Combining single-processor prototyping and code generation for the development of distributed embedded systems

Thesis submitted for the degree of
Master of Philosophy
at the University of Leicester

by

Peter J. Vidler

Department of Engineering
University of Leicester

April 2010
Combining single-processor prototyping and code generation for the development of distributed embedded systems

Peter J. Vidler

Abstract

Distributed embedded systems can be very challenging to test — not only is there increased complexity in the design of the software and operating systems, but we may also have to evaluate different scheduling and fault-tolerance approaches for suitability. The result is that system-level testing will generally take place in hardware at a much later time than would be possible with a single-processor system.

Carrying out high-level testing at an early stage of the software development process offers a number of advantages in the common scenarios of changing and poorly specified requirements; this thesis aims to address these concerns. A tool-supported process is introduced, which is capable of converting a single-processor prototype embedded system into various multi-processor equivalents, allowing the creation and use of a testable system at an earlier stage of development.

The process is then expanded to incorporate the automatic generation of source-code supporting redundancy, essentially allowing a developer to focus on writing code and leave the exploration of multi-processor scheduling and fault-tolerance mechanisms to the process described in this thesis.

These approaches are illustrated by means of empirical studies.
Acknowledgements

The work presented in this thesis was supported by the UK Government (EPSRC DTA Award). I would like to thank Dr. Michael Pont for his supervision, guidance and enthusiasm throughout this project. My thanks also go to the members of the Embedded Systems Laboratory, for their support and for providing some much needed relief during periods of stress.

I would also like to thank my parents for their support throughout this project. Finally, I would like to thank my partner, Hannah and our son, Benjamin, for putting up with me during the long hours required to get this finished and for providing encouragement when I needed it the most.
# Table of Contents

**Table of Contents**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Publications</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vi</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Software testing | 1 |
1.2 Embedded systems | 2 |
  1.2.1 Time-triggered scheduling | 2 |
  1.2.2 Scheduling multiple tasks | 3 |
1.3 Testing and safety concerns | 6 |
  1.3.1 Certification of safety-critical systems | 7 |
1.4 Research goals | 8 |
1.5 Contributions | 8 |
1.6 Layout | 9 |

## 2 Fault-Finding in Embedded Systems

2.1 Overview | 10 |
2.2 Static source-code analysis | 11 |
  2.2.1 Impact of programming language choice | 11 |
2.3 Testing and test-driven development | 12 |
  2.3.1 Hardware based unit testing | 12 |
  2.3.2 Simulation based unit testing | 13 |
## Table of Contents

2.4 Summary  

### 3 Modelling and Design-Space Exploration  

3.1 Overview  
3.2 Graphical models  
3.2.1 Applying UML throughout the development process  
3.2.2 The use of UML in embedded systems  
3.3 Source-code models  
3.4 Design-space exploration  
3.5 Summary  

### 4 Source-code parallelisation  

4.1 Overview  
4.2 Parallel architectures  
4.2.1 Shared memory architectures  
4.2.2 Distributed memory architectures  
4.3 Software tools and parallelisation  
4.3.1 Programming language extensions  
4.3.2 Automatic parallelisation  
4.4 Summary  

### 5 Code Generation Supporting Distributed Designs  

5.1 Program transformation  
5.1.1 Refactoring  
5.1.2 Challenges in refactoring C source-code  
5.1.3 Graph transformation  
5.2 Assumptions  
5.3 Basic formalism  
5.3.1 Splitting independent tasks  
5.3.2 Splitting linked tasks  
5.3.3 Splitting tasks on slave nodes  
5.4 Implementation  
5.4.1 Where do we convert?
5.4.2 Conversion process 38
5.4.3 Problems with pointers 40
5.5 Summary 41

6 Distributing a Cruise Controller 42
6.1 Overview 42
6.2 Multi-processor architectures 43
   6.2.1 TTC-SC1 scheduler 44
   6.2.2 TTC-SC3 scheduler 44
   6.2.3 Architectures 45
6.3 Testing 46
6.4 Summary 48

7 Code Generation Supporting Redundancy 49
7.1 Redundancy 49
   7.1.1 Hot standby 50
   7.1.2 Master-driven redundancy 50
7.2 Extending the formalism 51
   7.2.1 Duplicating independent nodes 52
   7.2.2 Duplicating linked nodes 53
   7.2.3 Duplicating networks 54
7.3 Supporting backup nodes 55
7.4 Summary 56

8 Making a Fault-Tolerant Cruise Controller 58
8.1 Overview 58
8.2 Redundant architectures 58
8.3 Testing 60
   8.3.1 Performance testing 60
   8.3.2 Failure testing 62
8.4 Summary 63
## Table of Contents

### 9 Tool Support and Automation

9.1 Overview .......................... 64
9.2 Generic schedulers ................. 66
  9.2.1 Matching schedulers and architectures .......... 66
  9.2.2 Interfacing tasks and schedulers ................. 68
9.3 Parsing .......................... 71
  9.3.1 A dynamic parser with Spirit ................... 71
  9.3.2 Generating the interface ....................... 73
9.4 Summary ........................ 74

### 10 Conclusion and Future Work

10.1 Evaluation ....................... 75
  10.1.1 Working with existing source-code and tools .... 76
  10.1.2 Limiting source-code structural changes ......... 76
  10.1.3 Adapting to architectural changes ............... 77
10.2 Discussion and future work ....... 78
  10.2.1 Source-code as a textual model ................ 78
  10.2.2 Conversion as design-space exploration ......... 79
  10.2.3 Conversion as refactoring ..................... 80
10.3 Conclusion ...................... 80

### A C99 Parser Source-code

A.1 Source of the syntax parser ........ 81

### B Target Specific Parser Source-code

B.1 Source of the implementation specific parser .... 99

### C Tool Support Source-code

C.1 Source of the conversion tool ....... 102

References .......................... 120
List of Publications

A number of papers were published during the course of the work described in this thesis. These are listed below (in 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.


List of Figures

1.1 A task executed by a time-triggered scheduler . . . . . . . . . . . 3
1.2 Co-operative scheduling preventing task pre-emption . . . . . . . 5
1.3 One possible redundancy design . . . . . . . . . . . . . . . . . . . . 6
5.1 One-way communication between two tasks . . . . . . . . . . . . . 31
5.2 Example conversion with two tasks . . . . . . . . . . . . . . . . . 32
5.3 A type graph for the rules . . . . . . . . . . . . . . . . . . . . . . . 33
5.4 The splitIndependentTasks rule . . . . . . . . . . . . . . . . . . . 34
5.5 The splitLinkedTasks rule . . . . . . . . . . . . . . . . . . . . . . . 36
5.6 The splitSlaveTasks rule . . . . . . . . . . . . . . . . . . . . . . . 37
5.7 Generated code mapping from task data to the scheduler . . . . . 39
5.8 Illustrating the difficulty of analysing code with pointers . . . . . 40
6.1 The cruise control system used for the case-study . . . . . . . . . . 43
6.2 Message timing for the TTC-SC1 scheduler . . . . . . . . . . . . . 44
6.3 Message timing for the TTC-SC3 scheduler . . . . . . . . . . . . . 44
6.4 The application of splitLinkedTasks to the CCS . . . . . . . . . . . 45
6.5 The application of splitSlaveTasks to the CCS . . . . . . . . . . . 46
6.6 The CCS system’s response to step changes in desired speed . . . 47
7.1 Voting in a system employing hot standby redundancy . . . . . . . 50
7.2 An updated type graph, including the aggregation relationship . . 51
7.3 The cloneSingeNode rule . . . . . . . . . . . . . . . . . . . . . . . . 52
7.4 The cloneOutputNode rule . . . . . . . . . . . . . . . . . . . . . . . 53
7.5 The cloneInputNode rule . . . . . . . . . . . . . . . . . . . . . . . . 54
7.6 The cloneFullNode rule . . . . . . . . . . . . . . . . . . . . . . . . 55
7.7 Illustrating the potential complexity of the problem . . . . . . . . 56
8.1 The first application of cloneFullNode . . . . . . . . . . . . . . . . 59
8.2 The second application of cloneFullNode .......................... 59
8.3 The performance of various architectures, some with redundancy 61
8.4 The effect of a node failure in 3, 4 and 5 node systems .......... 62

9.1 Steps in a typical build process .................................... 65
9.2 Steps in the modified build process ............................... 66
9.3 Simplified graph of the five node CCS architecture .......... 67
9.4 Metadata provided only to the MasterNode ...................... 67
9.5 Metadata provided only to nodes receiving input ............... 68
9.6 Simplified task code for Node2 and backups .................. 68
9.7 Data unions and buffers for the two global variables .......... 69
9.8 Functions to map variables to buffers and back again ........ 70
9.9 C++ source-code for part of a C99 parser ....................... 72
9.10 A rule allowing for non-standard extensions .................. 72
9.11 The basic format of an SCH_Add_Task call ..................... 73
CHAPTER 1

Introduction

Einstein argued that there must be simplified explanations of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer.

— FREDERICK P. BROOKS, JR.
“No Silver Bullet”

This chapter introduces the importance of testing in large-scale software development, as well as giving an overview of embedded systems and the complications of employing modern testing methods in their implementation. Finally, an outline is given of how this and other related problems are addressed through the research presented in this thesis.

1.1 Software testing

Of all the modern engineering disciplines, software engineering stands alone in having no established development procedure, or even a fully standardised higher-level notation for design. This may be due to the relative youth of the field, but still poses a serious problem for an area that encompasses numerous applications where safety is of critical concern. It is partly because of this lack that testing has become such a significant part of modern software development.

Studies have shown that more than half of the time and cost of the average software development project is spent on testing (Hambling, 1993; Harrold, 2000), and even more in the case of safety-critical applications. It is therefore important that such testing be made as efficient and, preferably, as cheap as possible without sacrificing its effectiveness (or the reliability of the system).

1Parts of this chapter have previously been published in (Vidler and Pont, 2006).
It has been demonstrated that finding and removing faults early in the development process can offer significant cost advantages (Boehm, 1982; Mosley and Posey, 2002; Shull et al., 2002). Since testing is a key method for finding faults in software projects, it is clear that continual testing from an early stage of a project can be desirable from both software quality and cost perspectives.

The key goal of thesis is to address the need for early and ongoing system testing within the context of distributed, time-triggered embedded systems.

1.2 Embedded systems

There are many possible definitions of what, precisely, constitutes an “embedded system”. For the purpose of this thesis, an embedded system is defined as being any application containing one or more processor (such as a microcontroller) that is not generally considered (by its users) to be computer-based (Pont, 2001). This includes a very wide range of systems, such as televisions, cars, aircraft, microwave ovens and Automated Teller Machines (ATMs) amongst a huge variety of others.

Many such embedded systems operate within very tight resource constraints and so run without the use of an off-the-shelf Real-Time Operating System (RTOS). These “deeply” embedded systems typically employ some form of simple (bespoke) scheduler that is responsible for ensuring that individual tasks (generally small units of code designed to perform a specific function) are executed at appropriate times.

1.2.1 Time-triggered scheduling

Embedded systems can be classified as either time-triggered (Kopetz, 1997) or event-triggered (Siemens et al., 2005). Flicking a switch, for example, is an external event that may cause a certain task to be executed in an event-triggered system. A time-triggered system may detect the same event by periodically executing a task that monitors the switch for changes and then acts appropriately if

---

2Where the term “time-triggered” indicates that all tasks in the system are executed continually at fixed intervals.
3Systems can be described as event-triggered if there are tasks in the system that are executed in response to external (typically aperiodic) events.
they are detected. This action of periodically checking for an event is commonly referred to as polling.

While this makes time-triggered systems sound more complicated than their event-triggered equivalents to design, the reality is that they can be made significantly more predictable. This is because tasks are executed periodically (as in Figure 1.1), so it is possible to know in advance when the task should execute — something that is impossible in the event-triggered system for the switch example, as it relies solely on an event that could be caused by the user at any time.

![Figure 1.1: A task executed by a time-triggered scheduler](image)

Consequently, the design of time-triggered systems can often involve finding the longest amount of time the system can take in responding to an event and using that time to determine a maximum value for use as the corresponding task’s period. For example, we may wish to detect and respond to the switch press within at most 100 ms, which makes that effectively the maximum value for the polling task’s period.

Because of their predictability, time-triggered systems are widely recognised as providing benefits to both reliability and safety (Allworth, 1981; Nissanke, 1997; Pont, 2001) in some of the more safety-critical applications (such as those used in the automotive and aerospace industries). All of the schedulers considered in this thesis share the same basic time-triggered nature, but this only specifies how (and when) tasks are started and not how they affect one another. Any scheduler that can execute more than one task must have a method of dealing with conflict resolution, when two or more tasks attempt to run simultaneously.

### 1.2.2 Scheduling multiple tasks

Schedulers that control more than one task generally fall into one of three main categories: pre-emptive (Moitra, 1986; Audsley et al., 1995), co-operative (Locke,
or a hybrid of the other two (Maaita and Pont, 2005; Pont, 2001). In a pre-emptive system, when one task needs to run while another is still being executed, the scheduler is free to interrupt the current task in favour of the new task (a process that requires context switching, or storing the state of a task so that it can be resumed later). The exact behaviour varies from scheduler to scheduler, but will often depend on the relative priorities that the developer assigns to each task.

Pre-emptive schedulers are powerful and capable of handling a wide variety of systems, but they are not without problems. For example, there is the possibility of data corruption that can occur should a high-priority task overwrite data that a lower priority task is in the middle of reading from. Working around this problem requires the introduction of concepts such as locking [4], that add significant complexity throughout the source-code for tasks.

Locking introduces its own share of problems, such as the priority inversion (Jie and Lee, 2003) that occurs when a low priority task locks a shared resource (typically for communicating or processing data) that is also required by a subsequent task with higher priority; the new task will not be able to pre-empt the old, despite its higher assigned priority. There are a number of solutions to this problem, such as priority inheritance, but each introduces problems and complications of its own and each further increases the complexity of the system.

By contrast, a co-operative scheduler may simply execute each task in turn, never allowing any task to interrupt (pre-empt) another (shown in Figure 1.2). This effectively avoids all of the problems listed above (as well as the overhead of context switching) and so provides the advantage of simplicity, amongst others (Bate, 2000).

Such simplicity comes with a cost: because no task can be interrupted, the frequency with which tasks can be executed is limited by their combined execution time. In other words, if the system contains (for example) both short tasks that must run frequently and very long tasks, it may not be suitable for use with a co-operative scheduler (as the more frequent tasks must still wait for the longer tasks to complete before running).

---

[4]While a resource is locked, any other task that tries to access it will be blocked from continuing until the task that owns the lock chooses to release it.
There are two possible compromises, offering much of the simplicity of co-operative schedulers while avoiding some of the problems of coping with long tasks. The first of these is a hybrid scheduler, which consists of a set of tasks that are co-operative with respect to one-another (they cannot interrupt any other task in the system) and one (typically short) task that can pre-empt any other (Maaita and Pont, 2005). The main advantage of such a hybrid system is that the problems of concurrent data access are limited to the single pre-emptive task, enabling a much simpler design while retaining many of the advantages of a pre-emptive system.

The second compromise is to employ multiple processors, and split the set of tasks amongst them (Vidler and Pont, 2005). This allows the use of a fully co-operative scheduler on each processor, which allows the developer to retain much of the simplicity of the actual task code (even though the scheduler must be made more complex to deal with the necessary communication between processors).

One of the main problems with such systems is that there are a vast array of different designs (both in terms of schedulers and multi-processor architectures) available for the developer to choose from, even within the constraint of Time-Triggered Co-operative (TTC) schedulers that is the focus of this thesis. The number and variety of design options, along with the possibility of the chosen design or underlying hardware platform changing at some point in the development process, make the early testing of such systems particularly challenging.
From this we can see that any process designed to facilitate the early testing of a distributed system must be capable of adapting to a number of different scheduler and architectural choices, as such things may be subject to change throughout the development process.

### 1.3 Testing and safety concerns

There are a number of factors that make embedded systems difficult to test, one of which is the fact that the system could be running on any of a wide variety of resource-constrained architectures. This means that the hardware must be considered during testing, either by carrying out tests on the actual hardware to be used (Smith et al., 2005), or through a software simulation of the hardware architecture. The former makes testing somewhat difficult and time consuming to accomplish with any frequency, while the latter cannot always easily simulate distributed architectures.

The use of multiple processors may actually be a requirement in many safety-critical embedded systems. In the design of such systems, the possibility that any single component can fail without warning must be considered (Pflanz and Vierhaus, 1998). The most effective way to be tolerant of such failures is to ensure that every component in the system (processor included) has at least one appropriate backup (as in Figure 1.3), along with a reliable mechanism (preferably fault-tolerant itself) to ensure that the backup takes over when the component fails (Isermann et al., 2002).

![Figure 1.3: One possible redundancy design, with a single backup per processor](image)

There are a vast number of different architectures that can be used to pro-
provide this redundancy for a system (Hammett, 2002). With so many choices, it can be difficult for developers to select an appropriate design; constraints on development time and cost also conspire to limit the number of such architectures that can be explored for a given project. Because the chosen architecture is important in existing embedded software testing methods (Smith et al., 2005), the connected process of testing different hardware designs complicates matters significantly.

Testing is not an optional process in the development of safety-critical software. Such systems are almost always required to pass some form of certification, which is very likely to impose further and more rigorous testing requirements.

1.3.1 Certification of safety-critical systems

Certification of safety-critical software is usually done to a given standard, such as the RTCA/DO-178B standard for aviation systems (RTCA, 1992), or the IEC 61508 standard for general safety-critical systems (IEC, 1998), amongst many others. Testing is a large part of such standards, but they additionally require evidence of the thoroughness of testing.

The testing discussed so far has been black-box (or functional) testing, where a system or unit of code is executed with a set of inputs and the outputs are observed to determine its functional correctness. Functional tests are written without knowledge of the source-code being tested and, as such, it can be impractical to show evidence of thoroughness using only functional testing.

By contrast, white-box (or structural) testing involves writing tests for the actual structure of the code. Their thoroughness can be directly (and formally) evaluated through the use of coverage metrics, which tell us how much of the code is being assessed by the suite of tests.

Structural testing with maximum coverage can be time-consuming and expensive to carry out (Dupuy et al., 2000), which has a number of implications in the design of safety-critical software. For example, systems may be built from components that already possess suitable test-suites, which can then be reused along with the component. While some additional integration-level testing will always be required, this method scales well for some of the more popular coverage metrics (Sundmark et al., 2008).
As white-box testing is always based on the actual structure of the component, it is clear that modifying any source-code may invalidate any existing testing. As a result, we can see that any source-code alteration should be minimised; such techniques, while useful, will greatly limit our ability to create and reuse structural tests throughout the development process.

1.4 Research goals

As mentioned in Section 1.1, the main goal of the research presented here is to look at how the need for early testing in distributed, time-triggered embedded systems may be addressed through a tool-supported process. From the context and discussion above, three additional goals and associated requirements can be identified:

1. To improve the ease and efficiency of testing and development in general, the process should work with existing source-code and tools as much as possible.

2. To be useful in real software development where requirements and design decisions may change, the process should be adaptable to a range of different architectural choices.

3. To be suitable for use in the development of systems requiring certification, the process should avoid altering the existing source-code structure as much as possible (and thus minimise the impact on any existing structural tests).

These goals will be used to evaluate the work presented here, as well as previous contributions in related areas.

1.5 Contributions

This thesis makes a number of contributions:

1. An original method of converting source-code targeting a single-processor prototype into source-code targeting multiple, distributed processors is introduced. This conversion is not carried out for performance reasons, but
to allow an inherently multi-processor design to be prototyped and tested more easily.

2. The conversion process is further expanded into a novel technique designed to produce source-code supporting fault-tolerance through several redundancy strategies.

3. A software tool is presented that automates much of the conversion process detailed throughout this thesis. This tool is designed to be integrated into a standard build-process, so that work may continue on the prototype source-code even when the target architecture, scheduler or redundancy mechanism are altered.

4. The above approaches are illustrated by means of empirical studies.

1.6 Layout

The remainder of this thesis is structured as follows. Chapters 2, 3 and 4 discuss further related work in the various areas covered in this thesis. Chapter 5 details the process necessary to convert from source-code for a single processor prototype system into distributed architectures, and Chapter 6 details an initial case-study used to verify the basic functionality of the process.

Chapter 7 discusses the changes made to the conversion process in order to allow the generation of source-code supporting redundancy, while Chapter 8 details a further case-study used to test the fault-tolerance of a generated system. Chapter 9 discusses the issues that are specific to automating the process presented in the previous chapters and Chapter 10 gives some concluding remarks.

Finally, Appendices A, B and C list various parts of the source-code for the tool used to automate the conversion process.
CHAPTER 2

Fault-Finding in Embedded Systems

_Beware of bugs in the above code; I have only proved it correct, not tried it._
— Donald E. Knuth

This chapter discusses related work in the areas of static source-code analysis and dynamic program analysis (testing), and how they relate to embedded systems and the specific focus of the work presented here.

2.1 Overview

As detailed in Chapter 1, finding and removing faults early in the development process can offer cost advantages over doing so at a later stage (Boehm [1982], Mosley and Posey [2002], Shull et al. [2002]). There are a wide variety of methods used to find faults in software, but in general many of them fall into one of two categories: static source-code analysis and dynamic program analysis.

Testing is a form of dynamic analysis (Harrold [2000]) that involves executing code (with certain test cases as input) and gathering output data for further analysis and verification. This is in addition to unit testing, which aims to verify that internal functional units meet certain requirements (which are generally defined through the unit tests themselves). However, both of these techniques involve the analysis of the system at runtime, whereas it is often possible to catch faults even earlier through the direct analysis of the source-code itself.
2.2 Static source-code analysis

Static analysis is often used as a generic term for any (generally automated) source-code inspection that does not require the program to be executed in order to be carried out. Tools that automate this analysis vary from simple syntax based guideline checkers (such as the well known “lint” program for checking C source-code) to complex systems offering the promise of formal verification of a program’s semantics (or even the mathematical verification of formal, abstract models describing the program’s behaviour). Regardless of the extent to which the source-code is analysed, all such tools depend on one thing: the programming language itself.

2.2.1 Impact of programming language choice

To some extent, the compiler acts as the most basic form of static analysis, enforcing the rules of the programming language before a program can be executed. The extent of the compiler’s analysis depends on how strict the rules of its programming language are. For example, the Ada programming language was developed under contract from the U.S. Department of Defence specifically as a language for developing safety-critical embedded systems (Marciniak, 1984). To aid in this mandate, Ada employs features such as (very) strong typing — where the compiler imposes strict checks as to the type of variables being used in the program and the validity of the operations being performed on them (for example, disallowing the implicit conversion between different types). Efforts have been made to further restrict the Ada language, theoretically making it safer still (Carre and Jennings, 1988; Smith, 1992; Guaspari et al., 1990; Michell and Saaltink, 1998). Despite this, the C programming language remains the most widely used language for programming embedded systems and, as such, is the language chosen for use in the work presented in this thesis.

The C programming language is weakly typed. This means that even though variables must be annotated with a specific type in C source-code, the compiler does not enforce such restrictions rigourously; as such, source-code written in C may (for example) implicitly convert been different types without the knowledge (or consent) of the developer. Such problems, amongst many others, can make C a difficult language to use for safety-critical systems. However, Hatton (1995)
argues that most of the problems with C are well known and are thus easier to avoid, and further that the relative safety of a programming language can only be examined fairly when the support of any additional verification tools is also taken into consideration.

2.3 Testing and test-driven development

The vast majority of previous work in Test-Driven Development (TDD), and Extreme Programming (XP) in general, has been based around desktop systems (Beck, 2002). However, the specific focus of this thesis is on embedded systems and there has been little work done in the area of employing TDD approaches with such systems. This is most likely due to the various problems with employing agile methods that were discussed in Section 1.3. Indeed, Dahlby (2004) notes that “. . . because embedded systems are more rigid in some aspects than other software systems, agile methods don’t apply universally, but rather need pragmatic adaptation”. Some of the more significant adaptations must deal with the difficulty of testing (and employing TDD) for use with embedded hardware.

2.3.1 Hardware based unit testing

Most of the existing attempts at introducing the TDD approach to embedded systems involve the adaptation of more traditional unit testing frameworks to operate on embedded hardware. For example, the E-TDD (Smith et al., 2005; Daeninck et al., 2006) method involves the use of a direct port of the “CppUnitLite” unit testing framework. This ported framework operates directly on the embedded hardware and includes the ability to check that constraints on execution time are met for a particular section of code. While this is a useful feature for testing embedded systems (where such constraints are often required), execution times can vary significantly due to a number of factors, so a great deal of care must be taken in the construction of such tests. They must effectively execute the code to be timed within a context (including the hardware and software states, as well as any required inputs) that would provide the worst-case execution time in order to be truly useful as a method of timing verification. This is difficult to achieve in general, and especially so in a TDD approach where such tests are supposed to be written prior to the code that they are designed to test.
Performing unit tests directly in embedded hardware also offers a number of difficulties, as it is impossible to include such internal testing without changing the behaviour of the system in some way (Harris 2006). For example, Smith et al. (2005) notes that including unit tests increases heap memory usage significantly, potentially beyond the memory limits of the system under test. The suggested workarounds — such as the use of additional external memory solely for such unit tests — often provide difficulties of their own, including the additional time required to access external memory (reducing the accuracy of execution time assertions). These problems, combined with the time and resources required to run frequent testing in hardware, can make simulation based approaches an attractive alternative.

### 2.3.2 Simulation based unit testing

Simulation offers several advantages for unit testing over hardware based techniques. In addition to the reduced time required to carry out such tests and the fact that no additional hardware must be set up for each developer, simulation based methods can be adapted to fit different hardware architectures with relative ease. This may not be the case with many hardware based approaches, which can be tied to their specific hardware architectures, such as in Schmitt (2004).

One exception to this is the approach, presented by Karlesky et al. (2006), allows the use of both simulation and hardware based methods. This approach draws on experience with utilising TDD in Graphical User Interface (GUI) based desktop applications, where the Model View Presenter (MVP) design can be used to separate an application’s function logic from its presentation (Fowler 2006). This separation can allow unit testing to be carried out automatically, without affecting the presentation itself (Alles et al. 2006). The same form of separation is used to decouple the system’s functional logic from its hardware (allowing simulation based unit testing as well as testing of the hardware itself).

Clearly such an approach depends on the source-code following a fairly specific design from very early in the software development process. It is unclear exactly how this approach would deal with changes to the underlying hardware and how (or even if) it could be used to represent multi-processor systems (with or without redundancy).
These embedded unit testing approaches, while interesting, do not address the same basic problems as the research in this thesis — the problems of facilitating testing at an early stage in the development process and of rapidly exploring the multitude of available design options. Similarly, none of these approaches support the development and testing of redundancy designs — something that is required in many safety-critical systems.

2.4 Summary

This chapter has discussed some of the previous work related to finding faults in embedded systems, including methods of both static and dynamic analysis. The static analysis of a program’s source-code can be useful in greatly increasing confidence in the correct operation of a system, through a formal mathematical proof. However, it is important to remember that such proof only applies to the source-code (or mathematical model) on which it was based and so cannot account for the reliability or correctness of any additional processes required to reach the final executable code; such processes may include compilation in the case of source-code analysis and additionally the implementation refinement in the case of more mathematical models.

For these reasons, dynamic analysis (usually in the form of testing) is required to further increase confidence in the system’s correctness. In embedded systems such analysis can take place directly in hardware, or through the use of simulation. The former potentially provides the advantage of accuracy (over the higher-level, model-based methods of simulation), while the latter is faster and so more suited to testing that must be carried out frequently. Both static and dynamic analysis are supported by the process described in this thesis, as both techniques benefit from the rapid development of a fully functioning system from a more abstract source-code based model.

The next chapter examines some related work in such modelling approaches, as well as the design-space exploration of embedded systems.
CHAPTER 3

Modelling and Design-Space Exploration

When you actually sit down to write some code, you learn things that you didn’t get from thinking about them in modelling terms... there is a feedback process there that you can only really get at from executing some things and seeing what works.

— Martin Fowler

This chapter discusses related work in the model-based design of embedded systems and the processes and tools used for design-space exploration in such systems.

3.1 Overview

Using the approach for designing embedded systems presented in this thesis, the prototype system used as the input of the process can be considered to be a text-(or more specifically a source-code) based modelling language. This is because it is more abstract than the resultant (generated) source-code, without detail specific to the target architecture and scheduler to be employed.

Proponents of model-driven approaches argue that developers using tools based around such models can produce many more lines of code in a day than is possible when hand-coding and — at the same time — eliminate errors introduced in this stage of development (Whalen and Heimdahl, 1999). It is also claimed that code-generation based on models provides support for software

\(^1\)Parts of this chapter have previously been published in Vidler and Pont, 2006.
CHAPTER 3. MODELLING AND DESIGN-SPACE EXPLORATION

maintenance (Bell, 1998).

There are many modelling languages available for use in software development, which come in nearly as many different varieties, but probably the most widely used are those employing graphical models.

3.2 Graphical models

Of all the graphical modelling techniques available, the Unified Modelling Language (UML) is perhaps the best known. The UML (Rumbaugh et al., 2005) consists of a unification (and standardisation) of different modelling approaches, many of which were already in widespread (if informally specified) use for software design and development. While this unification process gives the advantage of an easier transition from the earlier notations on which it is based, it can also result in the loss of simplicity in the modelling language (Paige et al., 2000). For example, several of the models present in UML offer overlapping (and even essentially duplicated) functionality, which can cause confusion in a language based on multiple models that must often interact to describe a complete system. A typical example is that of the UML’s sequence and collaboration diagrams, which are effectively equivalent (Paige and Ostroff, 1999).

Despite its relative complexity, UML has been increasingly used in the design of desktop software. There are several options for the use of such models to be extended throughout the remainder of the development process, including the automated generation of source-code from UML models.

3.2.1 Applying UML throughout the development process

Executable UML (Rumpe, 2002) allows the majority of development to take place on the model itself, between testing phases (also carried out directly on the model). This allows the use of UML models as a major part of the development process (beyond the early design phases) and is similar to the approach presented in this thesis in that changes are made on the model, rather than on the generated code, but still differs in the higher level (and largely visual) nature of UML itself. There is, however, some concern over the ability of such high-level

---

2As well as providing formal standardisation for such models, which were often informal and varying in notation from one use to the next.
models to adequately represent the lower level details that are required in many embedded systems (Torngren and Redell, 2000; Graaf et al., 2003).

Another key approach to utilising UML throughout the development process is to employ round-trip engineering (Demeyer et al., 1999). This involves generating source-code from the high-level model, carrying out some development on the generated code, then in some way synchronising (or reverse engineering) the source-code back into model form. This offers the advantage of allowing direct changes in the lowest level of source-code, thus bypassing the usual limitations of using such a high-level model. Unfortunately, many of the more useful and abstract models in UML cannot easily be used with round-trip engineering without further refinement (Paige et al., 2000) and it has been argued that UML itself is not entirely suitable for this purpose (Demeyer et al., 1999).

3.2.2 The use of UML in embedded systems

Despite the advantages of UML based code-generation, this approach is not yet the norm amongst developers of embedded systems. Some have suggested that this is because there is a mismatch between UML design models and embedded software (Martin et al., 2001; Schätz et al., 2003). As mentioned previously, it has also been suggested that this is because of the inability of UML to address specific low-level constraints such as: timing; the handling of periodic time-triggered systems; limited memory; power use and predefined hardware platform technology (Torngren and Redell, 2000; Graaf et al., 2003).

Efforts are being made within the embedded systems community to support UML. For instance, various extensions to UML have been proposed (Kukkala et al., 2005; Vanderperren and Dehaene, 2005). Of these extensions, the Systems Modelling Language (SysML) may be the most promising. SysML is “a domain-specific language for systems engineering applications” (SysML Partners, 2005). It is based on a subset of UML, but also includes additional components for representing systems that are not entirely software based (it can represent hardware architectures, for example).

Both SysML and UML are actually made up of a collection of different (but almost entirely graphical) models, which can be combined to describe a complete system. Each of these models uses its own notation, which is entirely separate from the others. This can make designing and developing systems with such
languages confusing, as multiple model types must be considered at once and each is written with its own unique notation.

An alternative to such graphical approaches is the use of text-based models, which can partially bypass some of these problems.

### 3.3 Source-code models

Giotto ([Henzinger et al., 2003](#)) is based on a mixture of high-level source-code supplemented with a custom domain-specific language for creating time-triggered embedded systems. As such, it is essentially a text-based modelling language that describes the inputs, outputs and functionality of a system and automatically produces an appropriate schedule for the system to use and generates the equivalent executable code.

To assist with this automatic scheduling, the Giotto program can be annotated with constraints. The purpose of such annotation is to provide more control over the scheduling process, such as mapping a task to a specific processor or assigning a specific priority to a task. This approach is somewhat different from the one presented here, which deals with the scheduling problem by supporting the generation of many different scheduler designs and thus allows the developer to test potential designs in a hardware-in-the-loop manner.

A general form of Giotto has also been developed — xGiotto ([Ghosal et al., 2004](#)) generalises the Giotto language to include support for event-triggered programming. Both Giotto and xGiotto can generate “E code” for an embedded virtual machine ([Henzinger and Kirsch, 2002](#)) in order to provide some measure of platform independence at runtime.

With respect to the goals in Section [1.4](#), Giotto does appear to provide the ability to produce a testable system at an early stage of the design process, but at the cost of introducing an entirely new language and syntax that must be learned. Also, it is not clear how structural testing would work with Giotto’s “E code” or with the generated scheduler.

Neither Giotto nor xGiotto appear to offer any direct support for the use of redundancy, or other fault-tolerance mechanisms. It may, however, be possible to generate such redundancy through the use of constraints, or by specifying backups manually in the Giotto source-code.
3.4 Design-space exploration

The process of converting from a single processor prototype to support many potential multi-processor architectures can also be used as the initial step in an exploration of the design-space.

The design space is the complete set of design alternatives applicable as the solution to a specific problem. Design-Space Exploration (DSE) is a general term applied to the process of searching this set of solutions in an attempt to find a workable design choice.

Any system with a significant number of design parameters may have a vast design space (Janneck and Esser, 2002). For example, the designer of a modern passenger car may have to choose between the use of one (or more) of the numerous communication protocols available. The resulting network may be connected in, for example, a bus or star topology (Tanenbaum, 1995). The individual processor nodes in the network may use event-triggered or time-triggered software architectures, or some combination of the two. The clocks associated with these processors may be linked using, for example, shared-clock techniques (Pont, 2001) or synchronisation messages (Hartwich et al., 2000). These individual processors may be C167 (Siemens, 1996), ARM (ARM, 2001), MPC555 (Bannatyne, 2003) or 8051 (Pont, 2001). A comparative analysis of some of these design choices has been carried out by Short et al. (2008).

The process of DSE for such systems may involve — for example — software simulation (Oliveira et al., 2006; Pimentel et al., 2002) or hardware-in-the-loop simulation (Rajagopal et al., 2004). Tool support is usually required in order to explore a number of design alternatives before a final decision is made on the nature of the hardware components and consequent software scheduling (Ernst, 1998).

Most existing automated methods of design-space exploration in embedded systems also attempt to automate some of the process of making design decisions. This is done through the analysis of some variable that is being explored in an attempt at optimisation. This is highly restrictive in embedded systems where such variables must be limited to factors that can be measured or modelled by design-space exploration tools. For this reason, most existing approaches are based on the use of one or more custom (often reconfigurable) processor cores in an SoC (System-on-Chip) architecture (Pimentel et al., 2002), where variables
such as the silicon area utilised by a design can be used as optimisation criteria for automating the basic design decisions \cite{Rajagopal2004}. Another common approach is to carry out design-space exploration in a specific area, such as on (cache) memory hierarchies, where factors such as cache size and the degree of associativity can be automatically altered and performance results obtained with relative ease \cite{Ghosh2003}.

As design spaces for embedded systems can be very large, a number of other approaches have been proposed for automating some of the process of exploring them. For example, \cite{Kienhuis1997} employed simulation to gather quantitative data about the performance of application-specific dataflow architectures. The gathered data is then used to allow different architectures to be rated, without the need to modify the algorithms describing the behaviour of the system.

The approach has been further extended by \cite{Keutzer2000} to include an “orthogonalisation of concerns”. This involves separating orthogonal aspects of the design\footnote{\cite{Keutzer2000} gives “function and architecture” or “communication and computation” as examples of orthogonal design aspects.} to simplify the exploration process. As with many related DSE approaches, including those proposed by \cite{Kwon2004} amongst others, it still relies heavily on the ability to easily simulate, quantify and rate the performance of different design alternatives at a fairly high level (above that of source-code).

Automation is important in DSE in order to reduce the full set of available choices to just those that are most desirable, which can then be tested and selected manually by the developer. The focus of this thesis is on systems employing multiple (redundant) CotS (Commercial, off-the-Shelf) processors, the exploration of which is difficult to automate without the use of higher-level models capable of being simulated or otherwise analysed \cite{Hsieh2000, Oliveira2006}.

### 3.5 Summary

This chapter has discussed some of the related work in component- and model-based design, as well as design-space exploration. Modelling allows a system to be specified and designed at a higher level than would otherwise be possible, but this can lead to difficulties during implementation due to the lack of low-
level details that many embedded systems require. There is also the common
difficulty of how subsequent model changes are applied to the implementation,
and how problems identified in testing may be mapped back into the model
for further analysis. Despite these issues, some modelling approaches can be
used to provide an unambiguous specification for the system, which is a major
advantage in their favour.

Design-space exploration is a way of coping with the combinatorial explo-
sion of architecture possibilities resulting from the wide array of potential design
choices. Most approaches rely on the (usually automated) testing and compari-
sion of one or more specific variables to be optimised (such as hardware silicon
usage for FPGA-based systems). Unfortunately, these approaches are not easily
adaptable to low-level embedded systems based on distributed CotS processors,
which can be difficult (or expensive) to simulate and almost impossible to auto-
matically reconfigure (as they exist in physical hardware). However, it is possible
to automate the first step of the exploration process, by generating appropriate
source-code for different architectures which might then be manually tested (or
simulated, if possible).

The next chapter examines some related work in the fields of parallel archi-
tectures and source-code parallelisation.
CHAPTER 4

Source-code parallelisation

Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

— DONALD E. KNUTH

THIS CHAPTER discusses related work in the area of parallel architectures, automated source-code parallelisation and communication in parallel code and tools.

4.1 Overview

The term parallelisation refers to the process of adapting code such that it is suitable for use with multiple processors. To be effective, techniques and tools that perform parallelisation will typically target one or more parallel architectures.

At the lowest level, many processors support a limited form of parallelism through the use of pipeline and superscalar architectures, along with the associated compiler support. This is a form of Instruction Level Parallelism (ILP) operating within a single processor and, due to the focus of this thesis being on distributed systems, will not be considered further here.

Higher levels of parallelism involve the use of multiple processors, which may or may not be running on separate physical chips. Such processors can communicate through a variety of mechanisms, from sharing a single address space to more network-oriented message passing schemes.
### 4.2 Parallel architectures

The full range of parallel architectures are too numerous to cover in detail here. Instead, we will focus on an overview of the various categories as defined by Patterson and Hennessy (2005).

#### 4.2.1 Shared memory architectures

Shared memory architectures include Symmetric MultiProcessor (SMP) — sometimes called Uniform Memory Access (UMA) — systems, as well as NonUniform Memory Access (NUMA) systems. In both cases multiple processors communicate through a single shared memory bus, but Patterson and Hennessy (2005) describe UMA systems as taking “the same time to access main memory no matter which processor requests it and no matter which word is requested”.

Duncan (1990) notes that sharing memory (in either of the above forms) can effectively bypass the latency problems that are often involved with message passing, but also that “other problems, such as data access synchronization and cache coherency, must be solved”. However, a shared memory architecture involving discrete processors will require the use of an external memory bus, which may itself suffer from latency issues over anything other than short distances (Fancey et al., 2001).

In the type of applications discussed in Chapter 1, it is sometimes necessary to employ discrete processors that must be spread over a distance for purely physical reasons. For example, an Antilock Brake System (ABS) in a passenger car may have a processor handling sensors and actuators for each wheel and a separate processor acting as a controller (Lee et al., 2004). In such a system the processors can be a significant distance apart — sharing a memory bus directly in the manner adopted by most shared memory systems may be impractical, due to the latency issues noted by Fancey et al. (2001).

In cases where shared memory is impractical or inadequate, distributed memory architectures may be preferred.

#### 4.2.2 Distributed memory architectures

Communication in distributed memory architectures is carried out by passing messages between processors, either directly in a point-to-point manner or more
commonly through a network. These networks will employ one of the various network topologies, such as a ring, mesh or tree configuration (Duncan, 1990) in addition to a suitable network protocol, such as the Controller Area Network (CAN) protocol. Note that while CAN provides a fully connected bus, it is still possible to produce more complex topologies (Muhammad and Pont, 2010).

As mentioned above, the message passing that is often required in distributed memory architectures brings with it the drawback of higher latencies. In addition, Li and Chen (1990) note that “it is widely recognized that directly programming this class of machines using explicit communication commands is inadequate: it requires explicit control and microscopic management of parallel resources; it is tedious, error-prone, and often unwieldy”.

A key issue that affects the performance and suitability of systems employing distributed memory architectures is the granularity of the threading model employed (Stone, 1993). This is a loose measure of the communication in the system — fine-grained parallelism involves short sequences of code with frequent communication, whereas coarse-grained parallelism involves long sequences of code with correspondingly infrequent communication.

Fine-grained systems have the advantage that parallelism can be exploited to a greater degree than may be possible for systems that use a coarse-grained threading model, but they also incur a cost in terms of latency from the increased communication required (Stone, 1993). By contrast, Kakulavarapu et al. (2000) notes that coarse-grained systems “can tolerate long latencies if the application provides enough parallelism because each task is long enough to amortize the communication overheads”.

As mentioned earlier, network latencies can be significant in the type of systems under consideration here. As we will see in Chapter 6, the coarse-grained threading model used throughout the work presented in this thesis shows little or no impact on performance from such network latency.

### 4.3 Software tools and parallelisation

Without the support of software tools, it can be difficult to successfully develop source-code for parallel architectures (Li and Chen, 1990). There are many such tools available, from tools that extend the programming language syntax
to include dedicated parallel constructs, to tools that attempt to automatically parallelise existing source-code.

### 4.3.1 Programming language extensions

Probably the most common approach to providing tool support for parallel software development is to extend the programming language with explicit support for parallelisation. For example, *Unified Parallel C* (UPC) provides additional language constructs such as `upc_forall`, which allows an otherwise standard C `for` loop to run on multiple processors (Cantonnet et al., 2004).

The problem with the approach taken by UPC, when combined with the low-level embedded systems discussed in Chapter 1, is that the compilers for such systems often provide extensions of their own for accessing hardware registers and peripherals. Providing additional support for the UPC extensions may mean modifying the compiler directly which, while possible in some cases, effectively rules out the use of commercial compilers or platforms, as well as any compiler that cannot be altered due to prior certification efforts.

Another programming language extension that deserves mention is *OpenMP*, which uses preprocessor pragmas (directives beginning with `#pragma omp`) and a runtime library instead of dedicated language constructs (Tian et al., 2003). This is superior to the approach used by UPC in that compilers which do not support OpenMP will simply ignore the extra directives (as required by the C standard), producing only sequential code instead. However, OpenMP suffers from the same problem as UPC — the requirement that a compiler for the target architecture must be altered for it to work.

Additionally, both OpenMP and UPC mainly produce fine-grained parallelism for shared memory architectures — UPC can also target distributed memory architectures, but not at a coarse granularity. OpenMP has more recently been updated to support *tasks*, which allow working with a coarse-grained threading model, but only when using a shared memory architecture (Ayguade et al., 2009). As discussed in Section 4.2.1, these architectures can have significant drawbacks for the type of application under consideration here, such as problems with latency.
4.3.2 Automatic parallelisation

While most tools in this area are focussed on allowing the developer to specify areas of code that should be executed in parallel, there are some that attempt this parallelisation automatically. Some of these tools work by generating parallelised source-code from models, such as the Giotto (Henzinger et al., 2003) approach that was discussed in Section 3.3.

Other tools aim to work on existing source-code; for example, Li and Chen (1990) detail a method of identifying and transforming communication in source-code targeting shared memory architectures to instead target distributed memory architectures. This essentially involves taking as input source-code written in an explicitly parallel C-based language — such as UPC — and replacing all of the parallel constructs with message passing equivalents.

Similar approaches exist for source-code that is written for OpenMP — Basumallik et al. (2007) and Renault et al. (2008) address the need for OpenMP support on distributed memory architectures through the use of source-code transformation. The result is essentially the same as provided by Li and Chen (1990), where the generated source-code uses explicit message passing to achieve its goal. The drawback of this approach is the same as for both UPC and OpenMP themselves — they all require changes to the compiler in order to work.

The approach employed in this thesis and detailed in Chapters 5 through to 9 works by identifying coarse-grained communication at the task level. The actual source-code transformation process is then performed as part of the build process — this takes place between the preprocessing and compilation phases and, as such, does not require changes to be made to the compiler itself, provided only that it supports separate preprocessing and compilation.

4.4 Summary

This chapter has discussed some of the related work in parallel architectures and software tool support for both automatic and explicitly specified parallelisation. There are essentially two categories that the majority of parallel architectures fall into — those that communicate through shared memory and those that communicate through message passing. Shared memory approaches provide advantages of (software) simplicity and relatively low latencies, but can
be impractical when multiple discrete processors must be placed at a significant distance apart.

By contrast, distributed memory architectures use a network connection that is more typically found in such cases, but which in turn can produce increased latency. This additional latency can be made more manageable through the use of a coarse-grained threading model.

Networks in distributed memory architectures are usually accessed through the use of explicit message passing routines, which can make the software more complex. This complexity can be addressed through the use of source-code transformation.

The next chapter (and the remainder of this thesis) details one such source-code transformation technique.
CHAPTER 5

Code Generation Supporting Distributed Designs

A notation is important for what it leaves out.

— Joseph Stoy

This chapter discusses the process used in this thesis to convert source-code for a single processor prototype system into code supporting any of a variety of equivalent architectures employing multiple processors

5.1 Program transformation

A key goal of the process presented here is that existing source-code can be modified to support different multi-processor architectures (including hardware, schedulers, redundancy schemes, etc) without altering the system’s overall behaviour. This includes the timing of tasks in the system and is somewhat different from Giotto (as described in Section 3.3), which specifically aims to alter such behaviour in an attempt at optimisation. Avoiding these alterations results in an approach that can be seen as a form of refactoring.

5.1.1 Refactoring

The term refactoring is usually used to refer to the practice of making internal improvements to a software system, without affecting the desired external behaviour (Fowler, 1999; Opdyke, 1992). The use of automated refactoring with

\footnote{Parts of this chapter have previously been published in (Vidler and Pont, 2005) and (Vidler and Pont, 2006).}
the form of embedded systems described in Section 1.2 provides a number of challenges, particularly in the area of ensuring that the system’s behaviour remains unaltered (Mens and Tourwé, 2004). For example, the behaviour of a control system is heavily dependent on timing; any refactoring performed on such a system must ensure either that the timing remains unaltered, or that the system is also altered to account for any change in timing (for example, parameters are adapted in a control algorithm). For the sake of simplicity, the process presented in this thesis opts for the former approach in that it attempts to avoid altering the timing of the system’s scheduled tasks as much as is possible.

This desire to avoid altering the timing, along with the context of embedded systems, has led to a somewhat different approach to refactoring than may typically be employed. There are a wide variety of formalisms dedicated to refactoring and specifically to providing guaranteed behaviour preservation, often enforcing pre- and post-conditions to do so (Opdyke, 1992; Roberts et al., 1997). In the systems considered here, these conditions would additionally need to prove the lack of timing alterations, which may require dynamic analysis. Such analysis can be both difficult and time consuming, as each possible architecture must either be fully constructed in hardware or accurately simulated before any testing can take place. Testing is already necessary in order to make an informed decision about the architecture to use, but having to undergo such additional regression testing each time can add significantly to the time needed for refactoring (and thus the cost of doing so).

An alternative approach, one that is employed in the work described in this thesis, would be to minimise the timing problem by restricting refactoring operations to the scheduler itself, leaving the source-code for tasks and their internal timing considerations intact. This allows a scheduler to be modified or even replaced automatically without significantly affecting the timing or performance of, for example, a typical control system (Vidler and Pont, 2006).

5.1.2 Challenges in refactoring C source-code

The application of any form of program transformation to source-code written in the C language can be particularly problematic, due in no small part to its frequent and even encouraged use of macro-style preprocessor directives (Garri- rido and Johnson, 2002). The C programming language is the “lingua franca”
for the developers of deeply-embedded systems (at least of the type detailed in Section 1.2 and under discussion here), so avoiding it in favour of a more easily parsed programming language is not always a practical option.

There have been several attempts at solving the preprocessor problem (Gar-ridó, 2005; Vittek, 2003), but perhaps the most common solution is to perform the refactoring directly on a translation unit that has already been pre-processed (Vittek et al., 2006). This approach may be the most reliable, as it operates on the same source-code level that is seen by the compiler.

Such refactoring of preprocessed translation units can remove much of the difficulty of dealing with the C programming language, but we must also consider exactly how such transformations may be formally specified. This can be achieved with graph transformation.

5.1.3 Graph transformation

Any program and its associated source-code can be represented as a graph, usually in the form of an abstract syntax tree (AST) or similar. Consequently, it is possible to formally express a given refactoring in terms of a graph transformation (Rozenberg, 1997; Ehrig et al., 1973) from a given initial state to the desired result of the refactoring (Mens et al., 2005). This approach of using graphs of the generic “before and after” cases provides a simple means of representing the graph transformation rules in a formal manner, without becoming too complex to easily follow (Heckel, 2006).

Although this approach allows a typical refactoring to be formally expressed for the modification of source-code or even higher-level models (Engels et al., 2002), the specific transformations discussed here must also express alterations in hardware. This can be dealt with if we are refactoring at the model level, but including such information in a source-code refactoring is less common. The remainder of this chapter discusses an original method of achieving this goal.

5.2 Assumptions

It is assumed that the input to the conversion process consists of the source-code for a prototype system designed for a single-processor time-triggered, co-operatively scheduled (TTC) architecture that utilises the same basic structure
and naming conventions as originally detailed by Pont (2001) and described in Section 1.2. This allows access to key information about the tasks in the system (including full source-code and timing requirements). For example, the names of tasks may be found by analysing the source-code for any call to the SCH_Add_Task function, which can then be used to find the definition of the task’s function and build a call graph for it.

This first assumption also means that communication between tasks in the prototype is carried out by reading from or writing to shared global variables. Note that in a TTC architecture, such communication is safe (even in the absence of locking mechanisms), because there can be no concurrency in a single-processor TTC scheduler. This, in turn, means that tasks are free from the problems of concurrent data access. The assumption can be used, through some careful analysis of the source-code, to determine the precise data flow between tasks. In Figure 5.1, Task1 writes to a particular (global) variable and Task2 reads from the same variable, which indicates that there is one-way communication from Task1 to Task2.

```c
// Global variable...
int some_value = 0;

void Task1(void)
{
    // Write access...
    some_value += 5;
}

void Task2(void)
{
    // Read access...
    int some_copy = some_value;
}

// Add all tasks (usually done in main)
SCH_Add_Task(Task1, 1, 5);
SCH_Add_Task(Task2, 2, 10);
```

Figure 5.1: One-way communication between two tasks

By making such assumptions, information can be acquired from the prototype that is useful for embedded systems but that can be lacking in many higher-level models (Torngren and Redell, 2000; Graaf et al., 2003). Meanwhile, details can be removed from the prototype that are unnecessary for understanding
the current problem and its solution, which has been a criticism of source-code based models in the past (Paige et al., 2000). This is important for achieving both the first and second research goals presented in Section 1.4.

Please note that the techniques discussed in this thesis can be adapted for use with other time-triggered system architectures, provided (only) that the tasks and communication schemes can be identified in the single-processor design.

5.3 Basic formalism

As noted in Section 5.2, the conversion process begins by identifying both the tasks in the prototype and the global variables that they access. We then require a formal way of representing this information, and the subsequent transformations that are carried out as we conduct the conversion. This allows us to specify the architectures that may be targeted as a result of the refactoring.

Figure 5.2 presents an example showing — in outline — the conversion of the single-processor design from Figure 5.1 into a multi-processor (two node) design.

![Diagram of single-processor design](image)

**Figure 5.2: Example conversion with two tasks**

More specifically, Figure 5.2 represents a system that has been analysed and found to contain two tasks with some communication between them; in this
case, Task1 is writing to the `some_value` global variable that Task2 accesses in a read-only manner (otherwise the link would be two-way). The conversion process can split Task2 onto a second processor node and we can see that the inter-task communication would happen between the two nodes instead of directly (a dashed arrow is left between tasks to represent the logical data flow that will be replaced).

We can view the conversion process more formally as a type of graph transformation, as discussed in Section 5.1.3. From the example above we can isolate the conceptual elements involved in the conversion process: the tasks and nodes of the systems (both before and after conversion). As described by Heckel (2006), we can use Unified Modelling Language (UML) class diagrams for detailing the concepts and relationships that can be used to represent a “snapshot” of any given system throughout the conversion process (which is in turn represented by a UML instance graph).

![Figure 5.3: A type graph showing concepts and the permissible relationships between them](image)

Figure 5.3 shows the type graph used to specify graphs representing the systems at various stages of the conversion process. It illustrates the design in significantly more detail than Figure 5.2, including the distinction between different node types: in particular, distinguishing between Master and Slave nodes is an important part of the conversion process as each requires different information about the target architecture (all of which can be derived from the graph that results from the transformation).

In Figure 5.3, the Task type represents individual tasks identified in the input...
source-code and so exists purely in software. These are unaltered by the conversion process, in order to fully preserve the behaviour of the original system and allow the reuse of existing structural tests. In contrast, the various node types (SingleNode, MasterNode and SlaveNode) represent a combination of both the physical hardware (the actual microcontroller and communications hardware) and the software (the scheduler).

### 5.3.1 Splitting independent tasks

The simplest scenario in the source-code conversion process is if we want to separate tasks that have no direct communication onto different processor nodes. As shown in Figure 5.4, the `splitIndependentTasks` rule simply moves the (non-empty) set of tasks $t_2$ onto the new node $n_2$. In this case, the conversion process will provide $n_2$ with a copy of the standard TTC scheduler and provide no provision for synchronising the two nodes (that is, the two nodes have independent timing).

![Figure 5.4: The splitIndependentTasks rule](image)

Of particular note in Figure 5.4 is that we employ UML composition to relate nodes and tasks. As with standard UML, use of the composition relationship
signifies that the lifetime of the task t1 (for example) is linked to the lifetime of the node n1. This becomes important when rules for supporting redundancy are introduced in Chapter 7, as it provides a mechanism to graphically differentiate tasks that rely on a single node from those that benefit from redundancy.

Also of note is the use of a negative application condition (Heckel, 2006), which is employed to specify the condition that none of the tasks in the set t2 may communicate with the task t1. This condition must be enforced, because communication between two such tasks would in turn require communication between the nodes n1 and n2, which is not provided for in the splitIndependent-Tasks rule.

### 5.3.2 Splitting linked tasks

Splitting independent tasks onto separate nodes is a relatively simple process. Figure 5.5 shows the splitLinkedTasks rule, or the transformation from aSingleNode with communicating tasks into one MasterNode and one SlaveNode connected via a single network (such as a CAN bus). As with the splitIndependentTasks rule, each of the two nodes (designated n1 and n2 here) is provided with a scheduler. It is important to note that the scheduler used is generic code that is not specifically customised to work with the architecture and tasks in question. In fact, we attempt to minimise the amount of source code actually generated as it is virtually impossible to test in isolation, whereas the unaltered scheduler source-code can be fully tested separately (as can the source-code for tasks).

The result of the transformation specified in the splitLinkedTasks rule contains details of both n1 and n2 nodes along with information about the architecture of the system, as well as providing communication details required by the schedulers of each node. These details are all obtained from the structure and connections of the resulting graph.

For example, we can see that there is now communication between n1 and n2 — specifically a value (d1) with a given type is now passed from n1 to n2. This information may be used by the scheduler on the MasterNode to send the required data to the SlaveNode at the appropriate times. It may also be used by the SlaveNode to determine what data is expected and act appropriately if it
does not arrive, but this is an implementation detail for the specific schedulers that are employed.

5.3.3 Splitting tasks on slave nodes

The initial application of the splitLinkedTasks rule results in a system with two nodes: one MasterNode and one SlaveNode. If we wish to further split the tasks on the SlaveNode, we can employ the splitSlaveTasks rule shown in Figure 5.6.

This rule is virtually identical to the splitLinkedTasks rule shown in Figure 5.5. The reason we must have both is so that there can only ever be one MasterNode (excepting backups, which are considered later, in Chapter 7). For example, we always start with a single-processor system, which is represented as one or more tasks running on aSingleNode. From here we can only apply the splitIndependentTasks and splitLinkedTasks rules — the latter of which results in a single MasterNode and removes the SingleNode (so that rule may not be applied twice). Further splitting of nodes must be carried out using the splitSlaveTasks rule, as this is the only transformation that begins with a SlaveNode, and that does not produce another MasterNode.

These three rules combine to allow the generation of source code supporting
a multi-processor system spanning any number of nodes, limited only by the number of tasks in the original single-processor prototype. Splitting a task across several nodes is not supported, as this greatly complicates the process (and the underlying goal is to keep the system as simple as possible, in order to enhance reliability).

5.4 Implementation

Now that we have rules specifying exactly what transformations may take place, we must consider how this information can be used to actually perform the conversion. First, we must consider where the conversion takes place.

5.4.1 Where do we convert?

As mentioned in Section 5.1.2 there are various challenges involved in refactoring C code, largely due to the inclusion of its macro preprocessor. To resolve these issues in a simple and reliable manner, we can perform the refactoring process on source-code that has already been preprocessed.

The simplest time to do this would be as a part of the build process, even though this is somewhat removed from the behaviour of typical refactoring tools.
that would expect to operate entirely independently of compilation. One downside to this approach is that the refactoring must be repeated with each build and any additional work must be carried out on the original prototype, instead of on the converted source-code.

### 5.4.2 Conversion process

As mentioned in Section 5.1.1, the conversion process (or refactoring) discussed here is not applied to tasks (which remain entirely unaltered), but is instead carried out on the scheduler. The process involves extracting the source-code for tasks that must run on a given node, adding a (multi-processor) scheduler and then generating new source-code to map the tasks onto the scheduler. By leaving tasks entirely unaltered, we are able to meet the third research goal presented in Section 1.4.

For reliability, we want to generate as little actual source-code as possible, so that the majority of the code can be tested and verified in isolation from the system as a whole — something that is not possible with generated source-code, as it may change during the build process. This can be achieved by using a very generic scheduler, one that is written in static code (i.e. not generated at all) but that calls generated functions in order to map its task’s data onto its required communications.

For example, a distributed scheduler may communicate data across a shared Controller Area Network (CAN) bus. Such a scheduler may send one (or more) packets of data during each tick interval, in order to keep all nodes on the network synchronised. In this case, the scheduler may be written to simply send this data directly from a buffer (and similarly any received data can be placed in a separate buffer). When data must be sent, the scheduler simply calls an external (generated) function which will fill the data buffer with the values that the given tasks should output.

Thus, the generated code simply maps from the global variables used by tasks to the generic data buffers employed by a given scheduler, as shown in Figure 5.7. This not only enables schedulers to be tested and verified in isolation, but also allows us to use any scheduler that provides the access to the same source-code hooks.
// The original task variable
int some_value;

// Generated data type
typedef int data_some_value_t;

// Generated data union
typedef union
{
    data_some_value_t data;
    uint8_t bytes[sizeof(data_some_value_t)];
} transfer_some_value_t;

// The buffer to be used by the scheduler for output
uint8_t Generated_output_buffer[sizeof(data_some_value_t)];

// Called by the scheduler before sending the data buffer
void Generated_Fill_Output_Buffer(void)
{
    int i = 0, j = 0;

    // This section is generated for each variable
    transfer_some_value_t output1;
    output1.data = some_value;
    for (i = 0; i < sizeof(output1.data); i++)
    {
        Generated_output_buffer[j++] = output1.bytes[i];
    }
}

Figure 5.7: Generated code mapping from task data to the scheduler
Figure 5.7 shows some of the generated interface code for the example presented in Section 5.3, specifically the code required to fill the output buffer. It is the responsibility of a conforming scheduler to call the output function and to then output the data from the provided array (using multiple packets, if required).

In order to easily map from task data to a scheduler’s buffer, we can employ a data union (Pont, 2001). This involves storing data in a union containing both the actual data type and a byte-based buffer of equivalent size. This approach is simple, but assumes that all nodes share the same byte-order. It also requires more memory than is strictly necessary, but as there is no dynamic allocation involved, the linker will produce an error if the available memory limit is exceeded.

5.4.3 Problems with pointers

A significant difficulty in analysing source-code to find global variable accesses, especially in C, is how we handle the use of pointers. Figure 5.8 illustrates the problem — it is relatively easy to detect the first read access to value, but access through pointers can be much more challenging.

```c
int value;

void Task(void)
{
    // Read access to value:
    int *ptr1 = &value;

    // Write access to value:
    *ptr1 = 20;

    int **ptr2 = &ptr1;
    // Write access to value:
    **ptr2 = 40;
}
```

Figure 5.8: Illustrating the difficulty of analysing code with pointers

Pointer analysis is an extremely complex topic with a great deal of research behind it — Hind (2001) offers a good summary of the issues and possible approaches involved. There are some assumptions that can be made, based on the nature of the source-code; specifically, we do not have to worry about analysing
pointers into stack or heap space, as the globals that we are interested in are always stored statically in locations assigned by the linker. Combined with the usage restrictions that are present in most C coding standards, a reasonably complete form of pointer analysis should be possible.

However, due to the sheer complexity involved, it has not been undertaken for the work presented in this thesis. To be suitable for commercial usage, the tool-supported process detailed in Chapter 9 would have to be expanded to include extensive pointer analysis. The lack of pointer analysis has not been an issue for the control systems that are used as case-studies in this thesis, as can be seen in Chapter 6.

5.5 Summary

This chapter has detailed a conversion process that we can use to convert from source-code targeting a single-processor prototype, to source-code for a number of different target architectures using a number of different schedulers. By separating the scheduler source-code (coupled with the distributed communication and synchronisation mechanisms) and the task source-code (coupled with the timing details), we can deal with each at a higher level than would otherwise be possible. Thus the given tasks can be seen as a form of source-based model, as discussed in Chapter 3.

This allows us to avoid the common modelling problems that result from a mismatch between high-level models and their implementations, as (in this case) both are in the same form. For example, if a problem is found during testing, it will always be clear and obvious which part of the model the problem is located in — the tasks, scheduler and their generated interface all map directly back to the model. Likewise, this separation still allows the verification tools discussed in Chapter 2 to be employed, as most of them will operate on source-code (before or after conversion). It also enables the reuse of structural tests that were originally written for the tasks and schedulers prior to conversion.

In the next chapter, this process is demonstrated through its application to a realistic case-study: a cruise-controller for a passenger car.
CHAPTER 6

Distributing a Cruise Controller

If everything seems under control, you’re just not going fast enough.
— MARIO ANDRETTI

This chapter discusses the application of the conversion process presented in Chapter 5 to the task of distributing a cruise controller for a passenger car.

6.1 Overview

The case study presented here is based on a hardware-in-the-loop simulation of the cruise-control system in a passenger car (Ayavoo et al., 2005). The Cruise Control System (CCS) consists of two basic parts (shown in Figure 6.1), the car model and the cruise controller itself. The car model outputs the current speed of the car as a Pulse Rate Modulated (PRM) signal and receives as input the new throttle setting from the cruise controller. Therefore the cruise controller will have to receive input as a PRM signal, convert it to a meaningful value, carry out some form of control algorithm to arrive at a desired throttle setting and finally output the throttle to the car model.

In choosing how to separate this system into tasks, it should be noted that the intent here is to test the conversion process, particularly that the behaviour (and to some extent timing) is preserved after refactoring. For this reason, the choice was made to partition the system into three separate tasks: computeSpeed, computeThrottle and outputThrottle. These tasks were initially designed to work with the TTC scheduled SingleNode, as required by the assumptions discussed
6.2 Multi-processor architectures

If a single processor is executing entirely independent tasks, then we could simply move the tasks onto separate processors and provide each with a TTC scheduler of its own (as discussed in Section 5.3.1). While this process sounds straightforward, tasks often have communication or synchronisation requirements and to meet these we must employ more advanced scheduling methods. For the purposes of this thesis, two different forms of shared-clock scheduler are considered (Ayavoo et al., 2007; Pont, 2001).

The shared-clock schedulers employed for this case-study both overcome the problem of keeping discrete processor nodes synchronised (and in communication) with a shared bus, specifically using the Controller Area Network (CAN) protocol. Both schedulers designate one node as a “master” that is triggered by a timer and then sends messages over the shared bus to trigger the “slave” nodes. Hence, the slave nodes are effectively sharing the master’s timer. Despite this similarity, each scheduler approaches the problem of inter-node communication differently.
6.2.1 TTC-SC1 scheduler

The TTC-SC1 scheduler employs a Time Division Multiple Access (TDMA) algorithm to trigger slave nodes separately (Ayavoo et al., 2007). This is shown in Figure 6.2.

![Figure 6.2: Message timing for the TTC-SC1 scheduler](image)

The use of a TDMA approach means that the master node must be triggered by a timer that is running with a tick interval at least $n$ times shorter than the TDMA round, where $n$ is the number of slave nodes in the system. This is particularly important when using the conversion process, in order to maintain the original task timing (as discussed in Chapter 5).

One of the main problems with this approach is that communication from one slave to another will require several ticks to accomplish, because each slave acknowledges separately. This may result in tasks on a slave node executing with older data, which may or may not be acceptable for the system in question, something that must be determined through testing.

6.2.2 TTC-SC3 scheduler

The TTC-SC3 scheduler (Ayavoo et al., 2007) works around some of the limitations of TTC-SC1 by requiring all slave nodes to share the same tick message and acknowledge within a single tick interval, as shown in Figure 6.3.

![Figure 6.3: Message timing for the TTC-SC3 scheduler](image)
This approach has the advantage of enabling slave nodes to synchronise their latest data at every tick, eliminating any chance of slaves executing with old data. This functionality comes at a cost however: the tick interval must now be long enough to accommodate the extra communication, which could itself be undesirable for some systems.

In addition, for every new message (other than the tick) that a node must have access to we must use another CAN message object (receive buffer). This is because nodes only access data on tick messages and we must prevent the prior messages from being overwritten. As CAN controllers typically provide a very limited number of message objects (up to 15), there are limits to the number of nodes that can be used with TTC-SC3.

### 6.2.3 Architectures

From the initial graph (shown in Figure 6.1), the graph transformation process detailed in Chapter 5 was employed to establish four multi-processor architectures. The splitLinkedTasks rule was used to produce a system operating on two nodes with the TTC-SC3 scheduler; this system executed the computeSpeed task on the Master node and the remaining two tasks executed on the Slave, as shown in Figure 6.4.

![Diagram](image)

**Figure 6.4:** The application of splitLinkedTasks to the CCS

Figure 6.4 shows the result of applying the splitLinkedTasks transformation. The information from this graph was then used, as discussed in Section 5.4.2.
to produce the interface layer. This was then combined with the relevant task source-code and the TTC-SC3 scheduler to produce two executables — one for each node of the two-node system.

Figure 6.5 shows how the three-node system was generated.

![Diagram showing the application of splitSlaveTasks](image)

**Figure 6.5: The application of splitSlaveTasks to the CCS**

In Figure 6.5, the **splitSlaveTasks** rule is employed to divide the slave node and provide the three node architecture. This was then combined with TTC-SC1 and then TTC-SC3 to produce two separate three-node systems for the sake of comparison.

### 6.3 Testing

The performance of each system was measured while issuing step changes in the cruise controller’s desired speed setting. This test was performed on the original single processor system, as well as the generated TTC-SC1 and TTC-SC3 architectures. Ideally, for the refactoring process to be successful the behaviour of the systems should be completely unaltered. In reality, the nature of the communications in a multi-processor system makes it impossible to achieve an identical result (due to slight timing differences from the network delays). We are therefore looking for the results to be as similar as possible in performance, preferably within the tolerances of the system: for the cruise controller case study, acceptable behaviour preservation is defined as being within two miles.
per hour of the single processor result, as a real driver would be unlikely to notice such a difference.

Figure 6.6 shows the changes in desired speed, as well as the actual measured speeds of the car model for each system under test. This data was taken from testing each system 10 times, with a single typical set shown in Figure 6.6 and averaged results shown in Table 6.1.

We can see that the performance of each of the multi-processor architectures closely matches that of the original single processor system, to the extent that they are nearly indistinguishable in the graph. In practical terms, this meets our informal definition of behaviour preservation for this system.

![Figure 6.6: The CCS system’s response to step changes in desired speed](image)

However, in order to examine the performance differences in greater detail, the Integral of the Absolute Error (IAE) values were calculated (Dorf and Bishop, 2004; Marti et al., 2002) across the range from 10 seconds to 80 seconds. These results (shown in Table 6.1) reflect the slight disparity between the performance of the various architecture choices.

Table 6.1 shows very little noticeable difference between the three architectures; it can be seen that the results are essentially identical, while the TTC-SC1
Table 6.1: IAE results for one, two and three node architectures

<table>
<thead>
<tr>
<th>Architecture</th>
<th>IAE Results (4 s.f.)</th>
<th>Std. Dev. (3 s.f.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 node TTC</td>
<td>389.8</td>
<td>0.659</td>
</tr>
<tr>
<td>2 node TTC-SC3</td>
<td>387.8</td>
<td>0.744</td>
</tr>
<tr>
<td>3 node TTC-SC3</td>
<td>388.7</td>
<td>0.512</td>
</tr>
<tr>
<td>3 node TTC-SC1</td>
<td>411.1</td>
<td>2.00</td>
</tr>
</tbody>
</table>

The scheduler shows a small degree of overhead. No attempt was made to adjust the controller parameters to compensate for these delays in this case-study.

In Table 6.1, the increased standard deviation also shows that there was greater spread of IAE values across the 10 samples than for the other architectures. It is difficult to pinpoint exactly why this is the case — one possible reason is the TDMA-based nature of the TTC-SC1 scheduler, which may alter the scheduler’s timing sufficiently to affect the performance of the control algorithm.

While the different designs vary in their effectiveness, each appears to operate correctly and within the specified limits defined for the single-processor system. Thus we can say that the conversion process has been successful — the fact that differences exist may even assist in making a choice between the available design choices.

### 6.4 Summary

This chapter discussed the implementation of a realistic and non-trivial case-study: a cruise-controller for a passenger car. Two of the rules presented in Chapter 5 (specifically the splitLinkedTasks and splitSlaveTasks rules) were employed to produce three different multi-processor architectures. Each of these designs was then tested under the same conditions a number of times and these results were compared in order to determine the effectiveness of the conversion process.

While the process was successful, none of the resulting systems could be considered as fault-tolerant. If this is a requirement, then we could benefit from a method of introducing redundancy. The next chapter examines what is involved in this and introduces further rules to incorporate backup nodes into the system.
CHAPTER 7

Code Generation Supporting Redundancy

“If a system does not do what it is supposed to do, then everything else about it matters little.”

— Bertrand Meyer

THIS CHAPTER discusses the additions that must be made to the conversion process presented in Chapter 5 to allow the generation of source code supporting redundancy.

7.1 Redundancy

There are many practical designs for incorporating redundancy in embedded systems (Hammett, 2002; Edwards et al., 2005). Consequently, there is no way to support redundancy without changing the existing conversion process. Fortunately, these changes can be kept relatively simple and so should not greatly impact the functionality of the system. It should be noted that, while the system presented here is very general and can be used in many situations, different redundancy methods might require further changes to the conversion process itself.

In order to be completely tolerant of any potential single point of failure, we must be able to support the generation of backups for every component in the system. There are essentially two ways that redundant nodes can operate in a system: either they are permanently active (on “hot standby”), or they must be activated by a master node (referred to here as master-driven redundancy).
7.1.1 Hot standby

In a system employing hot standby redundancy [Isermann et al., 2002], all the backup nodes are active at all times. This has one significant benefit: any output from a redundant node will also be duplicated and, with an odd number of identical nodes (at least three), we can employ voting in order to determine the validity of the outputs.

In other words, with the data-flow shown in Figure 7.1, Node2 should be receiving three effectively identical inputs; if one should be different from the others, it is likely to be an erroneous value and may be discarded (or otherwise dealt with).

![Voting in a system employing hot standby redundancy](image)

In this way hot standby redundancy provides additional safety — not only does it help to ensure that values are received, but also helps to assess their validity. Unfortunately, this comes at the cost of increased power consumption (hence the name), which may or may not be acceptable for any given system.

7.1.2 Master-driven redundancy

One alternative to hot standby is master-driven redundancy, where a backup node remains idle until it is specifically activated by the master node. This scheme utilises less power than hot standby, but it also relies on the master node being able to detect a failure.

Timing is also a significant factor with master-driven redundancy in a shared-clock distributed system (of the type described in Section 6.2). If the master node is sending and receiving acknowledgements on each timer tick, this will dictate the maximum length of time necessary to detect a fault and activate a
backup. For a TTC-SC1 scheduler (detailed in Section 6.2.1), which employs a TDMA approach, this worst-case could take the entire length of the TDMA round.

It is also important to note that both of these approaches only assist with hardware failures. Neither will help with the case where a failure is caused by an inherent software or architecture defect, because they rely on backup nodes operating in the same way as the original.

7.2 Extending the formalism

In order to deal with the added complexity of supporting redundancy, the basic formalism presented in Chapter 5 must be extended. Up until now, we have used UML composition to define the relationship between nodes and their tasks, indicating ownership (and that their lifetimes are linked). With redundancy this is no longer the case, so instead we can employ aggregation to show that the relationship is looser. Figure 7.2 shows the new type graph, updated to include aggregation.

Thus, when a node and its backup share tasks (as they always must), only one copy of each task appears in the graph — with aggregation relationships to any and all nodes that run it.
In Figure 7.2, we have three different types of node — any of which may be duplicated to create redundancy. Because it would take a number of graphs to represent every possible transformation, to save space and reduce complexity, an abstract type (Node) has been introduced using the standard UML generalisation relationship (i.e. inheritance). This will be used in the rules below to represent either a master or a slave node, as appropriate.

### 7.2.1 Duplicating independent nodes

Figure 7.3 shows the cloneSingleNode rule, which supports the introduction of a backupSingleNode into the system. Note that, as mentioned previously, the relationship between the nodes and the (non-empty) set of tasks t1 changes from composition to aggregation. This signifies that the lifetime of t1 is no longer restricted to that of n1 after the transformation, because of the existence of the backup node (b1).

![Figure 7.3: The cloneSingleNode rule](image)

The cloneSingleNode rule is presented here because it is the only exception to the following rule — only a SingleNode can be independent of any communication. However, as there is no communication in this case there is also nothing a scheduler can do to activate or otherwise make use of the backup — this must be dealt with using additional hardware.
7.2.2 Duplicating linked nodes

As mentioned previously, we must be able to support the creation of backups for every node type. Figure 7.4 shows the cloneOutputNode rule, used to support backup nodes of either MasterNode or SlaveNode types. It should be noted that the type of the backup node will always be the same as the type of the node that it is cloned from. Other than this restriction, any combination of master and slave node types is allowed.

Figure 7.4: The cloneOutputNode rule

In Figure 7.4 it is important to note that the communication (the data flow d1) is also duplicated. Figure 7.5 shows the duplication for nodes that only have input and no output, using the cloneInputNode rule.

The information from the graph resulting from these transformations may be used by the scheduler on the n2 nodes to implement hot standby redundancy, but that is an implementation detail — the information is provided to each node in the form of a list of all nodes and their backups, as well as a list of where each node's input data comes from; it is up to the scheduler to make use of this information to implement a specific redundancy scheme.

Finally, combining the two previous rules provides a rule for duplicating
Figure 7.5: The cloneInputNode rule

nodes with both input and output — the cloneFullNode is shown in Figure 7.6.

Note that, in the case of Figure 7.6, both the input and output data flows (d1 and d2) are duplicated. The additional outputs from the node n2 do not affect the generated source-code at all; communication is assumed to be over a shared bus, so the scheduler for the backup node b1 simply has to acquire the same data as the original node for this to work.

The additional outputs require more effort, as the scheduler may be employing hot standby redundancy. Fortunately, this is effectively the same as for the cloneOutputNode rule, so the same procedure will work for both situations.

7.2.3 Duplicating networks

The schedulers presented in Chapter 6 use a shared bus for all communication, including a “tick” message sent from the master node at the start of each time period. Duplicating any given node will increase the amount of communication and therefore the system’s reliance on the network bus. If this bus should fail, then the system as a whole will fail — so it too must be made redundant. It is quite possible for the schedulers discussed to be adapted to work with two or
more buses (Short and Pont, 2007).

However, because networks in this model are simply a hardware connection, support for them is entirely encapsulated within the schedulers employed. The only additional information required is the number of redundant buses in the system, which is not available in the graph-based representation used here.

### 7.3 Supporting backup nodes

For the purposes of this chapter we will look at the implementation of both forms of redundancy described in Section 7.1. The key to both systems is that the node that will be made redundant is simply duplicated, with each duplicate receiving the same input and (if all is functioning correctly) producing the same output.

Both the backup nodes and the original node perform exactly the same function without any awareness of each other. Therefore, in this redundancy architecture, generating code for the backup node is the relatively simple task of retrieving the set of functions for the specified task, duplicating it, and applying the same scheduler. If master-driven redundancy is to be employed, it is expected that the scheduler used for the master node will do the work of activating backups as required. For hot standby redundancy, the scheduler of any node receiving input from backups must handle the additional values appropriately.
(either by voting or some other mechanism). This can become quite complex, such as in the situation shown in Figure 7.7.

![Diagram of redundant nodes](image)

**Figure 7.7:** Illustrating the potential complexity of the problem

We cannot know during the program transformation phase exactly which redundancy mechanism is to be employed by the scheduler (if any). This means that the scheduler of each MasterNode must be provided with all the available information about the backups in the system, in case master-driven redundancy is employed. Similarly, the scheduler of any node that accepts input must be provided with details of any backups of the nodes that send it data, in case a form of hot standby redundancy is employed.

It should be noted that the case where the node to benefit from redundancy produces output (or receives input) that is external to the system is not considered in this thesis. This is because such situations are outside the control of the embedded system’s software, and thus outside the possible scope of the software-based process presented here.

### 7.4 Summary

This chapter has discussed some extensions to the formalisation presented in Chapter 5 that are designed to support several methods of improving fault-tolerance. Hot standby redundancy keeps backup nodes operating in parallel
with the original, continually producing additional data that can be compared in order to detect discrepancies that may be caused by a node failure. In contrast, master-driven redundancy reduces power requirements by keeping backup nodes idle until they are needed. Both approaches have advantages and either may be the most appropriate for any given application.

We can generate source-code supporting either fault-tolerance technique and so we can simply test them both to determine which is a better fit for the system. This helps to meet the second research goal presented in Section 1.4 and is demonstrated in the next chapter, where the cruise controller from Chapter 6 is made more fault-tolerant using the methods detailed here.
CHAPTER 8

Making a Fault-Tolerant Cruise Controller

Simplicity is prerequisite for reliability.
— EDGAR W. DIJKSTRA
“How do we tell truths that might hurt?”

This chapter combines the additional graph transformation rules presented in Chapter 7 to the cruise controller discussed in Chapter 6, in order to improve its fault-tolerance.

8.1 Overview

The same test bed from the study in Chapter 6 is employed in this chapter to perform several tests. These tests were intended to study the comparative performance and fault-tolerance of the different architectures employed, and the effectiveness of the conversion process in producing such different designs.

As such, the case-study will use two additional architectures. The first, a system with one backup node, employs master-driven redundancy. The second implements hot standby redundancy with voting from the original node and two additional backups.

8.2 Redundant architectures

Starting with the three node architecture from Chapter 6, we can use the clone-FullNode rule from Section 7.2.2 to produce a system with one backup for
Node2, as shown in Figure 8.1. This architecture is used with the TTC-SC1 scheduler, which supports master-driven redundancy.

Figure 8.1: The first application of cloneFullNode

Figure 8.2 shows one further application of the cloneFullNode rule, to produce a five node system with two backups for Node2. This architecture is used with the TTC-SC3 scheduler, which supports hot standby redundancy with voting.

Figure 8.2: The second application of cloneFullNode

In both cases Node3 must select which input it will use for its own processing.
CHAPTER 8. MAKING A FAULT-TOLERANT CRUISE CONTROLLER

In the TTC-SC1 scheduler, the node will always use the same input as it has used previously, provided it is available. If not, it will use the next available equivalent (usually from the backup node).

The TTC-SC3 scheduler receives all three inputs (from Node1, Backup1 and Backup2) and uses majority voting to decide which one to use. This behaviour is part of the given scheduler and thus may be altered or replaced without affecting any of the generated interface source-code.

8.3 Testing

Once again the performance of each system was measured while issuing step changes in the cruise controller's desired speed setting. The experiment carried out here is identical to the one detailed in Section 6.3, so that the results can be directly compared.

8.3.1 Performance testing

As a test of the relative performance impact of adding backups to the two scheduler types, the same experiment as presented in Section 6.3 is repeated here but with the new redundant architectures added. The results are shown in Figure 8.3.

Once again, there is little visible difference between any of the results. Neither of the redundancy approaches has had a significant impact on the performance of the system, in that neither approach causes a deviation of more than two miles per hour (fitting the informal specification from Section 6.3).

For comparison with the results from Chapter 6, the Integral of the Absolute Error (IAE) values must be calculated (Dorf and Bishop, 2004; Marti et al., 2002) across the same range — from 10 seconds to 80 seconds. These results are shown in Table 8.1.

Table 8.1 again shows very little noticeable difference between the three architectures, with or without redundancy. There is a small decrease in performance for the five node TTC-SC3 system, possibly due to the increased communication (all four slaves must acknowledge every tick message from the master).
Figure 8.3: The performance of various architectures, some with redundancy

Table 8.1: IAE results for various architectures, some with redundancy

<table>
<thead>
<tr>
<th>Architecture</th>
<th>IAE Results (4 s.f.)</th>
<th>Std. Dev. (3 s.f.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 node TTC</td>
<td>389.8</td>
<td>0.659</td>
</tr>
<tr>
<td>2 node TTC-SC3</td>
<td>387.8</td>
<td>0.744</td>
</tr>
<tr>
<td>3 node TTC-SC3</td>
<td>388.7</td>
<td>0.512</td>
</tr>
<tr>
<td>3 node TTC-SC1</td>
<td>411.1</td>
<td>2.00</td>
</tr>
<tr>
<td>4 node TTC-SC1</td>
<td>409.4</td>
<td>1.84</td>
</tr>
<tr>
<td>5 node TTC-SC3</td>
<td>409.2</td>
<td>0.694</td>
</tr>
</tbody>
</table>
With performance largely unaffected, the next test was to determine the effectiveness of the fault-tolerance mechanisms.

## 8.3.2 Failure testing

The failure of a specific node (Node2 in this case) was simulated by holding the microcontroller in reset such that it effectively produces a total failure of the node. This does not allow us to model a node that fails partially, unless such failure is caught by the CAN error detection and therefore does not supply a valid input. If the node were to instead become a *babbling idiot* ([Kopetz, 1997](#)) (or suffer some other failure undetectable by this system), the redundancy mechanism provided here would not be able to cope.

In this test, the CCS was set to maintain a constant speed of 60 mph and then a failure in the second node (the node benefiting from redundancy) was simulated. This test was duplicated on a three-node system without redundancy, as well as the four node TTC-SC1 architecture version with master-driven redundancy and the five node TTC-SC3 architecture employing hot-standby redundancy. The results are shown in Figure [8.4](#).

![Figure 8.4: The effect of a node failure in 3, 4 and 5 node systems](#)
Figure 8.4 shows that, as expected, the CCS without any redundancy fails once the node fails. Both of the generated redundancy schemes work, assuming the role of the failed node successfully. However, the CCS employing a hot standby redundancy scheme coped with the failure without any noticeable affect on the system, whereas the master-driven redundancy shows a small drop in speed before the backup node can take over.

This speed drop is over the two miles per hour limit defined in our informal specification, and as such may be noticeable by the driver. While it may be appropriate to inform the driver of any such failure, a sudden drop in speed could cause panic, which would be potentially dangerous.

Clearly then, the hot standby method is superior in performance (at least for this case-study). However, this performance improvement is associated with increased component costs (one additional node in this case) as well as higher system power consumption (from having redundant nodes active at all times).

In other words, the conversion process has been successful in producing several viable alternatives that may be used directly or further tested as part of the ongoing development process.

8.4 Summary

This chapter discussed the application of the rules presented in Chapter 7 to improving the fault-tolerance of the cruise controller from the case-study in Chapter 6. Two new architectures were used: a four node system employing the TTC-SC1 scheduler, and a five node system employing the TTC-SC3 scheduler.

After testing the different redundancy schemes, it was found that the hot standby approach (using voting to select appropriate input values) offered superior fault-tolerance, but at higher cost and with increased power consumption. Source-code for all the nodes of both architectures were generated from the original single processor prototype, as in Chapter 6. The next chapter discusses the issues and challenges involved in producing the tool that is used to automate this process.
CHAPTER 9

Tool Support and Automation

Give me six hours to chop down a tree and I will spend the first four sharpening the axe.
— ABRAHAM LINCOLN

THIS CHAPTER expands on the work presented over the last four chapters by presenting a tool to automate some of the conversion process.

9.1 Overview

In Chapter 5 it was decided that the conversion tool should operate during the build process. Before considering the low-level details of this conversion, a decision must first be made on exactly where in the build process the conversion should take place. Figure 9.1 shows some of the typical steps involved in the build process for a single processor embedded system. The source-code conversion could be performed at any one of the four major steps, each with its own advantages and disadvantages. For example, performing the conversion at the compiler or linker stages would involve modifying those tools, incurring a significant penalty in both cost and portability.

Eliminating those choices leaves only the build tool and preprocessor as the two remaining viable options. The build tool controls the entire process and has access to all source-files (whereas the preprocessor is fed source files one at a time), which are required to analyse all the tasks in the prototype system we are converting. The preprocessor on the other hand, is already responsible for a degree of source-code conversion and thus has a mechanism for communicating changes in source-code to the compiler (in order for compiler errors to be
correctly reported): the “#line” directive.

Section 5.1.2 discussed the refactoring of C source-code, where the difficulty of altering source files that will almost certainly rely on the macro preprocessor can make transformations error-prone. For this reason an alternative is employed here — perform the conversion on translation units that have already been passed through the preprocessor. By also altering the build scripts we can group processed translation units together and pass them all to the conversion tool at the same time.\(^1\)

Thus the tool receives as input one translation unit for each C source file. In order to produce a working multi-processor system, it then outputs one translation unit for each task. Each of these can then be compiled individually to produce the executable code for each node, respectively. This allows us to use the conversion process with any compiler (and toolchain) that supports separate preprocessing and compilation, which helps to meet the first research goal presented in 1.4. The new process is shown in Figure 9.2.

This means that a typical build process consisting of one translation unit per source file is altered into a modified build process (shown in Figure 9.2) consisting of one translation unit per node. As a consequence of this design, the generated translation units must contain all the source-code required to run on each microcontroller, including the scheduler.

\(^1\)This may also be possible in an Integrated Development Environment (IDE) by changing the build settings, depending on the IDE in question.
9.2 Generic schedulers

Before the prototype system's translation units can be analysed for tasks and their communication, the conversion tool must know which scheduler is to be employed in the generated system. As discussed in Section 5.4, the scheduler consists of static source-code that is made generic enough to work with any selection of tasks. The interaction between tasks and schedulers is carried out by an interface layer, that is generated automatically by the conversion tool.

9.2.1 Matching schedulers and architectures

In order to be made generic and largely independent of tasks and architectures, a scheduler must be provided with an interface layer containing a certain amount of metadata about the structure of the system in question. For example, Figure 9.3 shows a simplified graph of the final cruise controller architecture from Section 8.2, with the tasks removed for clarity.

In order to provide the scheduler with useful information about the nodes, each must be provided with a numeric identifier. For the example in Figure 9.3, the backups have been renamed as nodes so that their numbers may serve as identifiers.

With five nodes in the system we will be generating five executables from five
task-based translation units. Each translation unit outputted from the conversion tool will contain source-code for the chosen scheduler (differing only between master and slave nodes), the tasks from the input prototype and the source-code required to interface them. In this specific system there are two backup nodes, so the schedulers for both Node1 and Node3 may require additional information about the backups to be dealt with (depending on which type of redundancy is employed).

As the conversion tool knows nothing about the specific scheduler being used, we must add this information to the interface layer for both nodes. Node1 is the MasterNode and thus also requires information about every node in the system (in order to send tick messages to the appropriate nodes). Some of this code is shown in Figure 9.4.

```c
// Number of slaves in the system, not including backups
const int Generated_num_slaves = 2;
const int Generated_backups_per_slave = 2;

// List of slaves in the system, not including backups
int Generated_slaves[2] = {2, 3};
// List of backups per slave, where 0 means none present
const int Generated_backups[2][2] = {{4, 5}, {0, 0}};
```

Figure 9.4: Metadata provided only to the MasterNode

Because of the broadcast nature of the shared-bus networks considered here, no special information is needed by nodes producing output — they simply send their data over the bus and rely on the receiving node to filter it appropriately. Thus, any node that receives input must know the identifiers of any node that
sends it data, along with the information about backups (for voting, if required). An example of this additional source-code is shown in Figure 9.5.

```c
// Tell the scheduler that it must handle input
const bool Generated_node_has_input = true;

// Number of inputs in the system, not including backups
const int Generated_num_inputs = 1;
const int Generated_backups_per_input = 2;

// List of inputting nodes the system, not including backups
int Generated_inputs[1] = {2};
// List of backups per input, where 0 means none present
const int Generated_input_backups[1][2] = {{4, 5}};
```

**Figure 9.5: Metadata provided only to nodes receiving input (Node3 here)**

A generic scheduler can use this metadata to, for example, initialise CAN filters and fill the input (and empty the output) buffers appropriately. After receiving input (and before sending output), the interface layer must synchronise the scheduler’s buffers with the task’s global variables.

### 9.2.2 Interfacing tasks and schedulers

As discussed in Section 5.2, the interface layer maps from a buffer to the global variables used by tasks and back again. For example, Figure 9.6 shows highly simplified source-code for the tasks running on Node2 and its backups (shown in Figure 9.3).

```c
// Global variables...
float Scaled_speed = 0.0; // Received from computeSpeed
int Throttle_setting = 0; // Sent to outputThrottle

// Destined to run on Node2 (as well as Nodes 4 and 5):
void computeThrottle(void)
{
    // Desired speed: 60 miles per hour in metres per second
    const float ref_speed = 60 / 2.23693629;
    const float error = ref_speed - Scaled_speed;
    Throttle_setting = PID_Control(error);
}
```

**Figure 9.6: Simplified task code for Node2 and backups**

Of particular note in the computeThrottle task function from Figure 9.6 is...
the last two lines — the first is a read access to the Scaled_speed variable and the second is a write access to the Throttle_setting variable. As these are global variables, this sets up the communication for the tasks and the nodes they will eventually run on, as detailed in Chapter 5. These global variables are transmitted and received by Node2 and its two backups, so the interface layer for all three nodes must contain data unions for both.

Figure 9.7 shows that the interface layer defines two large (but byte-indexed) buffers: one for all the combined input and one for all the combined output. It is up to the scheduler implementation to transmit the contents of the output buffer, and to fill the input buffer with all the received data in a given tick period.

```c
typedef float data_scaled_speed_t;
typedef union
{
    data_scaled_speed_t data;
    uint8_t bytes[sizeof(data_scaled_speed_t)];
} transfer_scaled_speed_t;

typedef int data_throttle_setting_t;
typedef union
{
    data_throttle_setting_t data;
    uint8_t bytes[sizeof(data_throttle_setting_t)];
} transfer_throttle_setting_t;

uint8_t Generated_input_buffer[sizeof(data_scaled_speed_t)];
uint8_t Generated_output_buffer[sizeof(data_throttle_setting_t)];
```

Figure 9.7: Data unions and buffers for the two global variables

These large buffers can significantly increase the memory usage of the application, effectively doubling the memory requirements of the global variables used for inter-task communication. However, there is a major advantage to this approach — it greatly simplifies the implementation of the generic scheduler, which now only needs to operate on two specific buffers. It should also be noted that spreading tasks across multiple nodes will reduce the memory required for each, which may help to offset this increase.

In Figure 9.7 the scheduler only has a small amount of data to send and receive — not enough to fill a complete CAN message. In this case, all the data may be sent and received in one go (directly from the buffer if possible). If there is too much data to fit in one message, the scheduler implementation must
arrange transmission and reception of multiple messages. Exactly how this is accomplished is considered an implementation detail for the scheduler — no specific method is required by the interface defined here.

While it is the scheduler's responsibility to map from the network data to the buffer, the interface layer is responsible for mapping from the buffer to the task's global variables. Figure 9.8 shows how the interface layer achieves this.

```c
void Generated_Fill_Output_Buffer (void)
{
    int i = 0, j = 0;

    // This section is repeated for every output:
    transfer_throttle_setting_t output1;
    output1.data = Throttle_setting;
    for (i = 0; i < sizeof(output1.data); i++)
    {
        Generated_output_buffer[j++] = output1.bytes[i];
    }
}

void Generated_Empty_Input_Buffer (void)
{
    int i = 0, j = 0;

    // This section is repeated for every input:
    transfer_scaled_speed_t input1;
    for (i = 0; i < sizeof(output1.data); i++)
    {
        input1.bytes[i] = Generated_input_buffer[j++];
    }
    Scaled_speed = input1.data;
}
```

Figure 9.8: Functions to map variables to buffers and back again

Figure 9.8 shows two functions, the first of which is intended to be called by the scheduler before outputting the contents of the buffer. It reads in the current value of each write-accessed global variable associated with each task on the node and then copies them, byte by byte, into the output buffer.

The second function is intended to be called after the scheduler fills the input buffer with newly received data. It simply performs the reverse operation — extracting values from the buffer and putting them into the appropriate globals.

These functions and declarations, along with those presented in the preceding section, make up the interface layer. This is all the code that will be gen-
erated by the conversion tool — everything else either comes from the input prototype, or the existing scheduler source-code.

Thus, to generate the interface layer the conversion tool must be able to find the names and types of all the global variables, as well as which tasks access them and what type of access is involved (read-only or read-write access). To do this reliably, the tool needs to be able to parse the C source-code.

9.3 Parsing

Embedded systems are developed for a wide variety of different processor and microcontroller architectures. These can differ greatly in terms of the features and peripherals available to the developer, and so many compilers offer additional source-code constructs to access such features. This means that source-code for embedded systems is rarely written entirely in standard C99 but, instead, often utilises extensions that may vary based on the specific platform and compiler suite employed by the developer.

As a consequence, the design of a C parser for the conversion process must be general enough to accept input in many different forms. It is also necessary for the parser to be highly reliable, which can be difficult to achieve in such circumstances. In order to address these problems, the parser detailed here has been built around the Spirit parser framework (de Guzman, 1998).

9.3.1 A dynamic parser with Spirit

The Spirit framework is a part of Boost (1998): a collection of peer-reviewed C++ libraries, which has been described as being “...one of the most highly regarded and expertly designed C++ library projects in the world” (Sutter and Alexandrescu, 2004). Spirit makes extensive use of template meta-programming techniques (such as expression templates) in order to allow the use of syntax closely resembling Extended Backus-Normal Form (EBNF) directly in standard C++. This allows the production of a working parser that is visually verifiable (to a limited extent) by deriving it directly from the formal C99 syntax contained in the ISO standard for the C programming language (as seen in Figure 9.9).

Figure 9.9 shows the actual (working and complete) C++ code for one Spirit
// primary-expression:
//    identifier
//    constant
//    string-literal
//    "(" expression ")"
primary_expression
    = no_node_d[ch_p(' ( ')]
            >> expression
            >> no_node_d[ch_p(' )')]
| identifier_g
| constant_g
| string_literal_g
;

Figure 9.9: C++ source-code for part of a C99 parser

rule — the comment above it is a direct quote from the syntax in the ISO C99 standard, showing just how close the two representations can be (the only differences are there to accommodate the backtracking, recursive-descent nature of the Spirit framework's parsers). This relatively simple code, when plugged into the Spirit framework, will become a parser capable of producing an Abstract Syntax Tree (AST) from standard C code.

However, as mentioned previously, the parser must be capable of handling a variety of non-standard extensions to the C99 standard. Fortunately, Spirit is also quite capable of producing such a dynamic parser. Each rule (such as the primary_expression and expression rules in Figure 9.9) is simply an ordinary C++ object (of a type derived from the framework). Consequently the grammar definition can be passed an additional object at construction that will recognise specific non-standard extensions.

// type-qualifier:
//    "const" "restrict" or "volatile"
type_qualifier
    = str_p("const")
         | str_p("restrict")
         | str_p("volatile")
         | impl.type_qualifiers
;

Figure 9.10: A rule allowing for non-standard extensions

Figure 9.10 shows the production rule for parsing standard C99 type qualifiers, and serves as an example of how the parser can support the inclusion of
dynamic, implementation-defined extensions. The impl.type_qualifiers rule allows the inclusion of non-standard type qualifiers, such as code or data that may be allowed by a given compiler, without having to hard-code them directly into the parser.

9.3.2 Generating the interface

From the parser the conversion tool acquires the AST of each preprocessed translation unit, which can then be analysed to find all the necessary data. The first step is to build a complete call graph for the application, including details of the functions that are called from every task.

Schedulers often invoke tasks through the use of function pointers, which makes it difficult to tell where a task is being called. This is another case that would require pointer analysis (discussed in Section 5.4.3), except that we can use our call graph to find calls to the SCH_Add_Task function. These calls can in turn be analysed to find the names, periods and offsets of every task from the parameters that are passed to them (as shown in Figure 9.11).

```c
// Tell the scheduler that the Task1 function is a task:
SCH_Add_Task(Task1, 10, 500); // Delay: 10, Period: 500
```

**Figure 9.11: The basic format of an SCH_Add_Task call**

The full set of functions and variables for each task is then isolated — including the task function, every function in its call graph, all global variables accessed by the task and every typedef and struct declaration. Each set is moved to a new translation unit, where the specified scheduler source-code is then added.

Finally, the ASTs are analysed once more and the communication between each task set is identified (based on read and write accesses to global variables, as discussed in Chapter 5) and the interface layers are generated by combining this information with the structure identified through the graph transformation rules to generate source-code, as detailed in Section 9.2.
9.4 Summary

This chapter has discussed how the conversion process detailed in Chapters 5 and 7 can be automated. The key here is that only a minimal amount of source-code is actually generated, because generated source-code is difficult to thoroughly test and verify. This “interface layer” was detailed in Section 9.2 along with the requirements that a scheduler must fulfil in order to be used with the conversion tool presented here.
CHAPTER 10

Conclusion and Future Work

Prediction is very difficult, especially about the future.

— Niels Bohr

This chapter summarises the work presented in the rest of this thesis, and provides some discussion and conclusions to examine how well the initial goals of the project have been met. A brief overview of potential future work is also provided.

10.1 Evaluation

The goal of the process described in this thesis is that it can be used to facilitate early and ongoing testing within the context of distributed, time-triggered embedded systems.

In Section 1.4, three additional goals and requirements were identified:

1. To improve the ease and efficiency of testing and development in general, the process should work with existing source-code and tools as much as possible.

2. To be useful in real software development where requirements and design decisions may change, the process should be adaptable to a range of different architectural choices.

3. To be suitable for use in the development of systems requiring certification, the process should avoid altering the existing source-code structure as much as possible (and thus minimise the impact on any existing structural tests).
10.1.1 Working with existing source-code and tools

Chapters 5 and 9 detailed how the conversion process works within the build process, but without requiring alterations to the compiler or toolchain in general. Essentially the translation units that are the output of the preprocessor are parsed and transformed, with the output being source-code that can then be compiled directly. This should be possible with any C compiler, provided only that it is capable of carrying out preprocessing separately from compilation. In this way the first research goal above — working with existing tools — was met.

In addition, Chapter 5 detailed the basic conversion process, with a focus on identifying inter-task communication in the form of shared memory. In the single processor prototype (used as the input to the system), this memory is local to the processor; when converted to a distributed architecture, the memory remains local but is monitored by a short section of automatically generated code that is called by the scheduler. The scheduler then simply communicates (synchronises) these ‘shared’ values over the network during the system tick.

Source-code for existing tasks that uses shared memory for communication may therefore be used directly with the conversion process. In many cases this source-code may be used unaltered, which can help meet the both the first and third research goals listed above.

10.1.2 Limiting source-code structural changes

Chapter 6 demonstrated that the conversion process can be used for the development of a real system, in this case a cruise controller for a passenger car. In the results from this case-study, the functional testing for the multi-processor system closely matched that of the single-processor prototype; with no source-code modifications required in the tasks, the structure of the task’s source-code remains entirely unaltered. This helps to meet the third goal above.

However, there are some limitations. While the case-study in Chapter 6 did not require any alteration, there are some cases where source-code will not work with the conversion process so easily. If the code employs a communication method other than shared memory, the parser will not be able to identify it; the resulting distributed system may not function correctly if the two communicating tasks are moved onto separate processors. In such cases, the tasks must
either remain on the same processor or must be altered to use shared memory communication.

In Section 5.4.3 another potential problem was highlighted — the conversion tool in its present state does cannot identify shared memory access through pointers. Thus, if two tasks are communicating through shared memory and access that shared memory through one or more pointers and are split off into two separate processors, then the resulting distributed system will not function correctly. As before, the tasks must either remain on the same processor or the code employing pointers must be rewritten.

Finally, the conversion tool cannot detect cases when a function pointer dereference is used in place of a normal function call. This detection of function calls is important in order to build up a complete call tree and to detect shared memory accesses in the called functions. If function pointers are employed then we can no longer guarantee that any communication will be detected and so they must be removed before using the conversion process as detailed here. It should be noted that this only applies to function pointers used from task functions, or any functions called by them; the scheduler itself and any isolated libraries — those that cannot be used for inter-task communication — will not require alteration.

10.1.3 Adapting to architectural changes

In Chapter 7 the conversion process was expanded to include the generation of source-code supporting redundancy. This was then demonstrated in Chapter 8 by generating code for a variety of architecture and scheduling options, meeting the second goal of being adaptable to a range of architectural choices.

Chapter 9 discussed the conversion process itself and how it can be (at least partially) automated. This is vital for easily adapting to new architectural choices, as it allows for an earlier assessment of suitability than might be possible with an entirely manual conversion process.

As discussed in Section 5.2 there are some assumptions made about the architectures that can be targeted. The scheduler for the target distributed system is assumed to work with periodic (time-triggered) tasks and be capable of communicating data during each ‘tick’ interval. Beyond this, the only hard requirement for a scheduler is that it should acquire data from tasks (and provide
it back) by calling the functions automatically generated by the conversion tool, as specified in Section 9.2.2.

10.2 Discussion and future work

In order to assess the effectiveness of the work presented here, we need to consider some of the claims made earlier in the thesis. These are discussed throughout this section.

10.2.1 Source-code as a textual model

In Chapter 3 it was argued that source-code, at a sufficiently high-level, can be seen as a form of textual model. Removing considerations such as the choice of scheduler, node distribution and redundancy support leaves the remaining source-code (the single-processor prototype) with only details of the behaviour and timing of the tasks in the system.

Throughout the research presented here, this approach has proved highly effective. The cruise controller case-study (first presented in Chapter 6) was written in under 100 lines of code, with the scheduler removed (and not counting comments or white-space) — this provided three tasks, enough for up to three nodes and almost unlimited backups\(^1\) with the approaches presented here.

One key advantage of using this higher-level source-code is that it can still be compiled and tested using a single-processor scheduler (as shown in Chapter 6). Unfortunately, it is not always possible for the multi-processor architecture to be functionally tested on a single processor. The slow speed of the microcontroller, combined with the relatively simple TTC scheduler that is assumed for the prototype, limit the range of systems that may be tested in this manner.

In theory, the source-code model can be made even higher-level by introducing a hardware abstraction library. This could allow the prototype system to be built and tested on a different architecture than the generated system (simply by swapping out the library); it is even possible for each node to run on a different type of microcontroller, though this is not without its own difficulties (differing clock speeds, communication protocols, and so on).

---

\(^1\) The limiting factor for the schedulers presented here would be the number of CAN message boxes and filtering rules available, but this depends on the hardware used and the scheduler implementation.
Consequently, it is possible for the prototype system to be built targeting a fast architecture (such as x86 desktop processors) and the generated system could still run on multiple slower microcontrollers. Unfortunately, constraints on time and resources prevented this area from being explored as part of this research, but it remains a viable topic for future work.

10.2.2 Conversion as design-space exploration

Chapter 3 also made the argument that the conversion process presented here can be seen as the first step in a form of design-space exploration. This is primarily backed up by the case-study in Chapter 8, where several different redundancy designs were combined with different schedulers in an attempt to find the most suitable design. The visible differences between the designs, shown in Section 8.3, would be of great assistance in making a choice between them.

While this is useful, to be properly described as a tool for design-space exploration we must be able to obtain feedback from the system and make design decisions automatically. As discussed in Section 3.4, it is impractical to attempt to automatically reconfigure physical hardware and so such feedback cannot be obtained with the current tool.

However, it is possible to obtain feedback from a simulation of the architecture under examination. Simulating entire systems accurately, including all hardware components (such as CAN controllers and their communication) may be difficult and slow, but it is not impossible. Another alternative is to replicate the actual hardware on a Field Programmable Gate Array (FPGA), which can be reconfigured to match each design. Unfortunately, to get accurate results we would have to be using a “soft” (FPGA-ready) version of the core from the actual microcontroller, which can be prohibitively expensive.

There is also the issue of deciding exactly what to measure and optimise — the performance testing from Chapter 8 could be carried out automatically, but we must also be able to determine which system is superior. Likewise, deciding exactly how to test redundancy may require additional information from the developer.

Finally, the process presented here requires user interaction to specify exactly which rules and scheduler is to be employed. To do more than just the initial phase of design-space exploration, the tool would need to be able to make some
of these decisions internally. There is scope for a considerable amount of further research in this area.

10.2.3 Conversion as refactoring

At the beginning of Chapter 5, the conversion process was introduced in the context of a type of program transformation known as refactoring. The key to refactoring is that it involves improving a system’s source-code internally, in such a way that the external behaviour is unaltered. Strictly speaking, the initial multi-processor conversion from Chapter 5 could be seen as fitting this description, although it is difficult to verify that the timing is not altered when the source-code is distributed.

However, the addition of redundancy support from Chapter 7 makes things more difficult, as we are deliberately altering the behaviour of the system in the presence of hardware failures (although not if everything is operating correctly). Also, when using refactoring the developer will usually continue working on the source-code after the transformation; with the process presented here, additional work is intended to be carried out on the prototype, as the conversion takes place as part of the build process.

10.3 Conclusion

The work described in this thesis has made three contributions. First, evidence has been provided demonstrating the effectiveness of a conversion process for generating source-code supporting a variety of multi-processor architectures and schedulers, from a single-processor prototype. Second, this conversion process was further extended to support the generation of source-code for several redundancy schemes, with evidence supporting their usefulness.

Finally, a tool was presented that automates the conversion process through program transformation. This tool, and the research on which it was based, enables us to develop a multi-processor system to a testable stage in much less time than may otherwise be possible, providing significant benefits in both cost and development time.
APPENDIX A

C99 Parser Source-code

I have always wished that my computer would be as easy to use as my telephone. My wish has come true. I no longer know how to use my telephone.

— Bjarne Stroustrup

This appendix presents source-code for the C99 parser rules, as discussed in Section 9.3. Only the syntax parser is included here, the lexical parsers and supporting code constructs have been removed in order to save space.

A.1 Source of the syntax parser

```
// primary-expression:
//   identifier
//   constant
//   string-literal
//   "(" expression ")"

primary_expression = no_node_d[ch_p( '(' )]
   >> expression
   >> no_node_d[ch_p( ')' )]
| identifier_g
| constant_g
| string_literal_g
;

// postfix-expression:
//   primary-expression
//   postfix-expression "[" expression "]"
//   postfix-expression "("
//   argument_expression_list<opt> ")"
//   postfix-expression "." identifier
//   postfix-expression "-" identifier
```
// postfix-expression "++"
// postfix-expression "--"
// "(" type-name ")" 
// 
// "(" type-name ")" "{" initializer-list "}"

postfix_expression = (  
    no_node_d[ch_p( '(' )]  
    >> type_name  
    >> no_node_d[ch_p( ' ' )]  
    >> no_node_d[ch_p( '{' )]  
    >> initializer_list  
    >> !no_node_d[ch_p( ',' )]  
    >> no_node_d[ch_p( ')' )]  
    | primary_expression  
    )  
    >> *(  
    ch_p( '[' )  
    >> expression  
    >> ch_p( ']' )  
    | ch_p( ')' )  
    >> !argument_expression_list  
    >> ch_p( ')' )  
    | root_node_d[ch_p( '.' )]  
    >> identifier_g  
    | root_node_d[str_p( "->" )]  
    >> identifier_g  
    | root_node_d[str_p( "++" )]  
    | root_node_d[str_p( "--" )]  
    )

// argument-expression-list:
// assignment-expression
// argument-expression-list "," assignment-expression
argument_expression_list = list_p(
    assignment_expression  
    , no_node_d[ch_p( ',' )]
)

// unary-expression:
// postfix-expression
// "++" unary-expression
// "--" unary-expression
// unary-operator cast-expression
// "sizeof" unary-expression
// "sizeof" "(" type-name ")"
// unary-operator:
// "&" "+" "*" "*" "-" "-" or "!

unary_expression
APPENDIX A. C99 PARSER SOURCE-CODE

= root_node_d[str_p("++")]  
  >> unary_expression
| root_node_d[str_p("--")]  
  >> unary_expression
| (  
    root_node_d[ch_p('&')]  
    | root_node_d[ch_p('*')]  
    | root_node_d[ch_p('+')]  
    | root_node_d[ch_p('-')]  
    | root_node_d[ch_p('~')]  
    | root_node_d[ch_p('!')]  
  )  
  >> cast_expression
| root_node_d[str_p("sizeof")]  
  >> (  
    no_node_d[ch_p('(')]  
    >> type_name  
    >> no_node_d[ch_p(')')]  
    | unary_expression  
  )  
  | postfix_expression  
  ;

// cast-expression:
// unary-expression
// "(" type-name ")" cast-expression
cast_expression
  = no_node_d[ch_p('(')]  
    >> type_name  
    >> no_node_d[ch_p(')')]  
    >> cast_expression  
    | unary_expression  
  ;

// multiplicative-expression:
// cast-expression
// multiplicative-expression "*" cast-expression
// multiplicative-expression "/" cast-expression
// multiplicative-expression "%" cast-expression
multiplicative_expression
  = cast_expression  
    >> *(  
      (  
        root_node_d[ch_p('*')]  
        | root_node_d[ch_p('/')]  
        | root_node_d[ch_p('%')]  
      )  
    )  
    >> cast_expression  
  )  
  ;

// additive-expression:
// multiplicative-expression
APPENDIX A. C99 PARSER SOURCE-CODE

```c
// additive-expression "+" multiplicative-expression
// additive-expression "-" multiplicative-expression
additive_expression
    = multiplicative_expression
    >> *( 
        ( 
            root_node_d[ ch_p( '+’ )]
        | root_node_d[ ch_p( '-’ )]
        )
    >> multiplicative_expression
    )
;

// shift-expression:
// additive-expression
// shift-expression "<<" additive-expression
// shift-expression ">>" additive-expression
shift_expression
    = additive_expression
    >> *( 
        ( 
            root_node_d[ str_p( "<<" )]
        | root_node_d[ str_p( ">>" )]
        )
    >> additive_expression
    )
;

// relational-expression:
// shift-expression
// relational-expression "<" shift-expression
// relational-expression ">" shift-expression
// relational-expression "<=" shift-expression
// relational-expression ">=" shift-expression
relational_expression
    = shift_expression
    >> *( 
        ( 
            root_node_d[ str_p( "<=" )]
        | root_node_d[ str_p( ">=" )]
        | root_node_d[ ch_p( '<' )]
        | root_node_d[ ch_p( '>' )]
        )
    >> shift_expression
    )
;

// equality-expression:
// relational-expression
// equality-expression "==" relational-expression
// equality-expression "!=" relational-expression
equality_expression
    = relational_expression
```
APPENDIX A. C99 PARSER SOURCE-CODE

```c
>> *( 
   (  
      root_node_d[str_p("==")]  
      | root_node_d[str_p("!")]  
   )  
   >> relational_expression  
)  
;

// AND-expression:  
// equality-expression  
// AND-expression "&" equality-expression  
AND_expression  
= equality_expression  
>> *(  
   root_node_d[ch_p('&')]  
   >> equality_expression  
)  
;

// exclusive-OR-expression:  
// AND-expression  
// exclusive-OR-expression "^" AND-expression  
exclusive_OR_expression  
= AND_expression  
>> *(  
   root_node_d[ch_p('^')]  
   >> AND_expression  
)  
;

// inclusive-OR-expression:  
// exclusive-OR-expression  
// inclusive-OR-expression "|" exclusive-OR-expression  
inclusive_OR_expression  
= exclusive_OR_expression  
>> *(  
   root_node_d[ch_p('|')]  
   >> exclusive_OR_expression  
)  
;

// logical-AND-expression:  
// inclusive-OR-expression  
// logical-AND-expression "&&" inclusive-OR-expression  
logical_AND_expression  
= inclusive_OR_expression  
>> *(  
   root_node_d[str_p("&&")]  
   >> inclusive_OR_expression  
)  
;
```
APPENDIX A. C99 PARSER SOURCE-CODE

// logical-OR-expression:
// logical-AND-expression
// logical-OR-expression "||" logical-AND-expression
logical_OR_expression
  = logical_AND_expression
     >> *(root_node_d[ str_p( "||" )]
       >> logical_AND_expression)
;

// conditional-expression:
// logical-OR-expression
// logical-OR-expression "?" expression ":"
// conditional-expression
conditional_expression
  = logical_OR_expression
     >> !(root_node_d[ch_p( '?')]
       >> expression
       >> no_node_d[ch_p( ':')]
       >> conditional_expression)
;

// assignment-expression:
// conditional-expression
// unary-expression assignment-operator
// assignment-expression
//
// assignment-operator:
// "==" "*=" "/=" "%=" "+=" "-="
// "<<=" ">>=" "&=" "^=" or "|="
assignment_expression
  = unary_expression
     >> (root_node_d[ch_p( '=' )]
       | root_node_d[str_p( "*=" )]
       | root_node_d[str_p( "/=" )]
       | root_node_d[str_p( "%=" )]
       | root_node_d[str_p( "+=" )]
       | root_node_d[str_p( "-=" )]
       | root_node_d[str_p( "<<=" )]
       | root_node_d[str_p( ">>=" )]
       | root_node_d[str_p( "&=" )]
       | root_node_d[str_p( "^=" )]
       | root_node_d[str_p( "|=" )]
     )
     >> assignment_expression
     | conditional_expression
     ;

// expression:
// assignment-expression
// expression "," assignment-expression
expression
    = assignment_expression
        >> *(root_node_d[ch_p( ', ')]
            >> assignment_expression
        )
;
// constant-expression:
// conditional-expression
constant_expression
    = conditional_expression
    ;

// declaration:
// declaration-specifiers init-declarator-list <opt> ";"
declaration
    = declaration_specifiers
        >> !init_declarator_list
            >> no_node_d[ch_p( '; ')] [typedefs.end()]
        ;

// declaration-specifiers:
// storage-class-specifier declaration-specifiers <opt>
// type-specifier declaration-specifiers <opt>
// type-qualifier declaration-specifiers <opt>
// function-specifier declaration-specifiers <opt>
declaration_specifiers
    = +(storage_classSpecifier
        | typeSpecifier
        | typeQualifier
        | functionSpecifier
    )
;
// init-declarator-list:
// init-declarator
// init-declarator-list , init-declarator
init_declarator_list
    = list_p
        (init_declarator
            , no_node_d[ch_p( ', ')]
        )
;
// init-declarator:
// declarator
// declarator "=" initializer
init_declarator
= declarator
    >> !(
        (root_node_d[ch_p('=')]
         | root_node_d[impl.init_decl_separators])
    )
    >> initializer

// storage-class-specifier:
//     "typedef" "extern" "static" "auto" or "register"
storage_class_specifier
    = str_p("typedef") [typedefs.start()]
      | str_p("extern")
      | str_p("static")
      | str_p("auto")
      | str_p("register")
      | impl.storage_class_specifiers
;

// type-specifier:
//     "void" "char" "short" "int" "long" "float" "double"
//     "signed" "unsigned" "_Bool" "_Complex" or "_Imaginary"
// struct-or-union-specifier
// enum-specifier
// typedef-name
type_specifier
    = str_p("void")
      | str_p("char")
      | str_p("short")
      | str_p("int")
      | str_p("long")
      | str_p("float")
      | str_p("double")
      | str_p("signed")
      | str_p("unsigned")
      | str_p("_Bool")
      | str_p("_Complex")
      | str_p("_Imaginary")
      | struct_or_union_specifier
      | enum_specifier
      | typedef_name
      | impl.type_specifiers
;

// struct-or-union-specifier:
//     struct-or-union identifier<opt> "{"
//     struct-declaration-list "}"
// struct-or-union identifier
struct_or_union_specifier
    = root_node_d[struct_or_union]
    >> (}
identifier_g
|| no_node_d[ch_p('}')]
>> struct_declaration_list
>> no_node_d[ch_p('}')]}
)
;

// struct-or-union:
// "struct" or "union"
struct_or_union
= str_p("struct")
| str_p("union")
;

// struct-declaration-list:
// struct-declaration
// struct-declaration-list struct-declaration
struct_declaration_list
= +struct_declaration
;

// struct-declaration:
// specifier-qualifier-list struct-declarator-list ";"
struct_declaration
= specifier_qualifier_list
>> struct_declarator_list
>> no_node_d[ch_p(';')]
;

// specifier-qualifier-list:
// type-specifier specifier-qualifier-list <opt>
// type-qualifier specifier-qualifier-list <opt>
specifier_qualifier_list
= +(type_specifier
 | type_qualifier
)
;

// struct-declarator-list:
// struct-declarator
// struct-declarator-list , struct-declarator
struct_declarator_list
= list_p
( struct_declarator
 , no_node_d[ch_p(',')] )
;

// struct-declarator:
// declarator
// declarator <opt> ":" constant-expression
struct_declarator
    = declarator
    ||
    ( root_node_d[ch_p('::')]
        >> constant_expression
    )
;
// enum-specifier:
// "enum" identifier <opt> "{ enumarator-list "}"
// "enum" identifier <opt> "{ enumarator-list ",", "}"
// "enum" identifier
enum_specifier
    = root_node_d[str_p("enum")]
    >>
    ( identifier_g
        ||
        no_node_d[ch_p('{')]
        >>
        enumerator_list
        >>
        !no_node_d[ch_p(',')]
        >>
        no_node_d[ch_p('}')]
    )
;
// enumerator-list:
// enumerator
// enumerator-list , enumerator
enumerator_list
    = list_p
    ( enumerator
        ,
        no_node_d[ch_p(',')] )
;
// enumerator:
// enumeration-constant
// enumeration-constant ":= constant-expression
enumerator
    = identifier_g
    >>
    !( root_node_d[ch_p('=')]
        >>
        constant_expression
    )
;
// type-qualifier:
// "const" "restrict" or "volatile"
type_qualifier
    = str_p("const")
    | str_p("restrict")
    | str_p("volatile")
    | impl.type_qualifiers
;
// function-specifier:
// "inline"
function_specifier
    = str_p("inline")
      | impl.function_specifiers

// declarator:
// pointer<opt> direct-declarator
declarator
    = !pointer
        >> direct_declarator

// direct-declarator:
// identifier
// "(" declarator ")"
// direct-declarator "[" type-qualifier-list<opt>
//     assignment-expression<opt>
// "]"
// direct-declarator "[" static type-qualifier-list<opt>
//     assignment-expression "]"
// direct-declarator "[" type-qualifier-list static"
//     assignment-expression "]"
// direct-declarator "[" type-qualifier-list<opt> "," "]"
// direct-declarator "(" parameter-type-list ")"
// direct-declarator "(" identifier-list<opt> ")"
direct_declarator
    =
        identifier_g [typedefs.set_name()]
        | no_node_d[ch_p('(')]
            >> declarator
            >> no_node_d[ch_p(')')] )

        >> *( 
            ch_p('[')
            >> !type_qualifier_list
            >> !assignment_expression
            >> ch_p(']')
            | ch_p('[')
            >> str_p("static")
            >> !type_qualifier_list
            >> !assignment_expression
            >> no_node_d[ch_p(']')] )
            | ch_p('[')
            >> type_qualifier_list
            >> str_p("static")
            >> !assignment_expression
            >> ch_p(']')
            | ch_p('[')
            >> !type_qualifier_list
            >> ch_p('*')
            >> ch_p(']')

91
| ch_p( '(' )
>> ( parameter_type_list
| !identifier_list
) >> ch_p( ')' ) >> !impl.post_function_declaration
)
;

// pointer:
// "*" type-qualifier-list <opt>
// "*" type-qualifier-list <opt> pointer
pointer
= +( !impl.type_qualifiers
>> root_node_d[ch_p( '*' )]
>> !type_qualifier_list
)
;

// type-qualifier-list:
// type-qualifier
// type-qualifier-list type-qualifier
type_qualifier_list
= +type_qualifier
;

// parameter-type-list:
// parameter-list
// parameter-list ":", ":...
parameter_type_list
= parameter_list
>> !( no_node_d[ch_p( ',",,,')]
>> str_p( "..." )
)
;

// parameter-list:
// parameter-declaration
// parameter-list ":", parameter-declaration
parameter_list
= list_p
( parameter_declarator
, no_node_d[ch_p( ',"')]
)
;

// parameter-declaration:
// declaration-specifiers declarator
// declaration-specifiers abstract-declarator<opt>
parameter_declaration = declaration_specifiers
    >> ( declarator
        | !abstract_declarator
    )
;
// identifier-list:
// identifier
// identifier-list, identifier
identifier_list = list_p
    ( identifier_g
        , no_node_d[ch_p( ',' )]
    )
;
// type-name:
// specifier-qualifier-list abstract-declarator<opt>
type_name = specifier_qualifier_list
    >> !abstract_declarator
;
// abstract-declarator:
// pointer
// pointer<opt> direct-abstract-declarator
abstract_declarator = pointer
    | direct_abstract_declarator
;
// direct-abstract-declarator:
// "(" abstract-declarator ")"
// direct-abstract-declarator<opt> "["
// assignment-expression<opt> "]"
// direct-abstract-declarator<opt> "[" "*" "]"
// direct-abstract-declarator<opt> "(" 
// parameter-type-list<opt> ")"
direct_abstract_declarator = no_node_d[ch_p( '(' )]
    >> abstract_declarator
    >> no_node_d[ch_p( ')' )]
    |+( ch_p( '[' )
        >> !assignment_expression
        >> ch_p( ']' )
    | ch_p( '['
        >> ch_p( '*' )
        >> ch_p( ']' )
    | ch_p( '(' )
```c
#define parameter_type_list

ch_p(')')

);

// typedef-name:
// identifier
typedef_name
    = typedefs.names
    ;

// initializer:
// assignment-expression
// 
// 

initializer
    = assignment_expression
    | no_node_d[ch_p('}')]
    >> initializer_list
    >> !no_node_d[ch_p('',')]]
    >> no_node_d[ch_p('}',')]]
    ;

// initializer-list:
// designation<opt> initializer
// initializer-list "," designation<opt> initializer
initializer_list
    = list_p
    {
        ( designation
            >> initializer
        , no_node_d[ch_p('','')]
        )
    };

// designation:
// designator-list "="
designation
    = designator_list
    >> root_node_d[ch_p('','=')]
    ;

// designator-list:
// designator
// designator-list designator
designator_list
    = +designator
    ;

// designator:
// 
```
// "identifier"
designator
    = ch_p( ']' )
        >> constant_expression
        >> ch_p( '[' )
    | root_node_d[ch_p( '.')]  // 
        >> identifier_g
;

// statement:
// labeled-statement
// compound-statement
// expression-statement
// selection-statement
// iteration-statement
// jump-statement
statement
    = compound_statement
        | selection_statement
        | iteration_statement
        | jump_statement
        | labeled_statement
        | expression_statement
    ;

// labeled-statement:
// identifier ":" statement
// "case" constant-expression ":" statement
// "default" ":" statement
labeled_statement
    = identifier_g
        >> no_node_d[ch_p( ':' )]
        >> statement
    | root_node_d[str_p( "case" )]
        >> constant_expression
        >> no_node_d[ch_p( ':' )]
        >> statement
    | root_node_d[str_p( "default" )]
        >> no_node_d[ch_p( ':' )]
        >> statement
    ;

// compound-statement:
// "{" block-item-list <opt> "}" compound_statement
    = root_node_d[ch_p( '{' )]
        >> !block_item_list
        >> no_node_d[ch_p( '}' )]
    ;

// block-item-list:
// block-item
// block-item-list block-item
block_item_list
    = +block_item
    ;

    // block-item:
    //    declaration
    //    statement
    block_item
        = statement
            | declaration
                ;

    // expression-statement:
    //    expression<opt> ";"
    expression_statement
        = !expression
            >> no_node_d[ch_p( ',')] ;

    // selection-statement:
    //    "if" "(" expression ")" statement
    //    "if" "(" expression ")" statement "else" statement
    //    "switch" "(" expression ")" statement
    selection_statement
        = root_node_d[str_p( "if" )]
            >> no_node_d[ch_p( '(' )]
            >> expression
            >> no_node_d[ch_p( ')' )]
            >> statement
            >> !(
                no_node_d[str_p( "else" )]
                    >> statement
            )
            | root_node_d[str_p( "switch" )]
                >> no_node_d[ch_p( '(' )]
                >> expression
                >> no_node_d[ch_p( ')' )]
                >> statement
                ;

    // iteration-statement:
    //    "while" "(" expression ")" statement
    //    "do" statement "while" "(" expression ")" ";"
    //    "for" "(" expression<opt> ";" expression<opt> ";"
    //    expression<opt> ")" statement
    //    "for" "(" declaration expression<opt> ";"
    //    expression<opt> ")" statement
    iteration_statement
        = root_node_d[str_p( "while" )]
            >> no_node_d[ch_p( '(' )]
            >> expression
            >> no_node_d[ch_p( ')' )]
            >> statement
A
APPENDIX A. C99 PARSER SOURCE-CODE

| root_node_d[ str_p( "do" )]
|-> statement
|-> no_node_d[ str_p( "while" )]
|-> no_node_d[ ch_p( '(' )]
|-> expression
|-> no_node_d[ ch_p( ')' )]
|-> no_node_d[ ch_p( ';' )]
| root_node_d[ str_p( "for" )]
|-> no_node_d[ ch_p( '(' )]
|-> !expression
|-> no_node_d[ ch_p( ';' )]
|-> !expression
|-> no_node_d[ ch_p( ';' )]
|-> !expression
|-> no_node_d[ ch_p( ')' )]
|-> statement

| root_node_d[ str_p( "for" )]
|-> no_node_d[ ch_p( '(' )]
|-> declaration
|-> !expression
|-> no_node_d[ ch_p( ';' )]
|-> !expression
|-> no_node_d[ ch_p( ')' )]
|-> statement

;

// jump-statement:
// "goto" identifier ";"
// "continue" ";"
// "break" ";"
// "return" expression<opt> ";"
jump_statement
  = root_node_d[ str_p( "goto" )]
  |-> identifier_g
  |-> no_node_d[ ch_p( ';' )]
  | str_p( "continue" )
  |-> no_node_d[ ch_p( ';' )]
  | str_p( "break" )
  |-> no_node_d[ ch_p( ';' )]
  | root_node_d[ str_p( "return" )]
  |-> !expression
  |-> no_node_d[ ch_p( ';' )]

;

// translation-unit:
// external-declaration
// translation-unit external-declaration
translation_unit
  = +external_declaration

;

// external-declaration:
// function-definition

197
APPENDIX A. C99 PARSER SOURCE-CODE

```c
// declaration
external_declaration
   = declaration
   | function_definition
;

// function-definition:
// declaration-specifiers declarator
// declaration-list <opt> compound-statement
function_definition
   = declaration_specifiers
     >> declarator
     >> !impl.post_function_definition
     >> !declaration_list
     >> compound_statement
;

// declaration-list:
// declaration
// declaration-list declaration
declaration_list
   = +declaration
;
```
Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.

— Antoine de Saint-Exupery

This appendix presents full source-code for the implementation specific parsers for the 8051 target and the Keil compiler. This represents all the necessary code to support the given target, and all that would have to be altered should a new target be required.

B.1 Source of the implementation specific parser

```cpp
#ifndef KEIL_8051_INCLUDED
#define KEIL_8051_INCLUDED

#include "c_lexical_grammar.hpp"
#include <boost/spirit/core.hpp>
#include <boost/spirit/symbols.hpp>

namespace c_parser {

using namespace boost::spirit;

struct post_function_grammar :
    public grammar< post_function_grammar > {

    template< class ScannerT >
    struct definition {

        definition( const post_function_grammar& )
        {
            post_function
                = +(
```
APPENDIX B. TARGET SPECIFIC PARSER SOURCE-CODE

```cpp
str_p( "small" )
| str_p( "compact" )
| str_p( "large" )
| str_p( "reentrant" )
| (  
  | str_p( "interrupt" )
  | str_p( "using" )
  | str_p( "_task_" )
  | str_p( "_priority_" )
)
>> int_const_g

BOOST_SPIRIT_DEBUG_TRACE_NODE( int_const_g, false );
BOOST_SPIRIT_DEBUG_NODE( post_function );
}

const rule< ScannerT, parser_tag< post_function_definition_id > >&
start() const
{
  return post_function;
}

int_const_grammar int_const_g;

rule< ScannerT, parser_tag< post_function_definition_id > >
  post_function;
};

struct keil_8051
{
  keil_8051()
  {
  type_specifiers.add( "bit" );
  type_specifiers.add( "sbit" );
  type_specifiers.add( "sfr" );
  type_specifiers.add( "sfr16" );

  type_qualifiers.add( "code" );
  type_qualifiers.add( "data" );
  type_qualifiers.add( "idata" );
  type_qualifiers.add( "bdata" );
  type_qualifiers.add( "xdata" );
  type_qualifiers.add( "far" );
  type_qualifiers.add( "idata" );

  function_specifiers.add( "alien" );

  init_decl_separators.add( "_at_" );

  100
```
BOOST_SPIRIT_DEBUG_TRACE_NODE( post_function_definition, false );
BOOST_SPIRIT_DEBUG_TRACE_NODE( post_function_declaration, false );

} symbols<> storage_class_specifiers;
symbols<> type_specifiers;
symbols<> type_qualifiers;
symbols<> function_specifiers;
symbols<> init_decl_separators;
post_function_grammar post_function_definition;
post_function_grammar post_function_declaration;
};
}

#endif // !KEIL_8051_INCLUDED
APPENDIX C

Tool Support Source-code

The designer of a new kind of system must participate fully in the implementation.

— DONALD E. KNUTH

This appendix presents source-code for the tool that automates the process of converting from single to multiple processors. Some parts have been removed in order to save space.

C.1 Source of the conversion tool

```cpp
#include "c_parser.hpp"

#include <map>
#include <vector>
#include <stdexcept>
#include <iostream>
#include <sstream>
#include <fstream>
#include <algorithm>
#include <string>
#include <boost/shared_ptr.hpp>

using namespace boost;
using namespace boost::spirit;
using namespace c_parser;
using namespace std;

struct variable
{
    variable(string str, iter_t st, iter_t en, bool ext)
        : name(str),
          start(st),
          end(en),
```
APPENDIX C. TOOL SUPPORT SOURCE-CODE

```cpp
external( ext )
{
}

variable() {} 

string name;
string type;
iter_t start;
iter_t end;
bool external;

struct function
{
  function( string str, iter_t st, iter_t en, iter_t bs )
    : name( str ),
    start( st ),
    end( en ),
    block_start( bs )
  {
  }
}

function() {}

string name;
iter_t start;
iter_t end;
iter_t block_start;

vector< string > function_calls;
map< string, bool > global_access;

struct task_io
{
  string var_name;
  int task_num;
};

struct task
{
  string function_name;
  int initial_delay;
  int period;

  int position;

  map< string, bool > global_access;
  vector< task_io > input;
  vector< task_io > output;

  vector< int > backup_tasks;
};
```
struct translation_unit
{
    vector< string > declarations;
    map< string, variable > globals;
    map< string, function > functions;
};

template< class T >
string node_string( T& node )
{
    return string( node.value.begin(), node.value.end() );
}

template< class T >
variable create_variable( T& identifier, iter_t start, iter_t end, bool ext )
{
    return variable( node_string( identifier ), start, end, ext );
}

void remove_from_string( string& str, const string& rmv )
{
    int start = str.find( rmv );
    
    while( start < str.length() )
    {
        str.erase( start, rmv.length() );
        start = str.find( rmv );
    }
}

template< class T >
bool build_variable( T& node, translation_unit& tu )
{
    static iter_t start, end;
    static bool external = false;

    if ( node.value.id() == declaration_id )
    {
        start = node.value.begin();
        end = node.value.end();
    }

    if ( node.value.id() == identifier_id )
    {
        string name = node_string( node );
        if ( tu.globals.find( name ) != tu.globals.end() )
        {
            if ( external )
                return false;
        }
tu.globals[name] = create_variable( node, start, end, external );
tu.globals[name].type = string( start, end );
remove_from_string( tu.globals[name].type, name );
remove_from_string( tu.globals[name].type, "static" );
remove_from_string( tu.globals[name].type, "extern" );
remove_from_string( tu.globals[name].type, ";" );

return true;
}

if ( ( node.value.id() == storage_class_specifier_id ) &&
( node_string( node ) == "extern" ) )
external = true;

// Typedefs can never be variable declarations.
if ( ( node.value.id() == storage_class_specifier_id ) &&
( node_string( node ) != "typedef" ) )
return true;

if ( ( node.value.id() == type_specifier_id ) ||
( node.value.id() == struct_or_union_specifier_id ) ||
( node.value.id() == struct_or_union_id ) ||
( node.value.id() == type_qualifier_id ) ||
( node.value.id() == pointer_id ) ||
( node.value.id() == typedef_name_id ) ||
( node_string( node ) == "[" ) ||
( node_string( node ) == "]" ) )
return true;

if ( node.value.id() == declaration_id )
external = false;

if ( ( node.value.id() == declaration_id ) ||
( node.value.id() == init_declarator_list_id ) ||
( node.value.id() == declaration_specifiers_id ) ||
( node.value.id() == declarator_id ) ||
( node.value.id() == abstract_declarator_id ) )
{
for( int i = 0; i < node.children.size(); ++i )
{
if ( !build_variable( node.children[i], tu ) )
return false;
}
return true;
}

// We don't need the whole initialisation expression, just the id.
if ( node.value.id() == init_declarator_id )
return build_variable( node.children[0], tu );

return false;
APPENDIX C. TOOL SUPPORT SOURCE-CODE

```cpp
template< class T >
void add_function_call( T& call, function& func,
                        vector< task >& tasks )
{
    string name = node_string( call.children[0] );

    // Deal with "SCH_Add_Task" calls without adding the function.
    if ( name == "SCH_Add_Task" )
    {
        if ( ( call.children.size() != 4 ) &&
             ( call.children[2].value.id() !=
               argument_expression_list_id ) )
        {
            throw invalid_argument( "Invalid SCH_Add_Task call." );
        }

        T& params = call.children[2];
        task tsk;
        tsk.function_name = node_string( params.children[0] );

        istream str1( node_string( params.children[1] ) );
        if ( ! ( str1 >> tsk.initial_delay ) )
            throw invalid_argument( "Invalid SCH_Add_Task call." );

        istream str2( node_string( params.children[2] ) );
        if ( ! ( str2 >> tsk.period ) )
            throw invalid_argument( "Invalid SCH_Add_Task call." );

        tasks.push_back( tsk );
    }
    else
    {
        if ( find( func.function_calls.begin(),
                   func.function_calls.end(),
                   name ) == func.function_calls.end() )
            func.function_calls.push_back( name );
    }
}

template< class T >
void handle_function_body( T& node, function& func,
                          vector< task >& tasks,
                          translation_unit& tu,
                          bool readonly = true )
{
    if ( node.value.id() == assignment_expression_id )
    {
        handle_function_body( node.children[0], func, tasks, tu, false );
        for( int i = 1; i < node.children.size(); ++i )
```
APPENDIX C. TOOL SUPPORT SOURCE-CODE

```c
{ 
    handle_function_body( node.children[i], func, 
                        tasks, tu, true );
}

return;
}

if ( ( ( node.value.id() == unary_expression_id ) || 
      ( node.value.id() == postfix_expression_id ) ) && 
   ( ( node_string( node ) == "++" ) || 
     ( node_string( node ) == "--" ) ) )
{
    handle_function_body( node.children[0], func, tasks, tu, false );
    return;
}

if ( ( node.value.id() == block_item_list_id ) || 
    ( node.value.id() == compound_statement_id ) || 
    ( node.value.id() == argument_expression_list_id ) || 
    ( node.value.id() == jump_statement_id ) || 
    ( node.value.id() == primary_expression_id ) || 
    ( node.value.id() == unary_expression_id ) || 
    ( node.value.id() == cast_expression_id ) || 
    ( node.value.id() == multiplicative_expression_id ) || 
    ( node.value.id() == additive_expression_id ) || 
    ( node.value.id() == shift_expression_id ) || 
    ( node.value.id() == relational_expression_id ) || 
    ( node.value.id() == equality_expression_id ) || 
    ( node.value.id() == AND_expression_id ) || 
    ( node.value.id() == exclusive_OR_expression_id ) || 
    ( node.value.id() == inclusive_OR_expression_id ) || 
    ( node.value.id() == logical_AND_expression_id ) || 
    ( node.value.id() == logical_OR_expression_id ) || 
    ( node.value.id() == conditional_expression_id ) || 
    ( node.value.id() == expression_id ) || 
    ( node.value.id() == selection_statement_id ) || 
    ( node.value.id() == iteration_statement_id ) )
{
    for( int i = 0; i < node.children.size(); ++i )
    {
        handle_function_body( node.children[i], func, tasks, tu );
    }
}

// Declarations are only of interest if they are initialised from 
// global variables, so only deal with the initialisation part.
if ( node.value.id() == declaration_id )
{
    for( int i = 0; i < node.children.size(); ++i )
    {
        if ( node.children[i].value.id() == init_declarator_id )
        {
            ...
        }
    }
}
```
handle_function_body( node.children[i].children[1],
    func, tasks, tu );
}
}

// Handle function calls.
if ( node.value.id() == postfix_expression_id )
{
    if ( node_string( node.children[1] ) == "(" )
    {
        add_function_call( node, func, tasks );
        handle_function_body( node.children[2], func, tasks, tu );
    }
}

if ( node.value.id() == identifier_id )
{
    string name = node_string( node );

    if ( tu.globals.find( name ) != tu.globals.end() )
    {
        if ( func.global_access.find( name ) !=
            func.global_access.end() )
        {
            // Once a write access, always a write access.
            if ( func.global_access[name] == true )
                func.global_access[name] = readonly;
            else
                func.global_access[name] = readonly;
        }
    }
}

template< class T >
bool build_function( T& node, translation_unit& tu,
    vector< task >& tasks )
{
    static iter_t start, end;
    static string name;

    if ( ( node.value.id() == typeSpecifier_id ) ||
    ( node.value.id() == struct_or_unionSpecifier_id ) ||
    ( node.value.id() == struct_or_union_id ) ||
    ( node.value.id() == type_qualifer_id ) ||
    ( node.value.id() == pointer_id ) ||
    ( node.value.id() == typedef_name_id ) ||
    ( node.value.id() == storage_classSpecifier_id ) ||
    ( node.value.id() == functionSpecifier_id ) ||
    ( node.value.id() == declaration_specifiers_id ) )
        return true;
if ( node.value.id() == identifier_id )
{
    name = string( node.value.begin(), node.value.end() );
    return true;
}

// Only pick up the function name from the declarator.
if ( node.value.id() == direct_declarator_id )
{
    return build_function( node.children[0], tu, tasks );
}

// The function body.
if ( node.value.id() == compound_statement_id )
{
    iter_t bs = node.value.begin();

    function func( name, start, end, bs );
    tu.functions[name] = func;
    handle_function_body( node, tu.functions[name], tasks, tu );
    return true;
}

if ( node.value.id() == function_definition_id )
{
    start = node.value.begin();
    end = node.value.end();

    for( int i = 0; i < node.children.size(); ++i )
    {
        if ( !build_function( node.children[i], tu, tasks ) )
            return false;
    }

    return true;
}

return false;

translation_unit process_tree( const tree_parse_info_t& ti, vector<task>& tasks )
{
    translation_unit ret;

    if ( ( !ti.full ) || ( !ti.match ) )
    {
        cerr << "Failed_parse" << endl;
        throw invalid_argument( "tree_parse_info_not_complete." );
    }

    if ( ti.trees[0].value.id() != translation_unit_id )
    {

```
cerr << "Failed to parse" << endl;
throw invalid_argument( "tree_parse_info has no t.u." );
}

for( int i = 0; i < ti.trees[0].children.size(); ++i )
{
    if ( !build_variable( ti.trees[0].children[i], ret ) )
    {
        if ( !build_function( ti.trees[0].children[i], ret, tasks ) )
        {
            ret.declarations.push_back
            ( node_string
                ( ti.trees[0].children[i]
            );
        }
    }
}

// Remove the main function as we add our own.
ret.functions.erase( "main" );

return ret;
}

function& find_function( string name, vector< translation_unit >& tu )
{
    for( int i = 0; i < tu.size(); ++i )
    {
        if ( tu[i].functions.find( name ) != tu[i].functions.end() )
            return tu[i].functions[name];
    }

    // In normal operation, this should never be reached.
    static function func;
    return func;
}

variable& find_global( string name, vector< translation_unit >& tu )
{
    for( int i = 0; i < tu.size(); ++i )
    {
        if ( tu[i].globals.find( name ) != tu[i].globals.end() )
        {
            if ( tu[i].globals[name].external == false )
                return tu[i].globals[name];
        }
    }

    // In normal operation, this should never be reached.
```
APPENDIX C. TOOL SUPPORT SOURCE-CODE

```c
static variable var;
return var;
}

void reduce_functionGlobals(task& tsk, function& func,
  vector< translation_unit >& tu )
{
  map< string, bool >::iterator iter = func.global_access.begin();
  for( ; iter != func.global_access.end(); ++iter )
  {
    map< string, bool >::iterator it;
    it = tsk.global_access.find( iter->first );
    if ( it != tsk.global_access.end() )
    {
      // Once a write access, always a write access.
      if ( it->second == true )
        it->second = iter->second;
    }
    else
      tsk.global_access.insert( *iter );
  }
  for( int i = 0; i < func.function_calls.size(); ++i )
  {
    function& funct = find_function( func.function_calls[i], tu );
    reduce_functionGlobals( tsk, funct, tu );
  }
}

void process_communication( int index, vector< task >& tasks )
{
  map< string, bool >::iterator i;
  for( i = tasks[index].global_access.begin();
  i != tasks[index].global_access.end();
  ++i )
  {
    for( int j = 0; j < tasks.size(); ++j )
    {
      // Ignore the current task
      if ( j == index )
        continue;
    }
    map< string, bool >::iterator k;
    k = tasks[j].global_access.find( i->first );
    // Found a matching variable access
    if ( k != tasks[j].global_access.end() )
    {
      // If we read and they write... input
      if ( ( i->second ) && ( !k->second ) )
```
{  
    task_io tskio_in, tskio_out;

    tskio_in.var_name = i->first;
    tskio_in.task_num = j;

    tskio_out.var_name = i->first;
    tskio_out.task_num = index;

    tasks[index].input.push_back(tskio_in);
    tasks[j].output.push_back(tskio_out);
}

// If we write and they read... output
else if ((!i->second) && (k->second))
{  
    task_io tskio_in, tskio_out;

    tskio_out.var_name = i->first;
    tskio_out.task_num = j;

    tskio_in.var_name = i->first;
    tskio_in.task_num = index;

    tasks[index].output.push_back(tskio_out);
    tasks[j].input.push_back(tskio_in);
}
else
    throw invalid_argument("Invalid task communication.");
}

int determine_position(int index, vector<task>& tasks)
{
    tasks[index].position = index + 1;
    return index + 1;
}

void process_tasks(vector<task>& tasks, 
                    vector<translation_unit>& tu)
{
    for(int i = 0; i < tasks.size(); ++i)
    {
        tasks[i].output.clear();
        tasks[i].input.clear();
        tasks[i].position = 0;
    }

    for(int i = 0; i < tasks.size(); ++i)
    {
        function& func = find_function(tasks[i].function_name, tu);
        reduce_function_globals(tasks[i], func, tu);
    }
process_communication( i, tasks );
determine_position( i, tasks );
}
}

string reduce_declarations( vector< translation_unit >& tu )
{
    vector< string > declarations;

    // Go through every translation unit
    for( int i = 0; i < tu.size(); ++i )
    {
        // Go through every declaration
        for( int j = 0; j < tu[i].declarations.size(); ++j )
        {
            // Find unique declarations and add them to the list
            if ( find( declarations.begin(), declarations.end(),
                tu[i].declarations[j] ) == declarations.end() )
                declarations.push_back( tu[i].declarations[j] );
        }
    }

    string ret;

    // Compress the list to a single string
    for( int i = 0; i < declarations.size(); ++i )
    {
        ret += declarations[i];
        ret += "\n";
    }

    return ret;
}

void flatten_functions( function& func, vector< function >& funcs,
    vector< translation_unit >& tu )
{
    for( int i = 0; i < func.function_calls.size(); ++i )
    {
        function new_func = find_function( func.function_calls[i], tu );
        funcs.push_back( new_func );
        flatten_functions( new_func, funcs, tu );
    }
}

string reduce_func_declarations( task& tsk,
    vector< function >& funcs,
    vector< translation_unit >& tu )
{
    vector< string > declarations;

    function task_func = find_function( tsk.function_name, tu );
    funcs.push_back( task_func );
appendix c. tool support source-code

flatten_functions( task_func, funcs, tu);

for( int i = 0; i < funcs.size(); ++i )
{
    string declaration = string( funcs[i].start,
        funcs[i].block_start );
    declaration += ";";
    declarations.push_back( declaration );
}

string declaration;
for( int i = 0; i < declarations.size(); ++i )
{
    declaration += declarations[i];
    declaration += "\n";
}

return declaration;
}

void create_task_unit( int num, task& tsk,
    vector< translation_unit >& tu,
    int num_tasks )
{
    ofstream task_stream;
    string filename;
    stringstream fname( filename );
    fname << "task" << tsk.position << ".c";

    task_stream.open( fname.str().c_str() );
    if( !task_stream )
        throw invalid_argument( "Unable to open task file for output" );

    vector< function > funcs;

    task_stream << "/\_Task\_index:" << num << endl;
    task_stream << "/\_Has\_input:";
        << ( tsk.input.empty() ? "false" : "true" )
        << endl;
    task_stream << "/\_Has\_output:";
        << ( tsk.output.empty() ? "false" : "true" )
        << endl << endl;

    // Handle initial declarations
    task_stream << "/\_Declarations\_accumulated:" << endl;
    task_stream << reduce_declarations( tu ) << endl << endl;

    // Handle the global variables
    task_stream << "/\_Global\_variables:" << endl;
    map< string, bool >::iterator it = tsk.global_access.begin();

    for( ; it != tsk.global_access.end(); ++it )
    {

variable& var = find_global(it->first, tu);

task_stream << "" //Variable" << endl;

/*
* Task stream
*/

task_stream << string(var.start, var.end) << endl << endl;

task_stream << 
"" //Data\_typedef" << endl;

(task_stream << "typedef" << var.type
<< "tData_" << var.name << ",");

(task_stream << endl << endl << 
"" //Access\_union" << endl;

(task_stream << "typedef\_union" << endl
<< "{" << endl
<< "tData_" << var.name << "); Data;" << endl
<< "tByte\_Bytes["sizeof(tData_" << var.name << "]);" << endl
<< "}\_uTransfer_" << var.name << "," << endl
<< "}" << endl;

static int inputs = 0;
static int outputs = 0;

if ( tsk.input.empty() == false )
{
    for ( int i = 0; i < tsk.input.size(); i++ )
    {
        if ( tsk.input[i].var_name == var.name )
        {
            task_stream << 
"uTransfer_" << var.name << ";
"Generated\_input\_variable_" << i + 1
<< "_G;" << endl;

            task_stream << 
"uTransfer_" << var.name << ";
"Generated\_ack\_variable_" << i + 1
<< "_G;" << endl << endl;
        }
    }
}

if ( tsk.output.empty() == false )
{
    for ( int i = 0; i < tsk.output.size(); i++ )
    {
        if ( tsk.output[i].var_name == var.name )
        {
            task_stream << 
"uTransfer_" << var.name << ";
"Generated\_output\_variable_" << i + 1
<< "_G;" << endl;

            task_stream << 
"uTransfer_" << var.name << ";
"Generated\_received\_ack_" << i + 1
<< "_G;" << endl << endl;
        }
    }
}

}
```cpp
// The code snippet is a part of a larger framework for processing tasks.
// The code is written in C++ and uses templates and macros for generating
// output code for tasks. The code snippet shows how the size of data,
// input, and output variables are determined and how they are
// generated in output variables.

// The code snippet is a part of a larger framework for processing tasks.
// The code is written in C++ and uses templates and macros for generating
// output code for tasks. The code snippet shows how the size of data,
// input, and output variables are determined and how they are
// generated in output variables.

// Some additional comms info
// The code snippet is a part of a larger framework for processing tasks.
// The code is written in C++ and uses templates and macros for generating
// output code for tasks. The code snippet shows how the size of data,
// input, and output variables are determined and how they are
// generated in output variables.

// Some additional comms info
```
APPENDIX C. TOOL SUPPORT SOURCE-CODE

```c
#include <iostream>

using namespace std;

void Read_Input_Generated(void)
{
    if ( tsk.input.empty() == false )
    {
        for ( int i = 0; i < tsk.input.size(); i++ )
        {
            tByte i = 0, j;
            for ( j = 0; j < tsk.input[i].var_name.size(); j++)
            {
                Generated_input_variable_G[i++]
            }
        }
    }
}

void Send_Output_Generated(void)
{
    if ( tsk.output.empty() == false )
    {
        for ( int i = 0; i < tsk.output.size(); i++ )
        {
            Generated_output_variable_G[i++]
        }
    }
}
```

---

117
APPENDIX C. TOOL SUPPORT SOURCE-CODE

<< "Generated_output_variable_"
<< i + 1 << "_G.Bytes[j];"
<< endl;
task_stream << "_G"
<< endl;
task_stream << "_G"
<< endl;
}
} task_stream << "}" << endl << endl;

// Handle the task's function declarations
task_stream << "//Function declarations for the task:
<< endl;
task_stream << reduce_func_declarations(tsk, funcs, tu)
<< endl << endl;

// Handle the scheduling code
task_stream << "//Domino scheduler:"
<< endl << endl;

// Output scheduler.c here
ifstream sch_file;
sch_file.open( "scheduler.c" );
if ( !sch_file )
    throw invalid_argument( "scheduler.c not found!" );
sch_file.unsetf( ios::skipws );
while( !sch_file.eof() )
{
    char ch;
sch_file >> ch;
task_stream << ch;
}
sch_file.close();

// Output sch_first.c, sch_middle.c or sch_last.c here
ifstream sched_file;
task_stream << endl << endl;
if ( !tsk.input.empty() == false )
{
    if ( !tsk.output.empty() == false )
        sched_file.open( "sch_middle.c" );
    else
        sched_file.open( "sch_last.c" );
}
else
    sched_file.open( "sch_first.c" );

if ( !sched_file )
    throw invalid_argument( "sch_*c not found!" );
sched_file.unsetf( ios::skipws );
while( !sched_file.eof() )
{
char ch;
sched_file >> ch;
task_stream << ch;
}
sched_file.close();
task_stream << endl << endl;

// Handle the actual functions
task_stream << " // Task function definitions:" << endl;
for(int i = 0; i < funcs.size(); ++i )
{
    task_stream << string( funcs[i].start, funcs[i].end )
        << endl;
}
task_stream << endl;

// Handle the setup and teardown
// Task entry point:
task_stream << " int main() " << endl
    << " { " << endl
    << " SCC_Init_CAN(); " << endl
    << " SCH_Add_Task(" << tsk.function_name << ", "
    << tsk.initial_delay << ", "
    << tsk.period
    << "); " << endl << endl
    << " SCC_Start(); " << endl << endl
    << " while(1) " << endl
    << " { " << endl
    << " SCH_Dispatch_Tasks(); " << endl
    << " } " << endl
    << " } " << endl << endl;

task_stream.close();
}
References


