of 3ms. The tasks are called as shown in

```
if (--SCH_tasks_G[Index].Delay == 0)
    {
    // The task is due to run
    (*SCH_tasks_G[Index].pTask)();  // Run the task
```

Listing 16: Executing tasks from the 'Dispatch' function

If the Update function runs the functions directly then - all the time the long task is being executed - the tick interrupts are effectively disabled. Specifically, two 'ticks' will be missed. This will mean that all system timing is seriously affected, and may mean that two (or more) tasks are not scheduled to execute at all.

If the Update and Dispatch function are separated, then system ticks can still be processed while the long task is executing. This means that we will suffer task 'jitter' (the 'missing' tasks will not be run at the correct time), but these tasks will, eventually, run.

The 'Start' function

The 'start' function is very simple. After all the tasks have been added, this function is called to begin the scheduling process. (Listing 17) The function achieves this by globally enabling interrupts in an 8051 implementation.

```
void SCH_Start(void)
   {
   TIMER0_TCR |= 0x01;    // Counter enable (Timer Counter Register)
   }
```

Listing 17: Starting the timer (LPC2000 family)

The 'Delete Task' function

When tasks are added to the task array, the 'add task' function returns the position in the task array at which the task has been added:

```
Task_ID = SCH_Add_Task(Do_X,1000,0);
```

Sometimes it can be necessary to delete tasks from the array. To do so, a 'delete task' function can be used as follows:

```
SCH_Delete_Task(Task_ID);
```

Reducing power consumption

An important feature of scheduled applications is that they can lend themselves to low-power operation. This is possible because all current members of many controllers provide an 'idle' mode, where the CPU activity is halted, but the state of the processor is maintained. In this mode, the power required to run the processor is typically reduced by around 50%.

```
void SCH_Go_To_Sleep()
   {
   // Lots of further power saving that can be done ...
   // - see user manual
   PCON = 1;
```

Listing 18: Sleep or power-save mode (LPC2000)

Listing 18 shows how the processor sleep mode can be used. This idle mode is particularly effective in scheduled applications because it may be entered under software control and the micro-controller returns to the normal operating mode when any interrupt is received. Because the scheduler generates regular timer interrupts as a matter of course, we can put the system 'to sleep' at the end of every dispatcher call: it will then wake up when the next timer tick occurs.

Reporting errors

Hardware fails; software is never perfect; errors are a fact of life. To report errors at any part of the scheduled application, we could use an (8-bit) error code variable Error_code_G.

To report these error codes, the scheduler has a 'report error' function, which is called from the Update function. Error reporting is an optional feature. The scheduler will work without an error reporting mechanism. Adding an error reporting feature, makes it easier to maintain the system and analyse any faults. The 8-bit error code can be written to a port periodically.

The simplest way of displaying these codes is to attach eight LEDs (with suitable buffers) to the error port, as discussed in IC DRIVER. (PONT, 2001)

Reliability and safety implications

In this section we consider some key reliability and safety implications.

Make sure the task array is large enough

See 'Solution' for details.

Take care with function pointers

See 'Background' and 'Solution' for details.

Dealing with task overlap

Suppose we have two tasks in our application (Task A, Task B). We further assume that Task A is to run every second, and Task B every three seconds. We assume also that each task has duration of around 0.5 ms.

Suppose we schedule the tasks as follows (assuming a 1ms tick interval):

```
SCH_Add_Task(TaskA,0,1000);
SCH_Add_Task(TaskB,0,3000);
```

In this case, the two tasks will sometimes be due to execute at the same time. On these occasions, both tasks will run, but Task B will always execute after Task A (see the code for

SCH_Add_Task() for details). This will mean that if Task A varies in duration, then Task B will suffer from 'jitter': it will not be called at the correct time when the tasks overlap.
Alternatively, suppose we schedule the tasks as follows:

```
SCH_Add_Task(TaskA,0,1000);
SCH_Add_Task(TaskB,5,3000);
```

Now, both tasks still run every 1000 ms and 3000 ms (respectively), as required. However, Task A is explicitly scheduled to run, always, 5 ms before Task B. As a result, Task B will always run on time.
In many cases, we can avoid all (or most) task overlaps simply by the judicious use of the initial task delays.

Determining the required tick interval

Since, our main focus is in applications which operate on a millisecond timescale. Thus, the various tasks you will be adding to the scheduler will typically have task intervals of (say) 12 ms, 3 ms and 1000 ms.
In most instances, the simplest way of meeting the needs of the various task intervals is to allocate a scheduler tick interval of 1 ms. This is easily done: see HARDWARE DELAY [Page 181] and Chapter 13 (Pont, 2001) for details.
Remember, however, that the scheduler itself will impose a CPU load on the microcontroller, and that this load will increase dramatically at low tick intervals (see 'Hardware resource implications'). To keep the scheduler load as low as possible (and to reduce the power consumption: see below), it can help to use a long tick interval.
If you want to reduce overheads and power consumption to a minimum, the scheduler tick interval should be set to match the 'greatest common factor' of all the task (and offset intervals). This is easily calculated, if you remember some simple high-school mathematics.
Suppose we have three tasks (X,Y,Z), and Task X is to be run every 10 ms, Task Y every 30 ms and Task Z every 25 ms. The scheduler tick interval needs to be set by determining the relevant factors, as follows:

- The factors[16] of the Task X interval (10 ms) are: 1 ms, 2ms, 5 ms, 10 ms.
- Similarly, the factors of the Task Y interval (30 ms) are as follows: 1 ms, 2 ms, 3 ms, 5 ms, 6 ms, 10 ms, 15 ms and 30 ms.
- Finally, the factors of the Task Z interval (25 ms) are as follows: 1 ms, 5 ms and 25 ms.

In this case, therefore, the greatest common factor is 5 ms: this is the required tick interval.

---

[16] Remember: the factors are integers (between 1 and X) by which we can divide X and obtain a remainder of 0.

Note that it may seem that if you have task intervals of (say) 5 ms, 25 ms and 1000 ms, then this process will be extremely tedious, because 1000 will have many factors. However, in practice, we are only concerned with the factors up to and including the smallest of the task intervals. In this case, therefore, we would be only interested in the factors of 5, 25 and 1000 between 1 and 5. The largest common factor being, in this case, 5 ms.

The situation becomes slightly more complicated if we consider the initial task delays.

If we go back to the example above, suppose we have decided to use a 5ms scheduler. We are adding three tasks to the scheduler as follows:

```
SCH_Add_Task(X, 0, 2);
SCH_Add_Task(Y, 0, 6);
SCH_Add_Task(Z, 0, 5);
```

Clearly, these tasks are going to frequently overlap. For example, every time Task Y is scheduled to run, so is Task X; on some occasions, all three tasks are due to run simultaneously. To avoid this, we can add some initial task delays, as follows:

- Task X is to be run every 10 ms: we start this task immediately.
- Task Z is to be run every 25 ms: we start this task after 2 ms.
- Task Y is to be run every 30 ms; we start this task after 1 ms.

When determining the required scheduler interval, we must now take into account both the task intervals and the initial delays. This, in this case, we now need to find the greatest common factor of 10, 25, 30, 1 and 2: this suggest a scheduler interval of 1 ms is now required.

Guidelines for predictable and reliable scheduling

1. For precise scheduling, the scheduler tick interval should be set to match the 'greatest common factor' of all the task intervals (see above).

2. All tasks should have a duration less than the schedule tick interval, to ensure that the dispatcher is always free to call any task that is due to execute. Software simulation can often be used to measure the task duration.

3. In order to meet Condition 2, all tasks **must** 'timeout' so that they cannot block the scheduler under any circumstances. Note that this condition can often be met by incorporating, where necessary, a LOOP TIMEOUT [Page 262] or a HARDWARE TIMEOUT [Page 268] in scheduled tasks.

   Please remember that this condition also applies to any functions called from within a scheduled task, including any library code provided by your compiler manufacturer. In many cases, standard functions (like `printf()`) do not include timeout features. They must not be used in situation where predictability is required.

4. The total time required to execute all of the scheduled tasks must be less than the available processor time. Of course, the total processor time must include both this 'task time' and the 'scheduler time' required to execute the scheduler update and dispatcher operations.

5. Tasks should be scheduled so that they are never required to execute simultaneously: that is, task overlaps should be minimised. Note that where **all** tasks are of a duration much less than the scheduler tick interval, and that some task jitter can be tolerated, this problem may not be significant.

Portability

A co-operative scheduler, like that described in this pattern, can be written entirely in 'C', for many different platforms. The TTC SCHEDULER[C, 8051], describes a pattern implementation example written in C-language for the 8051 family of microcontrollers.

Overall strengths and weaknesses

The overall strengths and weaknesses of a co-operative scheduler may be summarised as follows:

☺ The scheduler is simple, and can be implemented in a small amount of code.

☺ The applications based on the scheduler are inherently predictable, safe and reliable.

☺ The scheduler is written entirely in 'C': it is not a separate application, but becomes part of the developer's code

☺ The scheduler supports team working, since individual tasks can often be developed largely independently and then assembled into the final system.

☹ Obtain rapid responses to external events requires care at the design stage.

☹ The tasks cannot safely use interrupts: the only interrupt that should be used in the application is the timer-related interrupt that drives the scheduler itself.

Related patterns and alternative solutions

For alternative solutions see:

- HYBRID SCHEDULER
- ONE-TASK SCHEDULER
- ONE-YEAR SCHEDULER
- STABLE SCHEDULER
- TTC-SL SCHEDULER
- TTC-ISR SCHEDULER

Further reading

-

TTC SCHEDULER [C, 8051]

{pattern implementation example}

Context

- You wish to implement a TTC SCHEDULER [this paper]
- Your chosen implementation language is C[17].
- Your chosen implementation platform is the 8051 family of microcontrollers.

Problem

How can you implement a TTC SCHEDULER for the Atmel 8051 family of microcontrollers?

Background

Function pointers and Keil linker options

When we write:

```
SCH_Add_Task(Do_X,1000,0);
```

…the first parameter of the 'Add Task' function is a *pointer* to the function `Do_X()`. This function pointer is then passed to the Dispatch function and it is through this function that the task is executed:

```
if (SCH_tasks_G[Index].RunMe > 0)
   {
   (*SCH_tasks_G[Index].pTask)();  // Run the task
```

The use of the 'C' function pointers on small microcontrollers presents a particular challenge. This is particularly true when function pointers are used as function arguments.

On desktop systems, function arguments are generally passed on the stack using the push and pop assembly instructions. Since the 8051 has a size limited stack (only 128 bytes at best and as low as 64 bytes on some devices), function arguments must be passed using a different technique: in the case of Keil C51, these arguments are stored in fixed memory locations. When the linker is invoked, it builds a call tree of the program, decides which function arguments are mutually exclusive (that is, which functions cannot be called at the same time), and overlays these arguments.

The linker has difficulty determining the correct call tree when function pointers are used as function arguments, as is the case with the 'Add Task' function. To deal with this situation, you have two realistic options:

[17] The examples in the pattern were created using the Keil C compiler, hosted in a Keil uVision 3 IDE.

1. You can prevent the compiler from using the OVERLAY directive by disabling overlays as part of the linker options for your project.

   Note that, compared to applications using overlays, you will generally require more RAM to run your program.

2. You can tell the linker how to create the correct call tree for your application by explicitly providing this information in the linker 'Additional Options' dialogue box.

   This solution generally uses less memory, but the compiler often cannot tell if you provide incorrect information: if you get this option wrong, your program can generate unpredictable results.

The linker options required are not difficult to understand. Suppose we have run our simple flashing LED example presented earlier, and we are scheduling a single task, as follows:

```
void main(void)
   {

   //...

   // Add the 'Flash LED' task (on for ~1000 ms, off for ~1000 ms)
   // - timings are in ticks (1 ms tick interval)
   // (Max interval / delay is 65535 ticks)
   SCH_Add_Task(LED_Flash_Update, 0, 1000);

   //...
```

The linker assumes - because the pointer `LED_Flash_Update` appears in `main()` - that the function is called from `main()`. Instead, the function is called from `SCH_Dispatch_Tasks`. We make this change explicit using the linker options below:

```
   OVERLAY
   (main ~ (LED_Flash_Update),
   SCH_Dispatch_Tasks ! (LED_Flash_Update))
```

Reporting errors

To report errors at any part of the scheduled application, we use an (8-bit) error code variable Error_code_G, which is defined in Sch51.C as follows:

```
// Used to display the error code
tByte Error_code_G = 0;
```

To record an error we include lines such as:

```
Error_code_G = ERROR_SCH_TOO_MANY_TASKS;
Error_code_G = ERROR_SCH_WAITING_FOR_SLAVE_TO_ACK;
Error_code_G = ERROR_SCH_WAITING_FOR_START_COMMAND_FROM_MASTER;
Error_code_G = ERROR_SCH_ONE_OR_MORE_SLAVES_DID_NOT_START;
Error_code_G = ERROR_SCH_LOST_SLAVE;
Error_code_G = ERROR_SCH_CAN_BUS_ERROR;
Error_code_G = ERROR_I2C_WRITE_BYTE_AT24C64;
```

Listing 19: Error Codes in the project header file.

These error codes are given in the file Main.H which is an example of the pattern PROJECT HEADER. (PONT 2001)

To report these error code, the scheduler has a function `SCH_Report_Status()`, which is called from the Update function. Note that error reporting may be disabled via the `Port.H` header file:

```
// Comment this line out if error reporting is NOT required
//#define SCH_REPORT_ERRORS
```

Where error reporting is required, the port on which error codes will be displayed is also determined via `Port.H`:

```
#ifdef SCH_REPORT_ERRORS
// The port on which error codes will be displayed
// ONLY USED IF ERRORS ARE REPORTED
#define Error_port P1

#endif
```

The simplest way of displaying these codes is to attach eight LEDs (with suitable buffers) to the error port, as discussed in IC DRIVER. (PONT, 2001)

Solution

A complete code example illustrating the implementation of a TTC SCHEDULER is given in Listing 20 and Listing 21.

```
/*---------------------------------------------------------------------*-

   2_01_10i.c (v1.00)

  --------------------------------------------------------------------

   *** THIS IS A SCHEDULER FOR 80C515C (etc.) ***
   *** For use in single-processor applications  ***

   *** Uses T2 for timing, 16-bit auto reload ***

   *** This version assumes 10 MHz crystal on 515c ***
   *** 1 ms (approx) tick interval ***

   *** Includes display of error codes ***


   COPYRIGHT
   ---------

   This code is adapted from the book:

   PATTERNS FOR TIME-TRIGGERED EMBEDDED SYSTEMS by Michael J. Pont
   [Pearson Education, 2001; ISBN: 0-201-33138-1].

   This code is copyright (c) 2001 by Michael J. Pont.

   See book for copyright details and other information.

-*---------------------------------------------------------------------*/

#include “Main.h”
#include “2_01_10i.H”

// ------ Public variable declarations ----------------------------

// The array of tasks (see Sch51.C)
extern sTask SCH_tasks_G[SCH_MAX_TASKS];

// Used to display the error code
// See Main.H for details of error codes
// See Port.H for details of the error port
extern tByte Error_code_G;

//  Used to indicate the number of times that the timer
//  has overflowed
long int Tick_count_G;

/*---------------------------------------------------------------------*-

  SCH_Init_T2()

  Scheduler initialisation function.  Prepares scheduler data
  structures and sets up timer interrupts at required rate.
  Must call this function before using the scheduler.

-*---------------------------------------------------------------------*/
void SCH_Init_T2(void)
   {
   tByte i;

   Tick_count_G = 0;

   // Sort out the tasks
   for (i = 0; i < SCH_MAX_TASKS; i++)
      {
      SCH_Delete_Task(i);
      }

   // Reset the global error variable
   // - SCH_Delete_Task() will generate an error code,
   //   (because the task array is empty)
```

```
   Error_code_G = 0;

   // Now set up Timer 2
   // 16-bit timer function with automatic reload
   // Crystal is assumed to be 10 MHz
   // Using c515c, so timer can be incremented at 1/6 crystal frequency
   // if prescalar is not used

   // Prescaler not used -> Crystal/6
   //T2PS = 0; // No prescaler in AT89C55? -- Pete

   // The Timer 2 resolution is  0.0000006 seconds (0.6 µs)
   // The required Timer 2 overflow is 0.001 seconds (1 ms)
   // - this takes  1666.666666667 timer ticks (can't get precise timing)
   // Reload value is 65536 - 1667 = 63869 (dec) = 0xF97D

  TH2 = 0xF9;
  TL2 = 0x7D;

   RCAP2H = 0xF9;
   RCAP2L = 0x7D;


   // Compare/capture Channel 0
   // Disabled
   // Compare Register CRC on: 0x0000;
   //CRCH = 0xF9;
   //CRCL = 0x7D; // Not available on AT89C55? -- Pete

   //  Mode 0 for all channels
   T2CON = 4; // 0x11; // Needs to be 00000100b -- Pete

   //  timer 2 overflow interrupt is enabled
   ET2 = 1;
   //  timer 2 external reload interrupt is disabled
   EXEN2 = 0;

   //  CC0/ext3 interrupt is disabled
   //EX3 = 0; // Not available on AT89C55? -- Pete

   // Compare/capture Channel 1-3
   // Disabled
   //CCL1 = 0x00;
   //CCH1 = 0x00;
   //CCL2 = 0x00;
   //CCH2 = 0x00;
   //CCL3 = 0x00;
   //CCH3 = 0x00; // Not available on AT89C55? -- Pete

   // Interrupts Channel 1-3
   // Disabled
   //EX4 = 0;
   //EX5 = 0;
   //EX6 = 0; // Not available on AT89C55? -- Pete

   // all above mentioned modes for Channel 0 to Channel 3
   //CCEN = 0x00; // Not available on AT89C55? -- Pete
   // ------ Set up Timer 2 (end) ---------------------------------
   }

/*---------------------------------------------------------------------*-

  SCH_Start()

  Starts the scheduler, by enabling interrupts.

  NOTE: Usually called after all regular tasks are added,
  to keep the tasks synchronised.

  NOTE: ONLY THE SCHEDULER INTERRUPT SHOULD BE ENABLED!!!

-*---------------------------------------------------------------------*/
```

```
void SCH_Start(void)
   {
   // Comment out as required, depending on compiler used
   EA = 1;   // Use with C51 v5.X
   //EAL = 1;  // Use with C51 v6.X
   }

/*-------------------------------------------------------------------*-

  SCH_Update()

  This is the scheduler ISR.  It is called at a rate determined by
  the timer settings in SCH_Init_T2().  This version is
  triggered by Timer 2 interrupts: timer is automatically reloaded.

-*-------------------------------------------------------------------*/
void SCH_Update(void) interrupt INTERRUPT_Timer_2_Overflow
   {
   TF2 = 0; // Have to manually clear this.

   Tick_count_G++;
   }

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

Listing 20: Scheduler functions as defined in 2_01_10i.C file

```
/*-------------------------------------------------------------------*-

   SCH51.C (v1.00)

  ------------------------------------------------------------------

   *** THESE ARE THE CORE SCHEDULER FUNCTIONS ***
   --- These functions may be used with all 8051 devices ---

   *** SCH_MAX_TASKS *must* be set by the user ***
   --- see “Sch51.H” ---

   *** Includes (optional) power-saving mode ***
   --- You must ensure that the power-down mode is adapted ---
   --- to match your chosen device (or disabled altogether) ---


   COPYRIGHT
   ---------

   This code is from the book:

   PATTERNS FOR TIME-TRIGGERED EMBEDDED SYSTEMS by Michael J. Pont
   [Pearson Education, 2001; ISBN: 0-201-33138-1].

   This code is copyright (c) 2001 by Michael J. Pont.

   See book for copyright details and other information.

-*-------------------------------------------------------------------*/

#include “Main.h”
#include “Port.h”

#include “Sch51.h”
// Gives access to Keil _idle_() function
#include “intrins.h”
// ------ Public variable definitions -----------------------------

// The array of tasks
sTask SCH_tasks_G[SCH_MAX_TASKS];

// Used to display the error code
// See Main.H for details of error codes
// See Port.H for details of the error port
tByte Error_code_G = 0;

// ------ Private function prototypes -----------------------------

static void SCH_Go_To_Sleep(void);

// ------ Private variables ---------------------------------------

// Keeps track of time since last error was recorded (see below)
static tWord Error_tick_count_G;

// The code of the last error (reset after ~1 minute)
static tByte Last_error_code_G;

extern long int Tick_count_G;


/*-------------------------------------------------------------------*-

  SCH_Dispatch_Tasks()

  This is the 'dispatcher' function.  When a task (function)
  is due to run, SCH_Dispatch_Tasks() will run it.
  This function must be called (repeatedly) from the main loop.

-*-------------------------------------------------------------------*/
```

```
void SCH_Dispatch_Tasks(void)
   {
   tByte Index;
   bit Update_again = 0;


   do {
      // NOTE: calculations are in *TICKS* (not milliseconds)
      for (Index = 0; Index < SCH_MAX_TASKS; Index++)
         {
         // Check if there is a task at this location
         if (SCH_tasks_G[Index].pTask)
            {
            if (--SCH_tasks_G[Index].Delay == 0)
               {
               // The task is due to run
               (*SCH_tasks_G[Index].pTask)();  // Run the task

               if (SCH_tasks_G[Index].Period)
                  {
                  // Schedule period tasks to run again
                  SCH_tasks_G[Index].Delay = SCH_tasks_G[Index].Period;
                  }
               else
                  {
                  // Delete one-shot tasks
                  SCH_tasks_G[Index].pTask  = 0;
                  }
               }
            }
         }

      // Disable Timer 2 interrupt
      ET2 = 0;

      if (--Tick_count_G > 0)
         {
         Update_again = 1;
         }
      else
         {
         Update_again = 0;
         }

      // Re-enable Timer 2 interrupt
      ET2 = 1;

      } while (Update_again);

   // Report system status
   SCH_Report_Status();

   // The scheduler enters idle mode at this point
   SCH_Go_To_Sleep();
   }

/*------------------------------------------------------------------*-

  SCH_Add_Task()

  Causes a task (function) to be executed at regular intervals
  or after a user-defined delay

  Fn_P   - The name of the function which is to be scheduled.
           NOTE: All scheduled functions must be 'void, void' -
           that is, they must take no parameters, and have
           a void return type.

  DELAY  - The interval (TICKS) before the task is first executed

  PERIOD - If 'PERIOD' is 0, the function is only called once,
           at the time determined by 'DELAY'.  If PERIOD is non-zero,
```

```
           then the function is called repeatedly at an interval
           determined by the value of PERIOD (see below for examples
           which should help clarify this).


  RETURN VALUE:

  Returns the position in the task array at which the task has been
  added.  If the return value is SCH_MAX_TASKS then the task could
  not be added to the array (there was insufficient space).  If the
  return value is < SCH_MAX_TASKS, then the task was added
  successfully.

  Note: this return value may be required, if a task is
  to be subsequently deleted - see SCH_Delete_Task().

  EXAMPLES:

  Task_ID = SCH_Add_Task(Do_X,1000,0);
  Causes the function Do_X() to be executed once after 1000 sch ticks.

  Task_ID = SCH_Add_Task(Do_X,0,1000);
  Causes the function Do_X() to be executed regularly, every 1000 sch ticks.

  Task_ID = SCH_Add_Task(Do_X,300,1000);
  Causes the function Do_X() to be executed regularly, every 1000 ticks.
  Task will be first executed at T = 300 ticks, then 1300, 2300, etc.

-*------------------------------------------------------------------*/
tByte SCH_Add_Task(void (code * pFunction)(),
                   const tWord DELAY,
                   const tWord PERIOD)
   {
   tByte Index = 0;

   // First find a gap in the array (if there is one)
   while ((SCH_tasks_G[Index].pTask != 0) && (Index < SCH_MAX_TASKS))
      {
      Index++;
      }

   // Have we reached the end of the list?
   if (Index == SCH_MAX_TASKS)
      {
      // Task list is full
      //
      // Set the global error variable
      Error_code_G = ERROR_SCH_TOO_MANY_TASKS;

      // Also return an error code
      return SCH_MAX_TASKS;
      }

   // If we're here, there is a space in the task array
   SCH_tasks_G[Index].pTask  = pFunction;

   SCH_tasks_G[Index].Delay  = DELAY + 1;
   SCH_tasks_G[Index].Period = PERIOD;

   SCH_tasks_G[Index].RunMe  = 0;

   return Index; // return position of task (to allow later deletion)
   }

/*------------------------------------------------------------------*-

  SCH_Delete_Task()

  Removes a task from the scheduler.  Note that this does
  *not* delete the associated function from memory:
  it simply means that it is no longer called by the scheduler.
```

```
  TASK_INDEX - The task index.  Provided by SCH_Add_Task().

  RETURN VALUE:  RETURN_ERROR or RETURN_NORMAL

-*---------------------------------------------------------------------*/
bit SCH_Delete_Task(const tByte TASK_INDEX)
   {
   bit Return_code;

   if (SCH_tasks_G[TASK_INDEX].pTask == 0)
      {
      // No task at this location...
      //
      // Set the global error variable
      Error_code_G = ERROR_SCH_CANNOT_DELETE_TASK;

      // ...also return an error code
      Return_code = RETURN_ERROR;
      }
   else
      {
      Return_code = RETURN_NORMAL;
      }

   SCH_tasks_G[TASK_INDEX].pTask   = 0x0000;
   SCH_tasks_G[TASK_INDEX].Delay   = 0;
   SCH_tasks_G[TASK_INDEX].Period  = 0;

   SCH_tasks_G[TASK_INDEX].RunMe   = 0;

   return Return_code;       // return status
   }


/*---------------------------------------------------------------------*-

  SCH_Report_Status()

  Simple function to display error codes.

  This version displays code on a port with attached LEDs:
  adapt, if required, to report errors over serial link, etc.

  Errors are only displayed for a limited period
  (60000 ticks = 1 minute at 1ms tick interval).
  After this the the error code is reset to 0.

  This code may be easily adapted to display the last
  error 'for ever': this may be appropriate in your
  application.

  See Chapter 10 for further information.

-*---------------------------------------------------------------------*/
void SCH_Report_Status(void)
   {
#ifdef SCH_REPORT_ERRORS
   // ONLY APPLIES IF WE ARE REPORTING ERRORS
   // Check for a new error code
   if (Error_code_G != Last_error_code_G)
      {
      // Negative logic on LEDs assumed
      Error_port = 255 - Error_code_G;

      Last_error_code_G = Error_code_G;

      if (Error_code_G != 0)
         {
         Error_tick_count_G = 60000;
         }
      else
         {
```

```
         Error_tick_count_G = 0;
         }
      }
   else
      {
      if (Error_tick_count_G != 0)
         {
         if (--Error_tick_count_G == 0)
            {
            Error_code_G = 0; // Reset error code
            }
         }
      }
#endif
   }


/*------------------------------------------------------------------*-

  SCH_Go_To_Sleep()

  This scheduler enters 'idle mode' between clock ticks
  to save power.  The next clock tick will return the processor
  to the normal operating state.

  Note: a slight performance improvement is possible if this
  function is implemented as a macro, or if the code here is simply
  pasted into the 'dispatch' function.

  However, by making this a function call, it becomes easier
  - during development - to assess the performance of the
  scheduler, using the 'performance analyser' in the Keil
  hardware simulator. See Chapter 14 for examples for this.

  *** May wish to disable this if using a watchdog ***

  *** ADAPT AS REQUIRED FOR YOUR HARDWARE ***

-*------------------------------------------------------------------*/
void SCH_Go_To_Sleep()
   {
   // Entering idle mode requires TWO consecutive instructions
   // on 80c515 / 80c505 - to avoid accidental triggering
   PCON |= 0x01;    // Enter idle mode (#1)
   //PCON |= 0x20;    // Enter idle mode (#2) not required on AT89C55 -- Pete
   }

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

Listing 21: Core scheduler functions defined in Sch51.C

Further Reading

-

Reference

Balanyi, Z.; Ferenc, R., 2003. Mining design patterns from C++ source code, Proceedings of International Conference on Software Maintenance, ICSM 2003, Publisher:IEEE Comput. Soc.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J., 1995. Design patterns: Elements of reusable object-oriented software, Addison-Wesley, Reading, MA.

Keller, Rudolf K.; Schauer, Reinhard; Robitaille, Sebastien; Page, Patrick, (1999). Pattern-based reverse-engineering of design components, Proceedings - International Conference on Software Engineering, 226-235, IEEE, Los Angeles, CA, USA.

Kurian, S. and Pont, M.J. (2005). Building reliable embedded systems using Abstract Patterns, Patterns, and Pattern Implementation Examples In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (2005) (Eds.) "Proceedings of the Second UK Embedded Forum" (Birmingham, UK, October 2005). Published by University of Newcastle upon Tyne

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

Pont, M.J. and Ong, H.L.R., 2003. Using watchdog timers to improve the reliability of TTCS embedded systems, In Hruby, P. and Soressen, K. E. (Eds.) Proceedings of the First Nordic Conference on Pattern Languages of Programs, ("VikingPloP 2002"), pp.159-200. Published by Micrsoft Business Solutions.

Pont, M.J., 2001. Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers, ACM Press / Addison-Wesley, UK

Pont, M.J., 2003. An object-oriented approach to software development for embedded systems implemented using C, Transactions of the Institute of Measurement and Control 25(3): 217-238.

Pont, M.J., Kurian, S., Maaita, A. and Ong, R. (2005) "Restructuring a pattern language for reliable embedded systems" ESL, Technical Report 2005-01. Available for download at http://www.le.ac.uk/eg/embedded/tech_reports.htm

Yoder, J. W.; Foote, B.; Riehle, D.; Tilman, M.(1998). Metadata and active objectmodels. In: Workshop Results Submission OOPSLA'98 Addendum, 1998.

A2.Other publications

This appendix documents papers published during the course of this research. The published material is not directly related to the topic being discussed in this report.

Meeting real-time constraints using “Sandwich Delays”

Michael J. Pont, Susan Kurian and Ricardo Bautista-Quintero
Embedded Systems Laboratory, University of Leicester,
University Road, LEICESTER LE1 7RH, UK.
M.Pont@le.ac.uk; sk183@le.ac.uk; rb169@le.ac.uk
http://www.le.ac.uk/eg/embedded/

Abstract

This short paper is concerned with the use of patterns to support the development of software for reliable, resource-constrained, embedded systems. The paper introduces one new pattern (SANDWICH DELAY) and describes one possible implementation of this pattern for use with a popular family of ARM-based microcontrollers.

Introduction

In this paper, we are concerned with the development of embedded systems for which there are two (sometimes conflicting) constraints. First, we wish to implement the design using a low-cost microcontroller, which has – compared to a desktop computer – very limited memory and CPU performance. Second, we wish to produce a system with extremely predictable timing behaviour.

To support the development of this type of software, we have previously described a “language” consisting of more than seventy patterns (e.g. see Pont, 2001). Work began on these patterns in 1996, and they have since been used it in a range of industrial systems and numerous university research projects (e.g. see Pont, 2003; Pont and Banner, 2004; Mwelwa et al., 2006; Kurian and Pont, in press a; Kurian and Pont, in press b).

This brief paper describes one new pattern (SANDWICH DELAY) and illustrates – using what we call a “pattern implementation example” (e.g. see Kurian and Pont, in press b) - one possible implementation of this pattern for use with a popular family of ARM-based microcontrollers.

Acknowledgements

Many thanks to Bob Hanmer, who provided numerous useful suggestions during the shepherding process. We also thank the contributors to our workshop session (Joe Bergen, Frank Buschmann, Neil Harrison, Kevlin Henney, Andy Longshaw, Klaus Marquardt, Didi Schütz and Markus Völter) for further comments on this paper at the conference itself.

Copyright © 2006 by Michael J. Pont, Susan Kurian and Ricardo Bautista. Permission is granted to copy this paper for use in association with the EuroPLoP 2006 conference.

SANDWICH DELAY

{design pattern}

Context

- You are developing an embedded system.
- Available CPU and / or memory resources are – compared with typical desktop designs – rather limited.
- Your system is based on a time-triggered scheduler rather than a "real-time operating system".
- Your system involves running two or more periodic tasks.
- Predictable timing behaviour is a key design requirement.

Problem

You are running two activities, one after the other. How can we ensure that the interval between the release times of the two activities is known and fixed?

Background

In many embedded applications (such as those involving control or data acquisition) variations in the start times of tasks or functions can have serious implications. Such timing variations are known as "release jitter" (or simply "jitter").

For example, Cottet and David (1999) show that – during data acquisition tasks – jitter rates of 10% or more can introduce errors which are so significant that any subsequent interpretation of the sampled signal may be rendered meaningless. Similarly Jerri discusses the serious impact of jitter on applications such as spectrum analysis and filtering (Jerri, 1997). Also, in control systems, jitter can greatly degrade the performance by varying the sampling period (Torngren, 1998; Mart et al., 2001).

In many embedded systems, we wish to keep the levels of jitter to a minimum.

Solution

A SANDWICH DELAY can be used to solve this type of problem. More specifically, a SANDWICH DELAY provides a simple but highly effective means of ensuring that a particular piece of code *always takes the same period of time to execute*: this is done using two timer operations to "sandwich" the activity you need to perform.

To illustrate one possible application of a SANDWICH DELAY, suppose that we have a system

executing two functions periodically, as outlined in Listing 22.

```
// Interrupt Service Routine (ISR) invoked by timer overflow every 10 ms
void Timer_ISR(void)
   {
   Do_X();  // WCET[18] approx. 4.0 ms
   Do_Y();  // WCET approx. 4.0 ms
   }
```

Listing 22: Using a timer ISR to execute two periodic functions.

According to the code in Listing 22, function `Do_X()` will be executed every 10 ms. Similarly, function `Do_Y()` will be executed every 10 ms, after `Do_X()` completes. For many resource-constrained applications (for example, control systems) this architecture may be appropriate. However, in some cases, the risk of jitter in the start times of function `Do_Y()` may cause problems. Such jitter will arise if there is any variation in the duration of function `Do_X()`. In Figure 21, the jitter will be reflected in differences between the values of *ty1* and *ty2* (for example).

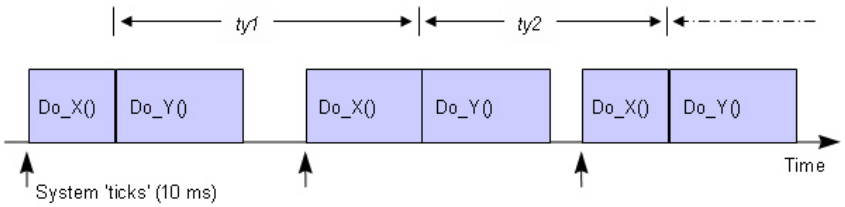


*Figure 21: The impact of variations in the duration of **`Do_X()`** on the release jitter of **`Do_Y()`**.*

We can use a SANDWICH DELAY to solve this problem: please refer to Listing 23.

```
// ISR invoked by timer overflow every 10 ms
void Timer_ISR(void)
   {
   // Execute Do_X() in a 'Sandwich Delay' - BEGIN
   Set_Sandwich_Timer_Overflow(5);       // Set timer to overflow after 5 ms
   Do_X();                               // Execute Do_X - WCET approx. 4 ms
   Wait_For_Sandwich_Timer_Overflow();   // Wait for timer to overflow
   // Execute Do_X() in a 'Sandwich Delay' - END

   Do_Y();  // WCET approx. 4.0 ms
   }
```

Listing 23: Employing a SANDWICH DELAY to reduce release in function `Do_Y()`.

In Listing 23, we set a timer to overflow after 5 ms (a period slightly longer than the worst-case execution time of `Do_X()`). We then start this timer before we run the function and – after the function is complete – we wait for the timer to reach the 5 ms value. In this way, we ensure that

[18] WCET = Worst-Case Execution Time. If we run the task an infinite number of times and measure how long it takes to complete, the WCET will be the longest execution time which we measure.

– as long as `Do_X()` does not exceed a duration of 5 ms - `Do_Y()` runs with very little jitter[19].

Figure 22 shows the tick graph from this example, with the SANDWICH DELAY included.

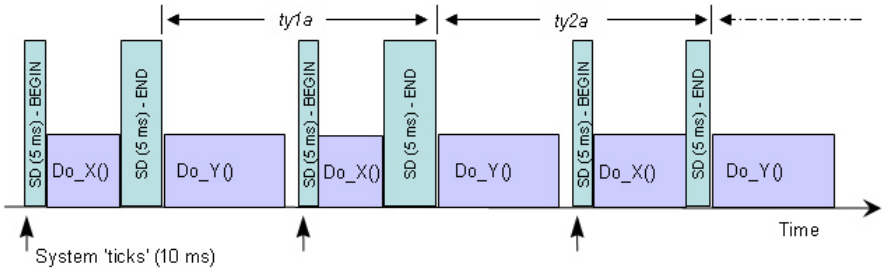


Figure 22: Reducing the impact of variations in the duration of ***`Do_X()`*** *on the release jitter of* ***`Do_Y()`*** *through use of a* ***SANDWICH DELAY****.*

Related patterns and alternative solutions

In some cases, you can avoid the use of a SANDWICH DELAY altogether, by altering the system tick interval. For example, if we look again at our `Do_X()` / `Do_Y()` system, the two tasks have the same duration. In this case, we would be better to reduce the tick interval to 5 ms and run the tasks in alternating time slots (Figure 23).

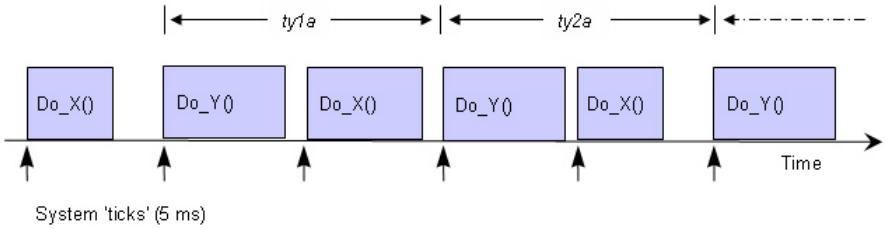


Figure 23: Avoiding the use of SANDWICH DELAYS through changes in the scheduler tick interval.

Please note that this solution will only work (in general) if the tasks in your system have similar durations. Where the tasks do not have the same duration, a scheduler involving multiple timer interrupts may be more appropriate: such a solution is beyond the scope of this paper but is described in detail elsewhere (Nahas and Pont, submitted).

Reliability and safety implications

Use of a SANDWICH DELAY is generally straightforward, but there are three potential issues of which you should be aware.

---

[19] In general, it is not possible to remove all jitter using this approach: we explain why under the heading "Reliability and safety implications".

First, you need to know the duration (WCET) of the function(s) to be sandwiched. If you underestimate this value, the timer will already have reached its overflow value when your function(s) complete, and the level of jitter will not be reduced (indeed, the SANDWICH DELAY is likely to slightly increase the jitter in this case).

Second, you must check the code carefully, because the "wait" function may never terminate if the timer is incorrectly set up. In these circumstances a watchdog timer (e.g. see Pont, 2001; Pont and Ong, 2003) or a "task guardian" (see Hughes and Pont, 2004) may help to rescue your system, but relying on such mechanisms to deal with poor design or inadequate testing is – of course - never a good idea.

Third, you will rarely manage to remove all jitter using such an approach, because the system cannot react instantly when the timer reaches its maximum value (at the machine-code level, the code used to poll the timer flag is more complex than it may appear, and the time taken to react to the flag change will vary slightly). A useful rule of thumb is that jitter levels of around 1 μs will still be seen using a SANDWICH DELAY.

Overall strengths and weaknesses

☺ A simple way of ensuring that the WCET of a block of code is highly predictable.

☹ Requires (non-exclusive) access to a timer.

☹ Will only rarely provide a "jitter free" solution: variations in code duration of around 1 μs are representative.

Example: Application of Dynamic Voltage Scaling

As we note in "Context", we are concerned in this pattern with the development of software for embedded systems in which (i) the developer must adhere to severe resource constraints, and (ii) there is a need for highly predictable system behaviour. With many mobile designs (for example, mobile medical equipment) we also need to minimise power consumption in order to maximise battery life. To meet all three constraints, it is sometimes possible to use a system architecture which combines time-triggered co-operative (TTC) task scheduling with a power-reduction technique know as "dynamic voltage scaling" (DVS). To achieve this, use of a SANDWICH DELAY is a crucial part of the implementation (and is used to ensure that the complex DVS operations do not introduce task jitter).

The use of SANDWICH DELAYs in this context is described in detail by Phatrapornnant and Pont (2006).

SANDWICH DELAY (C, LPC2000)

{pattern implementation example}[20]

Context

- You wish to implement a SANDWICH DELAY [this paper]
- Your chosen implementation language is C[21].
- Your chosen implementation platform is the NXP[22] LPC2000 family of (ARM7-based) microcontrollers.

Problem

How can you implement a SANDWICH DELAY for the NXP LPC2000 family of microcontrollers?

Background

As with all widely-used microcontrollers, the LPC2000 devices have on-chip timers which are directly accessible by the programmer. More specifically, all members of this family have two 32-bit timers, known as Timer 0 and Timer 1. These can each be set to take actions (such as setting a flag) when a particular time period has elapsed.

In the simplest case, these timers (and other peripheral devices) will be driven by the "peripheral clock" (plck) which - by default - runs at 25% of the rate of the system oscillator (Figure 24).

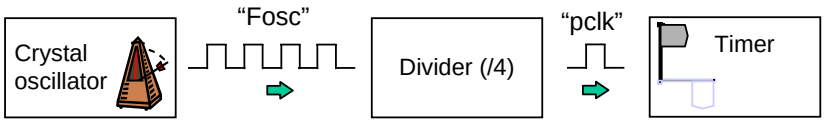


Figure 24: The link between oscillator frequency and timer updates in the LPC2000 devices (default situation).

By taking into account the link between the oscillator frequency and the timer hardware, the timers can be configured so that (for example) a flag is set after a period of 10ms has elapsed. The resulting delay code can be made highly portable.

Both Timer 0 and Timer 1 are 32-bit timers, which are preceded by a 32-bit pre-scalar. The pre-

---

[20] As the name might suggest, PIEs are intended to illustrate how a particular pattern can be implemented. This is important (in the embedded systems field) because there are great differences in system environments, caused by variations in the hardware platform (e.g. 8-bit, 16-bit, 32-bit, 64-bit), and programming language (e.g. assembly language, C, C++). The possible implementations are not sufficiently different to be classified as distinct patterns: however, they do contain useful information. We say more about PIEs in another paper at EuroPLoP 2006 (Kurian and Pont, in press b).

[21] The examples in the pattern were created using the GNU C compiler, hosted in a Keil uVision 3 IDE.

[22] Formerly Philips Semiconductors.

scalar is in turn driven by the peripheral clock. This is an extremely flexible combination. As an example, suppose that we wished to generate the longest possible delay using Timer 0 on an LPC2100 device with a 12 MHz oscillator. The delay would be generated as follows:

- Both the pre-scalar and the timer itself begin with a count of 0.
- The pre-scalar would be set to trigger at its maximum value: this is 2^{32}-1 (=4294967295). With a 12 MHz oscillator (and the default divider of 4), the pre-scalar would take approximately 1432 seconds to reach this value. It would then be reset, and begin counting again.
- When the pre-scalar reached 1432 seconds, Timer 0 would be incremented by 1. To reach its full count (4294967295) would take approximately 200,000 years.

Clearly, this length of delay will not be required in most applications! However, very precise delays (for example, an hour, a day – even a week) can be created using this flexible hardware.

As a more detailed example, suppose that we have a 12 MHz oscillator (again with default divider of 4) and we wish to generate a delay of 1 second. We can omit the prescalar, and simply set the match register on Timer 1 to count to to the required value (3,000,000 – 1).

We can achieve this using the code shown in Listing 24.

Solution

A code example illustrating the implementation of a SANDWICH DELAY for an LPC2000 device is given in Listing 26.

```
// Prescale is 0 (in effect, prescalar not used)
T1PC = 0;

// Set the “Timer Counter Register” for this timer.
// In this register, Bit 0 is the “Counter Enable” bit.
// When 1, the Timer Counter and Prescale Counter are enabled for counting.
// When 0, the counters are disabled.
T1TCR &= ~0x01;   // Stop the timer by clearing Bit 0

// There are three match registetrs (MR0, MR1, MR2) for each timer.
// The match register values are continuously compared to the Timer Counter value.
// When the two values are equal, actions can be triggered automatically
T1MR0 = 2999999;  // Set the match register (MR0) to required value

// When the match register detects a match, we can choose to:
// Generate an interrupt (not used here),
// Reset the Timer Counter and / or
// Stop the timer.
// These actions are controlled by the settings in the MCR register.
// Here we set a flag on match (no interrupt), reset the count and stop the timer.
T1MCR = 0x07;     // Set flag on match, reset count and stop timer

T1TCR |= 0x01;    // Start the timer

// Wait for timer to reach count (at which point the IR flag will be set)
while ((T1IR & 0x0001) == 0)
   {
   ;
   }

// Reset the timer flag (by writing “1”)
T1IR |= 0x01;
```

Listing 24: Configuring Timer1 in the LPC2000 family. See text for details.

```
/*-------------------------------------------------------------------*-

   Main.C (v1.00)

  ------------------------------------------------------------------

   Simple “Sandwich Delay” demo for NXP LPC2000 devices.

-*-------------------------------------------------------------------*/

#include “main.h”

#include “system_init.h”

#include “led_flash.h”
#include “random_loop_delay.h”

#include “sandwich_delay_t1.h”

/*-------------------------------------------------------------------*-

  int main (void)

-*-------------------------------------------------------------------*/
int main(void)
   {
   // Set up PLL, VPB divider, MAM and interrupt mapping
   System_Init();

   // Prepare to flash LED
   LED_FLASH_Init();

   // Prepare for “random” delays
   RANDOM_LOOP_DELAY_Init();

   while(1)
      {
      // Set up Timer 1 for 1-second sandwich delay
      SANDWICH_DELAY_T1_Start(1000);

      // Change the LED state (OFF to ON, or vice versa)
      LED_FLASH_Change_State();

      // “Random” delay
      // (Represents function with variable execution time)
      RANDOM_LOOP_DELAY_Wait();

      // Wait for the timer to reach the required value
      SANDWICH_DELAY_T1_Wait();
      }


   return 1;
   }

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

Listing 40: Implementing a SANDWICH DELAY for the LPC2000 family (`main.c`)

```
/*-------------------------------------------------------------------*-

   sandwich_delay_t1.c (v1.00)

  ------------------------------------------------------------------

   “Sandwich delay” for the LPC2000 family using Timer 1.


-*-------------------------------------------------------------------*/

#include “main.h”

/*-------------------------------------------------------------------*-

  SANDWICH_DELAY_T1_Start()

  Parameter is - roughly - delay in milliseconds.

  Uses T1 for delay (Timer 0 often used for scheduler)

-*-------------------------------------------------------------------*/
void SANDWICH_DELAY_T1_Start(const unsigned int DELAY_MS)
   {
   T1PC = 0x00;    // Prescale is 0
   T1TCR &= ~0x01; // Stop timer

   // Set the match register (MR0) to required value
   T1MR0 = ((PCLK / 1000U) * DELAY_MS) - 1;

   // Set flag on match, reset count and stop timer
   T1MCR = 0x07;

   T1TCR |= 0x01;   // Start timer
   }

/*-------------------------------------------------------------------*-

  SANDWICH_DELAY_T1_Wait()
  Waits (indefinitely) for Sandwich Delay to complete.

-*-------------------------------------------------------------------*/

void SANDWICH_DELAY_T1_Wait(void)
   {
   // Wait for timer to reach count
   while ((T1IR & 0x01) == 0)
     {
     ;
     }

   // Reset flag (by writing “1”)
   T1IR |= 0x01;
   }

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

Listing 26: Implementing a SANDWICH DELAY for the LPC2000 family (example): file ***(sandwich_delay_t1.c)***

References

Cottet, F. and David, L. (1999), "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.

Furber, S. (2000) "*ARM System-on-Chip Architecture",* Addison-Wesley.

Jerri, A. J. (1997), "The Shannon sampling theorem: its various extensions and applications a tutorial review", Proc. of the IEEE, Vol. 65, pp. 1565-1596.

Hughes, Z.H. and Pont, M.J. (2004) "Design and test of a task guardian for use in TTCS embedded systems". In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004 (Birmingham, UK, October 2004), pp.16-25. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0180-3].

Key, S.A., Pont, M.J. and Edwards, S. (2004) "Implementing low-cost TTCS systems using assembly language". Proceedings of the Eighth European conference on Pattern Languages of Programs (EuroPLoP 2003), Germany, June 2003: pp.667-690. Published by Universitätsverlag Konstanz. ISBN 3-87940-788-6.

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

Kurian, S. and Pont, M.J. (in press b) "Restructuring a pattern language which supports time-triggered co-operative software architectures in resource-constrained embedded systems". In: Proceedings of the Eleventh European conference on Pattern Languages of Programs (EuroPLoP 2006), Germany, July 2006.

Mart, P., Fuertes, J. M., Ramamritham, K. and Fohler, G. (2001), "Jitter Compensation for Real-Time Control Systems", 22nd IEEE Real-Time Systems Symposium (RTSS'01), London, England, pp. 39-48.

Mwelwa, C., Athaide, K., Mearns, D., Pont, M.J. and Ward, D. (2006) "Rapid software development for reliable embedded systems using a pattern-based code generation tool". Paper presented at the Society of Automotive Engineers (SAE) World Congress, Detroit, Michigan, USA, April 2006. SAE document number: 2006-01-1457. Appears in: Society of Automotive Engineers (Ed.) "In-vehicle software and hardware systems", Published by Society of Automotive Engineers. [ISBN: 0-7680-1763-7].

Nahas, M. and Pont, M.J. (submitted) "The impact of scheduler implementation on task jitter in real-time embedded systems".

Phatrapornnant, T. and Pont, M.J. (2006) "Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling" IEEE Transactions on Computers, **55** (2): 113-124.

Philips (2004) "LPC2119 / 2129 / 2194 / 2292 / 2294 User Manual", Philips Semiconductors, 3 February, 2004.

Pont, M.J. (2001) "Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers", Addison-Wesley / ACM Press. ISBN: 0-201-331381.

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

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

Pont, M.J. and Ong, H.L.R. (2003) "Using watchdog timers to improve the reliability of TTCS embedded systems", in Hruby, P. and Soressen, K. E. [Eds.] *Proceedings of the First Nordic Conference on Pattern Languages of Programs, September, 2002 ("VikingPloP 2002")*, pp.159-200. Published by Microsoft Business Solutions. ISBN: 87-7849-769-8.

Pont, M.J., Norman, A.J., Mwelwa, C. and Edwards, T. (2004) "Prototyping time-triggered embedded systems using PC hardware". Proceedings of the Eighth European conference on Pattern Languages of Programs (EuroPLoP 2003), Germany, June 2003: pp.691-716. Published by Universitätsverlag Konstanz. ISBN 3-87940-788-6.

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

Patterns which help to avoid conflicts over shared resources in time-triggered embedded systems which employ a pre-emptive scheduler

Huiyan Wang, Michael J Pont and Susan Kurian
Embedded Systems Laboratory, University of Leicester,
University Road, LEICESTER LE1 7RH, UK.
hw79@le.ac.uk; M.Pont@le.ac.uk; sk183@le.ac.uk
http://www.le.ac.uk/eg/embedded/

Abstract

This paper is concerned with the use of patterns to support the development of software for reliable, resource-constrained, embedded systems. The specific focus is on systems with a time-triggered architecture in which task pre-emption can occur. The paper introduces one new abstract pattern (CRITICAL SECTION), and four new design patterns (DISABLE TIMER INTERRUPT, RESOURCE LOCK, PRIORITY INHERITANCE PROTOCOL and IMPROVED PRIORITY CEILING PROTOCOL).

Acknowledgements

We thank Juergen Salecker, who provided numerous useful suggestions on earlier drafts of this paper during the shepherding process.

This work is supported by a DTA award to Huiyan Wang from the UK Government (EPSRC).

Copyright

Copyright © 2005-2007 by Huiyan Wang, Michael J. Pont and Susan Kurian.
Permission is granted to copy this paper for purposes associated with EuroPLoP 2007.

Introduction

We are concerned with the development of embedded systems for which there are two (sometimes conflicting) constraints. First, we wish to implement the design using a low-cost microcontroller, which has – compared to a desktop computer – very limited memory and CPU performance. Second, we wish to produce a system with extremely predictable timing behaviour.

To support the development of this type of software, we have previously described a "language" consisting of more than seventy patterns for time-triggered (TT) embedded systems (e.g. see Pont, 2001). Work began on these patterns in 1996, and they have since been used it in a range of industrial systems and numerous university research projects (e.g. see Mwelwa et al., 2007; Kurian and Pont, 2007; Ayavoo et al., 2007).

The patterns presented in this paper mark the start of a new area of work. Unlike the majority of papers we have presented at previous PLoP conferences, the work described here is based on the use of a pre-emptive scheduler. This brief paper describes one new abstract pattern (CRITICAL SECTION), and four new design patterns (DISABLE TIMER INTERRUPT, RESOURCE LOCK, PRIORITY INHERITANCE PROTOCOL and IMPROVED PRIORITY CEILING PROTOCOL). The solutions presented in this set of patterns are from published papers: they have been adapted (as necessary) to work with TT architectures and documented in pattern format.

Pattern structure

We now have a collection of well over 70 patterns for time-triggered systems. As our experience with this collection has grown, we have felt that there were ways in which the overall architecture could be improved in order to make the collection easier to use, and to reduce the impact of future changes. We are therefore in the process of re-working the collection into different pattern categories: Abstract Pattern, Design Pattern and PIE (Pattern Implementation Example). Abstract patterns address common design decisions faced by developers of embedded systems. Design Pattern identifies implementation details and is extensive references to Abstract Pattern. PIE identifies implementation specific design issues and includes extensive references to Abstract Pattern and relevant Design Pattern (Kurian and Pont, 2006).

References

Ayavoo, D., Pont, M.J., Short, M. and Parker, S. (2007) "Two novel shared-clock scheduling algorithms for use with CAN-based distributed systems", *Microprocessors and Microsystems*, **31**(5): 326-334.

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

Kurian, S. and Pont, M.J. (2007) “Maintenance and evolution of resource-constrained embedded systems created using design patterns”, *Journal of Systems and Software*, **80**(1): 32-41.

Mwelwa, C., Athaide, K., Mearns, D., Pont, M.J. and Ward, D. (2007) “Rapid software development for reliable embedded systems using a pattern-based code generation tool”. *SAE Transactions: Journal of Passenger Cars (Electronic and Electrical Systems)*, **115**(7): 795-803.

Pont, M.J. (2001) “Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers”, Addison-Wesley / ACM Press. ISBN: 0-201-331381.

CRITICAL SECTION

{abstract pattern}

Context

- You are developing an embedded system using a computer system with CPU and / or memory resources which are – compared with typical desktop designs – rather limited.
- Your system employs a single CPU.
- Your system employs a TT SCHEDULER [Pont et al., 2007: this conference].
- Your system supports task pre-emption.
- Predictable system behaviour is a key design requirement: in particular, predictable task timing is a concern.

Problem

How can you avoid conflicts over shared resources during the execution of critical sections?

Background

Scheduling and TT architectures

For general background information about scheduling (and scheduling of time-triggered systems in particular), please refer to the pattern TT SCHEDULER [Pont et al., 2007: this conference]. TT SCHEDULER provides background information on key concepts such as TTC, TTH and TTRM scheduling.

Shared resources and critical sections

Our focus in this pattern will be on TTH and TTRM designs in which task pre-emption can occur. Our particular concern will be with the issue of resources which may be accessed by more than one task at the same time. Such "shared resources" may – for example - include areas of memory (for example, two tasks need to access the same global variable) or hardware (for example, two tasks need to access the same analogue-to-digital converter). The code which accesses such shared resources is referred to as a "critical section".

Suppose that there are two tasks, $Task_A$ and $Task_B$ in a system, which are illustrated in Figure 25. There is one shared resource. In the figure, N represents the normal section and C represents the critical section (that is, the section which involve access to a shared resource). From t1 to t4, $Task_A$ and $Task_B$ are attempting to run "simultaneously".

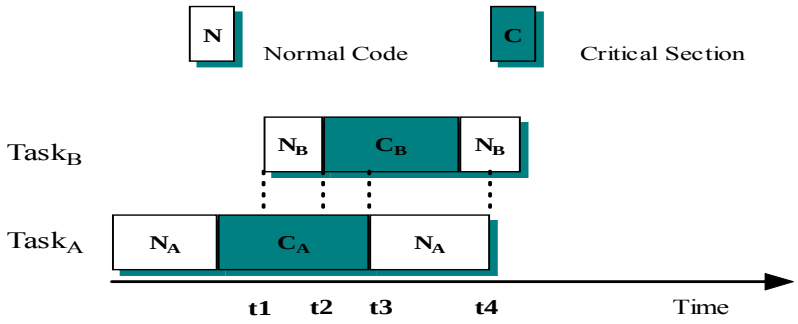


Figure 25 Two Tasks with a shared resource

In a TTC system, a task cannot be pre-empted by another task and the two tasks shown in Figure 25 are scheduled as shown in Figure 26. As in this example, there are no conflicts caused by the shared resources in TTC systems.

Figure 26 Two tasks scheduled using a TTC scheduler

However, if the same tasks are scheduled in a pre-emptive system, there may be conflicts. For example, suppose the priority of $Task_B$ is higher than that of $Task_A$. $Task_B$ will then pre-empt $Task_A$ at time t1 while it is running the critical section (Figure 27).

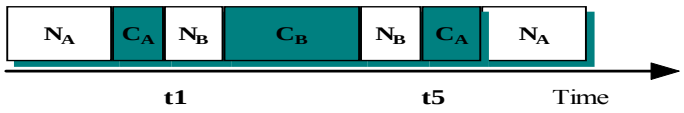


Figure 27 Two tasks scheduled in a pre-emptive system

Assume that the shared resource in the above example is some shared data (for example, numerical data stored in an array). The two tasks write and read the data in the critical section. $Task_B$ pre-empts $Task_A$ at t1 while it is reading the data: we will assume that $Task_A$ has read from half of the array at the time it is interrupted. We will further assume that $Task_B$ $Task_B$ has updated all of the data values in the array. After $Task_B$ finishes, $Task_A$ then continues, reading the remaining values from the second half of the array: it then processes a combination of "new" and "old" data, possibly leading to erroneous results (e.g. see Kalinsky, 2001).

In general, tasks must share data and / or hardware resources. However, the system designer must ensure that each task has exclusive access to the shared resources to avoid conflicts, data

corruption or “hanging tasks” (Pont, 2001; Labrosse, 2002; Laplante, 2004).

What is a resource lock?
A lock is the most common way to protect shared resources.

Before entering a critical section, a semaphore is checked. If it is clear, the resource is available. The task then sets the semaphore and uses the resource. When the task finishes with the resource, the semaphore is cleared.

Resource locking in this way requires care but is comparatively straightforward to implement. and affects only those tasks that need to take the same semaphore (Simon, 2001).

The main drawback is that it causes priority inversion in a priority based system (Sha, 1990; Burns, 2001; Renwick, 2004): see the next section for further details.

What is priority inversion?
In a priority-based system, each task is assigned a priority. In a TTC design, the scheduler will – when deciding which task to run next – always run the task with the highest priority. In a pre-emptive system, a high priority task may interrupt a lower-priority task while it is executing.

Priority inversion can occur in pre-emptive designs when resource locks are used. For example, suppose that a very low-priority task is using a resource. The resource will be locked. If a high-priority task is then scheduled to run (and use the resource) it will not be able to do so: in effect, the low-priority task will be given greater priority than the high-priority task.

See, for example, Figure 28 and Figure 29. Figure 28 shows an intended operation sequences of two tasks, $Task_H$ and $Task_L$, sharing a critical section C. Figure 29 shows how the priority inversion takes place. When $Task_L$ owns C, and $Task_H$ attempts to access it (at t_3), $Task_H$ is blocked and has to wait until time t_4 before it can run.

Please note that this is sometimes called “bounded priority inversion” (Burns, 2001; Renwick, 2004) or “controlled priority inversion” (Locke, 2002). In this case, the blocking time of $Task_H$ will not exceed the duration of the critical section C of $Task_L$.

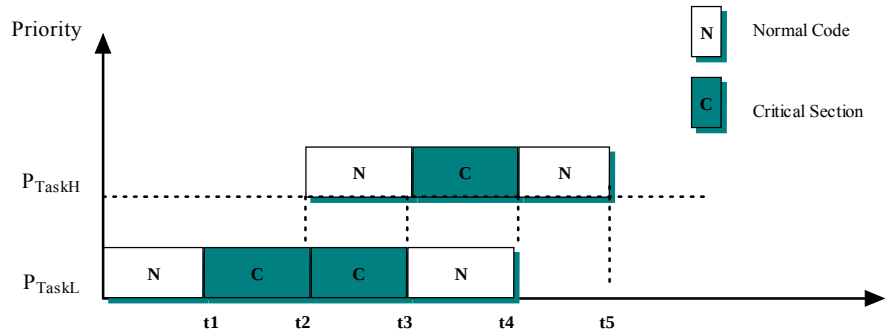


Figure 28 Operation Sequences of TaskH and TaskL

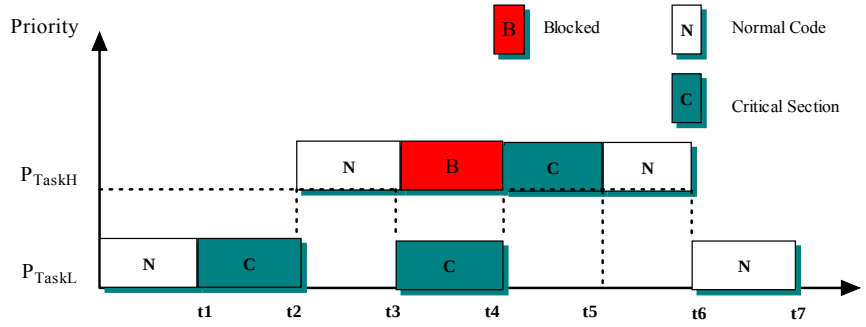


Figure 29 Bounded Priority Inversion

We can further suppose that $Task_M$ (with "medium" priority) pre-empts $Task_L$ when $Task_H$ is blocked by $Task_L$, the owner of the shared resource. $Task_H$ then has to wait until $Task_M$ relinquishes control of the processor and $Task_L$ completes the critical section. For example, see Figure 30: here, at t_4, $Task_M$ pre-empts $Task_L$.

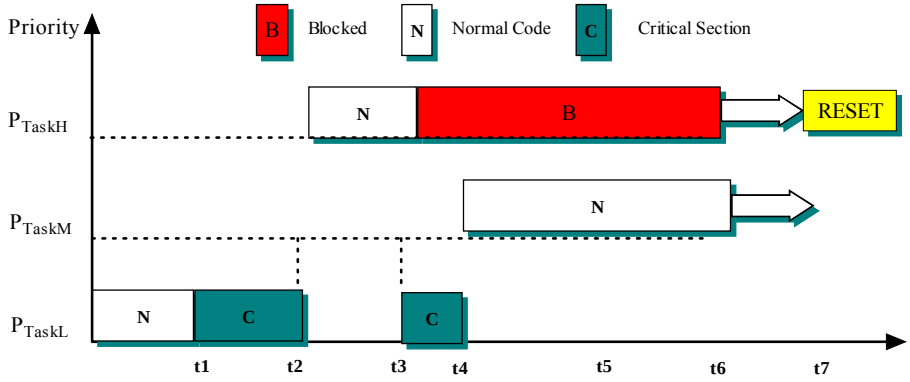


Figure 30 Unbounded Priority Inversion

In these circumstances, the worst-case waiting time for $Task_H$ is the sum of the worst-case execution times of $Task_M$ and the critical section of $Task_L$. This is called unbounded priority inversion (Renwick, 2004). If $Task_M$ runs for a long time (or "for ever"), $Task_H$ is likely to may

miss its deadline, with potentially serious consequences (shown as a system reset in Figure 30).

Unbounded priority inversion can be particularly problematic. For example, in 1997, the Mars Pathfinder mission nearly failed because of an undetected priority inversion (Jones, 1997).

What about deadlock and chained blocking?

As noted above, a locking mechanism may lead to priority inversion. However, this is not the only problem which is introduced by the use of locking mechanisms.

For example, suppose that $Task_H$ is waiting for a resource held by $Task_L$, while $Task_L$ is simultaneously waiting for a resource held by $Task_H$: neither task is able to proceed and – as a result - a deadlock is formed (see Figure 31 and Figure 32 for a simple example).

Figure 31 shows two tasks $Task_H$ and $Task_L$ which share two resources (C1 and C2): in this case, it is assumed that C1 is nested in $Task_L$ and C2 is nested in $Task_H$

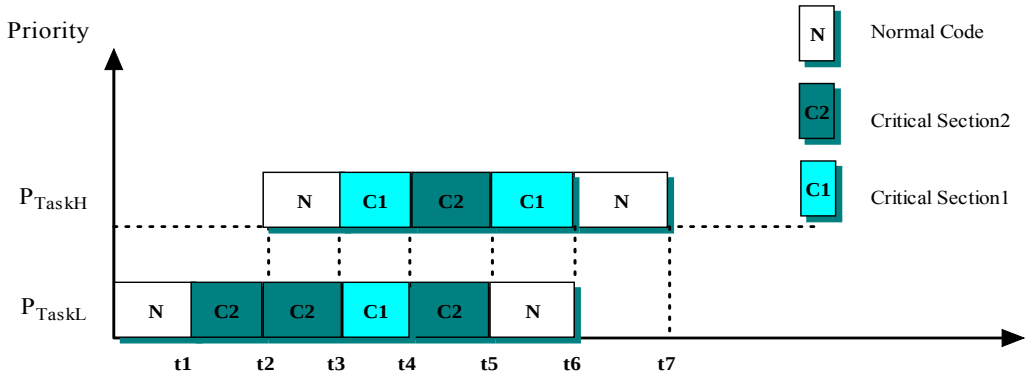


Figure 31 Operation Sequences of TaskH and TaskL

In Figure 32, at t_1, $Task_L$ locks C2, at t_2 $Task_H$ pre-empts $Task_L$ and starts to run, locks C1 at t_3, then is requiring C2 at t_4. Due that $Task_L$ has locked C2, $Task_H$ is blocked and $Task_L$ resumes running at t_4. At t_5, $Task_L$ requires C2 which is locked by $Task_H$, and both of tasks are blocked.

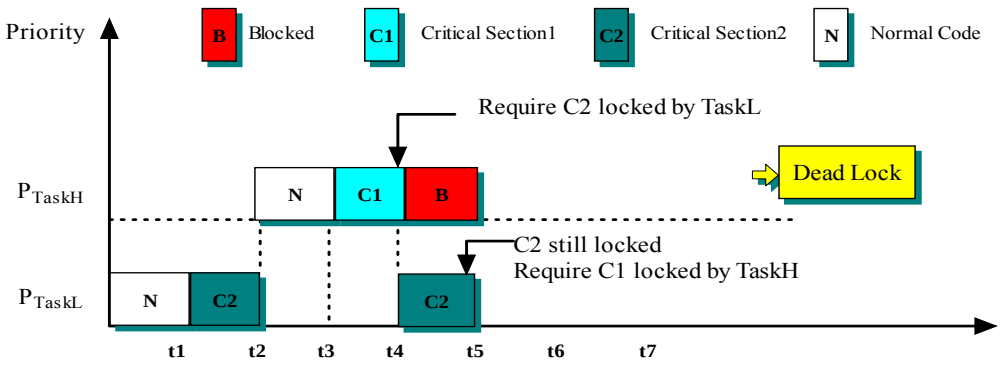


Figure 32 Deadlock: Operation Sequences of Tasks without Priority Protocols

In addition, locking mechanisms can cause chained blocking. Suppose $Task_H$ is waiting for a resource held by $Task_M$, while $Task_M$ is waiting for a resource held by $Task_L$, and so on (see Figure 33 and Figure 34). $Task_H$ needs to sequentially access resources C3 and C2, $Task_M$ accesses C2 with nested C1 and $Task_L$ accesses C1. Figure 34 shows that $Task_M$ is blocked by $Task_L$ at t_4 when it is requiring C1 which is locked by $Task_L$; $Task_H$ is blocked by $Task_M$ at t_7 when it is requiring C2 which is locked by $Task_M$. Therefore, the highest priority task $Task_H$ would be blocked for the duration of two critical sections, one is to wait for $Task_L$ to release C1, and another is to wait for $Task_M$ to release C2. As a result, a blocking chain is formed (Sha, 1990).

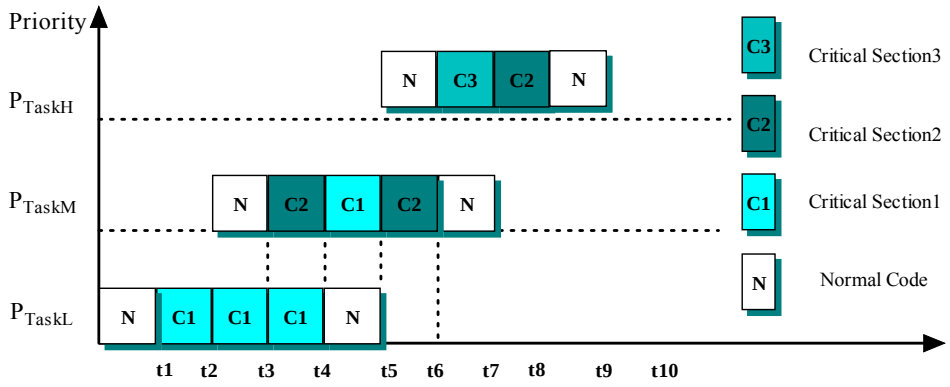


Figure 33 Operation Sequences of Three Tasks

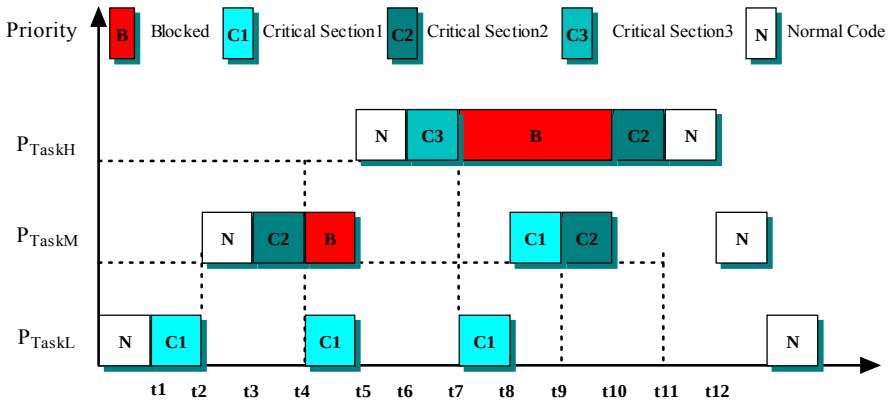


Figure 34 A blocking chain: Operation Sequences of Three Tasks without priority protocols

Solution

This pattern is intended to help answer the question: "How can you avoid conflicts over shared resources during the execution of critical sections?"

In general, the answer to this question is straightforward: you need to ensure that only one task attempts to access each shared resource at any time. There are two common ways of achieving this in a TT system architecture:

1. As noted in "Background", the simplest way of avoiding conflicts over shared resources in a time-triggered system is to use a TTC scheduler. This solution avoids the need for any of the mechanisms discussed in this paper.
2. You can disable interrupts and / or using a locking mechanism (probably in conjunction with a protocol that will help you avoid priority inversions). These solutions are discussed in "Related patterns".

Of these solutions, the first is the simplest and generally the most effective. No matter what you do in a pre-emptive design to protect your shared resources, they will still be shared and only one task can use them at a time. **As such, any form of locking mechanism provides only a partial solution to the problems caused by multi-tasking.**

Consider an example. If the purpose of Task A is to read from an ADC, and Task B has locked the ADC when the Task A is invoked, then Task A cannot carry out its required activity. Use of locks, or any other mechanism, will not solve this problem; however, they may prevent the system from crashing. Of course, by using a co-operative scheduler, these problems do not arise.

Please note that there may – in some circumstances – be two further options for you to consider:

1. Pre-runtime scheduling (e.g. Xu and Parnas, 1990). Use of a "pre-run time schedule design" may allow you to adapt your pre-emptive system schedule in order to ensure that – even with pre-emption – there are never conflicts over shared resources. Such techniques are not trivial to implement and are beyond the scope of the present paper.
2. Planned pre-emption. Adi and Pont (2005) have described an approach called "planned pre-emption" which avoids the need for locking mechanisms in TTH scheduler designs.[23]

Related patterns

This pattern is an abstract pattern, which provides background knowledge related to shared resources in embedded systems.

The following patterns describe some solutions to avoid shared resources conflicts and priority inversion:

DISABLE TIMER INTERRUPT

Disable interrupt is the simplest and fastest approach considered in this paper. However it may affect the response times of all other tasks in the system.

RESOURCE LOCK

Lock is the most common way to protect shared resources because it affects only those tasks that need to take the same semaphore. However, basic use of locking mechanisms can give rise to problems of priority inversion.

PRIORITY INHERITANCE PROTOCOL

The Priority Inheritance Protocol is intended to address problems with priority inversion.

IMPROVED PRIORITY CEILING PROTOCOL

The Improved Priority Ceiling Protocol is also intended to address problems with priority inversion.

Reliability and safety implications

If a system is to implement in pre-emptive architecture, applying the mechanisms discussed in this pattern will generally help you to construct a reliable and safe embedded system.

---

[23] Planned Pre-emption will form the basis for a future pattern. It is not considered further in this paper.

Overall strengths and weaknesses

☺ Increase system reliability

☹ Increase the complexity of systems implementation

☹ May be difficult to implement

☹ Some implementation may have to be written in assembly language

Context

- You are developing an embedded system using a computer system with CPU and / or memory resources which are – compared with typical desktop designs – rather limited.
- Your system employs a single CPU.
- Your system employs a TT SCHEDULER [Pont et al., 2007: this conference].
- Your system supports task pre-emption.
- Predictable system behaviour is a key design requirement: in particular, predictable task timing is a concern.

Problem

What is the simplest way of ensuring safe access to shared resources in your system?

Background

We provide some relevant background material in this section.

What is a shared resource?

For background information about shared resources, please refer to the abstract pattern CRITICAL SECTION [this paper].

The role of interrupts in TT systems

In general, an interrupt is a signal that is used to inform the processor that an event has occurred. Such event may (in general) include a timer overflow, completion of an A/D conversion or arrival of data in a serial port.

In TT systems, we only have a single interrupt source, linked to a timer overflow[24]. Disabling the timer interrupt (for short periods) may have little or no impact on the system behaviour.

Solution

This pattern is intended to describe the simplest way of avoiding conflicts over shared resources in a TT system which involves task pre-emption.

As noted in Background, in a time triggered system, only a single interrupt is enabled. This

[24] It is possible – using a Super Loop – to create very simple TTC designs which involve no interrupts (at all). Such architectures are not suitable for use with pre-emptive task sets and are not considered in this set of patterns: see Kurian and Pont (2007) for further details.

interrupt will be used to drive the scheduler (Pont, 2001). If we disable this interrupt, the scheduler will be disabled.

This gives us a simple mechanism to avoid conflicts over resources, as follows:

- When a task accesses a shared resource, it disables the timer interrupt.
- When the task has finished with the resource it re-enables the timer interrupt.
- During the time that our task is using the shared resource, the scheduler is disabled. This means that no context switch can occur, and no other task can attempt to gain access to the resource.

Overall, this is a very simple (and fast) way of dealing with issues of shared resources in a TT design. However, it may have an impact on all the tasks in the system. Therefore, interrupts should be disabled as little as possible (and for a very short period of time).

Related patterns and alternative solutions

The following patterns describe some solutions related to avoid shared resources conflicts:

- PRIORITY INHERITANCE PROTOCOL
- IMPROVED PRIORITY CEILING PROTOCOL
- ORIGINAL PRIORITY CEILING PROTOCOL

Reliability and safety implications

Disabling the interrupt of a system affects the response times of the interrupt routine and of all other tasks in the system. It is not safe if it keeps interrupt disable for long time. However, if the critical section is very short (e.g. we wish to access a single global variable), it is a fast and easy solution.

Overall strengths and weaknesses

☺ Easy to implement

☺ Only way to protect critical sections if ISRs and tasks share resources

☺ It is faster than other protection mechanisms, such as locks

☹ Increases interrupt latency

☹ May decrease system's ability to respond to external events

☹ Need carefully recognise the situation in which interrupts should be disabled

RESOURCE LOCK

{design pattern}

Context

- You are developing an embedded system using a computer system with CPU and / or memory resources which are – compared with typical desktop designs – rather limited.
- Your system employs a single CPU.
- Your system employs a TT SCHEDULER [Pont et al., 2007: this conference].
- Your system supports task pre-emption.
- Predictable system behaviour is a key design requirement: in particular, predictable task timing is a concern.

Problem

How can you implement a resource lock for your embedded system?

Background

We provide some relevant background material in this section.

What is a shared resource?

For background information about shared resources, please refer to the abstract pattern CRITICAL SECTION [this paper].

The role of interrupts in TT systems

For background information about interrupts in TT systems, please see DISABLE TIMER INTERRUPTS [this paper].

Solution

The pattern is intended to help you answer the question: "How can you implement a resource lock for your embedded system?"

A lock appears, at first inspection, easy to implement. Before entering the critical section of code, we 'lock' the associated resource; when we have finished with the resource we 'unlock' it. While locked, no other process may enter the critical section.

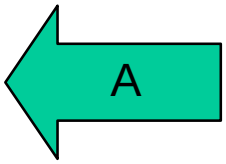
This is one way we might try to achieve this:

1. Task A checks the 'lock' for Port X it wishes to access.
2. If the section is locked, Task A waits.
3. When the port is unlocked, Task A sets the lock and then uses the port.

4. When Task A has finished with the port, it leaves the critical section and unlocks the port.

Implementing this algorithm in code also seems straightforward, as illustrated in Listing 42.

```
#define UNLOCKED   0
#define LOCKED     1

bit Lock;  // Global lock flag

// ...

// Ready to enter critical section
// - Wait for lock to become clear
// (FOR SIMPLICITY, NO TIMEOUT CAPABILITY IS SHOWN)
while(Lock == LOCKED);

// Lock is clear
// Enter critical section

// Set the lock
Lock = LOCKED;

// CRITICAL CODE HERE //

// Ready to leave critical section
// Release the lock
Lock = UNLOCKED;

// ...
```



Listing 42: Attempting to implement a simple locking mechanism in a pre-emptive scheduler. See text for details.

However, the above code cannot be guaranteed to work correctly under all circumstances.

Consider the part of the code labelled 'A' in Listing 42. If our system is fully pre-emptive, then our task can reach this point at the same time as the scheduler performs a context switch and allows (say) Task B access to the CPU. If Task Y also wants to access the Port X

We can then have a situation as follows:

- Task A has check the lock for Port X and found that the port is not locked; Task A has, however, not yet changed the lock flag.
- Task B is then 'switched in'. Task B checks the lock flag and it is still clear. Task B sets the lock flag and begins to use Port X.
- Task A is 'switched in' again. As far as Task A is concerned, the port is not locked; this task therefore sets the flag, and starts to use the port, unaware that Task B is already doing so.
- …

As we can see, this simple lock code violates the principal of mutual exclusion: that is, it allows more than one task to access a critical code section. The problem arises because it is possible for the context switch to occur after a task has checked the lock flag but before the task changes the

lock flag. **In other words, the lock 'check and set code' (designed to control access to a critical section of code), is itself a critical section.**

This problem can be solved. For example, because it takes little time to 'check and set' the lock code, we can disable timer interrupt for this period (see DISABLE TIMER INTERRUPT [this paper]).

Related patterns and alternative solutions

The following patterns describe some solutions related to avoid shared resources conflicts:

- DISABLE TIMER INTERRUPT
- PRIORITY INHERITANCE PROTOCOL
- IMPROVED PRIORITY CEILING PROTOCOL

Reliability and safety implications

As discussed in CRITICAL SECTION [this paper], use of a resource lock can give rise to problems of priority inversion. The patterns PRIORITY INHERITANCE PROTOCOL and IMPROVED PRIORITY CEILING PROTOCOL provide (partial) solutions to this problem.

Overall strengths and weaknesses

☺ Easy to implement

☹ May give rise to "priority inversion" if not implemented with care.

Context

- You are developing an embedded system using a computer system with CPU and / or memory resources which are – compared with typical desktop designs – rather limited.
- Your system employs a single CPU.
- Your system employs a TT SCHEDULER [Pont et al., 2007: this conference].
- Your system supports task pre-emption.
- Predictable system behaviour is a key design requirement: in particular, predictable task timing is a concern.

Problem

How can you ensure that access to shared resources in your system is mutually exclusive and avoids priority inversion?

Background

For background information about shared resources, please refer to the abstract pattern CRITICAL SECTION [this paper].

Solution

The pattern is intended to help you answer the question: "How can you ensure that access to shared resources in your system is mutually exclusive and avoids priority inversion?"

To avoid unbounded priority inversion, Sha et al (1990) introduced the priority inheritance protocol.

In the priority inheritance protocol, a low priority task inherits the priority of a high priority task if the high priority task requires access to the shared resource owned by the low priority task. The high priority task is blocked and the low priority task can continue executing its critical section until it releases the resource. Then its priority returns to the original and the high priority task starts to run.

This process is illustrated in Figure 35. In this example, the priority of $Task_L$ is raised to the priority of $Task_H$ once the higher-priority task tries to access the critical section (at t_3).

If an medium-priority $Task_M$ pre-empts $Task_L$ while executing the critical section, due to the

fact that the priority of $Task_L$ has been raised to the priority of $Task_H$, $Task_M$ has to wait until $Task_H$ completes and $Task_L$ finishes the critical section. Therefore, the highest-priority task $Task_H$ is not pre-empted by the medium-priority $Task_M$.

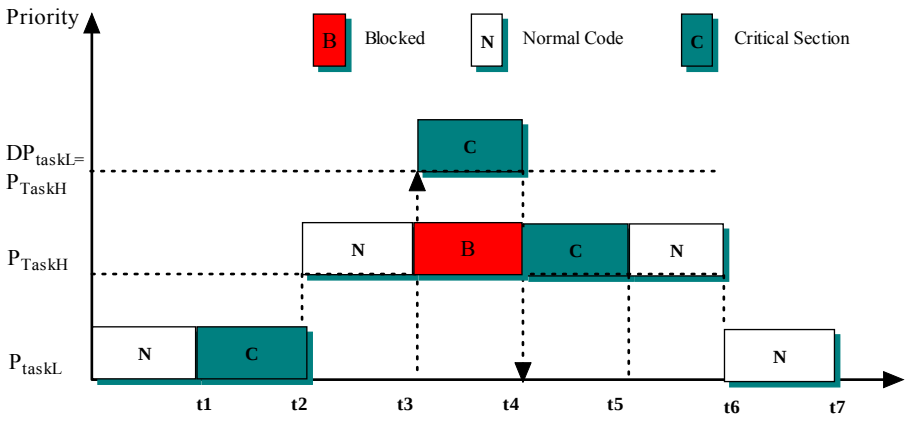


Figure 35 Priority Inheritance Protocol

Related patterns and alternative solutions

The following pattern describe some solutions related to avoid shared resources conflicts:

- IMPROVED PRIORITY CEILING PROTOCOL

Reliability and safety implications

Use of priority inheritance protocol avoids priority inversion, increases the stability of a system. Most of commercial real time operating system support this feature, or as additional package, such as µC/OS-II, eCOS, FreeRTOS and RTLinux etc.

Although priority inheritance protocol is generally found to be an effective and powerful technique to prevent priority inversion, it is not without its critics (e.g. see Yodaiken, 2002).

Of particular concern is that this protocol cannot avoid deadlock and blocking chains when tasks have nested shared resources. Therefore, to use PIP safely, an appropriate software architecture design is needed that avoids unnecessary coupling between tasks through shared resources (Locke, 2002), and it is important to avoid nested resources in applications.

Overall strengths and weaknesses

☺ Prevents priority inversion

☺ Has better average –case performance than Priority Ceiling protocol. When a critical section is not contended, priorities do not change, there is not context switches and no additional overhead.(Lcoke,2002; Renwick,2004)

☹ Difficult to implement when compared with DISABLE TIMER INTERRUPT [this paper].

☹ Does not prevent deadlock and blocking chains (Sha, 1990).

☹ Wastes processor time if there are not immediate tasks ready to run during the time that a higher-priority task is blocked by a lower-priority task

☹ Worst-case performance is worse than the worst-case performance for priority ceiling protocol since nested resource locks increase the wait time (Lcoke,2002; Renwick,2004).

Context

- You are developing an embedded system using a computer system with CPU and / or memory resources which are – compared with typical desktop designs – rather limited.
- Your system employs a single CPU.
- Your system employs a TT SCHEDULER [Pont et al., 2007: this conference].
- Your system supports task pre-emption.
- Your tasks may have nested shared resources.
- Predictable system behaviour is a key design requirement: in particular, predictable task timing is a concern.

Problem

How can we ensure that the shared resources are mutually exclusive and that priority inversion, deadlock and blocking chains are avoided?

Background

For background information about shared resources, please refer to the abstract pattern CRITICAL SECTION [this paper].

Solution

The pattern is intended to help you answer the question: "How can we ensure that the shared resources are mutually exclusive and that priority inversion, deadlock and blocking chains are avoided?"

Nested resource locks are the underlying cause of deadlock and blocking chains. Therefore, the simplest solution is to avoid nested resource locks at the design stage (indeed, some operating systems do not allow use of nested locks).

Sha et al. (1990) presented an alternative solution to the priority inheritance protocol: this was the priority ceiling protocol (PCP). However, this original priority ceiling protocol is expensive to implement. A simplified version of the original PCP is widely used (Locke, 2002). In this pattern we refer to this as the "Improved Priority Ceiling Protocol" (IPCP)[25].

[25] IPCP is of often incorrectly referred to as the priority ceiling protocol. What we refer to as IPCP here is also known as the Priority Ceiling Emulation in Real-Time Java, Priority Protect Protocol in POSIX and as the Immediate Ceiling Priority Protocol (Burns, 2001).

In IPCP, each task has an assigned static priority; each resource has also been assigned a priority which is the highest priority of tasks that need access it (i.e. its priority ceiling: Burns, 2001). When a task acquires a shared resource, the task is raised to its ceiling priority. Therefore, the task will not be pre-empted by any other tasks attempting to access the same resource with the same priority. When the task releases the resource, the task is returned to its original priority.

The deadlock case shown in Figure 32 is illustrated in Figure 36 to explain how IPCP works. $Task_H$ and $Task_L$ access both resources C1 and C2. Thereby the ceiling priorities of C1 (P_{c1}) and C2 (P_{c2}) are the priority of $Task_H$ (P_{TaskH}). $Task_L$ runs first. At t_1, it needs to access C2, according to IPCP, it will be raised to the ceiling priority of C1, which equals to $P_{TaskH.}$ At t2, $Task_H$ is ready to run. However, its priority is the same as the dynamic priority of $Task_L$. It will not able to pre-empt $Task_L$ until $Task_L$ completes the critical sections and returns to the original priority. Therefore, the deadlock is prevented.

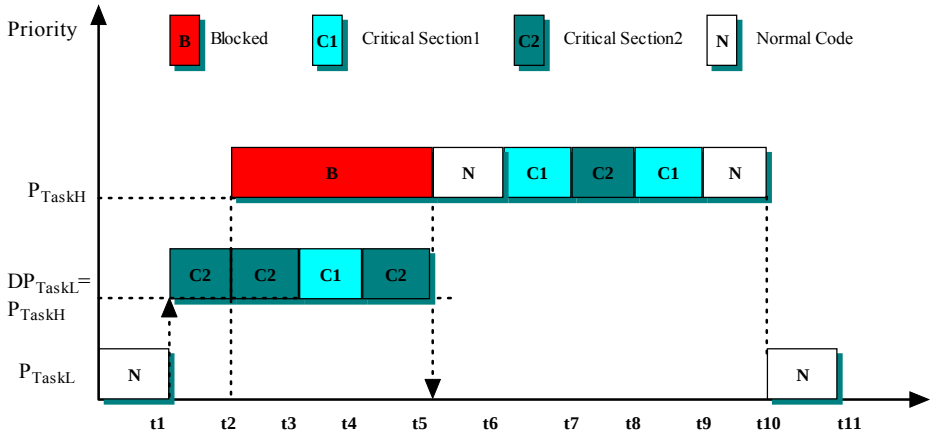


Figure 36 Operation Sequences of Tasks with IPCP

Related patterns and alternative solutions

Sha et al (1990) originally presented the priority ceiling protocol (PCP).

In original priority ceiling protocol (OPCP), each task has an assigned static priority; each resource has also been assigned a priority which is the highest priority of tasks that need access it, i.e. its priority ceiling, which are the same as the IPCP. There are two differences. One is that each task's dynamic priority is the maximum of its own static priority and its inheritance priority due to it blocking higher-priority tasks. The second is that a task can only lock a

resource if its dynamic priority is higher than the ceiling priority of any currently locked resource (Burns, 2001).

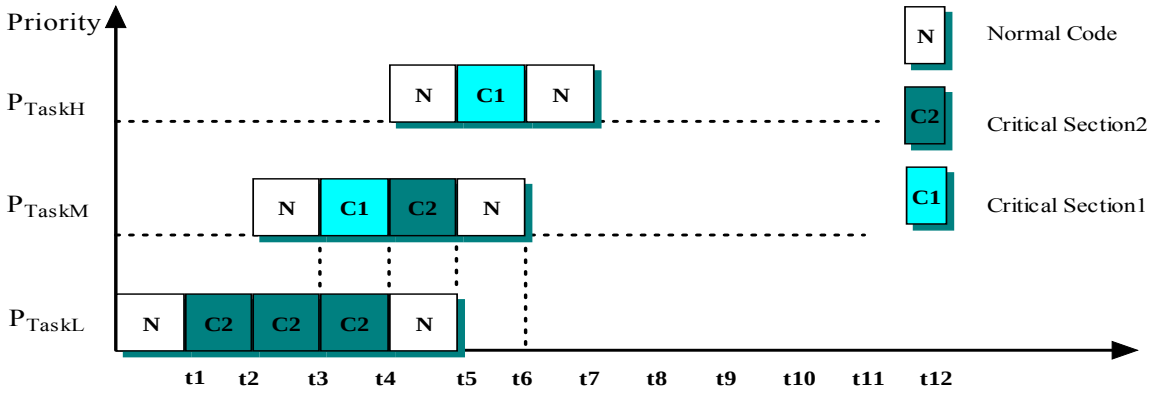


Figure 37 Operational sequences of three tasks with two shared resources

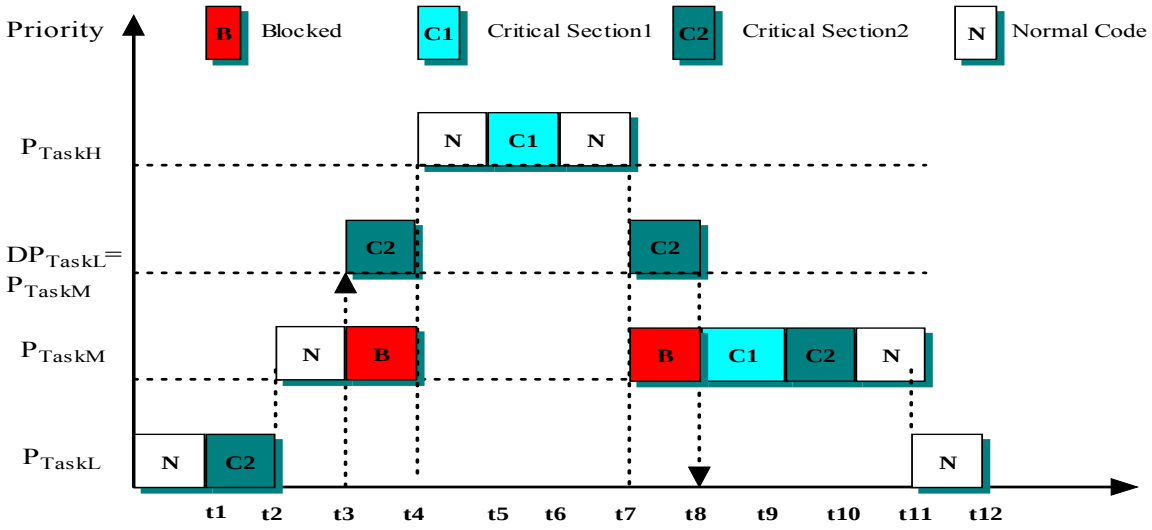


Figure 38 Operational sequences of three tasks with OPCP

Figure 38 explains how OPCP works. Three tasks intend to run shown in Figure 37. At t2, $Task_M$ pre-empts $Task_L$, at t3 $Task_M$ is attempting to lock the resource C1. However, due that its dynamic priority P_{TaskM} is not higher than the ceiling priority of C2 ($P_{c2} = P_{TaskM}$) which is currently locked by $Task_L$, it cannot lock C1, and is blocked by $Task_L$. $Task_L$ inherits $Task_M$ priority due to it blocking $Task_M$ and continues running in a higher priority. At t4 $Task_H$ starts to run and at t5, it is attempting to lock C1. Because its priority is higher than Pc2, it successfully locked C1 and runs to completion. After TaskL releases C2 and returns to its original priority at t8, $Task_M$ locks C1 and runs to finish.

To compare with improved priority ceiling protocol, the solution using IPCP in this case is illustrated in Figure 39. From Figure 38 and Figure 39 it is seen that there are 6 times context switches using OPCP and 4 times context switches in IPCP. In addition, OPCP needs to check blocking information for tasks dynamic priority. Therefore, OPCP is more difficult to implement.

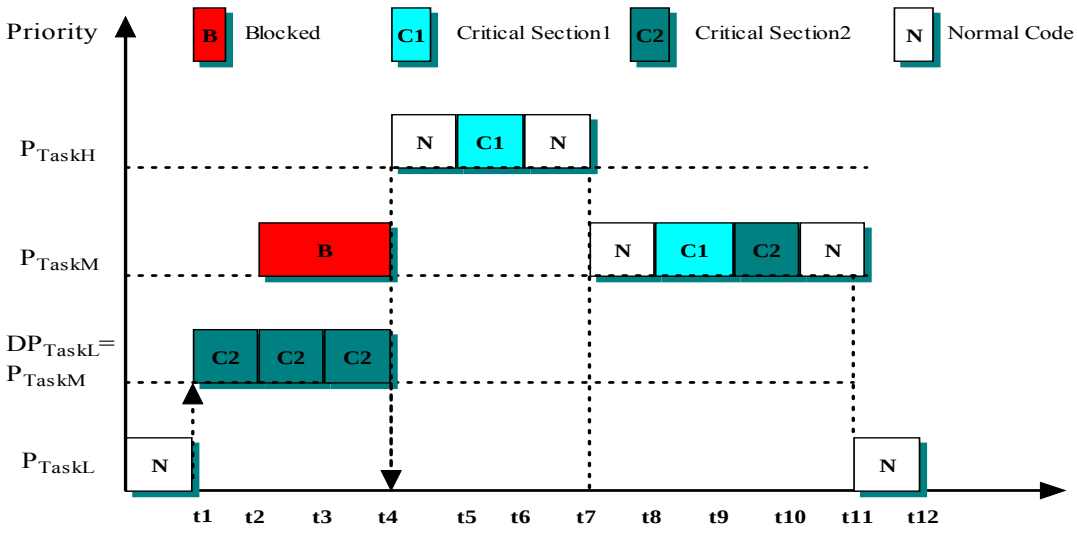


Figure 39 Operational sequences of three tasks with IPCP

Reliability and safety implications

IPCP is an attractive choice when there may be nested locks among tasks. Preventing deadlock and blocking chain increases the stability of a system.

Comparing with DISABLE TIMER INTERRUPT, IPCP is more difficult to implement (and test). An appropriate software architecture design is needed that avoids unnecessary coupling between tasks through shared resources (Locke, 2002), and avoids nested resources if possible.

If several tasks do use the same resource, designers should consider combining them into a single task (Renwick, 2004).

Overall strengths and weaknesses

☺ Prevents priority inversion

☺ Prevents deadlock and blocking chains

☺ Has better worst-case performance than PIP. The worst-case wait time for a high priority task waiting for a shared resource is limited to the longest critical section of any lower priority tasks that accesses the shared resource.

☹ Difficult to implement

☹ Requires static analysis of a system to find the priority ceiling of each critical section.

☹ Average –case performance is worse than PIP. IPCP changes a task's priority when it requires a resource, regardless of whether there is contention for the resource or not, resulting in higher overhead and many unnecessary context switches and blocking in unrelated tasks (Locke,2002)

References

Burns, A., Wellings, A. (2001) “Real-Time Systems and Programming Languages”, Addison-Wesley / ACM Press. ISBN: 0-201-72988-1.

Jones, M. (1997) “What really happened on Mars?” *http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html*

Kalinsky, D. (2001) “Context Switches” *Http://www.embedded.com/showArticle.jhtml?articleID=9900051*

Labrosse, J.J. (2002) “MicroC/OS-II: The Real-Time Kernel”, CMP Books. ISBN:1-57820-103-9

Laplante, P.A. (2004) “Real-Time Systems Design and Analysis”, Wiley-Interscience. ISBN: 0-471-22855-9.

Locke, D. (2002) “Priority Inheritance: The Real Story”, *http://www.linuxdevices.com/articles/AT5698775833.html*

Renwick, K., Renwick, B. (2004) “How to use priority inheritance”, *Http://www.embedded.com/showArticle.jhtml?articleID=20600062*

Sha, L., Rajkumar, R., Lehoczky, J.P. (1990) “Priority Inheritance Protocols: An Approach to Real-Time Synchronization”, *IEEE Transactions on Computers*, **39**(9): 1175-1185.

Simon, D.E. (2001) “An Embedded Software Primer”, Addison-Wesley / ACM Press. ISBN: 0-201-61569.

Xu , J. and Parnas, D.L. (1990) “Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations,’’ IEEE Transactions on Software Engineering, 16(3), pp. 360-369.

Yodaiken, V. (2002) “Against priority inheritance”, *http://www.linuxdevices.com/articles/AT7168794919.html*