Abstract

In seeking to gain a competitive edge, companies are searching for alternative methods for developing computer-based systems more effectively and efficiently. Application family engineering is seen as one potential solution.

A principal idea of application family engineering is to identify commonalities and variability among systems of a family, into analysis, design and code assets. New systems can be generated from these assets by filtering out requirements, design and code solutions that do not trace to the particular needs of the new system. The embedded systems industry has been reluctant to embrace application family engineering and other reuse methods, because of a concern that application family assets include provisions for variability that may bear an unpredictable execution time and memory space overhead. In environments where time and space are in short supply, reuse (and maintenance) may be undesirable. However, there is little empirical evidence to support this concern.

In this work we investigate the behaviour of the potential time and space overhead incurred by one design mechanism for managing variability in application family designs: common interfaces. Our approach is case study based and compares the performances of systems with one-off designs with no variability to the same systems
having application family designs with variability. Five common interface techniques for handling variability are examined, based on the components of five prevalent architectural styles, including Object-Orientation, Main/Subroutine, Event-Based, Pipe and Filter, and Repository architectural styles. For each architectural style we selected a case study application family that may typically be designed using that style.

Our results show that all five architectural styles exhibit a clear correlation between execution time and memory space overhead, and the amount of variability. Our data implies that although a trade-off between reuse and performance may exist, it can be predictable and possibly even quantified. The implication that overhead can be predicted and possibly quantified may help decide the economic viability of reusing application family assets in an embedded environment.
CHAPTER 1

Introduction

1.1 Introduction

Companies are often unable to develop system solutions effectively and efficiently enough to meet demand. To tackle this problem, one software development approach is application family engineering. Examples of such families include mobile phones, pension plans, and satellite instrument control units. A product of application family engineering is an application family model, comprising of a set of reusable assets for a family of applications. It identifies commonality and points of variability of existing and future systems of that family. New systems emerge from the interaction of two activities: application family engineering to create reusable assets and single system engineering to build single systems using those assets (Mannion et al [1.1], [1.2]).

Embedded software systems are found in a number of applications including aircraft flight control, reactor safety systems, medical electronic devices, washing machines, and microwave ovens. They are characterised by having to operate within stringent constraints regarding physical space, memory space, safety, and timing. Under such constraints hardware and software must be compact and well designed.
Embedded systems are primarily concerned with the control of external instruments. They often lack direct user involvement or supervision, and hence require very efficient and highly reliable software that must respond quickly to real-world events.

Embedded systems are not new, as engineers have been integrating computing power into electronics for more than two decades. What is new is that technologies such as distributed computing, intelligent systems, mobile computing, networking, and the Internet have expanded the scope and the power of embedded systems (Lear [1.3]).

It is often assumed that reuse and typically the extra modularization required to maintain it, introduce extra time and memory space overheads. In an environment where time and space are crucial, the likelihood of increased overheads is a deterrent to reuse. Hence, many companies that build embedded systems are reluctant to deploy reuse techniques (Mrva [1.4]).

There are examples where application family engineering was used in the production of embedded systems ([1.5] – [1.8]). However, very little is published about the performance considerations during reuse. Our work is focused on investigating performance issues during reuse, and provides some insights on the differences in behaviour of execution time and memory space requirements between one-off system solutions, built without reuse, and their application family equivalents.

1.2 Embedded Systems Software Design

Until recently, embedded systems design placed more emphasis on hardware design, than on software design. Typically, quick software-hacks were produced to tell the hardware how to function. This was often sufficient because embedded systems were generally based on 8-bit processors and were programmed in assembly
language to define a small and immutable set of functions - simple programs that brought some intelligence to mechanical devices.

Today’s embedded systems often use 32-bit or 64-bit processors running millions of lines of code. The cost of developing an embedded system has shifted from hardware to software. Embedded systems are evolving rapidly. The requirement is for more powerful, more efficient, and more flexible systems, with lower production costs and delivery times. Reuse is seen as a potential approach to address this requirement (Diaz-Herrera et al [1.9]). However, software development methods for embedded systems have evolved slower than non-embedded systems. Similarly, mainstream reuse methods have not evolved to cater for embedded software.

Most embedded systems follow stringent constraints for real-time operation, memory usage, and safety. They are designed to fit into a compact environment in a secure, physical and logical manner. The maintainability of embedded system designs often reflects this, as tight schedules and stringent memory constraints often make embedded system designs highly coupled and very inflexible.

Recent real-time design methods (such as Selic et al [1.10]) have tried to improve management of real-time, memory and safety constraints in traditional one-off system designs. However, engineering reusable, variable components into an environment with constraints on real-time operation, memory usage, and safety introduces a whole new set of challenges.

1.3 Variability in Embedded Systems

Typically, a system built from an application family model will include extra components to aid the maintenance of variability. These components are rarely present in traditional one-off system designs, where concepts of variability are less of an issue. There is a concern, though, that such components may conflict with existing
real-time and memory space constraints. There are two situations where this concern may be warranted:

- **During creation of an application family.** An application family is often partially, if not completely, built from resources of existing systems (a *ploughed field* approach). Existing design and code components that are mined to become reusable application family assets may have existing constraints on execution time and memory space. The introduction of extra components to aid the maintenance of variability may interfere with existing execution deadlines and memory constraints.

- **During evolution of an application family.** As an application family model evolves, some directly reusable components may become variable. New components to manage the new variability are typically wrapped around the component to enable its reuse in systems that require it and its replacement when alternative functionality is required. Execution time and memory space constraints that applied to the original directly reusable component may no longer be attainable.

Our research investigates one approach to reengineering new design and code components to maintain variability – a common interface approach. For example, consider existing embedded software that controls an antilock braking system (ABS), which is to be reengineered as a reusable asset for an application family of vehicle braking systems.
**Figure 1.1:** Example existing ABS braking system

Figure 1.1 shows the braking sequence of an existing ABS system. To prevent the vehicle from skidding in an emergency, the brakes are applied 15 times per second, until the brake pedal is released. We are assuming that the braking frequency is a real-time requirement and any less than 15 times per second is unacceptable.

Typically, a common interface is reengineered for the ABS braking component to aid maintenance of variant braking requirements (such as braking with or without ABS). Figure 1.2 shows how reengineering a common interface can integrate our existing ABS system from Figure 1.1 into an application family of braking systems.

In Figure 1.2 the exclusive OR symbol \( \otimes \) denotes a choice that an engineer must make, in building a system from application family assets. An engineer can either select a braking system that has no ABS, or a reengineered alternative with ABS. At build-time, the common interface *Activate Brakes* will permit the engineer’s design decision to be realised, resulting in the required braking system.

---

1 In Figure 1.1 one ABS brake application takes 0.066 s, which equates to 1/15 s.
Figure 1.2: Application family of braking systems

However, the code implementing **Activate Brakes**, the common interface, may introduce both execution time and memory space overhead. While the overhead of a single instance of a common interface is probably negligible, loops and recursive calls can multiply its effects enough to make it significant. Depressing the brake for just a few seconds will multiply the effects of the common interface by 15 times per second, which may be enough to breach the initial stringent real-time constraint.²

Our hypothesis is that:

> there exists a correlation between execution time and memory space overhead that reuse may introduce and the amount of variability incorporated into a single system.

---

² While similar problems may exist with safety constraints and their applicability to varying components in an application family, this is beyond the scope of this work.
Evidence that our hypothesis holds true would imply the following:

- The amount of potential overhead can be predicted and even quantified.

- A one-off single system, or parts thereof, with real-time and memory constraints may no longer be able to meet constraints if reengineered as application family design and code assets. The implication that overhead can be predicted and possibly quantified may help decide the economic viability of a reengineering process.

- Existing real-time and memory constraints may no longer be valid if an application family design and implementation is evolved to include new variability. The implication that overhead can be predicted and possibly quantified may help decide the economic viability of evolving an application family.

We use an experimental approach to support our claim. The results from our experiments will aid system engineers to determine situations where constraints may be breached. Our work may help inform the trade-off decision as to whether to build a new component that meets constraints, or to reuse an old one.

There is some evidence [1.11] that suggests that execution times will vary if different common interface approaches are used to realise application family variability. We have assumed here that any potential overhead that is associated with common interfaces will differ with platform, architectural style, design method and programming language. Our experiments investigate execution time and memory space overhead for five different common interface methods (grouped by Shaw et al [1.12]).

We benchmark two systems with the same specification: a one-off single-system implementation, and an application family counterpart that accounts for variability. Using comparisons of different case study pairs per common interface method, we
show that there is a high probability that overhead is correlated to the amount of implemented variability.

1.4 Thesis Structure

- **Chapter 2** summarises the current state of software reuse, available reuse processes and methods. It also introduces application family engineering.

- **Chapter 3** introduces the MRAM application family engineering method that we are basing the variability notation in our experiments on.

- **Chapter 4** looks at application family design issues and guidelines that we must consider for our case studies.

- **Chapter 5** characterises embedded systems. It singles out how they vary from other types of systems. We describe issues concerning reuse of embedded systems.

- **Chapter 6** describes our experiment and evaluation approach. It details the individual common interface methods that we scrutinise.

- **Chapter 7** introduces our case studies and presents their individual results.

- **Chapter 8** provides some discussion and our overall conclusions.
1.5 References


CHAPTER 2

An Introduction to Software Reuse

2.1 Software Reuse

Slowing advancements in software productivity and increased demand means that most software organisations increasingly find themselves in the midst of a software crisis that inhibits their ability to produce manageable, high-quality, cost-effective software (Cox [2.1], Pressman [2.2]). Software reuse is seen as one solution to reducing time-to-market, and potentially improving the productivity and quality of software development (Freeman [2.3], Wegner [2.4]).

The funding profile for reuse projects is quite different from conventional software projects. Typically, several years of initial investment is needed before a return is realised (Griss [2.5]). In a competitive cost and schedule focused market, it can be difficult to gain financial support for long-term reuse investments. The economic benefits of reuse are often not visible until the initial investments for building reusable assets are returned. However, once they become visible they can be significant enough to help ensure the long-term survival of an organisation ([2.6] details two reuse examples reporting savings of $1.5m and $2.7m, respectively).
Figure 2.1 shows an example of the cumulative cost of building systems with and without reuse (Coplien et al [2.7]). The benefits become apparent after the lines in Figure 2.1 have converged. Tracz et al [2.8] have gathered evidence that suggests that the average convergence occurs after three systems. Poulin et al [2.6] suggest a number of metrics and a return-on-investment model to help assess the economic viability of reuse to individual business cases.

Typically, a failure to obtain a return-on-investment is due to both non-technical and technical reasons (Berard [2.9]). The non-technical influences for the lack of successful reuse are psychological, sociological and economic (Tracz [2.10]). These are outside of the scope of this research. The most common technical influences that prevent software reuse are the insufficient quality of potentially reusable components and the lack of systematic methods for locating resources to solve a particular problem (Tracz [2.10], Biggerstaff et al [2.11]).
Early attempts to attain reuse were founded on the development of general software libraries (McIlroy [2.12]). This metaphor has inspired several researchers to investigate various retrieval methods based on faceted classification (Prieto-Diaz [2.13]), keywords, or free-text document retrieval (Frakes et al [2.14]). While library-based reuse has shown some success within stable, well understood, low-level application areas such as user interface libraries, mathematical and statistical packages, it has failed to yield a significant change in the way that software is developed. Part of the difficulty is that software modules have to interact with each other; they need to be designed to be compatible with the systems that intend to reuse them and need to fit into a maintenance cycle. Analysts’ implicit assumptions about requirements may produce components that cannot be reused as they conflict with the systems that are required to reuse them. This may be the result of components being built with only one intended system in mind. Latour [2.16] refers to this as the product-centric syndrome. Cybulski [2.17] recognised that this single minded system engineering is attributable to an engineer’s natural instinct to concentrate on the single solution at hand rather than to design systems with reuse in mind. The potential time-to-market benefits associated with reuse can be lost if excessive adaptation and subsequent regression testing are required. Biggerstaff et al [2.15] conclude that reuse is only worthwhile if the long-term cost of building applications with reuse is less than without.

Recently approaches to library methods have had more success. These include software kits (Beach et al [2.18]), task neutral problem solving (Beys et al [2.19]), generic building blocks (Mili et al [2.20]), and component-based software engineering (Szyperski [2.21]). Their emphasis is on producing library components that are extremely flexible and modular. Although this may solve some compatibility issues, one fundamental problem still persists: software products are not engineered from existing components. Typically a design stage is followed by a search for the possible
existence of code components that might be reusable: This is still a product-centric approach.

Most library reuse methods focus on the integration of reusable components with few or no processes to initially engineer reusable assets. Arango [2.42] identifies that libraries of components need to be engineered using a well-defined process. This understanding has lead to the discipline of application family engineering.

### 2.2 Application Family Engineering

To understand the concept of application family engineering, we must first define the contextual meaning of Application Family. Berard [2.22] gives two definitions:

- A collection of current and future software applications that share a set of common characteristics.

- A well defined set of characteristics that accurately, narrowly, and completely describes a family of problems for which computer application solutions are being, and will be, sought.

Examples of application families include mobile phones, washing machines, pension products, and satellite instrument control units. New products emerge from the integration of two separate activities:

- Application family engineering to create reusable assets.

- Single systems engineering to build systems using those assets.

---

1 By reusable asset we mean any document, architectural design, detailed design, source code, component, test specification and data, or any part thereof, as relevant in context.

2 Synonyms of application family engineering that are often found in literature include product line engineering, product family engineering and domain engineering.
An essential part of application family engineering is the generation of an application family model, which must identify and represent the commonalities and the variability between systems of a family. It is typically constructed out of a combination of knowledge from domain experts and assets from past systems that fall into the current family (*ploughed-field* construction). *Green-field* construction (where no domain knowledge or past system assets are available) is equally plausible but can carry a higher risk of failure.

**Figure 2.2:** Application family engineering lifecycles

Figure 2.2 shows the correlation between processes of application family engineering and system engineering lifecycles. The horizontal arrows indicate the direction of development and the vertical arrows indicate a filtering and feedback process. For each lifecycle phase, filtering the relevant reusable assets from an application family model yields single systems products. Feedback from single system models will update and refine an application family model. For example, an application family design model may be filtered to extract relevant assets for the
construction of a single system design. Feedback from the single system design will then update and refine the application family design model for future system designs.

Different methods of application family engineering focus on different lifecycle phases. For example, MRAM [2.52] only considers the analysis phase of the application family engineering lifecycle where the reusable assets of the application family model are application family requirements. Other methods, for example ODM (Simos et al [2.46], [2.48]) encompass more of the lifecycle. We are particularly interested in the design and implementation issues of application family engineering and adopt MRAM’s discrimination mechanisms for managing variability in our application family designs and implementations.

2.3 Methods of Application Family Engineering

Figure 2.3 shows a genealogy of the more significant application family engineering approaches. The arrows indicate the influences. For example, Jaworski’s approach [2.25] was influenced by Parnas’ work on information hiding, and by Coad’s [2.24] object-oriented analysis method. The following sections outline a few of the more substantial application family engineering methods. MRAM is omitted here, as it is covered in Chapter 3.
2.3.1 Feature Oriented Domain Analysis

The Software Engineering Institute at Carnegie Mellon University developed the Feature Oriented Domain Analysis (FODA) approach in 1990 (Kang et al [2.29]). It is based on identifying features of a problem domain (an application family), which describe both system commonalities as well as differences within that domain. FODA concentrates on two of the lifecycle phases in Figure 2.2, that is, analysis and design.

There are three main tasks to FODA: feature modelling; information analysis; and operational analysis.
Feature modelling produces a hierarchical tree diagram in which the features are typically one-word terms, which are meaningful to stakeholders. Variability is modelled by permitting features to be mandatory, alternative and/or optional.

Information analysis captures and defines the data requirements of the applications of a family. The output of this task may take a combination of forms including entity-relationship diagrams, class diagrams, and structure charts.

Operational analysis identifies the control and data flow of the applications of a family. The product of this operational analysis may also include data flow diagrams, class interaction diagrams and state transition diagrams.

Recent changes (Paterson et al [2.30], Griss et al [2.51]) have adapted FODA to include object-oriented developments. Hamilton Technologies have developed the 001 tool [2.35] which, although intended for their own purposes is able to partially automate FODA. Applied examples include a family of windowing interface applications [2.29] and a family of army movement control systems [2.45].

2.3.2 Reuse-Driven Software Engineering Business

Reuse-Driven Software Engineering Business (RSEB) is a framework developed by Jacobson et al [2.28] to define a set of guidelines and models that help ensure success with large-scaled, object-oriented reuse. The RSEB framework deals with business, process, architecture, and organisation in an application family.

The RSEB framework can be classified into three main categories: component system engineering; application family engineering; and application system engineering.
Component system engineering is a process responsible for creating component systems. Each component system is a set of customisable, configurable software components where a component is typically either a data type, class, or any work-product that has been specially engineered to be reusable.

Application family engineering (in the context of RSEB) creates an overall system architecture and identifies relevant component systems that may be used within the architecture to create systems.

Application systems engineering is the process of creating a specific system that belongs to a family by selecting, specialising and assembling the relevant component systems.

Subcontractors of Telecom Italia successfully use RSEB for telecommunications systems [2.28], and RSEB has recently been combined with FODA to create FeatuRSEB [2.51].

2.3.3 Organisation Domain Modelling

The Organisation Domain Modelling (ODM) method (Simos [2.46], [2.48]) was developed in 1995 as a part of the Software Technology for Adaptable Reliable Systems project (STARS). ODM is not restricted to any software-engineering paradigm and can be customised to individual business and process standards.

ODM is an extremely large method encompassing an excess of 25 individual processes, which can be generalised into three main categories: domain planning; domain modelling; and asset base engineering.
Domain planning involves assessing the suitability and boundaries of a family of applications.

Domain modelling identifies and depicts commonality and variability of concepts and features within a family of applications.

Asset base engineering focuses on selecting, modelling and implementing features that shall be supported in a reusable asset base. An ODM asset can be a combination of products from any application family engineering lifecycle (including documentation, designs, source-code, and even compilers).

ODM has been applied on a small-scale by a variety of organisations, including the Lockheed Martin Corporation, the Air Force Comprehensive Approach to Reusable Defence Software (CARDS) program, the Software Engineering Institute, Logicon Corporation, and the Rolls-Royce University Technology Centre. Larger applications of ODM have been attempted at Hewlett-Packard and by the US Army’s STARS demonstration project (Simos [2.46], Kelly et al [2.54]).

2.3.4 Domain Specific Software Architecture

The Domain Specific Software Architecture (DSSA) project [2.8] was developed for the Advanced Research Projects Agency (ARPA) in 1994. It supports the development of a domain model, and reference architectures for a particular family of applications.

The Domain Specific Software Architecture method consists of two main processes: domain analysis; and systems engineering.

Domain analysis is used to identify components and constraints inherent in a family of applications. The result of domain analysis is a reference architecture, which is a blueprint for a single system generator.
- **System engineering** defines systems by integrating sets of parameterised, plug-compatible components into the reference architecture.

Organisations that use DSSA include the Army Armament Research Development and Engineering Centre, MICOM Software Engineering Directorate, and Honeywell [2.8].

### 2.3.5 Synthesis / Reuse-Driven Software Process

The Software Productivity Consortium developed the Synthesis / Reuse-Driven Software Process (RSP) method (Campbell *et al* [2.32]) as a systematic approach for identifying similarities and differences across business requirements, to exploit commonalities in families of applications.

The Synthesis / RSP method spans all four lifecycle phases in Figure 2.2. It is divided into four main processes: domain management; domain analysis; domain implementation; and project support.

- **Domain management** is an activity of the Synthesis / RSP method for managing business-area resources to achieve assigned business objectives. It identifies strategies for integrating and monitoring existing company practices with application family engineering disciplines.

- **Domain analysis** is a process of Synthesis / RSP that formalises a family description to standardise and prioritise knowledge of how recurring and varying customer requirements affect the form and content of a products of that family.
Domain implementation is an activity for implementing product and process support for application engineering projects in a business area (that is, application family). Individual systems are built using the relevant products and processes described by the domain analysis process.

The project support process of Synthesis / RSP validates individual systems implementations. The project support activity will co-ordinate feedback to the other Synthesis / RSP processes and products.

Synthesis / RSP has been used successfully to initiate and refine application family engineering processes in companies such as Rockwell [2.57] and Lockheed-Martin. It was also a basis for the Boeing/Navy project of the DARPA STARS [2.58] program, and has been adopted as a corporate standard by Thomson-CSF [2.59].

2.3.6 Product Line Software Engineering

The Fraunhofer IESE (Institut für Experimentelles Software Engineering) in Germany developed the Product Line Software Engineering (PuLSE) method in an attempt to create a systematic, iterative approach to developing systems from reference architectures (DeBaud et al [2.55], Bayer et al [2.56]). The PuLSE method spans all application family lifecycles from Figure 2.2.

The main processes of PuLSE are: domain definition; evaluation management; and single system generation.

Domain definition is a process of PuLSE that defines a domain scope, which provides the boundaries of a particular family. A scope definition, along with a decision model and generic workproducts (that is, the application family differences and commonalities), provide the inputs for a reference architecture.
An evaluation process manages and maintains the evolution of the reference architecture, and provides constant feedback to early lifecycle stages.

Single system generation will populate the reference architecture generating systems from reusable workproducts.

2.3.7 Joint Integrated Avionics Working Group’s Object-Oriented Domain Analysis

The Joint Integrated Avionics Working Group’s Object-Oriented Domain Analysis (JODA) method [2.26] is largely influenced by the object-oriented analysis and design methods by Coad et al [2.24]. The main focus of the JODA method is on application family design (though some application family analysis guidelines are mentioned).

There are three main activities to JODA: business and methodology planning; domain engineering; and application engineering.

Business and methodology planning assess the feasibility and define the application family engineering procedures that are most applicable to a particular family.

Domain engineering captures common functionality amongst applications in a family. The product of domain engineering is an application family design that is modelled using notation from Coad et al [2.24]. Variations of system components within an application family are abstracted into generic objects.

Application engineering instantiates single systems from the application family design. Feedback from new systems will evolve the application family designs.
The JODA documentation includes a small example of a family of office building elevators and mentions its application to a family of avionics store management systems (although there is no reference to further documentation).

2.4 Summary

In this chapter we introduced application family engineering as a method for engineering reuse. The main practice of application family engineering is to define commonalities and variability between family assets. New systems are specified and built by selecting the relevant family assets.

The processes and products between the individual application family engineering methods vary as different methods focus on different lifecycle stages with different technical and business objectives (Mili et al [2.60]). However, they all share the same principle: single systems assets are derived from application family assets.

Our particular interest lies in how the different methods deal with application family variability in architectures, designs and implementations, and their impact on execution time and memory space. The embedded systems industry has been reluctant to embrace application family engineering because of a concern that the inclusion of provisions for handling variability may bear an unpredictable execution time and memory space overhead (see Chapter 5). In an environment where execution time and memory space are in short supply a possible trade-off with ease of reuse (that is, maintenance) may be undesirable.

In Chapter 3 we provide a detailed introduction to MRAM, the application family engineering method that we will be applying to our experiments. We are particularly interested in the mechanisms that MRAM uses to manage variability, namely discriminants, as we carry the same mechanisms through to our application family designs.
2.5 References


3.1 Introduction

Method for Requirements Authoring and Management (MRAM) [3.1], was developed as a joint venture between Napier University, Siemens AG and the European Space Operation Centre (ESOC). Most application family engineering methods concentrate on reusing analysis, design and code. For example, RSEB [3.2] depicts variability with use-cases and JODA [3.3] uses generic classes. However, there has been little work on reusing requirements (Sutcliffe [3.4]). Typically, the process is to map application family requirements and requirement clusters onto analysis and design modelling structures (such as use-cases, entity-relationships, and functional models). These are then mapped to reusable code components. When a new application is built, the desired requirements are selected from the application family requirements and compromises are made when parts of clusters are desired. However, there is little explanation about the specification of application family requirements.
We concur with Latour [3.5] and Cybulski [3.6] that reuse needs to start at the requirements phase of development. In Chapter 2, Figure 2.2 shows the application family engineering lifecycles. MRAM’s main focus is on the analysis phase.

We have chosen to use MRAM as the basis for our experiments, as MRAM is not tied to any design method or programming language. It provides guidelines for writing just reusable application family requirements and covers very little application family design and implementation aspects. This gives us the freedom to specify systems from an application family model, but design and implement them using a variety of styles for capturing and modelling variability, giving us a clearer view of the behaviour of potential overhead.

MRAM suggests mechanisms for denoting and discriminating variability between application family requirements. We believe this mechanism to be both intuitive and representative of discrimination mechanisms of many other methods. We continue the same discrimination mechanisms into our application family designs.

Typically application family engineering methods use a free-selection approach to pick appropriate application family assets for a new system. MRAM uses a discriminant-based selection process to build single systems from an application family model. This can help ensure that conflicting requirements do not coexist in new systems. We use the MRAM discriminant-based selection process to specify all systems of our experiments.

MRAM was used to generate an application family of spacecraft control operating systems at ESOC (Mannion et al [3.7]). To automate and support the management of the application family we developed a Tool for Requirements Authoring and Management (TRAM).
3.2 MRAM Application Family Model

The MRAM application family model depicts commonality and variability of requirements between systems in a family. Requirements in the application family model are structured hierarchically, following guidelines by Keepence et al [3.8]. The hierarchy of requirements is achieved by splitting the specific from the generic. It is often the case that a requirement contains generic and specific parts. Although such requirements are valid, they may reduce the level of reuse. Here is an example:

**REQ 1** A mobile phone shall support text messaging using SMS text messages and SMTP electronic mail messages.

Text messaging systems for all mobile phones typically use SMS text messages, but e-mails, as a messaging system, are specific to mobile phones with WAP capabilities. Hence the requirement could be written as follows:

**REQ 1** A mobile phone shall support text messaging.
**REQ 1.1** A mobile phone shall support text messaging using SMS text messages.
**REQ 1.2** A mobile phone shall support text messaging using SMTP electronic mail messages.

![Figure 3.1: Example hierarchy attributed to splitting the specific from the generic](image)

The result is two directly reusable requirements (REQ 1 and REQ 1.1) and one point of variability (REQ 1.2) that can be structured hierarchically (see Figure 3.1).
The MRAM application family model represents commonalities and variability in the form of directly reusable requirements and variable requirements.

Figure 3.1 shows how the generalisation forces the hierarchy into a tree structure where requirements have a parent-child relationship. The deeper the requirement is within the hierarchy, the more specific it is likely to be.

### 3.2.1 Discriminants

One technique for specifying variable requirements is to label those requirements as discriminants (Mannion et al [3.1]). A discriminant is any requirement that differentiates between systems. These can take on one of three basic forms:

- **Single adaptor discriminant**: A set of mutually exclusive requirements from which only one can be chosen in any system in the application family. For example, a mobile phone requires the network services of a telecommunications carrier, but it will only ever use one carrier at a time. In a requirements specification this might be written as:

  **REQ 2** The mobile phone requires the network services of one
  `Single Adaptor` telecommunications carrier at a time.

  **REQ 2.1** The telecommunications carrier shall be X1.

  **REQ 2.2** The telecommunications carrier shall be X2.

Requirement 2 is the single adaptor discriminant, and requirements 2.1 and 2.2 are the mutually exclusive variants. Figure 3.2 shows how the single adaptor example may be structured hierarchically.
Multiple adaptor discriminant: A list of alternatives that are not mutually exclusive, but at least one must be chosen. For example, on any mobile phone there can be several ways of making a phone call, but there must be at least one. In a requirements specification this might be written as:

REQ 3

Multiple Adaptor

REQ 3.1 A telephone call shall be made by dialling a number on the numeric keypad.

REQ 3.2 A telephone call shall be made by pressing a memory recall button to recall a stored number.

REQ 3.3 A telephone call shall be made by pressing a ring-back facility to dial the number of the last incoming call.

REQ 3.4 A telephone call shall be made by using speech recognition technology to interpret voice commands.

Requirement 3 is the multiple adaptor discriminant, and requirements 3.1 to 3.4 are the alternative variants. Figure 3.3 shows how the multiple adaptor example can be structured hierarchically.
Optional discriminant: a single optional requirement that may or may not be chosen. For example, a mobile phone may or may not have an Internet connection. In a requirements specification, this might be written as:

**REQ 4.2** The mobile phone shall have an Internet connection.

*Optional*

Figure 3.4 shows how optional discriminants fit into the hierarchy of an application family model.

Discriminants may be combined to denote more intricate variability. For example, we can combine an optional discriminant with a single adaptor discriminant to create lists of completely optional alternatives:
**REQ 5.2**  The mobile phone shall have an Internet connection.

*Optional*

**REQ 5.2.1**  The Internet connection shall use one of the following protocols.

*Single Adaptor*

**REQ 5.2.1.1**  WAP protocol.

**REQ 5.2.1.2**  PPP protocol.

Figure 3.5 shows how the combined discriminants may be represented graphically within the application family model.

![Figure 3.5](image)

**Figure 3.5:** Example hierarchy of combined optional and single adaptor discriminants

### 3.2.2 Parameters

Where discriminants describe qualitative variability, parameters are able to manage quantitative variability (Mannion *et al.* [3.7], [3.9] and Wheadon *et al.* [3.10]). Some requirements may contain numeric values describing coefficients, array lengths, units of time, or simply constants. Often these requirements can be reused if their numeric values are changeable. Consider the following example:

---

1 We are introducing the concept of parameters for completeness; we do not use them in our experiments. Chapter 4 describes how we model quantitative variability using discriminants.
REQ 6.2 The mobile phone shall establish a connection to its telecommunications provider in no longer than 4.8 seconds.

Requirement 6.2 is only reusable if the prerequisite that all systems in the application family always have a fixed connection deadline of less than 4.8 seconds, is fulfilled. If this is not the case for all systems of that application family, then the requirement is not easily reusable. We can rewrite requirement 6.2 and replace the constant numeric value (4.8) with a parameter:

REQ 6.2 **Parameter**

The mobile phone shall establish a connection to its telecommunications provider in no longer than \( X \) seconds.

![Diagram](image)

**Figure 3.6:** Example of parameter in application family model hierarchy

Figure 3.6 shows how a parameterised requirement fits into the hierarchy of the MRAM application family model. Two types of parameters can be distinguished. These are:

- **Local parameters:** The scope of a local parameter is exclusive to the single requirement within which that parameter is contained.

- **Global parameters:** The scope of a global parameter is the set of application family requirements. Any change to the existence, meaning or value of a global parameter will affect every requirement referencing that parameter.
3.2.3 Parameterised Discriminants

In [3.7] we further describe how discriminants themselves are able to contain a parameterised part. Parameterised discriminants are a mechanism for combining quantitative and qualitative variability into a single requirement. Consider the following situation where a parameter value can effect future discriminant selection decisions:

REQ 7    All mobile phones have a range of operation of $X$ miles

The value of this range, for example, 100-mile radius or 900-mile radius, may determine the set of features available to the user. To make this variability more explicit, requirement 7 can be re-written as separate requirements contained in a single adaptor discriminant, each distinguished by different values for the range of operation.

REQ 7    All mobile phones have a defined range of operation

Single Adaptor

REQ 7.1    The range of operation shall be less than $X$ miles

Parameter

REQ 7.2    The range of operation shall be greater than $Y$ and less than $X$ miles

Parameter

REQ 7.3    The range of operation shall be greater than $Y$ miles

Parameter

3.3 Conformity Amongst Application Family Engineering Methods

FODA [3.11] and FeatuRSEB [3.12] take a similar approach to modelling as MRAM, but they use features and not requirements. There are three phases in FODA: feature modelling; information analysis; and operational analysis. A feature model is
a hierarchical tree diagram in which the features are typically one-word terms, which are meaningful to stakeholders. Variability is modelled by permitting features to be mandatory (that is directly reusable), alternative (single adaptor discriminant) and optional (optional discriminant).

RSEB [3.1] and JODA [3.3] adopt an object-oriented approach, where abstraction is used to manage variable assets. Mutually exclusive assets are abstracted into different sub-class, all with the same base-class. Mechanisms equivalent to the optional and multiple adaptor discriminants of MRAM are represented by the cardinality of class aggregates. For example, a mobile phone class may have one or more classes responsible for different methods of dialling a telephone number. Specifying the required amount (that is cardinality) and types of the classes responsible for dialling, creates new mobile phone systems.

Other methods such as Synthesis (Campbell et al [3.13]) and ODM (Simos [3.14], [3.15]), that are not linked to specific design and programming languages, have similar guidelines for dealing with variable assets at different stages of the application family engineering lifecycle. Both Synthesis and ODM deal with concepts of mutual exclusion, list of alternatives, optional features and quantitative variability. However, the actual mechanisms to handle variability are not well defined.

The MRAM discriminant categories may not always be sufficient to represent qualitative variability, but we do believe them to be intuitive and representative of the discrimination mechanisms of many other methods (Mannion et al [3.7]).

3.4 Single System Definition

When engineering single systems, a system model can be derived from the application family model. A common approach is free-selection where relevant assets are selected in no particular order. The requirements engineer browses the application
family model and simply copies and pastes single requirements from anywhere in the model to the new instance – a new system. However, *free-selection* has its limitations:

- All assets, directly reusable and variable, need to be evaluated individually for their applicability to the system they are intended for. There can be an untenable number of choices.

- A random choice can mean an illegal choice. For example, if two mutually exclusive requirements are accidentally selected, or if requirements that must be included are not selected.

### 3.4.1 Discriminant Based Selection

An alternative approach was proposed by Mannion *et al* [3.1], which uses a discriminant-based, depth-first selection process. In this method, rather than evaluate every application family requirement for its applicability to a particular system, only discriminant requirements are browsed. This means that a systems engineer is only forced to make a choice on reaching a point of variability. Directly reusable requirements are added to the single system without any evaluation for applicability to that particular system.

We will be applying the same principle in the designs for our experiments later on. Explicitly denoting points of variability in our application family designs and corresponding implementations means that we need not attend to every design and code asset when building a new system – only the points of variability.
Figure 3.7 shows an example MRAM application family model with 16 requirements (11 directly reusable requirements and five points of variability).

![Diagram of MRAM application family model]

**Figure 3.7:** Example MRAM application family model

In a discriminant-based, depth-first selection process we start selecting application family assets for a single system at the topmost requirement. In Figure 3.7 we start selection with requirement 1. The following depth-first sequence is required to build a system from the example application family:

- Requirement 1 is directly reusable and can be added without any evaluation.

- Requirement 1.1 is a parameterised requirement, and the systems engineer needs to provide the relevant values.

- Requirement 1.1.1 is optional, and the system engineer needs to evaluate this requirement. If it is applicable to the current system, then it is selected, if not, it is omitted.

- Requirement 1.1.2, 1.1.2.1 and 2 are directly reusable and added automatically.
Requirement 2.1 is a single adaptor, and the systems engineer is required to select one alternative from its possible choices (2.1.1, 2.1.2 or 2.1.3). The preferred option is included in the single system, and the other two are omitted.

Requirement 2.2 is an option. If it is applicable to the current system then it is selected, else requirement 2.2 and all its children are omitted.

Requirement 2.2.1 is a multiple adaptor, and the systems engineer is required to select one or more from a list of alternatives (2.2.1.1, 2.2.1.2 and/or 2.2.1.3).

When building a system, requirements that are directly reusable are added automatically. The benefit of a discriminant-based selection process over *free-selection* becomes apparent when we compare the amount of decisions that a system engineer is required to make during any particular system build. In Figure 3.7 we have to make four or five decisions during any single system specification depending on whether Option 2.2 is selected or not. *Free-selection* would have required 16 decisions.

![Figure 3.8: Example single system specification](image-url)
Figure 3.8 shows an example system that is built from the application family model in Figure 3.7. The requirements that were selected for our single system specification are marked in grey. We were required to make four decisions in specifying the single system, these were:

- System specific values provided for parameters in requirement 1.1.
- Optional requirement 1.1.1 was discarded.
- Single Adaptor alternative 2.1.3 was selected.
- Optional requirement 2.2 was discarded.

Without an application family model to select from, the typical approach to the generation of requirements for a new instance would be *copy, paste then edit*. The advantage of a model - the application family model - is that requirements engineers can make better use of their time focusing on the implications of a requirement’s inclusion in a new instance without thinking about additional definition, specification, linkage and traceability issues. The hierarchical structuring combined with the discriminant-based, depth-first selection avoids selecting conflicting requirements.

### 3.5 Summary

In this chapter we noted the concept of discriminants to model points of variability in an application family model and how they are used to aid single system selection. We have chosen to use MRAM in our experiments for a number of reasons. Firstly, the discrimination mechanisms are very clearly defined, and although MRAM only deals with variability of requirements, we have observed that many other application family engineering methods use similar methods for different lifecycle products. MRAM is also not tied to any design method or programming language, which gives
us the freedom to specify the systems of our case studies from application family models, and design and implement them using a variety of styles for capturing and modelling variability.

Many other application family engineering methods are not restricted to a particular design method and could equally be used to specify our case studies; these including FODA [3.11], Synthesis [3.13], ODM [3.14], PuLSE [3.16], and DSSA [3.17]. Our results should be equally significant to any application family engineering method, provided that they chose to employ a common interface approach (see Chapter 4) for modelling points of variability in an application family design.

Chapter 4 will introduce the concept of application family design and demonstrate how we use common interfaces to model variability based on the MRAM discrimination mechanisms.

3.6 References


CHAPTER 4

Application Family Design

4.1 Application Family Design

Chapter 3 outlined application family analysis and the construction of single system specifications from an application family model (also see Chapter 2, Figure 2.2). This chapter discusses application family design and the construction of single system designs from an application family design.

New system designs are engineered from relevant components of an application family design that implement the desired functionality of a system. There are many application family design methods, each of which use different mechanisms and techniques to incorporate, and manage, commonality and variability. This chapter introduces and discusses our preferred mechanism of designing variability – using common interfaces. We show how common interfaces can use the discrimination mechanism of MRAM to depict points of variability in application family designs.
4.2 A Taxonomy of Mechanisms to Manage Variability

Cortese et al [4.1] present a taxonomy of five mainly object-oriented techniques and mechanisms for handling application family variability in application family designs and implementations. Their taxonomy is insufficient for our purposes as we investigate both object-oriented and non-object-oriented application family design approaches. By generalising their taxonomy, we can note five mechanisms to handle variability that are independent of object-orientation. While there may be more fine-tuned ways of defining this taxonomy; the following five categories cover the majority of mechanisms for dealing with application family commonality and variability.

4.2.1 Parameterisation

One category in the object-oriented taxonomy in [4.1] is Property-Based Customisation. Classes are customised by having them reveal a number of their properties. Figure 4.1 shows a class called Ringing Mechanism that is responsible for the ringing melody that a mobile phone makes on receiving a call. It exposes one property that can be customised – RingType. Defining the value of the RingType property will define the melody that the mobile phone plays when a call is received.

<table>
<thead>
<tr>
<th>Ringing Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>RingType : Enum</td>
</tr>
<tr>
<td>Ring ()</td>
</tr>
</tbody>
</table>

**Figure 4.1:** Example of a class that has customisable properties
We can generalise the property-based customisation technique to make it equally applicable to non-object-oriented design methods, and use a broader term to denote this type of mechanism – Parameterisation.

In the object-oriented approach, a class will behave differently depending on its properties. Cortese et al [4.1] use Java Beans as an example of components which can be customised by setting the appropriate parameter values – the properties of the Java Bean. However, many design and code components (not just objects and classes) may be customised depending on their inputs, properties or parameters. For example, a function may accept a parameter dictating its actions. The function will typically evaluate the parameter at run-time, and perform the required operation. An if or a switch statement may be used to evaluate each parameter. Figure 4.2 shows a function called Ring that takes one parameter – RingType. The pseudo-code demonstrates how a switch statement is used to evaluate the parameter (RingType), and depending on the value of RingType the phone plays a different melody.

Figure 4.2: Example parameterised function

Macro pre-processor instructions, such as #ifdef, are often perceived as parameters which may be evaluated at build-time rather than at run-time. However, macro pre-processors are normally supplementary language primitives that are embedded into a fully functional programming language (such as Pascal, C or C++). These typically cannot be used to evaluate inputs, properties or parameters of the fully functional programming language; as they can only be used to evaluate other macro
pre-processor arguments (such as \#define instructions). Macro pre-processors are discussed as a separate topic in Section 4.2.4.

While parameterisation is potentially a very powerful way of handling variability, there are very few design methods that can denote such detail. Most design methods are concerned with the structure, process-flow, data-flow, states or data organisations of a system; and they rarely cater for parameter instantiation. We feel that because of the lack of design support for this discrimination mechanism, it is ill suited to application family design. For this reason, we do not use parameterisation in our experiments in Chapter 7.

4.2.2 Metamodels

Metamodels provide explicit representation of how the data of systems of an application family shall be designed and constructed, rather than immediate solutions or part-solutions. A metamodel dictates generic rules, which designs of system solutions must adhere to. Typically, the rules of metamodels describe common data and/or their structure. For example, consider an application family of mobile phones that have an integrated electronic address book. A metamodel may be used to describe common data-structures for all mobile phones of that family. Figure 4.3 shows an example metamodel for electronic address book systems using an entity-relationship diagram.
All electronic address book systems that are built to a metamodel must comply with the common data-structure denoted in the metamodel. The application family design may consist of many different variants of an electronic address book, each with different functionality, for many different types of mobile phone. However, the underlying data and data structure remains consistent.

The metamodel evolves with every new system that is built from the application family. As it matures, application family knowledge is increasingly embedded in the common data and the data structure, and less by logic hardwired into actual design and code components. While Cortese et al [4.1] present a metamodel that can be customised and evolved at run-time, such as Active Object Models (Foote et al [4.3]), build-time evolution is equally viable.

Metamodels may be a very powerful way of modelling and evolving application family commonality. However, they are typically not able to denote variability without a complementary mechanism, like for example, common interfaces at points of variability (see Section 4.2.5). For this reason, the application family designs of our experiments are not based on metamodels.
4.2.3 Domain-Specific Languages

Domain-specific languages are yet another mechanism to aid the management of variable code components. They define systems in terms of domain-concepts, enabling systems to be programmed in high-level abstractions, pushing variability from detailed and abstruse code to a forefront. Compilers then translate domain-specific language specifications into source code (Batory et al [4.4]). Some application family engineering methods (such as FODA [4.5] and FeatuRSEB [4.6]) rely on components to encapsulate the implementations of basic family-features that are common to systems of a family. Systems are then derived from the addition or removal of feature components.

**Figure 4.4:** Example mapping between application family feature and domain and non-domain-specific language implementations

Figure 4.4 shows an example mapping between an application family feature for a family of mobile phones (feature *Connect Call*), and example domain and non-domain-specific language implementations of that feature. In the non-domain-specific language, including or excluding the feature to connect calls may require the inclusion or exclusion of many lines of code, spread out over many source code
components. In Figure 4.4 the domain-specific language implementation of the feature is encapsulated into one call: \texttt{ConnectCall()}. 

Batory et al [4.4] present a domain-specific language to model state diagram specifications for a mainly event-based application family. They use an extended version of Java (JavaSM) that allows increased control of state and event specifications, which are typically hidden in the detail of Java code.

Schönwälder et al [4.7] present a domain-specific language called Scotty for programming network management applications that use the Simple Network Management Protocol (SNMP [4.8]). Scotty is Tcl/Tk based and abstracts network management features into high-level Tcl scripts.

We do not use domain-specific languages for any of our case studies, as MRAM does not use a \textit{feature} model to specify the application family; it uses natural language requirements. While we do not make judgement over either approach; we believe that detailed and abstruse design and code components can contain variability, so long as they are identified as such. In Section 4.3 we will show how we use common interfaces to highlight and maintain points of variability.

\textbf{4.2.4 Macro Pre-processors and Code Generators}

Macro pre-processors and code generators may also be used to manage application family variability in source code. Macro pre-processors are usually supplementary instructions that are embedded into a fully functional programming language, like Pascal, C or C++. Macro pre-processors include or exclude code based on command line arguments passed to the compiler; the compiler then proceeds to build the resulting program.

A macro pre-processor can be found in many compilers and includes instructions like \#define, \#include and \#ifdef. Typically a code component that is variable, is
embraced by a conditional compound pre-processor statement (for example, in C or C++ this might be a \texttt{#ifdef ... #endif} compound). Before compilation, each of the conditions are evaluated using command line arguments (or arguments specified using \texttt{#define}); the embraced code is then either included in or excluded from the build depending on the arguments.

\begin{verbatim}
#ifdef OPTIONAL_GAMES
    void DisplayGameMenu()
    {
        ...
    }

    void PlayGame()
    {
        ...
    }
#endif
\end{verbatim}

\textbf{Figure 4.5:} Example optional component, using macro-pre-processors to handle variability

Consider an application family of mobile phones that can include optional games on some phones, but exclude them from others. Figure 4.5 shows how macro-commands may be used to include or exclude the optional games using C. By specifying the command line argument \texttt{OPTIONAL_GAMES} to the compiler, the code will be included in the build; else it is excluded.

Code generators are a different approach to managing variability in application family implementations. However, the results of both the macro pre-processor approach and the code generator approach are identical; because they both include or exclude application family code components at build-time, no run-time evaluation is required. A code generator will produce source code by following a knowledge base of rules dictating the constraints of the software. These rules can be based on a textual syntax or they can be based on a graphical model (typically the application family design). Unlike macro pre-processors that are usually embedded into a
programming language, a code generator is normally separate from the programming language and compiler. Some case-tools, like Rational Rose, have an integrated code generator, which translate design into code.

Engelhart \textit{et al} [4.9] present a code generator that is used to manage variability in an application family of guidance, navigation and control algorithms, called ControlH. It accompanies the DSSA application family engineering method [4.10], to generate single system implementations in Ada or C. ControlH uses both textual and graphical syntaxes for selecting design components for generation.

Both macro pre-processors and code generators handle variability in source code; they are not design mechanisms. They are typically used to complement implementations that are based on application family designs. While all the application family designs for our experiments use a common interface approach (see 4.2.5) to manage variability, some of our application family implementations may use macro pre-processors too. Macro pre-processors may complement a common interface approach in programming languages that do not provide mechanisms to manage build-time discrimination (such as C and Pascal). We discuss this later on in this chapter. No code-generators are used in our experiments.

\textbf{4.2.5 Common Interfaces}

The final category in this taxonomy is a common interface approach to managing variability in application family design. The term common interface in the context of software systems stems from early object-oriented documentation and the notion of abstraction and polymorphism (for example Rumbaugh \textit{et al} [4.11]). An abstract base class is said to provide a common interface to its child classes though a clearly defined public interface. In an application family design, a base class can define a common interface to variant classes of an application family design. New systems are created by aggregating the appropriate sub-classes that realise the desired variability.
We use the term common interface in a slightly looser way; a common interface is any design or code component that provides a public interface to one or more variant components of the same design method or the same type of code component. All access to variant components is via the common interface. The common interface shall be instantiated to access the appropriate variant components. Example common interfaces may include functions, objects, data-structures, filters, and even software libraries. The application family designs for our experiment consistently use common interfaces at points of variability.

In the requirements document for a traditional one-off single system points of variability are not required, hence not accounted for. The corresponding design will reflect this. We can demonstrate this using part of a mobile phone system.

REQ 2 The mobile phone requires the network services of one telecommunications carrier at a time.

REQ 2.1 The telecommunications carrier shall be X1.

The requirements do not indicate any points of variability; hence their design will reflect this. Figure 4.6 shows how the mobile phone may be designed using the UML notation (Booch et al [4.12]).
Figure 4.6: Example one-off single system design without variability considerations

The design in Figure 4.6 shows a class called Mobile Phone that has one member data of type CarrierX1. No variability is identified; hence no common interfaces are present. Building the same mission planning system using an application family approach means that points of variability are identified. The application family model for the family of mobile phones may include the following variability:

**REQ 2** The mobile phone requires the network services of one telecommunications carrier at a time.

**REQ 2.1** The telecommunications carrier shall be X1.

**REQ 2.2** The telecommunications carrier shall be X2.

The application family model notes that there may be two types of carrier. As no two carriers can be used simultaneously, the requirements 2.1 and 2.2 are mutually exclusive – a single adaptor. The application family design will reflect this decision and use a common interface to the different types of carrier classes. Figure 4.7 shows how the application family may be designed.
Figure 4.7: Example application family design with an abstract base class providing a common interface to variability

The design in Figure 4.7 shows a class called Mobile Phone that has one member data of type Carrier. At build-time a systems engineer must specify the required version of Carrier, CarrierX1 or CarrierX2. Should future systems require more types of carrier (such as X3, X4…), an extension is possible by adding another subclass to the common interface Carrier. Figure 4.8 shows an example instantiation, where a system is built that uses X1 as its carrier.

Figure 4.8: System instantiation, using Carrier X1
A common interface provides a single point of access to a variable design (and code component). Components are often accessed by a number of other program units. In systems that do not identify and highlight variability, designing an alternative solution may require changes to many parts of the system. Oversight of the changes is more likely to be lost. Coupling is an indication of the strength of interconnections between program units (Sommerville [4.13], Conger [4.14], Page-Jones [4.15]). The less coupling there is between program units, the easier maintenance becomes. Common interfaces help ensure low coupling at points of variability.

Although we use the UML notation [4.12] to demonstrate how common interfaces may be used to model variability, most design methods are equally capable (although sometimes an additional notation may be required, see Section 4.3). Many application family engineering methods use a common interface approach:

- RSEB [4.16] and JODA [4.17] methods adopt an object-oriented approach, where abstraction is used to capture common interfaces to variable classes.

- DSSA [4.10] and PuLSE [4.18] capture variability in components of a reference architecture, where the desired components, with the correct common interface, can plug into. Both methods are independent of any design method or programming language. Section 4.5 discusses reference architectures in greater detail.

- Batory [4.19] suggests extensions to an object-oriented design paradigm, where object-oriented design patterns of common interfaces are used to denote points of variability within a reference architecture.

Typically, common interfaces can discriminate between alternative variability at both build-time and run-time. We feel that discriminating application family
variability is a build-time issue and not a run-time issue. When designing a new system from components of an application family design, relevant alternatives are selected long before the system is compiled. Hence, we use a build-time common interface approach in our experiments.

4.3 Using Common Interfaces for Qualitative Variability

MRAM proposes three mechanisms to denote qualitative variability at the application family analysis phase: single adaptor discriminants, multiple adaptor discriminants, and optional adaptors.

Keepence et al [4.20] describe an object-oriented solution to designing qualitative variability based on the discriminant types of MRAM (much like our example in Figure 4.7). We propose a similar approach, which extends to include non-object-oriented design methods too. Based on our interpretation of the term common interface, we suggest three patterns for modelling application family variability using discriminants. While we acknowledge that there is unlikely to be a one-to-one mapping between application family requirements and application family design components, providing equivalent mechanisms to handle variability in the design stage will undoubtedly help the transition.

Single adaptor discriminants are used to discriminate between mutually exclusive requirements. The pattern we propose for modelling mutual exclusion in alternative design solutions is shown in Figure 4.9. A common interface is used to access alternative components. For design methods that are unable to denote build-time variability we propose to add an ⊗ (exclusive or) symbol to component connectors, denoting a build-time decision point. Design methods like UML have their own symbols for modelling build-time variability (such as static polymorphism).
When deriving a single system from the application family design, a systems engineer must make a decision at points of variability – at the common interfaces. In Figure 4.9 a systems engineer must select the appropriate connector between the common interface and the appropriate alternative solution (that is, Alternative 1 or Alternative 2 or Alternative n). No change is required to the other components of the system as they access all alternatives in the same way: through the common interface.

Multiple adaptor discriminants are used to denote a list of possible alternatives. The pattern we propose for modelling alternative solutions is demonstrated in Figure 4.10. Unlike the pattern in Figure 4.9 each design component may cover multiple solution alternatives. Again, a systems engineer is required to select the appropriate connector between the common interface and the appropriate solution (that is, Alternative 1 or Alternative 2 or Alternative 1 & 2, and so on).
Finally, in MRAM, optional adaptor discriminants are used to denote optional requirements, which may or may not be included in systems that are derived from an application family model. Figure 4.11 illustrates the pattern we propose for modelling optional components.

Figure 4.10: Multiple adaptor design pattern

Figure 4.11: Optional adaptor design pattern
In Figure 4.11, a systems engineer is required to make one decision when building a single system using the optional adaptor design pattern. This is whether to include or exclude the optional component (Option).

While the patterns in Figure 4.9, Figure 4.10 and Figure 4.11 are able to model qualitative variability using the discrimination mechanism recommended in MRAM, they are just recommendations. We do not assume that they are sufficient to cover all requirements for modelling variability and accept that some situations may require derivative, ad-hoc patterns. Nonetheless, we use these patterns in our experiments (in Chapter 7) to model qualitative variability in application family designs using discriminants.

4.4 Using Common Interfaces for Quantitative Variability

MRAM supports quantitative variability in the form of parameters. Our application family designs and implementations may be required to handle quantitative variability. In Section 4.2.1 we noted that few design modelling notations are able to model numeric attributes and place more value on modelling higher-level concepts like structure, process-flow and data-flow, states or data organisations. However, an application family design that follows an application family analysis phase with both qualitative and quantitative discrimination should be able to model both concepts.

Although common interface approaches are rarely able to model quantitative variability as numeric attributes, we can express numeric variations by means of qualitative variability. Consider a parameterised requirement that takes a value denoting a time limit for establishing a mobile phone connection.

**REQ 6.2** Parameter

The mobile phone shall establish a connection to its telecommunications provider in no longer than $X$ seconds.
We could rewrite requirement 6.2 to express the numeric variability in terms of qualitative variability. Instead of selecting a parameter, the connection time can be given as a finite (but extendible) set of mutually exclusive alternatives. Extra single adaptor options may be added if the desired connection time is not listed as an alternative.

**REQ 6.2** The mobile phone shall establish a connection to its telecommunications provider in within a set time.

**REQ 6.2.1** The connection must be established within 1 second.

**REQ 6.2.2** The connection must be established within 2 seconds.

**REQ 6.2.3** The connection must be established within 2.7 seconds.

**REQ 6.2.4** The connection must be established within 3 seconds.

**REQ 6.2.5** The connection must be established within 3.1 seconds.

... ... ...

This same principle can be used to model requirement 6.2 in application family design. Figure 4.12 shows how qualitative variability can model quantitative variability; the alternative connection time limits are accessed through a common interface. The design uses process-flow diagrams according to Yourdon [4.21] and is based on our single adaptor pattern from Figure 4.9 to model qualitative variability.
Using common interfaces to denote quantitative variability in terms of qualitative variability can help maintain consistency throughout an application family design as all points of variability in the design use the same mechanism.

4.5 Reference Architectures and Application Family Frameworks

Components in an application family design are often presented as groups or sub-collections with a public, common interface. Some common interface approaches have an underlying framework (Fayad et al [4.2]) or a reference architecture (DSSA [4.10], PuLSE [4.18]), where the desired components, with the correct common interface, can plug into.

Matsumoto et al [4.22] define a reference architecture as a core portion of all systems of a family, which represents common structure, functionality, non-functionality and variability. Reference architectures are often ad-hoc, high-level conceptual representations of an application family. Typically, common interfaces are defined at points of variability in the reference architecture. New systems are
composed by plugging the required components, chosen from lists of alternative components that conform to the common interface, into the reference architecture.

Fayad et al [4.2] suggest that application family frameworks are often based on reference architectures. While reference architectures denote core components and application family variability at a conceptual level, a framework is less abstract and normally very code oriented. Application family frameworks represent a more detailed design of core components for systems of a family. They are typically aimed at a specific design method or programming language.

![Diagram of application family scope and single systems](image)

**Figure 4.13:** Example application family scope and single systems within that scope that share a common core

However, both reference architectures and application family frameworks assume that all systems in the application family share a large common core. We can demonstrate this using a Venn diagram. Figure 4.13 illustrates the boundaries (the scope) of a family of applications; four systems of that family are shown. The grey area in Figure 4.13 demonstrates the core portion of all four systems: the reference architecture; or application family framework. The problems of having a central core, which is common to all systems within a family, are usually related to scoping and
defining reference architectures and application family frameworks (Schmid [4.23]), such as:

- Defining a scope, and a corresponding core, that is too large may result in few resources being available to successfully complete individual systems. It may add extra complexity and costs (to the reference architecture or application family framework), which can result in added risks and a possible reduction in the overall benefits.

- A scope that is too small may limit the overall number and type of systems that can be derived from a reference architecture or framework. Systems may require additional work, as their core is too inflexible to satisfy all requirements.

Reference architectures and application family frameworks have demonstrated a great deal of success for families with a well-definable scope. However, with poorly defined common cores there is a danger that the scope may end up being too large or too small. Our belief is that systems can still reside in the same application family, even without core assets that are common to all systems. Figure 4.14 shows the scope of an application family, where, although some systems share common assets, no asset is shared by all systems.
Figure 4.14: Example application family scope and single systems within that family that have no common core

An example of an application family where not all systems share a common core can be demonstrated using systems that are derived from part of an MRAM application family model for a family of mobile phones:

**REQ 1**  The mobile phone software shall use one of the following operating systems.

*Single Adaptor* Operating system type A.

*REQ 1.1* Operating system type B.

*REQ 2* The mobile phone shall consist of one of the following software packages.

*Single Adaptor* Software package type 1. (such as LCD support and calling features ABC).

*REQ 2.1* Software package type 2. (such as no LCD support and calling features XYZ).

Consider three mobile phone systems that are derived from our example application family (system α, system β and system χ). System α includes operating system A with software package type 1, system β also uses operating system A with
software package type 2, and system $\chi$ uses operating system B with software package type 1. Figure 4.15 shows a Venn diagram of the commonalities between system $\alpha$, system $\beta$ and system $\chi$. We can observe that while the systems are arguably all a part of the same application family, they do not share a common core.

![Venn diagram](image)

**Figure 4.15**: Example application family of mobile phones that do not share a common core

While Figure 4.13 and Figure 4.14 show two extremes, the typical scope of an application family is likely to lie somewhere in between the two. While there may be some form of core, an application family may equally have functionality that is common to a number of systems, but not to all of them.

Johnsson *et al* [4.24] support our argument of not using reference architectures and application family frameworks. Their work concentrates on quantifying the cost and feasibility of maintaining reference architectures, as the business needs of the application family evolve. They imply that the overall benefits of using application family engineering can be lost if reference architectures must change in response to new or modified business requirements before the investment costs of the architecture have been recouped.
In our application family designs we make no explicit requirement for reference architectures or frameworks. In our designs, a point of variability – a common interface – may manage variability at a very high architectural level right down to a detail design component. We feel that there is no need for the existence of core components that exist in all systems of a family as this may inhibit evolution that is driven by changing business objectives.

4.6 Evolving Application Family Designs

An important aspect of application family design is reengineering to evolve application family designs. This section describes how reengineering may be used to aid the integration of existing on-off system designs into an application family design (ploughed-field evolution) and how application family design components may be reengineered to incorporate additional points of variability. We will see in Chapter 5 how this impacts on our thesis.

Chikofsky et al [4.25] define software reengineering as the examination and alteration of a subject system to reconstitute it in a new form and subsequent implementation of that form. The result of a reengineered subject system is a new system or subsystem with the same functionality as the original one, but for use in a different solution-space. Subject systems are typically split into several abstraction levels, each of which represents a particular view of the subject system.

Although reengineering is most commonly used as a process to help to understand and model large legacy systems that lack documentation, it may be applied during two processes of application family design: the first is to aid the integration of mined one-off design components; and the second is to help the evolution of application family design components.
Organisations may choose to adopt a reuse approach when existing systems, even legacy systems, are identified as being within the same application family (*ploughed-field* approach). Under such circumstances, many resources may already exist to help construct application family assets. These assets usually span from system specifications to source-code and executables, although, at this stage, we are only interested in design components. Design components that are considered directly reusable rarely pose a problem. However, consider the situation, where a component is identified as a point of variability. Typically a common interface is *reengineered* for the component to enable its reuse in systems that require it and its replacement when alternative functionality is required.

Existing application family design components may evolve. As the business objectives of an application family change, components (or even part-components) that were previously thought to be directly reusable may evolve and become variable. A reengineering process can be used to add a common interface to such components.

Software reengineering is well researched and applied in many domains (such as Biggerstaff [4.26], Britcher [4.27], Brodie *et al* [4.28], Schwanke [4.29], Müller *et al* [4.30], DeBaud *et al* [4.31]). The individual examples vary in the type of asset that is reengineered; these include requirements documents, design assets, code components, and business procedures. However, the concept in context with application family design is generally the same. We can demonstrate this concept using a simple example.

Figure 4.16 shows a process flow diagram for two, over-simplified, example one-off system designs. Both systems are assumed to be in the same family of applications. A comparison of the two systems reveals one point of variability (marked in grey).
To make the possible variants more maintainable, reengineering is required. Typically a common interface to the variant is wrapped around the components, providing a more loosely coupled interface to the variability. Figure 4.17 shows how an application family may be constructed from the example one-off system designs in Figure 4.16. The process node marked CI (short for common interface) in Figure 4.17 provides the reengineered common interface to the one point of variability using the single adaptor design pattern from Figure 4.9.
A systems engineer will select the required alternative functionality at the common interface at build-time. Figure 4.18 shows how the common interface might be implemented in C; pre-processor definitions may be used to complement build-time selection of the required alternative (if no other mechanisms are available).

```c
#define ALTERNATIVE 1 /* or 2 */
void CI( ... )
{
    #if ALTERNATIVE == 1
        /* Call routines for alternative 1 */
        ...
    #else
        /* Call routines for alternative 2 */
        ...
    #endif
}
```

**Figure 4.18:** Example C implementation of common interface CI from Figure 4.17
The process of reengineering existing, directly reusable application family design assets as they are evolved into new points of variability, is exactly the same.

4.7 Summary

In this chapter we introduced the notion of application family design. We present a taxonomy of five techniques and mechanisms for handling application family variability in application family designs and implementations, and justified using a common interface approach for our experiments.

We also presented three patterns that capture the discrimination mechanisms of MRAM into common interfaces to help model variability in application family designs. These patterns can be adapted to cover many design methods and programming languages. We will make extensive use of these patterns in later chapters when we experiment with different approaches to modelling application family designs.

In the next chapter we introduce embedded systems and highlight problems that are specific to application family design of embedded systems (with a particular focus on reengineering). These criteria will become the foundations of our research.

4.8 References


CHAPTER 5

Embedded Systems

5.1 Introduction

In the previous chapters we introduced application family engineering and design concepts. This chapter will provide an insight into embedded systems and explain the issues governing reuse in embedded systems. We will outline some of the technical challenges that are specific to application family engineering of embedded systems. One of these – the perceived introduction of execution time and memory space overhead – will become the focal point of our research.

Embedded software systems typically run on dedicated hardware, and are systems in which a primary objective is typically to control external devices. Embedded software systems are found in a number of applications; such as in aircraft flight control, reactor protection systems, medical electronic devices, washing machines, and mobile phones. A lack of direct user involvement or supervision requires highly reliable and efficient software, which must typically work in real-time, as embedded systems respond to and control real-world events (Calvez [5.1], Nissanka [5.2], and Lear [5.3]).
5.2 An Overview of Embedded Systems

Embedded systems are not new; engineers have been integrating computing power into electronics for more than two decades. What is new is that technologies such as distributed computing, intelligent systems, mobile computing, networking, and the Internet have begun to expand both the reach and the power of embedded systems.

Until recently, the design processes of embedded systems have placed more emphasis on the hardware design, than its software. Quick software-hacks produced small programs, which told the hardware how to operate. Embedded systems were typically based on 8-bit processors and were programmed in assembly language to define an immutable set of functions; they were simple programs that brought some intelligence to mechanical things.

Today, embedded systems typically use 32-, 64- or even 128-bit processors, and are mainly programmed in a higher-level language such as Ada, C and C++. In many cases they do not just run equipment but often inter-operate with other systems. This requires that an embedded system gather, partially process, and transmit data. Typically, embedded software provides between 80-90% of the overall functionality, and development costs, of an embedded system (Diaz-Herrera et al [5.4]).

As the hardware of embedded systems evolve, the software demands grow with it. This makes the choice of an engineering technique, programming language, and platform for an embedded system very significant. Most embedded systems have an operating system that forms the core (and in some cases the whole) of its software. Unlike desktop computing, the embedded market currently supports many different operating systems. These are often cut to a bare minimum, as they are usually required to be highly reliable and compact. Some processors come with proprietary operating systems, while others may use commercial operating systems such as QNX, VxWorks, LynxOS, pSOSystem, OSE, and Windows CE.
Many embedded systems follow stringent constraints for real-time operation, and safety (Selic et al [5.5]). A real-time constraint dictates the time limits within which a system must perform its functions. Real-time constraints, like most constraints, usually start as non-functional requirements that relate to properties of the physical world (Loucopoulos et al [5.6]). These high-level real-time requirements are typically mapped down through the levels of system design to apply to derived requirements for the software components. Examples of real-time requirements may include:

- The system shall take no longer than 10 seconds to execute.

- Each acquisition cycle shall not exceed 2 seconds.

- Following error 5122, the recovery sequence must take no longer than 0.5 seconds to activate and no longer than 3 minutes to complete.

Different systems have different requirements for meeting deadlines. At one end of the spectrum are hard real-time systems, where missing even a single deadline is considered unacceptable. Examples of hard real-time systems can be found in life-critical systems, such as nuclear power stations, medical equipment, and aircraft control. At the other end of the spectrum are soft real-time systems where missing a deadline occasionally is acceptable. For example, telephony systems contain a requirement that a subscriber will receive a dial tone within a certain time period of lifting the handset. Failure to produce a timely dial tone is not catastrophic, provided the frequency of failure is within acceptable boundaries.

Physical space constraints and cost constraints often limit the number of processors and memory devices that an embedded system can contain. A typical example is a satellite system where overall size constraints limit the hardware and thus the computing power. The software is still required to interact with the same number of
onboard devices and to the same real-time constraints. Constraints on memory usage are imposed when a software’s memory consumption is close to its physical limit. They dictate the memory limits within which a system must operate. For example:

- The module’s memory usage shall not exceed 50 kilobytes.
- The processing queue may not exceed 30 items.

Execution time and memory space constraints are often heavily associated. Improving the execution time of a system to meet deadlines may be at the expense of memory space, and vice-versa. Using techniques like pre-emptive memory allocation instead of dynamic allocation may increase run-time performance, but more memory will be required. For example, a function may calculate the cosine of any given angle. Pre-allocating a look-up table of angles against their cosine values can speed-up the function, however the look-up table will require additional memory space.

Real-time specification languages typically formalise expressions for timing and memory constraints and in some cases allow proofs of program properties based on these constraints. Examples of specification languages include ACSR [5.7], GCST [5.8] and RTL [5.9]. Based on these formalisms, real-time programming languages will check assertions or even optimise code to ensure adherence to timing constrains. Example real-time programming languages include Tomal [5.10], Pearl [5.11], Real-time Euclid [5.12], RTC++ [5.13], Real-time concurrent C [5.14], Chaos [5.15], Flex [5.17], TCEL [5.17], Ada95 [5.18], MPL [5.19], and CaRT-Spec [5.20].

Safety constraints are imposed when the behaviour, or failure, of a system can lead to accidents involving loss of life, injury or environmental damage. Safety requirements usually relate to the physical world. For example, consider some requirements for the control of water levels in a water tank. Examples of safety constraints might include:
- The water level shall not exceed 12 litres.

- If the water tank overflows, an alarm shall sound.

- There shall be a backup alarm system.

Safety requirements often specify the required system redundancy, back-up facilities, and levels of exception handling. Safety constraint (and other quality attributes including functionality, security, interoperability, and efficiency) will often dictate the *hardness* of a real-time system. The greater the required safety, the harder the deadlines.

Safety does not imply correctness. A system may be incorrect, and unreliable, but safe because its failure modes are not hazardous. Similarly a system may be correct, but unsafe, due to errors in the specification (McDermid [5.21]).

Unlike execution time and memory space constraints, safety is an issue that has not yet been addressed by many programming languages. Safety analysis is traditionally a manual process, though recent work focuses on formal specifications as the basis for partial automated hazard analysis (Ratan et al [5.22], Liu et al [5.23]). Safety analysis is generally performed using semi-formal hazard analysis techniques such as Fault Tree Analysis [5.24]. Such techniques are used to decompose an identified environmental hazard into its causal failure modes, where the failure modes relate to software and complementary safety requirements can be derived. Assurance of software safety is then generated by demonstrating that the software specification satisfies the derived safety properties and that the specification is implemented correctly ([5.25], [5.26], [5.27]).

Considerable work has been undertaken to improve the process and project management of software development projects over the last 10-15 years. However, embedded systems development has evolved in isolation from mainstream software
applications development. The traditional light software requirements of embedded systems gave rise to the perception that the overhead of a formal development process and management was not cost effective. Whilst informal ad-hoc development may have produced working solutions two decades ago when the software did not exceed 1’000 lines of code, embedded systems today often exceed one million lines of code, and this approach is no longer efficient or reliable. Recent publications have shown the growing need for process and project management in embedded systems (Bergsma [5.28], Brands [5.29]).

According to a study by market researchers BCC [5.30], the US embedded systems market is expected to increase by an average annual growth rate exceeding 13%, from $32 billion in 1998 to over $66 billion in 2004. Embedded systems have shifted from simple industrial automation to a variety of new markets including:

- **Consumer electronics** (such as TV set-top boxes, in-vehicle computing devices, consumer digital cameras, gaming consoles, handheld PCs, and microwaves).

- **Retail automation** (such as point-of-sale systems, bar-code readers, and interactive kiosks).

- **Medical devices** (such medical stations for the home, portable care stations, and non life-critical devices).

- **Telecommunications** (such as smart phones, and WAP protocols).

The expansion of business objectives to new fields is a driving factor for smarter, more functional and flexible systems. Through increasing competition the embedded systems industry is turning to mainstream development methods, seeking for ways to reduce development costs and delivery times. The case for reuse has never been stronger.
5.3 Reuse in Embedded Systems

Reuse and management methods developed for the mainstream market incorporate few processes that consider real-time, memory usage or safety effects of a reused component.

Attempts have been made to build embedded systems, using mainstream application family engineering methods (JODA [5.31], FODA [5.32], ODM [5.33], and DSSA [5.34]). None of the examples mention any consideration for time, space and safety constraints. We can only assume that the importance of the constraints was secondary to the reuse process, thus not explored.

There are a number of reasons why many reuse techniques are not easily applied to embedded systems, including many technical drivers (Mrva [5.35]). Section 5.3.1, 5.3.2 and 5.3.3 address some of these technical issues.

5.3.1 Highly Coupled Designs

Tight execution environments often mean that embedded system designs are highly coupled. One reason for this can be the usage of optimisation techniques to help meet constraints. For example, consider a function in an application family of an embedded system that calculates the cosine of an angle.

![Figure 5.1: Timeline of cosine function before optimisation](image)

Figure 5.1: Timeline of cosine function before optimisation
Figure 5.1 shows a typical sequence of a function to calculate cosine values for a given angle, before any optimisation. On each execution of the function its memory requirements are allocated; a cosine value is calculated; and following the operation, the memory is then freed. Optimising the function for performance, using a pre-allocation technique, will shift much of the memory allocation and processing from run-time to system initialisation.

Figure 5.2 shows a pre-allocation solution of the cosine function in Figure 5.1. A look-up table is created during system initialisation containing many angles and their corresponding cosine values. At run-time, rather than calculating cosine values with every call to the function, only a look-up is performed, typically increasing performance. We can observe a significant increase in coupling. Functionality that was grouped into a single function in Figure 5.1 is spread out over a number of execution phases in Figure 5.2 (including system initialisation, run-time and system deallocation phases). This phenomenon is primarily considered a problem with the inflexibility of existing real-time design methods (Askit et al [5.36], Burns et al [5.37]).

A second reason for the high coupling in embedded designs is outlined by Young [5.38]. Safety is often a prime requirement of embedded systems. Type-safe and structure-safe programming languages (such as Ada) and corresponding designs are often used to shift much of the error detection from run-time to build-time. The
overall safety of the software is normally increased, as many errors can be detected early. There is also no need for extra code that performs run-time type checking. However, type-safe and structure-safe software are usually very highly coupled; a dependency between software units is not just functional, but structural and type dependant too. Reducing the type-safety and structure-safety is often seen as one way of aiding flexibility. However, this can compromise the reliability of a system and is often omitted.

In Section 4.2.5 we argued that the application family engineering process tries to promote loose coupling for variant assets. In environments where execution time and memory space are important constraints, like embedded systems, loose coupling may not always be achievable. We discuss this issue further in Chapter 8.

5.3.2 Existing Constraints Restrict Possible Alternatives

An execution time, memory or safety constraint imposed on a reusable asset may not be applicable to all its variant forms. Constraints are usually specified as formalisms (including ACSR [5.7], GCST [5.8], RTL [5.9]) or natural language requirements. Consider a point of variability in an application family implementation that is subject to such a constraint. The constraint may restrict possible alternatives, which are unable to comply with it.

For example, an application family of mobile phones may include different connection mechanisms, including one-to-one phone calls and three-way calling. The calling mechanisms of all systems derived from the application family may be subject to a memory constraint; for example, all calling mechanisms must run within a 100 kilobyte memory limit. If, for example, the one-to-one mechanism requires 80 kilobytes of memory and the three-way calling mechanism requires 90 kilobytes, then an extension to include a multi-way, conference calling alternative at 125 kilobytes is not possible without restructuring the application family design and implementation.
While MRAM can provide a mechanism for varying constraints, it does not check to ensure that these constraints can be met. The following example demonstrates how the mobile phone can support different constraints for different discriminant options.

<table>
<thead>
<tr>
<th>REQ</th>
<th>The mobile phone shall support at least one of the following calling mechanisms:</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td><strong>Multiple Adaptor</strong></td>
</tr>
<tr>
<td>8.1</td>
<td>One-to-one calls.</td>
</tr>
<tr>
<td>8.1.1</td>
<td>One-to-one calls require a 100 KB memory frame.</td>
</tr>
<tr>
<td>8.2</td>
<td>Three-way calling.</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Three-way calling requires a 100 KB memory frame.</td>
</tr>
<tr>
<td>8.3</td>
<td>Multi-way, conference calling.</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Multi-way, conference calling requires a 125 KB memory frame.</td>
</tr>
</tbody>
</table>

Many application family engineering methods do not start specifying reusable assets until a design stage (such as RSEB [5.39], JODA [5.31]). By this time constraints are usually already heavily associated with individual components and can be hard to transfer to variant parts. Ravindran *et al* [5.40] have produced a reusable specification language for real-time systems called Palette, which is able to reason with varying constraints in much the same way as MRAM. However, Palette specifies an application family model using formalisms rather than natural language requirements.

The issue of *existing constraints restricting alternatives* is beyond the scope of our research. Rather than focus on the anticipation and resolution of constraint breaches, our work will concentrate on potential constraint breaches during creation and evolution of an application family – the problem of perceived performance and memory overhead.
5.3.3 Performance and Memory Overhead

There is a perception that reuse techniques may introduce performance and memory overhead into systems. This may not be desirable if stringent time and memory constraints are present.

In Chapter 4 we observed that common interfaces at points of variability are not present in systems that are built without reuse (no points of variability). These extra design concepts may introduce extra source-code into single systems built from an application family. It is often perceived that this extra source-code will delay execution time and require extra memory. However, there is little empirical evidence to support this claim. Our research will focus on investigating the possible execution time and memory space overhead that may be introduced by the addition of common interfaces at points of variability.

In Chapter 4 we observed how reengineering can aid the creation and evolution of application family designs. Reengineering introduces extra common interfaces to application family design components or existing one-off system designs drawn into an application family, that are not present in systems that are built using a traditional on-off approach.

Consider a component in an existing one-off system, or a directly reusable component, both with stringent timing constraints that are required to evolve and become points of variability in an application family design. A typical procedure is to reengineer a common interface for that component to aid maintenance of its variant forms. If we assume that a common interface introduces execution time and memory overhead, the stringent real-time and space constraints imposed on the components may be compromised. We can demonstrate this by example.

Figure 5.3 shows part of a design for an existing one-off system that shall be reengineered into a variable application family design asset. The figure illustrates that
a time constraint is imposed: processes 7 to 10 must complete their tasks in no longer than 5 seconds. Figure 5.3 also contains a loop; Nodes 8 and 9 are looped 1’000 times.

![Diagram](image)

**Figure 5.3:** Example one-off single system with a time constraint

Let us assume that in Figure 5.3, Process 9 has been identified as a possible point of variability. Figure 5.4 shows how we can reengineer a common interface for Process 9. We have applied the design pattern to model a single adaptor discriminant (Figure 4.9). In Figure 5.4 all access to our point of variability (formerly Node 9) is now through a common interface node marked CI. Selecting the appropriate alternative will result in the desired single system. For example, selecting Alternative 1 will result in our original system, but with an extra function call – the common interface node CI.
Figure 5.4: Example application family solution with reengineered common interface for process 9

The time constraint that applies to the one-off system processes 7 through 10 (in Figure 5.3) will also apply to the reengineered application family equivalents, at the very least when Alternative 1 is required. It may even apply to alternatives 2 to $n$.

We can observe in Figure 5.3 that Process 9 is inside a loop and is repeated 1’000 times per system cycle. In the application family equivalent in Figure 5.4 the common interface to Process 9 is also called 1’000 times (which will in turn call Process 9 or one of its variants).
Table 5.1: Example average execution times of processes (in seconds)

Table 5.1 shows average execution times for a cycle of both our original one-off system and the reengineered application family equivalent using Alternative 1. If we assume, hypothetically, that the common interface introduces a relatively small execution time overhead (such as 0.00001 seconds) then the application, over 1’000 loops, breaches the initial time constraint that was imposed on the components of the original system.

One way to solve the constraint breach in our example is to include the whole loop in the variable alternative, so that the common interface is only called once. Figure 5.5 shows how the application family from Figure 5.4 may be changed to enable it to meet the 5-second time constraint. Techniques for restructuring application family designs to meet local constraints, if common interfaces cause constraints to be breaches, are beyond the scope of our research.
Figure 5.5: Reworked application family that can meet original time constraint

We have demonstrated one example of how execution time may potentially be affected by mining application family design and code assets from existing one-off systems, or evolving existing application family design and code components. This will be the focal point of our research; investigating potential execution time and memory space overhead that may be introduced by reengineered common interfaces.

5.4 Experiment

In Lewis et al [5.41] we present a trial experiment to inform our experimental design in Chapter 6. The trial experiment is designed to provide an initial feel for execution time overhead of a common interface technique in a small, controlled
setting, using real-life data. While the experiment only focuses on object-oriented common interfaces, it provides able foundations for more in-depth experimentation.

In Chapter 4 we describe a taxonomy of techniques to realise application family variability. In [5.41] we compare the execution times of three implementations of a system, each using a different technique to realise variability. Our trial experiment compares a parameterisation approach (structured decision tree) to two common interface approaches (static polymorphism and dynamic polymorphism).

The programming community often assumes that object-oriented languages suffer from performance problems when compared with structured languages (Weinstein [5.42], Hölzle et al [5.43]). This experiment was prompted by the industry concerns that polymorphic function calls, are unnecessarily slow (Rowe [5.44]). Polymorphic function calls do additional processing that standard function calls do not do. At the assembly language level, the calling mechanism that provides the polymorphic facility needs to perform a tag lookup, which requires increased processing time [5.45]. This is not visible to the programmer, though it is this increase in processing time that is perceived to be one barrier in the adoption of polymorphism as a discrimination mechanism for real-time, embedded systems development. Hence, many organisations will emulate the mechanism using a parameterisation approach, like structured decision trees typically using switch and case statements.

We show that polymorphism in an object-oriented language can outperform a structured decision tree. We conducted an experiment in which we compare the performances of a same telemetry validating module designed using the three different techniques.

The Rhea Group Space Division is a software development provider to the space industry, who have developed skills in constructing software for satellite instrument control units (ICU). A satellite ICU performs the data handling, monitoring and
control of on-board scientific instruments. Rhea are keen to develop a set of reusable ICU components to enable them to be more competitive on price and time to market. They were interested in this experiment because although they use an object-oriented language (Ada95) they have avoided using its polymorphic language features.

![Diagram of ICU process model]

**Figure 5.6: Simplified ICU process model**

Figure 5.6 shows an ICU Controller being commanded, from Earth, through macro commands. The Controller initialises the Monitor process, which runs continuously. The Controller also initiates data-acquisition from the Data Handler, which acquires telemetry-data from the onboard instruments. It relays the data to Earth and to the Monitor, which validates the telemetry data. A telemetry status is reported to the Controller and appropriate action is taken. For example, if an instrument needs to be turned off, a shutdown message is sent from the Controller to the Data Handler. The requirements of the monitoring module are [5.46]:

**REQ 1** Monitoring is run as a cyclic timer based event, and there is a thread that performs the monitoring function. After a collection cycle the values of monitored telemetry variables are checked and the required shutdowns are performed.
REQ 2 The monitored telemetry variables consist of six Boolean, thirty-two 16-bit integers and two 32-bit integers.

REQ 3 The Boolean and the 32-bit integer telemetry variables shall be monitored against an expected status.

REQ 4 The 16-bit integer telemetry variables shall be monitored against maximum and minimum limits.

REQ 5 The values of the monitored telemetry variables shall be compared against limits or expected status values. If an out-of-limit or unexpected status is detected then a log entry shall be made and shutdown enabled.

There are wide ranges of satellite instruments, even for performing the same task. These instruments will typically come from different manufacturers and will carry out the same tasks in different ways, or even implement a slightly different task. An application family of instrument control units must respond to this change. In our paper [5.41] we identified two common ways in which this module is most likely to change:

- In **REQ 2**, the monitored variables may vary in number and type.

- In **REQ 3** and **REQ 4**, different types of validation may be required for the variable types.

We implemented three single systems in Ada95; all three were derived from three different application families. Each of the three application families used a different discrimination mechanism at points of variability: structured decision tree; static polymorphism; and dynamic polymorphism, respectively. To evaluate the performance of each single system we measured the average processor time taken to collect data from the Data Handler and validate it by the Monitor. We used a test harness to emulate the Controller and Data Handler and to invoke the Monitor.
Three experiments were conducted for each of the discrimination mechanisms. In each experiment we varied the number of monitoring cycles. In each cycle the Controller calls the Data Handler to acquire data from a simulated data instrument. The data is passed to the Monitor, which then validates the data and sends a message to the Controller. The system clock was used to measure the execution time for acquiring the data and validating it.

The unit of measurement of the clock is 10 millisecond. A typical cycle time is of the order of one millisecond. To eliminate any inaccuracies due to the difference in resolution of the unit of measurement we ran experiments of 1’000, 5’000 and 10’000 monitoring cycles.

In many real-time, process monitoring experiments, there will be slight differences in absolute values recorded. This can be attributed to background tasks being performed by the operating system, or other concurrent tasks. To reduce the effect of these differences we ran each of the 1’000, 5’000 and 10’000 cycle sets three times for each technique.

5.4.1 Experimental Results

Table 5.2 shows the average execution time per cycle of each technique for each set of 1’000, 5’000 and 10’000 cycles.

<table>
<thead>
<tr>
<th>Method</th>
<th>1’000 cycles</th>
<th>5’000 cycles</th>
<th>10’000 cycles</th>
<th>Average per Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision tree</td>
<td>2.18 s</td>
<td>10.90 s</td>
<td>21.77 s</td>
<td>0.002178125 s</td>
</tr>
<tr>
<td>Static</td>
<td>0.68 s</td>
<td>3.40 s</td>
<td>6.79 s</td>
<td>0.000679375 s</td>
</tr>
<tr>
<td>Dynamic</td>
<td>1.54 s</td>
<td>7.69 s</td>
<td>15.39 s</td>
<td>0.001538750 s</td>
</tr>
</tbody>
</table>

Table 5.2: Performance results
The average execution time per cycle for the static polymorphic solution is 220.6% faster than the structured decision tree solution, and the dynamic polymorphic solution outperforms the structured decision tree solution by 41.6%.

5.4.2 Discussion

Our results in Table 5.2 show that polymorphism can outperform a structured decision tree. More importantly, from this work we can draw the following conclusions:

- The perception that a common interface approach introduces an unacceptable performance overhead is not always well founded and at least merits further investigation.
- Different mechanisms for implementing a point of variability can produce very different performance results. To better understand these performance differences we need to conduct further experiments for a variety of styles.
- Our experiments only account for a fixed number of points of variability (two points). Further investigation is required to evaluate the common interface mechanisms when the number of points of variability changes.

5.5 Summary

Technologies, such as distributed computing, intelligent systems, mobile computing, networking, and the Internet have begun to expand the scope and the power of embedded systems. The development costs for embedded systems have shifted from hardware to software. Traditional development approaches for embedded systems software are not sufficient to cope with the required flexibility and
maintainability of today’s market. Application family engineering is seen as one solution.

In this chapter we have highlighted three technical problems that are associated with application family engineering of embedded system: highly coupled designs can make incorporating variability difficult; existing constraints can restrict possible alternatives; and a possible performance and memory overhead may inhibit reuse.

Common interfaces are a popular technique for introducing flexibility and reuse into software designs. However, they may introduce an undesirable execution time and memory space overhead. Our trial experiments in [5.41] shows us that different mechanisms for implementing a point of variability can produce very different performance results. Based on this evidence, we are assuming that any potential overhead that is associated with common interfaces will differ with platform, architectural style, design method and programming language.

Our hypothesis is that there exists a correlation between execution time and memory space overhead that reuse may introduce and the amount of variability incorporated into a single system. Our research is aimed at showing that there is a high probability that such a correlation will exist for different common interface approaches, on different platforms, in different architectural styles, design methods and programming languages. Chapter 6 will detail an experimental approach to substantiate this claim.

5.6 References


CHAPTER 6

Experiment Design

6.1 Introduction

This chapter describes the rationale and design of the experiments for which results are reported in Chapter 7. We will investigate the possible introduction of execution time and memory space overhead for a number of different common interface methods. It is not feasible to examine the performance of every possible common interface approach on every platform, using every architectural style, design method and programming language. We have chosen to examine the behaviour of five application family design methods, based on five common architectural styles, each using a very different type of common interface at points of variability. We believe that grouping our experiments by different common architectural styles will provide a more representative account of the behaviour of potential overhead. A correlation between overhead and number of points of variability for all five styles would be a very strong indication that other common interface approaches correlate similarly.

In Section 6.2 we introduce the five architectural styles on which we base the application family designs in our case studies. We will demonstrate examples of how our universal design patterns in Chapter 4 can be used with the components of each
design method. Later sections will detail our intended experiments and evaluation techniques.

6.2 Software Architectures

There is currently no universally accepted definition of software architecture. Typically an architecture describes a software system’s structure as a decomposition into elements and their interconnections. The IEEE ECBS TC Architecture Working Group’s definition of architecture reads [6.1]:

- A system’s essential, unifying structure defined in terms of components, connections and constraints along with the system’s interaction with its environment.

6.2.1 Architectural Styles

The IEEE ECBS TC Architecture Working Group’s definition of an architectural style is [6.1]:

- The language, notation and structuring guidelines for representing system architectures. They provide specific analysis based on the formality of the system representation.

An architectural style provides the formal building blocks of an architectural design. There are a finite number of commonly referenced architectural styles. Many of these styles are based on common design and programming methods. For example, the Main/Subroutine style (Shaw et al [6.2]) is based on structured, functional design methods. Some architectural styles model systems at such an abstract level that their components are not easily mapped to components of a design or programming
language. As these architectural styles are not based on design methods or programming languages, their components are unlikely to be able to implement common interfaces to points of variability. One style like this is arguably the *Layered* architectural style [6.2].

Sections 6.2.2 to 6.2.6 describe the five architectural styles that we group our experiments by. Each section outlines a design method that is representative of each respective style. We also demonstrate examples of how the universal design patterns from Chapter 4 may be applied to denote variability in each of the design methods. Table 6.1 summarises the architectural styles and design methods that we have chosen for our experiments.

<table>
<thead>
<tr>
<th>Section</th>
<th>Architectural Style</th>
<th>Design Method used in our Case Studies</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.2</td>
<td>Main/Subroutine</td>
<td>Process-flow diagram</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Data-Abstraction, Object-Orientation</td>
<td>UML</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Pipe and Filter</td>
<td>Pipe and Filter</td>
</tr>
<tr>
<td>6.2.5</td>
<td>Event-Based, Implicit Invocation</td>
<td>State-transition diagram</td>
</tr>
<tr>
<td>6.2.6</td>
<td>Repository</td>
<td>Data-flow diagram</td>
</tr>
</tbody>
</table>

*Table 6.1: Experiments grouping by architectural style*

### 6.2.2 Main/Subroutine Organisations

Shaw *et al* [6.2] note that the organisation of systems often reflects the programming language in which a system is written. Languages that do not support high-level abstraction often result in an organisation around a main program and a set of subroutines. The main program typically provides a control loop, which sequences the subroutines in the correct order, where the subroutines provide the necessary modularization. Most programming languages are capable of implementing this architectural notation. An illustration of this architectural style is typically in the form of a process-flow diagram, where subroutines are modelled by processes (such as Yourdon [6.3], MASCOT [6.4], DeMarco [6.5]). Figure 6.1 illustrates an example
Main/Subroutine architectural style. It shows a main program that sequences the required subroutines, and calls them in the required order – Subroutine 1 then Subroutine 2. The subroutines themselves may, in turn, sequence other subroutines, creating a hierarchy of processes.

**Figure 6.1:** Example Main/Subroutine architectural style

We will investigate the behaviour of potential performance overhead using process-flow diagrams from Yourdon’s design method [6.3]. Points of variability in the application family design are encapsulated by a function that provides a common interface to variant solutions. For example, Figure 6.2 shows a washing machine case study in which function CI_PreWash provides a common interface to a washing machine’s pre-wash phase alternatives. At build-time the systems engineer must specify which version of pre-wash is applicable, *pre-wash 30°* or *pre-wash 40°*. Figure 6.2 applies the single adaptor pattern defined earlier in Section 4.9.
6.2.3 Data Abstraction and Object-Oriented Organisation

Object-orientation is widely used, with many different methods currently available (OMT [6.6], Booch [6.7], ROOM [6.8], UML [6.9], Shlaer-Mellor [6.10], Fusion [6.11], and Jacobson [6.12]). In an Object-oriented architectural style, data representations and their operations are encapsulated in an abstract data-type or object. Shaw et al [6.2] refer to objects as managers because they are responsible for maintaining the integrity of a resource (usually by preserving some invariant over it). The representation and detail of a particular object is hidden from other objects. To access a particular object’s functionality or data, other objects only need to know its interface. Objects interact with each other by invoking member functions. These invocations provide the architectural connectors.

Figure 6.3: Example Object-oriented architectural style
Figure 6.3 shows an example of an Object-oriented architectural style with four objects (A, B, C and D) and their procedure invocations. We are assuming that an object’s member data is always private, and totally hidden from other objects. Member data may be accessed only through the object’s member functions. For example, in Figure 6.3, Object D might consist of member data q and p. So that B can access q and p, the encapsulating object D should provide functions like Get_q, Set_q, Get_p and Set_p. This is important as it provides us with a purely functional, and thus consistent, interface. This method of using access functions to read and write an object’s attributes follows guidelines by Mili and Li [6.13].

Our second case study uses UML [6.9] as a representative design method of the Object-oriented architectural style. Points of variability are captured through an abstract base-class which provides a common interface to variant solutions. Parameterised classes are instantiated with the appropriate subclass that implements the desired variant functionality. For example, Figure 6.4 shows a parameterised class called Mobile_Phone, taken from an application family design the family of mobile phones in Section 4.2.5.

![Diagram](image_url)

**Figure 6.4:** Example base-class providing a common interface to variant assets
It takes one parameter; a sub-class of the abstract base class Carrier. Systems are built by instantiating Mobile_Phone with the appropriate variation of Carrier: CarrierX1 or CarrierX2. Figure 6.5 shows an example instantiation of Mobile_Phone. The desired variant that was selected is sub-class CarrierX1, which is derived from the abstract base-classes Carrier.

![Diagram](image)

**Figure 6.5:** Example instantiated mobile phone

For this case study we do not need to apply our universal design patterns to model points of application family variability. UML and object-orientation provide their own mechanisms to support build-time variability using common interfaces – static polymorphism, as demonstrated in Figure 6.4.

### 6.2.4 Pipe and Filter

Figure 6.6 shows a Pipe and Filter style consisting of a set of components (that is, filters) each with a set of inputs and outputs (that is, pipes). A component’s inputs and outputs are transmitted streams of data. Outputs may be produced before the entire input is consumed - hence components are termed filters. The connectors linking the output of one filter to the input of the next filter are termed pipes.
Important characteristics of the Pipe and Filter style include the condition that
filters must be independent entities and not share state with other filters. They should
not be aware of the identity or existence of preceding or subsequent filters. Some
examples of a pure application of this style include pipeline-processing [6.14], batch
systems, and Unix shell applications [6.15]. The Pipe and Filter style can be found in
many application areas including functional programming, signal processing, parallel
programming, and distributed systems.

For this case study we use the design notation described in Shaw et al [6.2], and
use threads to implement filters, and data-streams (the pipes) to connect each thread.
This is different to the Main/Subroutine architectural style, where typically the
architectural components all share a thread of control and the connectors indicate the
process-flow, not the data-flow as pipes do. Points of variability in the application
family are accessed through a filter that provides a common interface to variant
solutions. For example, Figure 6.7 shows a common interface to variant sensor input
algorithms (CI Read Sensor). At build–time the systems engineer must specify if
the sensor readings are acquired from Sensor A or Sensor B. Figure 6.7 demonstrates
how we might integrate the universal single adaptor pattern at points of variability in
application family designs using a Pipe and Filter notation.

Figure 6.6: Example Pipe and Filter architectural style
6.2.5 Event-Based, Implicit Invocation

Event-based systems have their historical roots in systems based on actors (Hewitt [6.16]), constraint satisfaction, daemons, and packet-switched networks. The concept of event-based systems involves components announcing (or broadcasting) one or more events. Other components in the system can register an interest in an event by associating a procedure with it. When an event is broadcast, all the procedures that are registered with that event are invoked. Thus, an event announcement implicitly causes the invocation of procedures in other modules.

An important characteristic of this style is that announcers do not know which components will be affected by which events. Components are unable to make assumptions about the order of processing, or even about what processing will occur as a result of their events. For this reason, most implicit invocation systems also include explicit invocation (that is, normal procedure calls) as a complementary form of interaction.

Event-based systems are used in programming environments to integrate tools (HP Softbench [6.17]), database management systems to ensure consistency constraints (Oracle8 [6.18]), user interfaces to separate presentation of data from its management (Windows [6.19]), and in syntax-directed editors to support incremental semantic checking (see Degano et al [6.20]).
Our fourth case study uses Harel state-transition diagrams [6.21], as a representative design method of the event-based/implicit invocation architectural style. A raised event invokes appropriate procedures typically forcing a system-transition, which usually results in a change of system-state. Points of variability in the application family design are encapsulated by a logical state that provides a common interface to variant solutions. As an example Figure 6.8 shows part of an application family of instrument control units. If a STABILISE event is raised within the control unit, a Stabilising state is entered (CI Stabilising) – the common interface. If CI Stabilising state is entered at run-time, the system’s event handler will automatically raise an event (in this case INTERNAL_STABILISE) to enter the appropriate variant state – Stabilising with or without internal instrument switching (MVR and USO). At built-time a systems engineer must associate the required variants with the event INTERNAL_STABILISE – Stabilising No MVR, No USO or Stabilising MVR=OFF, USO = ON. The appropriate version that is registered with the event-handler is then invoked at run-time.

![Diagram](image)

**Figure 6.8:** Example state providing a common interface to variants in a family of control units

### 6.2.6 Repository

The Repository architectural style has two distinct types of components: a central data structure holding the current state; and a collection of independent components
processing the central data structure. Procedural control may occur in two ways: independent components may execute as a result of the central structure’s state; or they may invoke other components directly. Figure 6.9 shows the Repository style, where the knowledge sources represent the independent components and the shared data structure is in the central Repository. The knowledge sources can be any type of functional module (such as a procedure, a class, a filter, and so on).

Repositories are found in a number of domains, and are often used in systems requiring complex interpretations of signal processing (Nii [6.22]), such as speech and pattern recognition.

![Figure 6.9: Example Repository architectural style](image)

Our fourth case study uses Yourdon’s data-flow diagrams [6.3]¹ as a design method capable of a Repository style notation. In Figure 6.9 the knowledge sources are represented in process bubbles, the Repository itself is denoted as using a storage

¹ Not to be confused with Yourdon’s process-flow diagrams that we use in the Main/Subroutine experiments.
symbol, and their interactions are shown as data-flows. Points of variability in the application family design are captured by a knowledge source that provides a common interface to variant solutions. Figure 6.10 shows an example common interface to variant knowledge sources that is taken from an application family of image processing systems. In this application family, an image-processing system can perform one of two types of colour-spectrum analysis in processing an image that is stored in a shared Repository: single pixel colour spectrum analysis; or area-based colour spectrum analysis. In Figure 6.10 the knowledge source \textit{CI Spectrum Analysis} provides a common interface to the two types of analysis. At build-time a systems engineer must specify the type of colour spectrum analysis that is required, and the appropriate knowledge source is the used at run-time.

![Diagram of common interface to application family variants](image)

\textbf{Figure 6.10}: Example knowledge source providing a common interface to application family variants

6.3 Experiment Proposal

Our hypothesis is that there is a correlation between execution time and memory space overhead that reuse may introduce and the amount of variability incorporated into a single system. To evaluate our hypothesis, we focus on determining any such correlation. It is clearly unfeasible to examine the performance of every possible common interface approach on every platform, using every architectural style, design
method and programming language. We have chosen an experimental approach using five case studies grouped by different architectural styles. These case studies examine the effects on execution time and memory space overhead using five different common interface approaches.

6.3.1 Measuring Overhead

For each architectural style we measured overhead for several pairs of systems designed using a representative design method. Both systems of each pair are derived from exactly the same specification. System pairs consist of one system that is built using a traditional one-off development approach with no points of variability, and one system derived from an application family design with points of variability. A comparison of the two systems provided a measure of both execution time and memory overhead. We compared the two systems of each pair over several different experimental runs. In each run we changed the number of points of variability. This data enables us to observe any trend that may exist between overhead and amount of variability for a particular common interface approach.

To help evaluate time and space overhead, each pair of systems ran within a test harness, which invoked each system individually, emulated I/O and timed their execution.

Each test harness used the system clock to measure execution time. To eliminate any inaccuracies due to the differences in resolution of the unit of measurement and the system clock, we ran a number of test cycles. For example, the unit of measurement of the system clock may be 10 milliseconds. A typical cycle time may be of the order of 1 milliseconds or 0.1 milliseconds. Experiments must repeat test cycles often enough to eliminate these inaccuracies. Memory usage was measured using an external tool – Microsoft Spy++. 
In many real-time experiments that measure execution time, there can be slight differences in the absolute values recorded. These can be attributed to background tasks being performed by the operating system. To reduce the effect of these differences an average value from multiple test-runs shall be taken for each experiment. Also, to ensure consistency, no compiler optimisations were used in any of our experiments as they influence both execution times and memory usage.

6.3.1.1 Measuring Execution Time Overhead

The equation we use to calculate the execution time overhead $O_{\text{exec}}$ for a system pair with a given number of points of variability is

$$O_{\text{exec}} = \frac{P_a - P_s}{C \times V_{\text{calls}}}$$

Equation 6.1

where $P_a$ is the execution time (in seconds) of a system built from an application family over many test cycles. We define a test cycle as one complete execution of either system in the pair. Typically, many test cycles are required to eliminate any inaccuracies due to the differences in resolution of the unit of measurement and the system clock (see section 6.3.1). $P_s$ denotes the average execution time (in seconds) for the same software built using a traditional single system approach over the same number of test cycles. $V_{\text{calls}}$ are the number of calls to a common interface per cycle. $C$ is the number of test cycles.

$P_a - P_s$ calculates the difference in total execution time between the two systems in every pair. This difference denotes the execution time overhead that is introduced by all common interfaces that are called in the application family solution ($P_a$) and are not present in the one-off solution ($P_s$).
It is unlikely that a system has the same number of common interface as it does points of variability. A common interface may be called from within a loop, or recursively. Hence, \( P_a - P_s \) denotes the execution time overhead for a number of calls to a common interface for many test cycles. In line with our hypothesis, the division by \( (C \times V_{calls}) \) will bias the calculation for execution time overhead to annul multiple calls to the same interface and multiple test cycles. \( O_{exec} \) is thus the execution time overhead introduced by a number of points of variability (and not common interfaces), for one test cycle.

To demonstrate, consider an example single system that is built from an application family model and implements 10 points of variability. In each test cycle of the system, two common interfaces denoting two different points of variability are called twice, all other common interfaces are only called once \( (V_{calls} = 12) \). The experiment runs over 100 test cycles \( (C = 100) \). The execution time for 100 test cycles and 12 calls to a common interface is 15.342 seconds \( (P_a) \). An equivalent on-off solution of the system executes 100 test cycles in 12.127 seconds \( (P_s) \). Hence, the overhead for 10 points of variability and one test cycle (one single execution) would be 2.68 milliseconds.

### 6.3.1.2 Measuring Memory Space Overhead

Run-time memory overhead may be introduced by common interfaces denoting a point of application family variability. Applications require segments of physical memory to execute. Rather than accessing memory directly using a physical address, applications can use a logical address, which is typically interpreted by a memory management unit. The logical addressing system allows processors to employ a segmentation mechanism to manage multiple logical memory segments; each can be independently protected and addressed. The segmentation means that different areas of memory can be dedicated to different memory needs of an application. There are
three types of memory segments that may be affected by the introduction of a common interface:

- **Code segments.** These are areas of memory containing the machine code instructions of an executing program (UNIX refers to this type of memory as text segments). Typically, a program cannot modify its code segments as it would be modifying its own machine code.

- **Data Segments.** These are areas of memory that programs use for dynamic memory allocations (such as heap allocations). Blocks of memory are allocated and freed in an arbitrary order and the pattern of allocation and size of blocks is not known until run-time.

- **Stack Segments.** These are typically referred to as memory that is used for storing items, which are accessed in a last-in, first-out, order. Most processors include support for stacks in their instruction set architectures. The most common use of stacks is to store subroutine arguments and return addresses, where the stack keeps track of the sequence of routines called in a program.

We are not interested in distinguishing between the three, as they all contribute to a run-time storage space overhead. To ensure that the memory space experiments have no effect on the execution time measurements, they were carried out in two separate tasks. The equation to calculate memory space overhead $O_{mem}$ for any number of points of variability is

$$O_{mem} = M_a - M_s$$  \hspace{1cm} \text{Equation 6. 2}$$

where $M_a$ is the memory usage for a test cycle of a system built from an application family. $M_s$ denotes the memory usage of the same software built using a traditional one-off approach.
To measure memory overhead, a test harness only needs to perform one test cycle. Unlike execution time that can be affected by low system clock resolutions and background processes, measuring memory is very deterministic and precise; no average measurements are required. We used Microsoft Spy++ to attain measurements of code segment, data segment and stack segment usage.

In our experiments we ensure that both data segments (that are typically allocated dynamically) and code segments are pre-allocated; this way they remain unchanged throughout the execution of a system, regardless of how often a common interface is called. With this precaution, we only need to take one measurement of data segment and code segment usage for each system, and thus reduce the possibility of human error.

Unlike data segments and code segments we cannot pre-allocate stack segments, as these are used to keep track of the sequence of routines called in a program. They are allocated when, for example, a machine code \texttt{CALL} instruction is executed. A high-level programming language once compiled will typically generate \texttt{CALL} instructions for every function, member function, filter, event-registered function, and knowledge source; that is one for every call to a common interface (Goody [6.23]).

As the systems of every pair are both identical, apart from one which contains common interfaces and one which does not; we are assuming that the only difference in stack memory usage between the two system pairs is the stack that is allocated with the extra \texttt{CALL} instructions for each common interface. We thus manually add the stack allocations for a single call to each common interface to $M_a$ separately. Only accounting for a single call of each common interface means that we are counting the stack usage of points of variability, not common interface calls. Hence $M_a = DS_a + CS_a + SS_{aCl}$ where $DS_a$ and $CS_a$ are the pre-allocated data segments and code segments respectively, and $SS_{aCl}$ is the stack required for a single call to each common interface rather than the total stack usage of the system.
Based on our assumption, stack allocations of points of variability are not done by systems built using a traditional one-off development approach. Hence, we calculate the memory usage of one-off systems as \( M_s = DS_s + CS_s \), where \( DS_s \) and \( CS_s \) are code segment and data segment usage, respectively. Although \( M_a \) and \( M_s \) may not reflect the actual memory usage of each system pair as only the difference in stack usage is considered, \( M_a - M_s \) is clearly representative of the memory overhead introduced at points of variability.

### 6.3.2 Experiment Format

We used a range of measurements of overhead for different amounts of variability. The more spread out the different amounts of variability are, the more likely we are to observe a correlation. It would be insufficient to measure overhead for a number of systems all with the same amount of points of variability. We need to use a range of different amounts of variability, for example, one system with one point of variability, another with five points of variability, yet another with 10 points of variability, and so on. We propose to take the following five sample measurements of execution time overhead for each architectural style:

1. One measurement of time overhead between a system implementing the minimum amount of variability (normally one point of variability) and a counterpart system build using a one-off development approach.

2. One measurement of time overhead between a system implementing the maximum possible number of points of variability (such as 10 points of variability) and its one-off counterpart.
3. & 4. At least two random intermediate readings of time overhead between minimum and maximum number of points of variability (such as three and seven points of variability).

5. At least one duplicate measurement of time overhead containing the same number of points of variability as an existing measurement, but using different variants to the existing measurement (such as a different three points of variability). We discuss the reason for a duplicate measurement in Section 6.4.

Similarly, we propose that the following five measurements of memory space overhead be taken for each architectural style:

1. One measurement of memory overhead between a system implementing the minimum amount of variability and a counterpart system build using a one-off development approach.

2. One measurement of memory overhead between a system implementing the maximum amount of variability and its one-off counterpart.

3. & 4. At least two random intermediate readings of memory overhead between minimum and maximum number of points of variability.

5. At least one duplicate measurement of memory overhead containing the same number of points of variability as an existing measurement, but using variants at points of variability.
This set of 10 experiments is then repeated for each of the five selected architectural styles providing 50 measured results. The format of expected results is shown in Table 6.2 and Table 6.3.

<table>
<thead>
<tr>
<th>Time Overhead</th>
<th>Case Study 1 Main Subroutine</th>
<th>Case Study 2 Object-Oriented</th>
<th>Case Study 3 Pipe and Filter</th>
<th>Case Study 4 Event Based</th>
<th>Case Study 5 Repository</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>$O_{mem} \text{ 11 s}$</td>
<td>$O_{mem} \text{ 21 s}$</td>
<td>$O_{mem} \text{ 31 s}$</td>
<td>$O_{mem} \text{ 41 s}$</td>
<td>$O_{mem} \text{ 51 s}$</td>
</tr>
<tr>
<td>Intermediate 1</td>
<td>$O_{mem} \text{ 12 s}$</td>
<td>$O_{mem} \text{ 22 s}$</td>
<td>$O_{mem} \text{ 32 s}$</td>
<td>$O_{mem} \text{ 42 s}$</td>
<td>$O_{mem} \text{ 52 s}$</td>
</tr>
<tr>
<td>Intermediate 2</td>
<td>$O_{mem} \text{ 13 s}$</td>
<td>$O_{mem} \text{ 23 s}$</td>
<td>$O_{mem} \text{ 33 s}$</td>
<td>$O_{mem} \text{ 43 s}$</td>
<td>$O_{mem} \text{ 53 s}$</td>
</tr>
<tr>
<td>Duplicate</td>
<td>$O_{mem} \text{ 14 s}$</td>
<td>$O_{mem} \text{ 24 s}$</td>
<td>$O_{mem} \text{ 34 s}$</td>
<td>$O_{mem} \text{ 44 s}$</td>
<td>$O_{mem} \text{ 54 s}$</td>
</tr>
<tr>
<td>Maximum</td>
<td>$O_{mem} \text{ 15 s}$</td>
<td>$O_{mem} \text{ 25 s}$</td>
<td>$O_{mem} \text{ 35 s}$</td>
<td>$O_{mem} \text{ 45 s}$</td>
<td>$O_{mem} \text{ 55 s}$</td>
</tr>
</tbody>
</table>

Table 6.2: Format of results for execution time overhead $O_{exe}$

<table>
<thead>
<tr>
<th>Space Overhead</th>
<th>Case Study 1 Main Subroutine</th>
<th>Case Study 2 Object-Oriented</th>
<th>Case Study 3 Pipe and Filter</th>
<th>Case Study 4 Event Based</th>
<th>Case Study 5 Repository</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>$O_{mem} \text{ 11 \ Bytes}$</td>
<td>$O_{mem} \text{ 21 \ Bytes}$</td>
<td>$O_{mem} \text{ 31 \ Bytes}$</td>
<td>$O_{mem} \text{ 41 \ Bytes}$</td>
<td>$O_{mem} \text{ 51 \ Bytes}$</td>
</tr>
<tr>
<td>Intermediate 1</td>
<td>$O_{mem} \text{ 12 \ Bytes}$</td>
<td>$O_{mem} \text{ 22 \ Bytes}$</td>
<td>$O_{mem} \text{ 32 \ Bytes}$</td>
<td>$O_{mem} \text{ 42 \ Bytes}$</td>
<td>$O_{mem} \text{ 52 \ Bytes}$</td>
</tr>
<tr>
<td>Intermediate 2</td>
<td>$O_{mem} \text{ 13 \ Bytes}$</td>
<td>$O_{mem} \text{ 23 \ Bytes}$</td>
<td>$O_{mem} \text{ 33 \ Bytes}$</td>
<td>$O_{mem} \text{ 43 \ Bytes}$</td>
<td>$O_{mem} \text{ 53 \ Bytes}$</td>
</tr>
<tr>
<td>Duplicate</td>
<td>$O_{mem} \text{ 14 \ Bytes}$</td>
<td>$O_{mem} \text{ 24 \ Bytes}$</td>
<td>$O_{mem} \text{ 34 \ Bytes}$</td>
<td>$O_{mem} \text{ 44 \ Bytes}$</td>
<td>$O_{mem} \text{ 54 \ Bytes}$</td>
</tr>
<tr>
<td>Maximum</td>
<td>$O_{mem} \text{ 15 \ Bytes}$</td>
<td>$O_{mem} \text{ 25 \ Bytes}$</td>
<td>$O_{mem} \text{ 35 \ Bytes}$</td>
<td>$O_{mem} \text{ 45 \ Bytes}$</td>
<td>$O_{mem} \text{ 55 \ Bytes}$</td>
</tr>
</tbody>
</table>

Table 6.3: Format of results for memory space overhead $O_{mem}$

Figure 6.11 and Figure 6.12 show how we may draw graphs of both execution time and memory space overhead for example results of a Pipe and Filter architecture. A similar graph shall be drawn for each of our five case studies for both time and memory overhead.
Figure 6.11: Plot of example execution time overhead of Pipe and Filter case study

Figure 6.12: Plot of example memory space overhead of Pipe and Filter case study
6.4 Statistical Analysis of Measured Results

We have chosen regression analysis (Krishnaiah [6.24]) to estimate a probable strength of correlation between overhead and number of points of variability for each of the five case studies. Regression analysis will approximate the best-fitting equation describing the relationships between execution time overhead and points of variability, and between memory space overhead and points of variability; for each of the two graphs of every case study (such as Figure 6.11 and Figure 6.12).

We calculate a residual error for each fitted equation, to denote the strength of correlation between the measured points and the fitted equation. Residual plots are used to assess the adequacy of the fitted equation, and a t-test is used to calculate the probability that our measured data is representative and not due to chance. Our statistical analysis approach is detailed in the Sections 6.4.1 to 6.4.4.

6.4.1 Approximating an Equation

We use a spreadsheet to estimate a best-fitting equation for each set of measured data, based on our initial observations of each graph. For example, if the measured points appear to have a linear relationship, we let the spreadsheet use a linear regression method to estimate and plot a linear trend-line. If the measured points appear to have a quadratic relationship, we let the spreadsheet use a non-linear regression method (such as least squares fit) to estimate and plot a quadratic trend-line.
Figure 6.13 shows an example plot of an estimated linear equation based on measurements of memory overhead against points of variability, for a single case study.

![Graph showing estimated linear equation](image)

**Figure 6.13**: Example plot of estimated equation for measured memory overhead against points of variability

We are assuming that a system incorporating no points of variability will have no reuse-related overhead. Hence all estimated equations, both linear and non-linear, must pass through the graph’s origin.

### 6.4.2 Residual Error

In any linear and non-linear analysis, it is necessary to assess the fit of the equation to the data. Residuals, as described in Bates *et al* [6.26], are a measure of the difference between measured data points and a fitted equation. Figure 6.14 shows
five example memory overhead measurements ($O_{\text{mem} 1}$ to $O_{\text{mem} 5}$), the plot for a fitted linear equation ($fo$) and five residuals ($R_1$ to $R_5$).

![Diagram of memory overhead and residuals](image)

**Figure 6.14:** Residuals $R_1$ to $R_5$ for example plot of memory overhead against points of variability

Residuals are used to calculate a residual error, which indicates a probability that estimated points from the equation ($f_0(1)$ to $f_0(5)$) are equal to measured points ($O_{\text{mem} 1}$ to $O_{\text{mem} 5}$), and vice-versa. This probability is a measure of the strength of correlation for each data set of every case study, and is given by:

$$R^2 = \frac{\sum_{i=1}^{N} \left( \frac{R_i}{f_0(y_i)} \right)^2}{N}$$

Equation 6.3

where $R_i$ is the residual for a measured data point $O_{\text{mem} i}$, $f_0(y_i)$ is the estimated data point from the equation, and $N$ is the total number of measured data points (e.g. $N = 5$ in Figure 6.14). Equation 6.3 can be used to calculate a residual error $R^2$, regardless
of the type of equation (including linear, quadratic, polynomial, logarithmic, and exponential equations).

By definition, finding an equation that can fit the plots of the number of points of variability to either memory overhead or execution time overhead implies that there is a correlation. A residual error of less than 0.05 is generally accepted as an indication that an equation correlates very strongly (James [6.25]), and a residual error less than 0.05 for all our experiments would be a satisfactory indication that our hypothesis is accurate. Residual errors that are higher than 0.05 may imply the following:

- The fitted equation for that residual error is inadequate. For example, a linear equation may have been fitted where a quadratic may have been more appropriate. We use residual plots to help assess the adequacy of each fitted equation (see Section 6.4.3).

- One or more of the measured data points may be unrepresentative or just due to chance. For example, accidentally running operating system background tasks (including checking for email messages, system backup, and system cleanup) alongside some experiments, and not on others, can potentially impact on the execution time of some systems and invalidate measurements. We compare duplicate measurements and use Student’s t-tests to assess the validity of the data points of every plot (see Section 6.4.4).

- There is simply no correlation, which would dispute our hypothesis.

### 6.4.3 Residual Plots

A plot of residuals may be used to assess the adequacy of fit of an estimated equation, as many equations can fit a set of measured data points. Figure 6.15 shows how two different equations are fitted to some example data; one is a linear equation
and the other is a quadratic equation. To the naked eye both fits seem perfectly reasonable. A residual plot can be used to rectify such situations.

![Figure 6.15: Example measured data points with 2 fitted equations](image)

A residual plot is a graph of the residuals \( R_i \) against their corresponding data point index \( i \). Attention is paid to whether or not the plotted residuals seem randomly distributed. If the plot is non-random (that is an obvious trend can be observed), there is a strong likelihood that the wrong equation was fitted to the initial data points. Using a different fit that accounts for the observed trend in a residual plot can reduce a residual error (by that trend). For example, an exponential trend in a residual plot may suggest that a different fit is more appropriate than the current linear one. Figure 6.16 shows a residual plot for an adequate fit; the residuals are distributed randomly. Figure 6.17 shows a residual plot for a fit of a linear equation that is probably not representative of the data points, the plotted residuals suggest that a quadratic or an exponential fit may be more appropriate.
We examine the residual plots for every execution time and for every memory space overhead experiment in each of our five case studies, and use a spreadsheet to calculate the residuals and residual errors, and to draw the residual plots.

Residual plots are not always accurate for fitted equations based on few data points, as there is almost always a visible pattern. However, they can still help detect small deviations from an ideal fit. In our situation we only have five measured data-points, which is insufficient to assume that residual plots are of any quantitative statistical significance. Nevertheless, we can still use them for a trial-and-error
assessment of our fits. Should a residual plot in our experiments seem non-random, a
different fit can be attempted. However, due to the small number of data-points, we
cannot accept or reject a fit on the basis of a residual plot alone. If the new fit has an
improved residual error and a more confident t-test result then we can assume that it is
more adequate than the previous one.

6.4.4 Student’s t-tests

Student’s t-tests, as described in [6.27], are a statistical test to help us examine the
validity of our linear model$^2$. In all regression analysis on plots of measured data,
some of the variation of $y$ (that is, the overhead) is a result of its dependence on $x$
(points of variability), and some is from random variation. While the residual error
describes the percentage of the variation of $y$ that is attributed to its dependence on $x$,
Student’s t-tests provide a percentage denoting the probability that the variation of $y$ is
not attributed to random variance. Rather than assess the validity of the fitted
equation, it assesses the validity of the measured data, and calculates a probability that
our measured data is representative and not due to chance. In situations where there
are few measured data points (that is, a low degree of freedom) even a single value
that is not representative can have a significant impact on our overall observations.
We only have five measured data points for each of our graphs, which equates to three
degrees of freedom (where degrees of freedom = number of data points – 2).

Student’s t-tests start by hypothesising that no relationship exists between the two
data sets $x$ and $y$, and that any relationship between the two is purely coincidental.
This hypothesis can be formalised as $m = 0$, where $m$ is the gradient of the equation
relating $x$ to $y$: the regression slope is the result of random variation. This is known as
the null hypothesis.

$^2$ Not to be confused with linear-fit. A linear model is a model in a vector space.
A t-test aims to show that the fitted equation, the relationship between \( x \) and \( y \), on the balance of probability, cannot be the result of random variance \((m \neq 0)\). This is known as the alternative hypothesis.

The t-test compares t-stat\(^3\) values in the standard t-distribution, as given in Table 6.4, to a calculated t-stat value. If our calculated t-stat value is greater or equal to a t-stat value in the t-distribution table for a particular probability \( p \), we can assume the probability \( p \) of our data being representative and not due to random variance. Table 6.4 shows a Student’s t-distribution for three degrees of freedom.

<table>
<thead>
<tr>
<th></th>
<th>( p = 80% )</th>
<th>( p = 90% )</th>
<th>( p = 95% )</th>
<th>( p = 99% )</th>
<th>( p = 99.9% )</th>
</tr>
</thead>
<tbody>
<tr>
<td>t-stat for 3</td>
<td>1.64</td>
<td>2.35</td>
<td>3.18</td>
<td>5.84</td>
<td>12.92</td>
</tr>
<tr>
<td>degrees of freedom</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 6.4:** Standard t-distribution for three degrees of freedom

The calculated t-stat value represents the ratio of the fitted equation to the standard error of the measured data points. We let a spreadsheet calculate a t-stat value for each measured data set of execution time and memory overhead, of our five case studies. A high value for the calculated t-stat tends to indicate a high probability that the data points are representative. A low calculated t-stat value tends to indicate a poor model.

For example, a calculated t-stat value of 4.23 is greater than the t-stat value 3.18 in Table 6.4, but less than 5.84. Hence there is a probability greater than 95\% that our data is representative and a less than 5\% chance that data points are as a result of random variation. If a 5\% probability provides sufficient confidence in our results, the null hypothesis must be rejected.

\(^3\) Some literature refers to t-stat values as t-values.
In most situations, a 95% confidence in measured results is considered to be sufficiently accurate to reject any null hypothesis (James [6.25]). t-tests for our experiments that reveal plots that have measured data points that are, on a balance of probability, less representative, suggest that we can have no confidence that our measurements for those plot are representative. Hence, we have no confidence in the residual error, and any correlation, drawn from those measurements, as they may be due to random variance. In situations, where there is little confidence in our experimental data, we can take more measurements to confirm, or reject, the null hypothesis.

For our experiments, we insist that each case study take one duplicate measurement of overhead for two different systems, both with the same number of points of variability (see Section 6.3.2). This is another technique to help assert confidence in our data. If the difference between the two independent data-points both with the same number of points of variability looks insignificant, then our measurements for that plot are less likely to be due to chance. Random variance by definition is not repeatable. However, while duplicate measurements can provide some visual confirmation about the probable presence of random variance, a t-test will quantify any such presence.

6.5 Summary

In this chapter we introduced five architectural styles by which we group our case studies, including Main/Subroutine, Object-oriented, Pipe and Filter, Event-Based/Implicit Invocation and Repository styles. We demonstrate examples of how our universal design patterns in Chapter 4 can be used with the components of each design method to model points of variability.

We proposed experiments that measure five values for execution time overhead and five values for memory space overhead for each of the five case studies. With
each measurement of overhead we vary the amount of points of variability, and we
detailed how individual measurements are taken.

Finally, we propose measures to evaluate each of the 10 sets of measurements of
overhead: residual errors, residual plots and Student’s t-tests. Our results will indicate
the strength of correlation between execution time and memory space overhead, and a
number of points of variability. A correlation between overhead and number of points
of variability for all five case studies would be a very strong indication that other
common interface approaches are potentially predictable, and very strong evidence
supporting our hypothesis.

Chapter 7 presents our case studies and results, and Chapter 8 will evaluate and
discuss our findings.

6.6 References

Draft Version 3.0, IEEE Architecture Working Group. (Currently under

0-13-182957-2.


Number 3, Pages 103-120, May 1986.


Oriented Modelling And Design. Prentice Hall, Englewood Cliffs, New Jersey,


CHAPTER 7

Case Studies

7.1 Introduction

This chapter presents the case studies we use to help evaluate our hypothesis. It is spilt up into five subsections; each describing the experiments and results for the five nominated architectural styles described in Chapter 6. For all experiments, data and results from statistical analysis can be found in Appendix G.

7.2 Case Study 1: Main/Subroutine

Increasingly washing machine manufacturers are turning from mechanical and simple electronic systems that control the wash cycles to more complex instrument control units run by software. Our first experiments are from an application family of washing machine cycles\(^1\) that have implementations based on a Main/Subroutine architectural style. A washing machine cycle is a wash-program that a user can select from a control panel. Washing machines usually have a range of wash cycles for different types of fabric, including a boil-wash for white linen, a fast-coloureds wash

\(^1\) Although our specifications for a family of washing machine cycles is based on information taken from Zanussi documentation, our solutions are completely independent of Zanussi. Any resemblance to their actual products is coincidental.
cycle for cotton, a mixed-fabric wash cycle, and a cool wash cycle for woollens. Each wash-cycle typically consists of a number of phases, which might include a pre-wash phase, a main-wash phase, a rinse phase, a spin phase, and a drying phase.

Typically a control panel on the front of a washing machine lets a user select a desired wash cycle. A user’s requests will be evaluated by the software, which in turn will control the washing machine’s hardware. The software monitors the progress of the hardware and the states of the machine are relayed back to a control panel – informing the user. Figure 7.1 shows the relationship between the embedded software and the hardware.

![Figure 7.1: Washing machine system layout](image)

We have constructed an MRAM application family model for a single washing machine cycle that may exist as a part of a washing machine’s embedded software. The application family model is based on features found in Zanussi documentation [7.1] and [7.2], and is detailed in Appendix B.1. It has thirteen points of variability (discriminants), including:

- A wash cycle shall have different phases including a pre-wash, a main-wash, a rinse phase, a spin phase and a tumble-dry phase – multiple adaptor discriminant.
- Different pre-wash phases that work at different temperatures – single adaptor discriminant.
Different main-wash phases that work at different temperatures – single adaptor discriminant.

Different tumble-dry phases shall spin the drum at different speeds – single adaptor discriminant.

There shall be an optional rinse-hold, eco-wash and reduced-spin facility – optional discriminants.

Indicator lights shall display the progress of a wash cycle on the control panel – optional discriminant.

An application family design was built to realise the features in the application family model (See Appendix B.1.1). This design takes the form of a Yourdon process-flow diagram [7.3]. Design notations are described in Appendix A.2. Points of variability in the application family design are modelled using our patterns from Chapter 4.

7.2.1 Main/Subroutine Experiments

Five requirements specifications were derived from the application family model for five different washing machine cycles. Each specification incorporates a different amount of variability:

- **System 1** is a rinse cycle incorporating two points of variability. (Appendix B.2).

- **System 2** is a low-temperature tumble-dry cycle with a delayed door release, incorporating four points of variability. (Appendix B.3).

- **System 3** consists of a low-temperature main-wash cycle with a rinse phase, also incorporating four points of variability. (Appendix B.4).
System 4 is a medium-temperature main-wash cycle, which includes pre-wash, main-wash, rinse and spin phases, incorporating seven points of variability. (Appendix B.5).

System 5 is a full, high-temperature wash and dry cycle, incorporating 10 points of variability. (Appendix B.6).

For each of the five system specifications we constructed two designs, each with corresponding implementations. The first design is derived from the application family design for all five systems (Appendix B.1.1) and includes common interfaces at points of variability (Appendix B.2.1, B.3.1, B.4.1, B.5.1 and B.6.1). The second design is built using a traditional one-off approach that does not allow for variability (Appendix B.2.2, B.3.2, B.4.2, B.5.2 and B.6.2). The corresponding system implementations of the five system pairs were done in Turbo Pascal. All source code for this case study can be found on the accompanying CD-ROM.

To evaluate the performance of each version of each system we measured the average CPU time and the memory usage that is required to complete a single wash-cycle. To compensate for any inaccuracies of the system clock, each wash-cycle was repeated 2’000’000 times. We used a test harness to invoke each wash-cycle and to emulate Hardware I/O. The results for each system pair provided a measured figure for overhead for a given amount of variability.

7.2.2 Main/Subroutine Results

Table 7.1 shows the results for each of the system pairs. It shows the execution times ($P_a$) for 2’000’000 cycles of the five systems built from an application family and the corresponding execution times ($P_s$) for the same five systems built using a one-off approach. Table 7.1 details the execution time overhead ($P_a - P_s$) per call to a common interface. Biasing $P_a - P_s$ by the total number of calls to a common interface
\((C \times V_{\text{calls}})\) provides a figure for the execution time overhead \(O_{\text{exec}}\) per points of variability for Systems 1 to 5. We calculate \(O_{\text{exec}}\) using Equation 6.1.

\[
\begin{array}{cccccc}
\text{System} & \text{Points of variability} & \bar{P}_s & \bar{P}_t & \bar{P}_s - \bar{P}_t & C \times V_{\text{calls}} & O_{\text{exec}} \\
1 & 2 \text{ (Minimum)} & 2.53 \text{ s} & 2.20 \text{ s} & 0.33 \text{ s} & 2'000'000 & 0.000000165 \text{ s} \\
2 & 4 \text{ (Intermediate)} & 4.82 \text{ s} & 3.84 \text{ s} & 0.98 \text{ s} & 2'500'000 & 0.000000390 \text{ s} \\
3 & 4 \text{ (Duplicate)} & 5.08 \text{ s} & 4.39 \text{ s} & 0.69 \text{ s} & 2'000'000 & 0.000000344 \text{ s} \\
4 & 7 \text{ (Intermediate)} & 8.47 \text{ s} & 7.08 \text{ s} & 1.39 \text{ s} & 2'000'000 & 0.000000697 \text{ s} \\
5 & 10 \text{ (Maximum)} & 12.61 \text{ s} & 10.49 \text{ s} & 2.12 \text{ s} & 2'200'000 & 0.000000965 \text{ s} \\
\end{array}
\]

**Table 7.1:** Execution time overhead \(O_{\text{exec}}\) for

Main/Subroutine case study

Figure 7.2 shows a plot of execution time overhead \(O_{\text{exec}}\) against number of points of variability for a single test-cycle of each system. By observation the graph seems linear, hence we used linear regression to derive an equation for the graph. The residual error between the fitted equation and the plotted points is 0.0075. A t-test confirms that the measurements for execution time overhead for the Main/Subroutine experiments have a probability \(p\) greater than 99.9\% of being representative and not due to random variance. (Appendix G.1).
Figure 7.2: Plot of execution time overhead against points of variability for Main/Subroutine case study

Table 7.2 shows the memory usage of each of the Main/Subroutine system pairs. It details memory space overhead $O_{mem}$ for systems with 2, 4 ($\times$2), 7 and 10 points of variability. $O_{mem}$ is the difference between the memory usage $M_a$ of a system built from an application family and the memory usage $M_s$ of a system built using a traditional one-off approach (see Equation 6.2).
Figure 7.3 shows a graph of memory space overhead $O_{mem}$ against number of points of variability. We used a linear regression to estimate the best fitting equation for the graph. The residual error between the equation and the plotted points is 0.0066. A t-test shows that there is a probability $p$ greater than 99.9% that these results are not due to chance. (Appendix G.1).

Table 7.2: Memory space overhead for Main/Subroutine case study

<table>
<thead>
<tr>
<th>System</th>
<th>Points of variability</th>
<th>$M_a$</th>
<th>$M_i$</th>
<th>$O_{mem}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 (Minimum)</td>
<td>229 Bytes</td>
<td>186 Bytes</td>
<td>43 Bytes</td>
</tr>
<tr>
<td>2</td>
<td>4 (Intermediate)</td>
<td>322 Bytes</td>
<td>234 Bytes</td>
<td>88 Bytes</td>
</tr>
<tr>
<td>3</td>
<td>4 (Duplicate)</td>
<td>351 Bytes</td>
<td>266 Bytes</td>
<td>85 Bytes</td>
</tr>
<tr>
<td>4</td>
<td>7 (Intermediate)</td>
<td>559 Bytes</td>
<td>402 Bytes</td>
<td>157 Bytes</td>
</tr>
<tr>
<td>5</td>
<td>10 (Maximum)</td>
<td>776 Bytes</td>
<td>538 Bytes</td>
<td>238 Bytes</td>
</tr>
</tbody>
</table>

Figure 7.3 shows a graph of memory space overhead $O_{mem}$ against number of points of variability. We used a linear regression to estimate the best fitting equation for the graph. The residual error between the equation and the plotted points is 0.0066. A t-test shows that there is a probability $p$ greater than 99.9% that these results are not due to chance. (Appendix G.1).
7.3 Case Study 2: Data Abstraction and Object-Orientation

Our second case study is taken from a family of modelling algorithms for Global Positioning Systems. The U.S. Global Positioning System (GPS) and the Russian Global Navigation Satellite System (GLONASS) are world-wide radio-navigation systems formed from two constellations each with 24 satellites, in six and three orbits respectively.

The satellites each transmit a unique signal to earth. Encoded in each signal are a satellite’s identity and the precise atomic time at which the signal was broadcast. The distance to a satellite can then be estimated given the exact time the signal took to...
transmit. Corrections must be made to consider atmospheric disturbance to the carrier signal.

![Diagram of satellite positioning](image)

**Figure 7.4:** Example of the principle of global positioning

A combination of distances to four satellites combined with the known locations of the four satellites allows us to pinpoint a position in space. Figure 7.4 shows a simplified two-dimensional example where a minimum of three satellites are required to calculate a position on earth (in three-dimensions, four satellites are required). In Figure 7.4, d₁, d₂ and d₃ denote the distances from each satellite (S₁, S₂ and S₃) to our location on earth. Triangulating the distances d₁, d₂ and d₃ provides a point of intersection, which is our relative position to the satellites. Combining the relative position with the exact locations of each satellite allows us to calculate a location on Earth.
Our case study focuses on the algorithms to model satellite positions at any given point in time. We have constructed an MRAM application family model of GPS and GLONASS modelling algorithms. The application family model is based on summaries of GPS and GLONASS presented in [7.4] and [7.5], and can be found in Appendix C.1. The points of variability that we have identified include:

- The algorithms shall model the positions of GPS and/or GLONASS satellite constellations – multiple adaptor discriminant.

- A satellite position shall be passed to a GPS device as an orbital phase rotation, or as a vector co-ordinate – single adaptor discriminant.

- A satellite position shall be calculated with respect to a viewpoint in space or an orthogonal view from earth – single adaptor discriminant.

- The four nearest satellites to a known position on earth shall be flagged. The nearer a satellite is, the better the signal will be – optional discriminant.

- Orbits, satellite identifiers and earth shall be displayed alongside each satellite – multiple adaptor discriminant.

- Up to seven landmarks may be displayed on earth – multiple adaptor discriminant.

Appendix C.1.1 details an object-oriented application family design that realises the application family model in Appendix C.1, where the application family design is modelled using UML [7.6] (design notation summarised in Appendix A.2).
7.3.1 Object-Oriented Experiments

We have derived five requirements specifications from the application family model for five different GPS and GLONASS modelling systems. Each of the five system specifications incorporates a different amount of variability.

- **System 1** calculates the orbit phase of the GPS satellite constellations at a given point in time, and the results are passed to the receiver. It incorporates two points of variability. (Appendix C.2).

- **System 2** displays the GPS and GLONASS satellites on the receiver’s LCD using an orthogonal viewpoint on earth, and it highlights the four closest satellites. It incorporates four points of variability. (Appendix C.3).

- **System 3** displays the GLONASS satellites from a viewpoint in space. The three GLONASS orbits and each satellite’s identifier are drawn. It incorporates four points of variability. (Appendix C.4).

- **System 4** uses the receiver’s LCD to model the GPS and GLONASS satellites in an orbit around earth. For reference it also displays three landmarks on earth – London, Sydney and Rio de Janeiro. It incorporates seven points of variability. (Appendix C.5).

- **System 5** models just the GPS constellation. It displays the six orbits, the satellite identifiers and seven landmarks on earth. For each landmark the optimum four GPS satellites are highlighted. This system incorporates eight points of variability. (Appendix C.6).

Two designs were constructed for each of the five system specifications. The first design is derived from the application family design (Appendix C.1.1) and includes common interfaces at all the points of variability (Appendix C.2.1, C.3.1, C.4.1, C.5.1
and C.6.1). The second design is built using a traditional one-off approach that does not allow for variability (Appendix B.2.2, C.3.2, C.4.2, C.5.2 and C.6.2). The implementations of the five system pairs were done in C++, and the source code for the object-oriented experiments can be found on the accompanying CD-ROM.

To evaluate the performance of each of the five systems we again measured the average CPU time and the memory usage that is required to model satellites over time. To minimise inaccuracies due to the lack of precision of the system clock, each test was repeated 2'880 times. A test harness was used to invoke each modelling-cycle and to emulate Hardware I/O. The results for each system pair provided a measured figure for overhead for a given amount of variability.

### 7.3.2 Object-Oriented Results

Table 7.3 shows the measured results from the object-oriented case study. It details the execution time overhead $O_{exec}$ for Systems 1 to 5. We calculate $O_{exec}$ using Equation 6.1.

<table>
<thead>
<tr>
<th>System</th>
<th>Points of variability</th>
<th>$P_e$</th>
<th>$P_r$</th>
<th>$P_e - P_r$</th>
<th>$C \times V_{calls}$</th>
<th>$O_{exec}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 (Minimum)</td>
<td>31.129 s</td>
<td>31.114 s</td>
<td>0.015 s</td>
<td>2'880</td>
<td>0.000005208 s</td>
</tr>
<tr>
<td>2</td>
<td>4 (Intermediate)</td>
<td>123.736 s</td>
<td>123.707 s</td>
<td>0.029 s</td>
<td>2'880</td>
<td>0.0000010069 s</td>
</tr>
<tr>
<td>3</td>
<td>4 (Duplicate)</td>
<td>127.320 s</td>
<td>127.081 s</td>
<td>0.239 s</td>
<td>24'480</td>
<td>0.0000009763 s</td>
</tr>
<tr>
<td>4</td>
<td>7 (Intermediate)</td>
<td>604.570 s</td>
<td>604.478 s</td>
<td>0.092 s</td>
<td>5'348.6</td>
<td>0.0000017201 s</td>
</tr>
<tr>
<td>5</td>
<td>8 (Maximum)</td>
<td>725.130 s</td>
<td>724.661 s</td>
<td>0.469 s</td>
<td>25'560</td>
<td>0.000018349 s</td>
</tr>
</tbody>
</table>

**Table 7.3:** Execution time overhead $O_{exec}$ for Object-Oriented case study

Figure 7.5 plots execution time overhead $O_{exec}$ against the number of points of variability. Through observation we assumed a linear relationship and used a linear regression to fit an equation to the measurements. A residual error of 0.0106 suggests that there is a strong correlation between the measured points and the equation. A t-
test shows that there is a probability $p$ of greater than 99.9% that the measurements for execution time overhead in our object-oriented case study are not due to random variance. (Appendix G.2).

![Figure 7.5: Plot of execution time overhead $O_{exec}$ against points of variability for Object-Oriented case study](image)

Table 7.4 shows the memory space measurements for the object-oriented experiments. It itemises the memory overhead $O_{mem}$ for systems with 2, 4 ($\times$2), 7 and 8 points of variability. We use Equation 6.2 to calculate $O_{mem}$. 

150
<table>
<thead>
<tr>
<th>System</th>
<th>Points of variability</th>
<th>$M_s$</th>
<th>$M_r$</th>
<th>$O_{mem}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 (Minimum)</td>
<td>835’072 Bytes</td>
<td>834’714 Bytes</td>
<td>358 Bytes</td>
</tr>
<tr>
<td>2</td>
<td>4 (Intermediate)</td>
<td>836’626 Bytes</td>
<td>835’895 Bytes</td>
<td>731 Bytes</td>
</tr>
<tr>
<td>3</td>
<td>4 (Duplicate)</td>
<td>864’256 Bytes</td>
<td>863’383 Bytes</td>
<td>873 Bytes</td>
</tr>
<tr>
<td>4</td>
<td>7 (Intermediate)</td>
<td>871’936 Bytes</td>
<td>870’656 Bytes</td>
<td>1’280 Bytes</td>
</tr>
<tr>
<td>5</td>
<td>8 (Maximum)</td>
<td>845’312 Bytes</td>
<td>843’878 Bytes</td>
<td>1’434 Bytes</td>
</tr>
</tbody>
</table>

**Table 7.4:** Memory space overhead $O_{mem}$ for Object-Oriented case study

Figure 7.6 plots $O_{mem}$ against the number of points of variability. A linear regression was used to estimate and plot an adequate equation. The residual error is 0.0271, and the results of a t-test show a probability $p$ of greater than 99.9% that the figures of our measurements are not due to chance. (Appendix G.2).

**Figure 7.6:** Plot of memory overhead $O_{mem}$ against points of variability for Object-Oriented case study
7.4 Case Study 3: Pipe and Filter

The experiments for the Pipe and Filter case study are taken from automated process control. Our experiments are derived from a family of systems that simultaneously regulate liquid levels in two large storage-tanks. Figure 7.7 shows the physical layout of our application family of liquid regulators. The tanks are controlled by a closed feedback loop using an on/off control algorithm, and the controller software is responsible for maintaining a constant set point in the tanks. Sensors measure the actual liquid levels and transmit them to the software. If the measured level is below the set point the respective valves are opened until the tank is full again. The states of the tanks and of the valves are logged for status and fault analysis.

![Diagram of Pipe and Filter case study](image-url)
We have constructed an MRAM application family model for controller software of liquid regulating systems. The application family model is based on automated process control examples in [7.7] and [7.8], and can be found in Appendix D.1. Our application family model of controller software notes 10 points of variability, including:

- Sensor values shall be acquired through a serial port or a parallel port for Tank 1 and/or Tank 2 – single adaptor discriminant.
- The sensor readings from Tank 1 and/or Tank 2 shall be converted from liquid weight to liquid volume – single adaptor discriminant.
- The set point for either of the tanks shall be adjustable through a remote user terminal – optional discriminant.
- For either of the tanks, 10 liquid level and valve status readings shall be averaged before logging – optional discriminant.
- Tanks levels and valve states shall be stored to hard disk and/or written to printer – multiple adaptor discriminant.
- A redundant secondary backup device shall be added – optional discriminant.

From the specifications in the application family model we constructed an application family design for our controller software (See Appendix D.1.1). The design uses a Pipe and Filter notation described in Shaw et al [7.9].

7.4.1 Pipe and Filter Experiments

Five requirements specifications were derived from our application family model for five controller systems that simultaneously regulate liquid levels in two tanks.
Again, each system derived from the application family model incorporates different amounts of variability. These are:

- **System 1** controls both tanks through a serial port connection; it requires no extra data processing. Acquired values are logged to hard disk. It incorporates three points of variability. (Appendix D.2).

- **System 2** controls Tank 1 through a serial port and Tank 2 through a parallel port. 10 measurements from Tank 1 are averaged before logging. Acquired values are logged to hard disk and a backup device. It incorporates five points of variability. (Appendix D.3).

- **System 3** is required to control Tank 1 through a parallel port and Tank 2 through a serial connection. Both tanks have an adjustable set point. Acquired values from Tank 1 are converted to measures of volume. 10 measurements from Tank 2 are averaged before logging. Acquired data is logged to hard disk, printer and a secondary backup device. This system incorporates eight points of variability. (Appendix D.4).

- **System 4** controls Tank 1 through a serial port and Tank 2 through a parallel port. Readings from both tanks are converted to measures of volume and average 10 readings before logging the measurements. Tank 2 has an adjustable set point. The acquired values are sent to a printer. It also incorporates eight points of variability. (Appendix D.5).

- **System 5** controls both tanks through a serial port connection. Readings from both tanks are converted to measures of volume. Tank 1 and Tank 2 both have adjustable set points. For each tank, 10 readings are averaged before they are logged. Readings are logged to hard disk, printer and a secondary backup device. It incorporates 10 points of variability. (Appendix D.6).
Two designs with corresponding implementations were created for each of the five system specifications. The first design is derived from the application family design (Appendix D.1.1) and includes common interfaces at all the points of variability (Appendix D.2.1, D.3.1, D.4.1, D.5.1 and D.6.1). The second design is built using a traditional one-off approach that does not allow for variability (Appendix D.2.2, D.3.2, D.4.2, D.5.2 and D.6.2). The 10 implementations for the five system pairs were programmed in C, and the source code for this experiment can be found on the accompanying CD-ROM.

To evaluate the performance of each system we measured the average CPU time and the memory usage. To compensate for inaccuracies of the system clock, each control cycle was repeated 10’000 times. A test-harness was used to invoke each control cycle and to emulate serial, parallel, hard disk and printer I/O. The results for each system pair provided a measured figure for execution time and memory space overhead for a given amount of variability.

7.4.2 Pipe and Filter Results

Table 7.5 details execution time overhead $O_{exec}$ for Systems 1 to 5. $O_{exec}$ is calculated using Equation 6.1.

<table>
<thead>
<tr>
<th>System</th>
<th>Points of variability</th>
<th>$P_s$</th>
<th>$P_t$</th>
<th>$P_s - P_t$</th>
<th>$C \times V_{min}$</th>
<th>$O_{exec}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Minimum</td>
<td>12.24 s</td>
<td>10.54 s</td>
<td>1.70 s</td>
<td>13'333.3</td>
<td>0.0001275 s</td>
</tr>
<tr>
<td>2</td>
<td>Intermediate</td>
<td>14.83 s</td>
<td>11.59 s</td>
<td>3.24 s</td>
<td>10'400</td>
<td>0.000311538 s</td>
</tr>
<tr>
<td>3</td>
<td>Intermediate</td>
<td>22.35 s</td>
<td>16.21 s</td>
<td>6.14 s</td>
<td>10'250</td>
<td>0.000599024 s</td>
</tr>
<tr>
<td>4</td>
<td>Duplicate</td>
<td>17.46 s</td>
<td>12.03 s</td>
<td>5.43 s</td>
<td>9'000</td>
<td>0.000603333 s</td>
</tr>
<tr>
<td>5</td>
<td>Maximum</td>
<td>19.88 s</td>
<td>13.19 s</td>
<td>6.69 s</td>
<td>8'400</td>
<td>0.000796429 s</td>
</tr>
</tbody>
</table>

Table 7.5: Execution time overhead $O_{exec}$ for Pipe and Filter case study
Figure 7.8 shows a plot of the execution time overhead measurements in Table 7.5 against the number of points of variability for a single controlling cycle. This time the equation is not linear. We used a least squares fit to estimate a quadratic equation relating the points of variability to the execution time overhead. A plot of the residuals (Appendix G.3) indicates that the quadratic equation is probably an ample fit. The residual error between the quadratic equation and the plotted points is 0.0044.

Figure 7.8: Plot of execution time overhead $O_{exec}$ against points of variability for Pipe and Filter case study

Table 7.6 shows the memory overhead results for the Pipe and Filter case study. It details the memory space overhead $O_{mem}$ for systems with 3, 5, 8 ($\times 2$) and 10 points of variability. Equation 6.2 from chapter 6 was used to calculate $O_{mem}$. 

156
Table 7.6: Memory space overhead $O_{mem}$ for Pipe and Filter case study

Figure 7.9 shows a plot of the memory space overhead against number of points of variability for each of the liquid level controlling systems. A linear regression was used to estimate the best fitting equation for the plot. The residual error between the equation and the plotted points is 0.0009. (Appendix G.3).
7.5 Case Study 4: Event-Based / Implicit Invocation

Our fourth case study is taken from an application family of onboard instrument control unit (ICU) software in satellites. In our application family the primary requirements of the ICU software are to: control and monitor onboard instruments; to verify and execute macro-commands from Earth; to generate and report real-time telemetry to Earth; to handle bus interface protocols; and to manage anomalies.

Figure 7.10 demonstrates typical interconnections between an ICU and its onboard instruments. The ICU software controls and monitors the onboard instruments and passes telemetry and status information to other systems through the data bus. It controls the onboard instruments in response to macro-commands and each instrument’s internal state.

![Conceptual layout of on-board instrument control unit](image)

**Figure 7.10:** Conceptual layout of on-board instrument control unit

Onboard instruments can have up to five operating modes: standby; initialising; stabilising; measuring; and shutdown/refuse.
In **Standby** mode onboard instruments are idle. The onboard instruments do not acquire any telemetry; they just poll for a mode-switch macro-command to the instrument control unit.

**Initialising** mode is used to prepare onboard instruments for measuring. Status information may be acquired during initialisation. Examples of onboard instruments that may require and initialise state include heated or cooled instruments, which have a set working temperature.

A **Stabilising** mode is sometimes required to adjust the internal calibration of onboard instruments. Instruments may transmit their status. Again, heated or cooled instruments may have a stabilising mode that adjusts their temperature back to the required set point.

The **Measuring** mode is an instrument’s operating mode. In measuring mode onboard instruments will transmit their status and any required telemetry.

**Shutdown/Refuse** mode is used in the event of an anomaly. The onboard instruments are switched off; any mode switch instructions other than to standby are ignored.

There are two ways in which an instrument may change its operating mode. A macro-command from earth may request a change of mode or the instrument itself may request a different mode. The ICU software must respond to either type of event and switch the onboard instrument appropriately.

We have constructed an application family model for the part of the ICU software that is responsible for controlling mode switching for a single onboard instrument. The application family model is based on control software for a Doppler Orbitograph
and Radio-positioning instrument (DORIS) \([7.10]\)^2, a Microwave Radiometer (MWR) \([7.10]\)^2, and a Radar Altimeter (RA-2) \([7.11]\)^2 that are all part of the ENVISAT–1 payload, and is detailed in Appendix E.1. Our application family model of ICU software notes 30 points of variability, including:

- Optional macro-commands to switch the onboard instrument into the required modes. Note that omitting all events that switch the instrument to an undesired mode will exclude that mode.

- The instrument shall have optional internal events that request it to be switched into a new mode.

- Macro-commands and internal instrument events shall change the system mode and/or the state of the internal instruments – multiple adaptor discriminant.

- An optional RESET macro-command shall restart instruments that have been shut down.

An application family design was built to realise the specifications in our application family model of onboard instrument control unit software (See Appendix E.1.1). The designs are Harel state-transition diagrams \([7.12]\) (design notations are noted in Appendix A.5).

### 7.5.1 Event-Based / Implicit Invocation Experiments

Five requirement specifications were derived from our application family model for five different instrument control units. Each specification incorporates a different amount of variability.

\(^2\) Although our specifications for a family of ICU software are based on specifications taken from the European Space Agency (ESA) documentation, our solutions are completely independent of ESA. Any resemblance to their actual products is coincidental.
System 1 can only switch its instrument from standby to measuring. It incorporates two points of variability. (Appendix E.2).

System 2 can switch its instrument from standby to initialising and back, and from initialising to measuring. It also responds to shutdown and reset events and incorporates 10 points of variability. (Appendix E.3).

System 3 can switch its instrument from standby to stabilising and back, from stabilising to measuring and back, and from measuring to standby. It also responds to the instrument’s internal states. This system incorporates 12 points of variability. (Appendix E.4).

System 4 can switch from standby to initialising, from initialising to stabilising and back, and from stabilising to measuring and back. From any mode it is able to switch back to standby or shutdown. It incorporates 20 points of variability. (Appendix E.5).

System 5 can switch from standby, initialising, stabilising and measuring to any state. It responds to the instrument’s internal states, to shutdown and to reset events and incorporates all 30 points of variability. (Appendix E.6).

For each of the five system specifications we constructed two designs, each with a corresponding implementation. The first design is derived from the application family design (Appendix E.1.1) and includes common interfaces at all the points of variability (Appendix E.2.1, E.3.1, E.4.1, E.5.1 and E.6.1). The second design is built using a traditional one-off approach that does not allow for variability (Appendix E.2.2, E.3.2, E.4.2, E.5.2 and E.6.2). The corresponding system implementations of the five system pairs were done in Ada95. All source code for this case study can be found on the accompanying CD-ROM.
To evaluate the performance of each system we measured the average CPU time and the memory usage. To compensate for any inaccuracies of the system clock each execution cycle was repeated 60'000 times. A test harness was used to invoke each cycle.

### 7.5.2 Event-Based / Implicit Invocation Results

Table 7.7 shows the execution time overhead $O_{exec}$ for Systems 1 to 5. $O_{exec}$ is calculated using Equation 6.1.

<table>
<thead>
<tr>
<th>System</th>
<th>Points of variability</th>
<th>$P_s$</th>
<th>$P_r$</th>
<th>$P_s \cdot P_r$</th>
<th>$C \cdot V_{est}$</th>
<th>$O_{exec}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Minimum</td>
<td>0.58 s</td>
<td>0.14 s</td>
<td>0.44 s</td>
<td>60'000</td>
<td>0.000007 s</td>
</tr>
<tr>
<td>2</td>
<td>Intermediate</td>
<td>7.82 s</td>
<td>1.22 s</td>
<td>6.60 s</td>
<td>60'000</td>
<td>0.000110 s</td>
</tr>
<tr>
<td>3</td>
<td>(~ Duplicate)</td>
<td>12.00 s</td>
<td>2.19 s</td>
<td>9.81 s</td>
<td>60'000</td>
<td>0.000164 s</td>
</tr>
<tr>
<td>4</td>
<td>Intermediate</td>
<td>27.94 s</td>
<td>3.75 s</td>
<td>24.19 s</td>
<td>60'000</td>
<td>0.000403 s</td>
</tr>
<tr>
<td>5</td>
<td>Maximum</td>
<td>62.99 s</td>
<td>8.92 s</td>
<td>54.07 s</td>
<td>60'000</td>
<td>0.000901 s</td>
</tr>
</tbody>
</table>

**Table 7.7**: Execution time overhead $O_{exec}$ for Event-Based / Implicit Invocation case study

Figure 7.11 shows a graph of the execution time overhead $O_{exec}$ against the number of points of variability for a single controlling cycle. Like the Pipe and Filter experiments, the equation was observed as non-linear. We used a *least squares fit* to estimate a quadratic equation relating the points of variability to the execution time overhead. A plot of the residuals (Appendix G.4) helped to assess the adequacy of fit. The residual error between the quadratic equation and the plotted points is 0.0002.
Figure 7.11: Plot of execution time overhead $O_{exec}$ against points of variability for Event-Based / Implicit Invocation case study

Table 7.8 shows the memory overhead results from the Event-Based/Implicit Invocation case study. It details the memory space overhead $O_{mem}$ for systems with 2, 10, 12, 20 and 30 points of variability. We calculate $O_{mem}$ using Equation 6.2.

<table>
<thead>
<tr>
<th>System</th>
<th>Points of variability</th>
<th>$M_e$</th>
<th>$M_r$</th>
<th>$O_{mem}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 (Minimum)</td>
<td>283'701 Bytes</td>
<td>283'617 Bytes</td>
<td>84 Bytes</td>
</tr>
<tr>
<td>2</td>
<td>10 (Intermediate)</td>
<td>284'934 Bytes</td>
<td>284'530 Bytes</td>
<td>404 Bytes</td>
</tr>
<tr>
<td>3</td>
<td>12 (- Duplicate)</td>
<td>284'973 Bytes</td>
<td>284'541 Bytes</td>
<td>432 Bytes</td>
</tr>
<tr>
<td>4</td>
<td>20 (Intermediate)</td>
<td>287'131 Bytes</td>
<td>286'407 Bytes</td>
<td>724 Bytes</td>
</tr>
<tr>
<td>5</td>
<td>30 (Maximum)</td>
<td>288'824 Bytes</td>
<td>287'744 Bytes</td>
<td>1'080 Bytes</td>
</tr>
</tbody>
</table>

Table 7.8: Memory space overhead $O_{mem}$ for Event-Based / Implicit Invocation case study
Figure 7.12 shows a plot of $O_{\text{mem}}$ against number of points of variability for each of the onboard instrument control systems. A linear regression was used to estimate the best fitting equation for the plot. The residual error between the equation and the plotted points is 0.0034. (Appendix G.4).

![Plot of memory overhead $O_{\text{mem}}$ against points of variability for Event-Based / Implicit Invocation case study](image)

**Figure 7.12:** Plot of memory overhead $O_{\text{mem}}$ against points of variability for Event-Based / Implicit Invocation case study

### 7.6 Case Study 5: Repository

Our final experiments are taken from an application family of image processing systems. Image processing is used in many fields from picture enhancement, shape and pattern recognition, through to compression.

Our case study is for systems that process weather images. Typically, a digital satellite photograph is processed to aid recognition of weather systems and fronts.
Our application family model performs an area-based colour-spectrum analysis on raw satellite images.

Figure 7.13 shows an example satellite image before and after processing. The colour–spectrum analysis is tuned to highlight cloud regions and their intensities. Other information may be added, including landmarks, maps, longitudinal and latitudinal information, a border, and labels.

Figure 7.13: Example weather images, before and after colour-spectrum analysis

An MRAM application family model was constructed for systems that process weather satellite images. The application family model is derived from information on digital image processing in [7.13] and [7.14], and is detailed in Appendix F.1. Our application family of image processing systems notes seven points of variability, including:
Two different types of colour-spectrum analysis, one to highlight cloud regions and their intensities, and one to highlight only the most intense regions – single adaptor discriminant.

Identified cloud regions shall be outlined – optional discriminant.

The original satellite image shall be merged with the processed image – optional discriminant.

Cartographic information shall be added. This includes lines of longitude and latitude, landmarks and/or a map outline – multiple adaptor discriminant.

A black border shall be added to the images – optional discriminant.

A time stamp shall be attached within the border – optional discriminant.

The processed image may be output to different media – single adaptor discriminant.

An application family design was engineered to realise the features in the application family model (See Appendix F.1.1). This design takes the form of a Yourdon’s data-flow diagram [7.3]. Design notations can be found in Appendix A.6.

7.6.1 Repository Experiments

Five requirements specifications were derived from the application family model for five different weather satellite image processing systems. Each specification incorporates a different amount of variability.
System 1 highlights the cloud regions and their intensity and adds lines of longitude and latitude, landmarks and a map outline. It incorporates three points of variability. (Appendix F.2).

System 2 displays the most intense cloud regions, merges the original image and adds landmarks. It incorporates four points of variability. (Appendix F.3).

System 3 shows outlined cloud regions and their intensities, merges the original image, adds landmarks and a map outline, and borders the image. It incorporates six points of variability. (Appendix F.4).

System 4 displays the most intense cloud regions, merges the original image, adds a map outline and lines of longitude and latitude, borders the image and adds a time stamp. It also incorporates six points of variability. (Appendix F.5).

System 5 is similar to System 3, with the addition of lines of longitude and latitude, and a time stamp in the border. This system incorporates seven points of variability. (Appendix F.6). Figure 7.13 shows an example output of System 5.

As with previous experiments, we constructed two system designs for each of the five system specifications. The first design is derived from the application family design (Appendix F.1.1) and includes common interfaces at all the points of variability (Appendix F.2.1, F.3.1, F.4.1, F.5.1 and F.6.1). The second design is built using a traditional one-off approach that does not allow for variability (Appendix F.2.2, F.3.2, F.4.2, F.5.2 and F.6.2). The implementations of the five system pairs were done in C. All source code for this case study can be found on the accompanying CD-ROM.

To evaluate the performance of each system we measured the average CPU time and the memory usage that was required to process a single image. To compensate for any inaccuracies of the system clock each processing cycle was repeated 10'000
times. A test harness was used to invoke each cycle. The results for each system pair provided a measured figure for execution time and memory space overhead for a given amount of variability.

7.6.2 Repository Results

Table 7.9 shows the measured results for the Repository case studies. It details the execution time overhead $O_{exec}$ for Systems 1 to 5. We calculate $O_{exec}$ using Equation 6.1.

<table>
<thead>
<tr>
<th>System</th>
<th>Points of variability</th>
<th>$P_s$ (s)</th>
<th>$P_i$ (s)</th>
<th>$P_s - P_i$ (s)</th>
<th>$C \times V_{set}$</th>
<th>$O_{exec}$ (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3 (Minimum)</td>
<td>381.85</td>
<td>238.48</td>
<td>143.37</td>
<td>10'000</td>
<td>0.014337</td>
</tr>
<tr>
<td>2</td>
<td>4 (Intermediate)</td>
<td>382.28</td>
<td>192.57</td>
<td>189.71</td>
<td>10'000</td>
<td>0.018971</td>
</tr>
<tr>
<td>3</td>
<td>6 (Intermediate)</td>
<td>626.59</td>
<td>336.75</td>
<td>289.84</td>
<td>10'000</td>
<td>0.028984</td>
</tr>
<tr>
<td>4</td>
<td>6 (Duplicate)</td>
<td>634.50</td>
<td>332.36</td>
<td>302.14</td>
<td>10'000</td>
<td>0.030214</td>
</tr>
<tr>
<td>5</td>
<td>7 (Maximum)</td>
<td>771.59</td>
<td>421.17</td>
<td>350.42</td>
<td>10'000</td>
<td>0.035042</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System</th>
<th>Points of variability</th>
<th>$P_s$ (s)</th>
<th>$P_i$ (s)</th>
<th>$P_s - P_i$ (s)</th>
<th>$C \times V_{set}$</th>
<th>$O_{exec}$ (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3 (Minimum)</td>
<td>381.85</td>
<td>238.48</td>
<td>143.37</td>
<td>10'000</td>
<td>0.014337</td>
</tr>
<tr>
<td>2</td>
<td>4 (Intermediate)</td>
<td>382.28</td>
<td>192.57</td>
<td>189.71</td>
<td>10'000</td>
<td>0.018971</td>
</tr>
<tr>
<td>3</td>
<td>6 (Intermediate)</td>
<td>626.59</td>
<td>336.75</td>
<td>289.84</td>
<td>10'000</td>
<td>0.028984</td>
</tr>
<tr>
<td>4</td>
<td>6 (Duplicate)</td>
<td>634.50</td>
<td>332.36</td>
<td>302.14</td>
<td>10'000</td>
<td>0.030214</td>
</tr>
<tr>
<td>5</td>
<td>7 (Maximum)</td>
<td>771.59</td>
<td>421.17</td>
<td>350.42</td>
<td>10'000</td>
<td>0.035042</td>
</tr>
</tbody>
</table>

Table 7.9: Execution time overhead $O_{exec}$ for Repository case study

Figure 7.14 shows a plot of execution time overhead $O_{exec}$ against the number of points of variability for a single image processing cycle. A linear regression was used to estimate the best fitting equation. The residual error between the equation and the plotted points is 0.0062. (Appendix G.5).
Figure 7.14: Plot of execution time overhead $O_{exec}$ against points of variability for Repository case study

Table 7.10 details the measured memory space overhead $O_{mem}$ for systems with 3, 4, 6 ($\times 2$) and 7 points of variability. $O_{mem}$ is calculated using Equation 6.2.

<table>
<thead>
<tr>
<th>System</th>
<th>Points of Variability</th>
<th>$M_s$</th>
<th>$M_i$</th>
<th>$O_{mem}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3 (Minimum)</td>
<td>2'187'264 Bytes</td>
<td>1'884'160 Bytes</td>
<td>303'104 Bytes</td>
</tr>
<tr>
<td>2</td>
<td>4 (Intermediate)</td>
<td>2'486'272 Bytes</td>
<td>2'082'960 Bytes</td>
<td>403'312 Bytes</td>
</tr>
<tr>
<td>3</td>
<td>6 (Intermediate)</td>
<td>3'092'480 Bytes</td>
<td>2'494'464 Bytes</td>
<td>598'016 Bytes</td>
</tr>
<tr>
<td>4</td>
<td>6 (Duplicate)</td>
<td>3'112'960 Bytes</td>
<td>2'510'848 Bytes</td>
<td>602'112 Bytes</td>
</tr>
<tr>
<td>5</td>
<td>7 (Maximum)</td>
<td>3'420'160 Bytes</td>
<td>2'716'456 Bytes</td>
<td>703'704 Bytes</td>
</tr>
</tbody>
</table>

Table 7.10: Memory space overhead $O_{mem}$ for Repository case study

Figure 7.15 shows a plot of memory space overhead $O_{mem}$ against number of points of variability for our image processing system examples. Again, a linear regression
was used to estimate an adequate equation for the graph. The residual error between the equation and the plotted points is 0.0002. (Appendix G.5).

![Graph](image)

**Figure 7.15:** Plot of memory overhead $O_{\text{mem}}$ against points of variability for Repository case study

### 7.7 Summary

In this chapter we detailed the experiments that we used to help evaluate our thesis. We measured sets of execution time and memory space overhead for five case studies, each case study used a different common interface approach based on a different architectural style. Chapter 8 will discuss our results.

### 7.8 References


CHAPTER 8

Conclusion

8.1 Introduction

Where Chapter 7 details our experimental results, this chapter assesses the strength and validity of those results with respect to our hypothesis. We discuss our findings and detail our conclusions.

8.2 Evaluation

The strength of correlation between execution time overhead and memory space overhead, and points of variability can be calculated as a residual error between the fitted equations and the measured data. All equations of the form \( y = f(x) \) have an exact correlation between \( x \) and \( y \), namely \( f \). A residual error establishes the degree to which a fitted equation represents the points it was derived from. The smaller the residual error is, the stronger the correlation. A residual error of less than 0.05 is accepted as an indication that an equation correlates exactly to the points it was derived from (James [8.1]).
Table 8.1 details the residual errors between the fitted equations (Chapter 7, Figures 7.2, 7.5, 7.8, 7.11 and 7.14) and the measured execution time overhead for each of our case studies. All figures imply a very strong correlation between points of variability and execution time overhead.

<table>
<thead>
<tr>
<th>Equation type</th>
<th>Main/Subroutine</th>
<th>Object-Oriented</th>
<th>Pipe and filter</th>
<th>Event-Based / Implicit Invocation</th>
<th>Repository</th>
</tr>
</thead>
<tbody>
<tr>
<td>Residual error</td>
<td>Linear</td>
<td>Linear</td>
<td>Quadratic</td>
<td>Quadratic</td>
<td>Linear</td>
</tr>
<tr>
<td></td>
<td>0.0075</td>
<td>0.0106</td>
<td>0.0044</td>
<td>0.0002</td>
<td>0.0062</td>
</tr>
</tbody>
</table>

**Table 8.1**: Residual errors for execution time experiments

Table 8.2 shows the residual errors between the fitted equations (Chapter 7, Figures 7.3, 7.6, 7.9, 7.12 and 7.15) and the measured memory space overhead for each of our case studies. We can observe a very strong correlation between points of variability and memory space overhead.

<table>
<thead>
<tr>
<th>Equation type</th>
<th>Main/Subroutine</th>
<th>Object-Oriented</th>
<th>Pipe and filter</th>
<th>Event-Based / Implicit Invocation</th>
<th>Repository</th>
</tr>
</thead>
<tbody>
<tr>
<td>Residual error</td>
<td>Linear</td>
<td>Linear</td>
<td>Linear</td>
<td>Linear</td>
<td>Linear</td>
</tr>
<tr>
<td></td>
<td>0.0066</td>
<td>0.0271</td>
<td>0.0009</td>
<td>0.0034</td>
<td>0.0002</td>
</tr>
</tbody>
</table>

**Table 8.2**: Residual errors for memory space experiments

Student’s t-tests [8.2] are a statistical test to help us examine the validity of measured data. They are used to calculate a probability that measured data is representative and not due to random variance. In situations where there are few measured data points (a low degree of freedom) even a single value that is not representative can have a significant impact on overall observations. In these case studies we only have five measured data points for each of our graphs (equates to three degrees of freedom).
Table 8.3 shows the calculated t-stat\textsuperscript{1} values for the measured execution time overhead and the closest t-stat values in a t-distribution table. The figures indicate a probability greater than 99.9\% that our execution time measurements are representative, and not due to random variance.

<table>
<thead>
<tr>
<th></th>
<th>Main / Subroutine</th>
<th>Object-Oriented</th>
<th>Pipe and Filter</th>
<th>Event-Based / Implicit Invocation</th>
<th>Repository</th>
</tr>
</thead>
<tbody>
<tr>
<td>t-stat calculated</td>
<td>47.49</td>
<td>51.37</td>
<td>24.96</td>
<td>73.96</td>
<td>88.62</td>
</tr>
<tr>
<td>t-stat from t-distribution</td>
<td>12.92</td>
<td>12.92</td>
<td>12.92</td>
<td>12.92</td>
<td>12.92</td>
</tr>
<tr>
<td>Probability</td>
<td>&gt; 99.9%</td>
<td>&gt; 99.9%</td>
<td>&gt; 99.9%</td>
<td>&gt; 99.9%</td>
<td>&gt; 99.9%</td>
</tr>
</tbody>
</table>

**Table 8.3**: Student’s t-test for execution time experiments

Table 8.4 shows the calculated t-stat values for the measured memory space overhead and the closest t-stat values in a t-distribution table. The figures indicate a probability greater than 99.9\% that our memory space measurements are representative, and not due to random variance.

<table>
<thead>
<tr>
<th></th>
<th>Main / Subroutine</th>
<th>Object-Oriented</th>
<th>Pipe and Filter</th>
<th>Event-Based / Implicit Invocation</th>
<th>Repository</th>
</tr>
</thead>
<tbody>
<tr>
<td>t-stat calculated</td>
<td>50.44</td>
<td>31.73</td>
<td>193.84</td>
<td>65.54</td>
<td>474.59</td>
</tr>
<tr>
<td>t-stat from t-distribution</td>
<td>12.92</td>
<td>12.92</td>
<td>12.92</td>
<td>12.92</td>
<td>12.92</td>
</tr>
<tr>
<td>Probability</td>
<td>&gt; 99.9%</td>
<td>&gt; 99.9%</td>
<td>&gt; 99.9%</td>
<td>&gt; 99.9%</td>
<td>&gt; 99.9%</td>
</tr>
</tbody>
</table>

**Table 8.4**: Student’s t-test for memory space experiments

Residual plots [8.3] were used to assess the adequacy of each fitted equation. Attention is paid to whether or not the plotted residuals seem randomly distributed. If the plot is non-random, there is a strong likelihood that the wrong equation was fitted to our initial data points. Our residual plots (Appendix G) are all distributed randomly. This is a good indication, that the fitted equations are adequate.

\textsuperscript{1} Explained in Chapter 6, Section 6.4.4.
Our experiments indicate a very strong correlation between both execution time overhead and number of points of variability, and memory space overhead and number of points of variability, for each of the examined common interface approaches, for all five case studies. This is good evidence that that other common interface approaches will have a similar correlation between overhead and number of points of variability.

8.3 Discussion

Many application family engineering methods could have been used to assess the problem of possible overhead when reengineering common interfaces. We chose to use MRAM [8.4] because it is not tied to any design method. This allowed us to experiment with many different ways of designing and implementing points of variability using common interfaces. Many other application family engineering methods are not tied to a design method and could also have been used to specify our case studies; these include FODA [8.5], ODM [8.6], Synthesis [8.7], PuLSE [8.8], and DSSA [8.9]. Our results should be equally significant to any application family engineering method, provided that they chose to employ a common interface approach for modelling points of variability in an application family design.

In Chapter 4 we noted a taxonomy of methods for handling variability in application family designs and implementations, based on Cortese et al [8.10]. Our preferred method of denoting points of variability in an application family design is though common interfaces to groups or sub-collections of application family design components. While this view is shared by a number of other application family engineering methods (including RSEB [8.11], FeatuRSEB [8.12], JODA[8.13], PuLSE [8.8], DSSA [8.9], Batory [8.14], Keepence et al [8.15]), it is not exclusive. Parameterisation, metamodels, domain-specific languages, macro pre-processors and code generators are arguably just as suitable for managing variability. Our research
has no value to application family engineering methods that do not use common interfaces at points of variability.

Chapter 4 also introduced universal design patterns to model qualitative and some quantitative points of variability using the MRAM discrimination mechanisms. Where possible we used these designs in our case studies. However, these patterns are just recommendations. We do not assume that they are sufficient to cover all situations for all design methods and accept that some designs may require alternative patterns. Nonetheless, our results should provide equally valuable in either situation, so long as a common interface approach is employed.

It is perfectly reasonable for an application family design to use different techniques for handling variability within a same design. For example, a single application family design may use both parameterisation and common interfaces to model points of variability. It is equally plausible that a single application family use a mixture of design methods and programming languages. Our case studies do not use hybrid discrimination techniques, design methods or programming languages. While this may not reflect all real-life situations, our results should still be applicable to individual parts of a hybrid application family design that denote variability using the same common interface. Further work should consider the applicability of our research to hybrid designs and code.

An alternative approach to conducting this experiment might have been to use case studies with no functional code content; that is, the structure of each system would exist but not do anything. For example, a Main/Subroutine case study could have been made up of many functions, but which contain no code. This would ensure that any performance differences between system pairs were not attributed to their coding differences. However, this is not realistic. The code content of each pair does affect execution time and memory usage. The challenge is to be sure that the code in each
of the system pairs is comparable. In our experiment we have tried to the best of our ability to satisfy this criteria.

8.4 Conclusion

Our hypothesis is that there exists a correlation between the number of points of variability incorporated into a single system, built from an application family model, and execution time and memory space overhead. We made six assumptions:

- Reuse is engineered using an application family engineering approach. Our preferred method is MRAM [8.4], detailed in Chapter 3.

- Common interfaces are used at points of variability in application family designs. In Chapter 4 we presented three universal patterns, based on Keepence et al [8.15], that model points of variability through a common interface.

- Application family designs consistently use the same type of architectural or design component to denote all points of variability.

- The corresponding application family implementations use the same type of code component to manage all points of variability.

- Execution time overhead is defined as the difference in execution time for a single system derived from an application family with common interfaces at points of variability, and an equivalent one-off system designed without reuse, with no common interfaces.

- Memory space overhead is defined as the difference in heap, stack and data segment memory that a single system derived from an application family with common interfaces at points of variability requires over an equivalent one-off system designed without reuse.
Our experimental approach shows that our hypothesis holds true for all five common interface approaches. The convincing correlation between overhead and number of points of variability for the five case studies is a very strong indication that other common interface approaches will, on a balance of probability, have a correlation between overhead and number of points of variability too. The implications of our results are as follows:

- By definition, a correlation between execution time and memory space overhead, and points of variability implies that overhead can be predicted and even quantified.

- A one-off single system, or parts thereof, with real-time and memory constraints may no longer be able to meet constraints if reengineered as application family design and code assets. The implication that overhead can be predicted and possibly quantified may help decide the economic viability of a reengineering process.

- Existing real-time and memory constraints may no longer be valid if an application family design and implementation is evolved to include new points of variability. The implication that overhead can be predicted and possibly quantified may help decide the economic viability of evolving an application family.

8.5 Future Work

In Chapter 5 we noted three technical reasons why many reuse techniques are not easily applied to embedded systems: Highly coupled designs; existing constraints restricting possible alternatives; and the potential introduction of execution time and memory space overhead. Our research only focused on the possible introduction of execution time and memory space overhead, and only for common interface
approaches. Future work in this area will include examining the potential introduction of overhead when non-common interface methods (e.g. parameterisation, metamodels, domain-specific languages, macro pre-processors and code generators) or hybrid common interface designs are used to manage variability.

In our experiments we can observe that there is always a correlation between overhead and points of variability; the behaviours of overhead are a different matter. Unlike, memory overhead that seems to be linearly correlated to the number of points of variability for all case studies, execution time overhead does not. We can observe two distinct behavioural patterns: linear behaviour (main/subroutine, object-orientation and repository) and quadratic behaviour (pipe and filter, event-based/implicit invocation). We can hypothesise that the behaviour of execution time overhead will not be linearly related to the number of points of variability, if the execution of a common interfaces is affected by the total number of common interfaces. We can demonstrate our reasoning behind this, with an example. Consider an event-based system, where an event-handler is responsible for dispatching a raised event to the relevant procedures. The more procedures that register with events, the more procedures need to be evaluated by the event-handler with every raised event. If the amount of registered procedures affect the event-handler’s evaluation time, the execution time of every event-call would clearly increase with new procedure registrations – a quadratic increase.

Further work could investigate this in more detail; identifying potential other types of correlation, other than linear and quadratic ones.

8.6 References


## Appendix A

### Glossary

## A.1 Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CI</td>
<td>Common Interface</td>
</tr>
<tr>
<td>DSSA</td>
<td>Domain Specific Software Architectures</td>
</tr>
<tr>
<td>FODA</td>
<td>Feature Oriented Domain Analysis</td>
</tr>
<tr>
<td>JODA</td>
<td>Joint Integrated Avionics Working Group’s Object-Oriented Domain Analysis</td>
</tr>
<tr>
<td>MA</td>
<td>Multiple Adaptor Discriminant</td>
</tr>
<tr>
<td>MRAM</td>
<td>Method for Requirements Authoring and Management</td>
</tr>
<tr>
<td>ODM</td>
<td>Organisation Domain Modelling</td>
</tr>
<tr>
<td>OP</td>
<td>Optional Adaptor Discriminant</td>
</tr>
<tr>
<td>PuLSE</td>
<td>Product Line Software Engineering</td>
</tr>
<tr>
<td>RSEB</td>
<td>Reuse-Driven Software Engineering Business</td>
</tr>
<tr>
<td>SA</td>
<td>Single Adaptor Discriminant</td>
</tr>
<tr>
<td>Synthesis / RSP</td>
<td>Synthesis / Reuse-Driven Software Process</td>
</tr>
</tbody>
</table>
A.2 Design Notations – Main Subroutine

- Process Bubble
- External Process
- Process/Control Flow
- Exclusive-Or. Used to denote build-time decisions in the application family design.

A.3 Design Notation – Object Orientation

- Package
- Class
  - Class Name
  - Attributes
  - Operations
  - Parameterised class / template class
    - Parameterised class that is instantiated with argument “Actual”
- Dependency between two or more packages or two or more classes
- Generalisation / Specialisation
- Aggregation
A.4 Design Notations – Pipe and Filter

- Filter
- Pipe
- Exclusive-Or. Used to denote build-time decisions in the application family design.

A.5 Design Notations – Event Based

- State
- Event
- Optional event. Used to denote build-time decisions in the application family design.
- Start
- Stop
- Exclusive-Or. Used to denote build-time decisions in the application family design.
A.6 Design Notations – Repository

- Knowledge source
- Repository / shared memory

Interaction between knowledge sources and repositories

Optional Interaction. Used to denote build-time decisions in the application family design.

Exclusive-Or. Used to denote build-time decisions in the application family design.
Appendix B

Main/Subroutine Case Study

B.1 Application Family Model of Requirements

1\textit{ MA} A wash-cycle on a washing machine shall consist of many phases.
1.1 Pre-wash phase
1.1.1 \textit{SA} Prior to a pre-wash phase the internal water container shall be heated.
1.1.1.1 The water shall be heated to 30\textdegree{} C.
1.1.1.2 The water shall be heated to 40\textdegree{} C.
1.1.2 The drum shall be filled with the heated water.
1.1.3 The drum shall spin at speed setting 1 for 40 time units.
1.1.4 Following the spin, the machine shall be drained.
1.2 Main-wash phase
1.2.1 \textit{SA} Prior to a wash phase the internal water container shall be heated.
1.2.1.1 The water shall be heated to 30\textdegree{} C.
1.2.1.1.1 The drum shall be filled with the heated water.
1.2.1.2 The drum shall spin at speed setting 2 for 80 time units.
1.2.1.2.1 The water shall be heated to 40\textdegree{} C.
1.2.1.2.2 The drum shall be filled with the heated water.
1.2.1.3 The water shall be heated to 50\textdegree{} C.
1.2.1.3.1 The drum shall be filled with the heated water.
1.2.1.3.2 The drum shall spin at speed setting 2 for 80 time units.
1.2.1.4 The water shall be heated to 60\textdegree{} C.
1.2.1.4.1 \textit{OP} Check the ECO button’s state.
1.2.1.4.1.1 If the button is pressed, only heat the water to 50\textdegree{} C.
1.2.1.4.2 The drum shall be filled with the heated water.
1.2.1.4.3 The drum shall spin at speed setting 3 for 90 time units.
1.2.1.5 The water shall be heated to 95° C.
1.2.1.5.1 \(OP\) Check the ECO button’s state.
1.2.1.5.1.1 If the button is pressed, only heat the water to 60° C.
1.2.1.5.2 The drum shall be filled with the heated water.
1.2.1.5.3 The drum shall spin at speed setting 3 for 90 time units.
1.2.2 Following the spin, the machine shall be drained.
1.3 Rinse phase
1.3.1 The drum shall be filled with cold water.
1.3.2 The drum shall spin at speed setting 1 for 25 time units.
1.3.3 \(OP\) Check the rinse-hold button’s state.
1.3.3.1 Wait for rinse-hold button to be released.
1.3.4 The machine shall be drained.
1.4 \(SA\) Spin phase
1.4.1 The spin phase shall be a short spin phase
1.4.1.1 The drum shall spin at speed setting 30 for 65 time units.
1.4.1.2 \(OP\) Check the reduced-spin button’s state
1.4.1.2.1 If pressed, the drum shall only spin for 45 time units.
1.4.1.3 Any excess water shall be drained.
1.4.2 The spin phase shall be a long spin phase
1.4.2.1 The drum shall spin at speed setting 30 for 120 time units.
1.4.2.2 \(OP\) Check the reduced-spin button’s state
1.4.2.2.1 If pressed, the drum shall only spin for 65 time units.
1.4.2.3 Any excess water shall be drained.
1.5 \(SA\) Tumble-dry phase
1.5.1 The drying phase shall do a low-temperature dry.
1.5.1.1 \(OP\) The dry indicator light shall be lit during the drying phase.
1.5.1.1.1 The light shall be switched off after the drying phase.
1.5.1.2 The machine shall be drained before the drying phase.
1.5.1.3 The drum shall be heated to 80° C.
1.5.1.4 The drum shall cycle at speed setting 2.
1.5.1.5 The external timer shall control the duration of the drying phase.
1.5.1.6 Any excess water shall be drained.
1.5.2 The drying phase shall do a high-temperature dry.
1.5.2.1 \(OP\) The dry indicator light shall be lit during the drying phase.
1.5.2.1.1 The light shall be switched off after the drying phase.
1.5.2.2 The machine shall be drained before the drying phase.
1.5.2.3 The drum shall be heated to 100° C.
1.5.2.4 The drum shall cycle at speed setting 2.
1.5.2.5 The external timer shall control the duration of the drying phase.
1.5.2.6 Any excess water shall be drained.
2 The drum’s door shall be locked prior to any wash cycle
2.1 \(SA\) The door shall be unlocked
2.1.1 Immediately after a cycle is complete.
2.1.2 After a 2 minute delay

B.1.1 Application Family Design

![Diagram](image_url)

Figure B.1: Context diagram

![Diagram](image_url)

Figure B.2: CI WashCycle (Context)

![Diagram](image_url)

Figure B.3: WashCycle (1)
**Figure B.4:** CI Prewash (1.2)

**Figure B.5:** CI Wash (1.3)

**Figure B.6:** Wash 60° (1.3.4)
Figure B.7: Wash 95° (1.3.5)

Figure B.8: CI CheckEcoButton (1.3.4.1)
Figure B.9: Rinse (1.4)

Figure B.10: CI CheckRinseHoldButton (1.4.2)

Figure B.11: CI Spin (1.5)
Figure B.12: ShortSpin (1.5.1)

Figure B.13: LongSpin (1.5.2)
Figure B.14: CI ReducedSpin (1.5.1.1)

Figure B.15: CI Dry (1.6)

Figure B.16: LowTempDry (1.6.1)
Figure B.17: HighTempDry (1.6.2)

Figure B.18: CI SetIndicatorLight (1.6.1.1)

Figure B.19: CI DoorUnlock (1.7)
B.2 Single System 1 – Requirements Specification

1. **MA** A wash-cycle on a washing machine shall consist of many phases.

1.3 **Rinse phase**

1.3.1 The drum shall be filled with cold water.

1.3.2 The drum shall spin at speed setting 1 for 25 time units.

1.3.4 The machine shall be drained.

2 The drum’s door shall be locked prior to any wash cycle

2.1 **SA** The door shall be unlocked

2.1.1 Immediately after a cycle is complete.

B.2.1 Single System 1 – Design with Variability

---

**Figure B.20:** Context diagram of single system 1

**Figure B.21:** CI WashCycle (Context)

**Figure B.22:** WashCycle (1)
B.2.2 Single System 1 – Design without Variability

Figure B.24: Context diagram of single system 1

Figure B.25: WashCycle (Context)

B.3 Single System 2 – Requirements Specification

1 $MA$ A wash-cycle on a washing machine shall consist of many phases.
1.5 $SA$ Tumble-dry phase
1.5.1 The drying phase shall do a low-temperature dry.
1.5.1.1 $OP$ The dry indicator light shall be lit during the drying phase.
1.5.1.1.1 The light shall be switched off after the drying phase.
1.5.1.2 The machine shall be drained before the drying phase.
1.5.1.3 The drum shall be heated to 80° C.
1.5.1.4 The drum shall cycle at speed setting 2.
1.5.1.5 The external timer shall control the duration of the drying phase.
1.5.1.6 Any excess water shall be drained.
2 The drum’s door shall be locked prior to any wash cycle
2.1 $SA$ The door shall be unlocked
2.1.2 After a 2 minute delay

Figure B.23: CI DoorUnlock (1.7)
B.3.1 Single System 2 – Design with Variability

![Diagram](image)

**Figure B.26:** Context diagram of single system 2

![Diagram](image)

**Figure B.27:** CI WashCycle (Context)

![Diagram](image)

**Figure B.28:** WashCycle (1)

![Diagram](image)

**Figure B.29:** CI Dry (1.6)
B.3.2 Single System 2 – Design without Variability

Figure B.30: LowTempDry (1.6.1)

Figure B.31: CI SetIndicatorLight (1.6.1.1)

Figure B.32: CI DoorUnlock (1.7)

Figure B.33: Context diagram of single system 2
B.4 Single System 3 – Requirements Specification

1. **MA** A wash-cycle on a washing machine shall consist of many phases.
2. **Main-wash phase**
   1.2 **Prior to a wash phase the internal water container shall be heated.**
   1.2.1 **The water shall be heated to 40° C.**
   1.2.1.1 **The drum shall be filled with the heated water.**
   1.2.1.2 **The drum shall spin at speed setting 2 for 80 time units.**
   1.2.2 **Following the spin, the machine shall be drained.**
3. **Rinse phase**
   1.3 **The drum shall be filled with cold water.**
   1.3.2 **The drum shall spin at speed setting 1 for 25 time units.**
4. **OP**
   1.3.3 **Check the rinse-hold button’s state.**
   1.3.3.1 **Wait for rinse-hold button to be released.**
   1.3.4 **The machine shall be drained.**
5. **Drum’s door shall be locked prior to any wash cycle**
6. **The door shall be unlocked**
   2.1 **Immediately after a cycle is complete.**
B.4.1 Single System 3 – Design with Variability

**Figure B.36**: Context diagram of single system 3

**Figure B.37**: CI WashCycle (Context)

**Figure B.38**: WashCycle (1)

**Figure B.39**: CI Wash (1.3)
B.4.2 Single System 3 – Design without Variability

**Figure B.40:** Rinse (1.4)

**Figure B.41:** CI CheckRinseHoldButton (1.4.2)

**Figure B.42:** CI DoorUnlock (1.7)

**Figure B.43:** Context diagram of single system 3
A wash-cycle on a washing machine shall consist of many phases.

1.1 Pre-wash phase

1.1.1 Prior to a pre-wash phase the internal water container shall be heated.

1.1.1.1 The water shall be heated to 30°C.

1.1.2 The drum shall be filled with the heated water.

1.1.3 The drum shall spin at speed setting 1 for 40 time units.

1.1.4 Following the spin, the machine shall be drained.

1.2 Main-wash phase

1.2.1 Prior to a wash phase the internal water container shall be heated.

1.2.1.3 The water shall be heated to 50°C.

1.2.1.3.1 The drum shall be filled with the heated water.

1.2.1.3.2 The drum shall spin at speed setting 2 for 80 time units.

1.2.2 Following the spin, the machine shall be drained.

1.3 Rinse phase

1.3.1 The drum shall be filled with cold water.

1.3.2 The drum shall spin at speed setting 1 for 25 time units.

1.3.3 Check the rinse-hold button’s state.

1.3.3.1 Wait for rinse-hold button to be released.

1.3.4 The machine shall be drained.

1.4 Spin phase

1.4.1 The spin phase shall be a short spin phase

1.4.1.1 The drum shall spin at speed setting 30 for 65 time units.

1.4.1.2 Check the reduced-spin button’s state

1.4.1.2.1 If pressed, the drum shall only spin for 45 time units.
1.4.1.3 Any excess water shall be drained.
2 The drum’s door shall be locked prior to any wash cycle
2.1 SA The door shall be unlocked
2.1.1 Immediately after a cycle is complete.

B.5.1 Single System 4 – Design with Variability

**Figure B.46**: Context diagram of single system 4

**Figure B.47**: CI WashCycle (Context)

**Figure B.48**: WashCycle (1)

**Figure B.49**: CI Prewash (1.2)
Figure B.50: CI Wash (1.3)

Figure B.51: Rinse (1.4)

Figure B.52: CI CheckRinseHoldButton (1.4.2)

Figure B.53: CI Spin (1.5)
Figure B.54: ShortSpin (1.5.1)

Figure B.55: CI ReducedSpin (1.5.1.1)

Figure B.56: CI DoorUnlock (1.7)
B.5.2 Single System 4 – Design without Variability

**Figure B.57:** Context diagram of single system 4

**Figure B.58:** WashCycle (Context)

**Figure B.59:** Rinse (1.4)
B.6 Single System 5 – Requirements Specification

1 $MA$ A wash-cycle on a washing machine shall consist of many phases.
1.1 Pre-wash phase
1.1.1 $SA$ Prior to a pre-wash phase the internal water container shall be heated.
1.1.1.2 The water shall be heated to $40^\circ C$.
1.1.2 The drum shall be filled with the heated water.
1.1.3 The drum shall spin at speed setting 1 for 40 time units.
1.1.4 Following the spin, the machine shall be drained.
1.2 Main-wash phase
1.2.1 $SA$ Prior to a wash phase the internal water container shall be heated.
1.2.1.4 The water shall be heated to $60^\circ C$.
1.2.1.4.1 $OP$ Check the ECO button’s state.
1.2.1.4.1.1 If the button is pressed, only heat the water to $50^\circ C$.
1.2.1.4.2 The drum shall be filled with the heated water.
1.2.1.4.3 The drum shall spin at speed setting 3 for 90 time units.
1.2.2 Following the spin, the machine shall be drained.
1.3 Rinse phase
1.3.1 The drum shall be filled with cold water.
1.3.2 The drum shall spin at speed setting 1 for 25 time units.
1.3.3 $OP$ Check the rinse-hold button’s state.
1.3.3.1 Wait for rinse-hold button to be released.
1.3.4 The machine shall be drained.
1.4 SA Spin phase
1.4.2 The spin phase shall be a long spin phase
1.4.2.1 The drum shall spin at speed setting 30 for 120 time units.
1.4.2.2 OP Check the reduced-spin button’s state
1.4.2.2.1 If pressed, the drum shall only spin for 65 time units.
1.4.2.3 Any excess water shall be drained.
1.5 SA Tumble-dry phase
1.5.2 The drying phase shall do a high-temperature dry.
1.5.2.1 OP The dry indicator light shall be lit during the drying phase.
1.5.2.1.1 The light shall be switched off after the drying phase.
1.5.2.2 The machine shall be drained before the drying phase.
1.5.2.3 The drum shall be heated to 100° C.
1.5.2.4 The drum shall cycle at speed setting 2.
1.5.2.5 The external timer shall control the duration of the drying phase.
1.5.2.6 Any excess water shall be drained.
2 The drum’s door shall be locked prior to any wash cycle
2.1 SA The door shall be unlocked
2.1.2 After a 2 minute delay

B.6.1 Single System 5 – Design with Variability

![Context diagram of single system 5](image)

**Figure B.61:** Context diagram of single system 5

![CI WashCycle](image)

**Figure B.62:** CI WashCycle (Context)
Figure B.63: WashCycle (1)

Figure B.64: CI Prewash (1.2)

Figure B.65: CI Wash (1.3)
Figure B.66: Wash 60° (1.3.4)

Figure B.67: CI CheckEcoButton (1.3.4.1)

Figure B.68: Rinse (1.4)
**Figure B.69:** CI CheckRinseHoldButton (1.4.2)

**Figure B.70:** CI Spin (1.5)

**Figure B.71:** LongSpin (1.5.2)
Figure B.72: CI ReducedSpin (1.5.1.1)

Figure B.73: CI Dry (1.6)

Figure B.74: HighTempDry (1.6.2)

Figure B.75: CI SetIndicatorLight (1.6.1.1)
B.6.2 Single System 5 – Design without Variability

Figure B.76: CI DoorUnlock (1.7)

Figure B.77: Context diagram of single system 5

Figure B.78: WashCycle (Context)
Figure B.79: Wash 60° (1.3.4)

Figure B.80: Rinse (1.4)
Figure B.81: LongSpin (1.5.2)

Figure B.82: HighTempDry (1.6.2)
Appendix C

Object-Oriented Case Study

C.1 Application Family Model of Requirements

1. The system shall model satellite positions with respect to earth for any given point in time.
2. The following satellite constellations shall be modelled.
   2.1. US Global Positioning Satellites (GPS) shall be modelled.
   2.2. Russian GLONASS satellites shall be modelled.
3. The global positioning device shall use these positions to estimate the best four satellites to use for positioning.
   3.1. The estimated satellite positions shall be passed to the global positioning device.
      3.1.1. Only the phase rotation of each satellite shall be passed.
      3.1.2. The satellite positions shall be modelled graphically on the device’s integrated LCD using vector co-ordinates.
         3.1.2.1. An orthogonal view shall be displayed. Showing satellites above a point on earth.
         3.1.2.1.1. 5°, 20° and 50° angles from the horizon shall be highlighted.
         3.1.2.1.2. The four optimum satellites shall be highlighted on the display.
         3.1.2.2. A perspective view shall be used. Showing the satellites orbiting earth from a point in space.
         3.1.2.2.1. Along with the satellites, other shapes may be displayed.
         3.1.2.2.1.1. The orbits of the satellites shall be drawn.
         3.1.2.2.1.2. The satellites shall have identifiers.
         3.1.2.2.1.3. Earth shall be drawn, rotated appropriately.
         3.1.2.2.1.3.1. Landmarks shall be displayed on earth.
A fixed number of landmarks shall be shown.

1 Landmark.
2 Landmarks.
3 Landmarks.
4 Landmarks.
5 Landmarks.
6 Landmarks.
7 Landmarks.

The displayed landmarks shall be

London 51.517 N, 0.105 W
New York 40.714 N, 74.006 W
Sydney 33.883 S, 151.217 E
Tokyo 35.700 N, 139.767 E
Rio de Janeiro 22.900 S, 43.233 W
Delhi 28.667 N, 77.217 E
Moscow 55.750 N, 37.583 E

For each landmark the optimum four satellites shall be noted.

C.1.1 Application Family Design

Figure C.1: Application family package dependencies
**Figure C.2**: Position Modeller (Package)

![Diagram of Position Modeller](image)

**Figure C.3**: Satellite Constellations (Package)

![Diagram of Satellite Constellations](image)
OutputType

PostSatellitePositions( The Satellites, Time)

Is a

Is a

OutputToGPSDevice

PostSatellitePositions( The Satellites, Time)

OutputToLCDScreen

ThePerspective : Perspective

PostSatellitePositions( The Satellites, Time)

Figure C.4: Output Types (Package)

Perspective

Draw( The Satellites, Time)

Is a

Is a

OrthogonalPerspective

TheDrawer : Orthogonal Drawer

Draw( The Satellites, Time)

OrthogonalDrawer

1

NormalPerspective

TheDrawer : NormalDrawer

Draw( The Satellites, Time)

NormalDrawer

1

Figure C.5: Perspective (Package)
Figure C.6: Normal Drawer (Package)
Figure C.7: Orthogonal Drawer (Package)

Figure C.8: Orbit Drawer (Package)

Figure C.9: Satellite ID Drawer (Package)
Figure C.10: Earth Drawer (Package)

Figure C.11: Landmark Drawer (Package)
Figure C.12: Landmark Array (Package)
C.2 Single System 1 – Requirements Specification

1. The system shall model satellite positions with respect to earth for any given point in time.

2. The following satellite constellations shall be modelled.

2.1. US Global Positioning Satellites (GPS) shall be modelled.

3. The global positioning device shall use these positions to estimate the best four satellites to use for positioning.

3.1. The estimated satellite positions shall be passed to the global positioning device.

3.1.1. Only the phase rotation of each satellite shall be passed.
C.2.1 Single System 1 – Design with Variability

![Diagram](image1)

**Figure C.15:** Test Harness (Package)

C.2.2 Single System 1 – Design without Variability

![Diagram](image2)

**Figure C.16:** Single system 1 package dependencies

![Diagram](image3)

**Figure C.17:** Test Harness (Package)
C.3 Single System 2 – Requirements Specification

1. The system shall model satellite positions with respect to earth for any given point in time.

2. The following satellite constellations shall be modelled.

2.1. US Global Positioning Satellites (GPS) shall be modelled.

2.2. Russian GLONASS satellites shall be modelled.
The global positioning device shall use these positions to estimate the best four satellites to use for positioning.

3.1 SA
The estimated satellite positions shall be passed to the global positioning device.

3.1.2 SA
The satellite positions shall be modelled graphically on the device’s integrated LCD using vector co-ordinates.

3.1.2.1
An orthogonal view shall be displayed. Showing satellites above a point on earth.

3.1.2.1.1
5°, 20° and 50° angles from the horizon shall be highlighted.

3.1.2.1.2
The four optimum satellites shall be highlighted on the display.

C.3.1 Single System 2 – Design with Variability

Figure C.21: Test Harness (Package)
C.3.2 Single System 2 – Design without Variability

**Figure C.22:** Single system 2 package dependencies

**Figure C.23:** Test Harness (Package)

**Figure C.24:** Position Modeller (Package)
OutputToLCDScreen
ThePerspective : OrthogonalPerspective
PostSatellitePositions( The Satellites, Time)

1

OrthogonalPerspective

Figure C.25: Output Types (Package)

OrthogonalPerspective
TheDrawer : OrthogonalWithGP OPT
Draw( The Satellites, Time)

1

OrthogonalWithGP OPT

Figure C.26: Perspective (Package)

OrthogonalWithGP OPT

Draw( The Satellites, Time)

Figure C.27: Orthogonal Drawer (Package)

GPS_GLONASS
TheSatellites[48] : Satellite
NumberOfSatellites : int

GPS_GLONASS()
GetSatellite() : Satellite[48]

Satellite

Same as in Application Family Design

Figure C.28: Satellite Constellations (Package)

C-xiv
C.4 Single System 3 – Requirements Specification

1. The system shall model satellite positions with respect to earth for any given point in time.

2. The following satellite constellations shall be modelled.
   2.2. Russian GLONASS satellites shall be modelled.

3. The global positioning device shall use these positions to estimate the best four satellites to use for positioning.
   3.1. The estimated satellite positions shall be passed to the global positioning device.
   3.1.2. The satellite positions shall be modelled graphically on the device’s integrated LCD using vector co-ordinates.
   3.1.2.2. A perspective view shall be used. Showing the satellites orbiting earth from a point in space.
   3.1.2.2.1. Along with the satellites, other shapes may be displayed.
   3.1.2.2.1.1. The orbits of the satellites shall be drawn.
   3.1.2.2.1.2. The satellites shall have identifiers.
Figure C.29: Test Harness (Package)
C.4.2 Single System 3 – Design without Variability

Figure C.30: Single system 3 package dependencies

Figure C.31: Test Harness (Package)
Position Modeller

The Output : OutputToLCDScreen
The Constellations : GLONASS
Current Time : Time

ModelSatellites()
SetCurrentTime()

GLONASS  OutputToLCDScreen

Figure C.32: Position Modeller (Package)

OutputToLCDScreen

The Perspective : NormalPerspective
Post Satellite Positions (The Satellites, Time)

NormalPerspective

Figure C.33: Output Types (Package)

NormalPerspective

The Drawer : NormalPerspectiveOrbitSatIDDrawer
Draw (The Satellites, Time)

NormalPerspectiveOrbitSatIDDrawer

Figure C.34: Perspective (Package)
Figure C.35: Normal Drawer (Package)

Figure C.36: Orbit Drawer (Package)

Figure C.37: Satellite ID Drawer (Package)

Figure C.38: Satellite Constellations (Package)
C.5 Single System 4 – Requirements Specification

1. The system shall model satellite positions with respect to earth for any given point in time.

2. The following satellite constellations shall be modelled.
   2.1 US Global Positioning Satellites (GPS) shall be modelled.
   2.2 Russian GLONASS satellites shall be modelled.

3. The global positioning device shall use these positions to estimate the best four satellites to use for positioning.

3.1. The estimated satellite positions shall be passed to the global positioning device.
   3.1.2 The satellite positions shall be modelled graphically on the device’s integrated LCD using vector co-ordinates.
       3.1.2.2 A perspective view shall be used. Showing the satellites orbiting earth from a point in space.
       3.1.2.1 Along with the satellites, other shapes may be displayed.

3.1.2.1.3 Earth shall be drawn, rotated appropriately.

3.1.2.1.3.1 Landmarks shall be displayed on earth.
   3.1.2.1.3.1.1 A fixed number of landmarks shall be shown.
   3.1.2.1.3.1.3 3 Landmarks.
   3.1.2.1.3.1.2 The displayed landmarks shall be:
       3.1.2.1.3.1.2.1 London 51.517 N, 0.105 W
       3.1.2.1.3.1.2.3 Sydney 33.883 S, 151.217 E
       3.1.2.1.3.1.2.5 Rio de Janeiro 22.900 S, 43.233 W
C.5.1 Single System 4 – Design with Variability

![Diagram of Test Harness (Package)](image)

**Figure C.39:** Test Harness (Package)
C.5.2 Single System 4 – Design without Variability

![Diagram]

**Figure C.40:** Single system 4 package dependencies

![Diagram]

**Figure C.41:** Test Harness (Package)
PositionModeller

TheOutput : OutputToLCDScreen
TheConstellations : GPS,GLONASS
CurrentTime : Time

ModelSatellites()
SetCurrentTime()

1

GPS,GLONASS
OutputToLCDScreen

Figure C.42: Position Modeller (Package)

OutputToLCDScreen

ThePerspective : NormalPerspective
PostSatellitePositions(TheSatellites,Time)

1

NormalPerspective

Figure C.43: Output Types (Package)

NormalPerspective

TheDrawer : NormalPerspectiveEarthDrawer
Draw(TheSatellites,Time)

1

NormalPerspectiveEarthDrawer

Figure C.44: Perspective (Package)
Figure C.45: Normal Drawer (Package)

Figure C.46: Earth Drawer (Package)

Figure C.47: Landmark Drawer (Package)
C.6 Single System 5 – Requirements Specification

1. The system shall model satellite positions with respect to earth for any given point in time.

2. The following satellite constellations shall be modelled.

2.1. US Global Positioning Satellites (GPS) shall be modelled.
The global positioning device shall use these positions to estimate the best four satellites to use for positioning.

The estimated satellite positions shall be passed to the global positioning device.

The satellite positions shall be modelled graphically on the device’s integrated LCD using vector co-ordinates.

A perspective view shall be used. Showing the satellites orbiting earth from a point in space.

Along with the satellites, other shapes may be displayed.

The orbits of the satellites shall be drawn.

The satellites shall have identifiers.

Earth shall be drawn, rotated appropriately.

Landmarks shall be displayed on earth.

A fixed number of landmarks shall be shown.

7 Landmarks.

The displayed landmarks shall be

London 51.517 N, 0.105 W
New York 40.714 N, 74.006 W
Sydney 33.883 S, 151.217 E
Tokyo 35.700 N, 139.767 E
Rio de Janeiro 22.900 S, 43.233 W
Delhi 28.667 N, 77.217 E
Moscow 55.750 N, 37.583 E

For each landmark the optimum four satellites shall be noted.
C.6.1 Single System 5 – Design with Variability

![Diagram](image.png)

Figure C.51: Test Harness (Package)

C-xxvii
C.6.2 Single System 5 – Design without Variability

Figure C.52: Single system 5 package dependencies

Figure C.53: Test Harness (Package)
Figure C.54: Position Modeller (Package)

Figure C.55: Output Types (Package)

Figure C.56: Perspective (Package)
Figure C.57: Normal Drawer (Package)

Figure C.58: Orbit Drawer (Package)

Figure C.59: Satellite ID Drawer (Package)

Figure C.60: Earth Drawer (Package)
LandMarkDrawerWithGPOPT
TheLandMarkArray : LandMarkArray?
TheGPOPTDrawer : NormalGPOPTDrawer
DrawFarLandmark( The Satellites, Time)
DrawNearLandmark( The Satellites, Time)

Figure C.61: Landmark Drawer (Package)

LandMarkArray?
Landmark1 : London
Landmark2 : NewYork
Landmark3 : Sydney
Landmark4 : Tokyo
Landmark5 : Rio
Landmark6 : Delhi
Landmark7 : Moscow
DrawFarLandmark( The Satellites, Time)
DrawNearLandmark( The Satellites, Time)

Figure C.62: Landmark Array (Package)

London
Longitude : double
Latitude : double
London()

NewYork
Longitude : double
Latitude : double
NewYork()

Moscow
Longitude : double
Latitude : double
Moscow()

Tokyo
Longitude : double
Latitude : double
Tokyo()

Sydney
Longitude : double
Latitude : double
Sydney()

Rio
Longitude : double
Latitude : double
Rio()

Delhi
Longitude : double
Latitude : double
Delhi()

Figure C.63: Landmark Types (Package)
Figure C.64: GPS Opt Drawer (Package)
Pipe and Filter Case Study

Appendix D

D.1 Application Family Model of Requirements

1 The system shall regulate two independent liquid tanks.
2 A closed feedback on/off control loop shall be used to regulate both tanks.
3 Each tank shall be controlled individually.
3.1 Measured liquid levels shall be logged.
4 Tank 1
4.1 SA Liquid tank 1 may be controlled through a serial port or a parallel port.
4.1.1 The serial port shall be used to communicate with tank 1
4.1.2 The parallel port shall be used to communicate with tank 1
4.2 OP The readings for tank 1 shall be converted from weight to volume measurements.
4.3 The initial set point shall be 10 litres.
4.4 OP A new set point shall be acquired remotely.
4.5 If the measured liquid level exceeds the set point, the inlet valve shall be switched off.
4.6 If the measured liquid level is equal to or below the set point, the inlet valve shall be opened.
4.7 OP 10 measurements taken from tank 1 shall be averaged before logging.
5 Tank 2
5.1 SA Liquid tank 2 may be controlled through a serial port or a parallel port.
5.1.1 The serial port shall be used to communicate with tank 2
5.1.2 The parallel port shall be used to communicate with tank 2
5.2 OP The readings for tank 2 shall be converted from weight to volume measurements.
5.3 The initial set point shall be 10 litres.
5.4 \textit{OP} A new set point shall be acquired remotely.
5.5 If the measured liquid level exceeds the set point, the inlet valve shall be switched off.
5.6 If the measured liquid level is equal to or below the set point, the inlet valve shall be opened.
5.7 \textit{OP} 10 measurements taken from tank 2 shall be averaged before logging.
6 \textit{MA} Measured liquid levels shall be logged.
6.1 They shall be stored to hard disk.
6.1.1 \textit{OP} Values shall also be stored on a secondary backup device.
6.2 They shall be spooled to a printer.
D.1.1 Application Family Design

Figure D.1: Application family design
D.2 Single System 1 – Requirements Specification

1. The system shall regulate two independent liquid tanks.
2. A closed feedback on/off control loop shall be used to regulate both tanks.
3. Each tank shall be controlled individually.
4. Measured liquid levels shall be logged.

4.1 Tank 1

4.1.1 The tank 1 may be controlled through a serial port or a parallel port.
4.1.1 The serial port shall be used to communicate with tank 1
4.3 The initial set point shall be 10 litres.
4.5 If the measured liquid level exceeds the set point, the inlet valve shall be switched off.
4.6 If the measured liquid level is equal to or below the set point, the inlet valve shall be opened.

5. Tank 2

5.1 The tank 2 may be controlled through a serial port or a parallel port.
5.1.1 The serial port shall be used to communicate with tank 2
5.3 The initial set point shall be 10 litres.
5.5 If the measured liquid level exceeds the set point, the inlet valve shall be switched off.
5.6 If the measured liquid level is equal to or below the set point, the inlet valve shall be opened.

6. Measured liquid levels shall be logged.
6.1 They shall be stored to hard disk.
D.2.1 Single System 1 – Design with Variability

Figure D.2: Single system 1 with points of variability
D.2.2 Single System 1 – Design without Variability

Figure D.3: Single system 1 without points of variability
D.3 Single System 2 – Requirements Specification

1 The system shall regulate two independent liquid tanks.
2 A closed feedback on/off control loop shall be used to regulate both tanks.
3 Each tank shall be controlled individually.
3.1 Measured liquid levels shall be logged.
4 Tank 1
4.1 "SA" Liquid tank 1 may be controlled through a serial port or a parallel port.
4.1.1 The serial port shall be used to communicate with tank 1
4.3 The initial set point shall be 10 litres.
4.5 If the measured liquid level exceeds the set point, the inlet valve shall be switched off.
4.6 If the measured liquid level is equal to or below the set point, the inlet valve shall be opened.
4.7 "OP" 10 measurements taken from tank 1 shall be averaged before logging.
5 Tank 2
5.1 "SA" Liquid tank 2 may be controlled through a serial port or a parallel port.
5.1.2 The parallel port shall be used to communicate with tank 2
5.3 The initial set point shall be 10 litres.
5.5 If the measured liquid level exceeds the set point, the inlet valve shall be switched off.
5.6 If the measured liquid level is equal to or below the set point, the inlet valve shall be opened.
6 "MA" Measured liquid levels shall be logged.
6.1 They shall be stored to hard disk.
6.1.1 "OP" Values shall also be stored on a secondary backup device.
D.3.1 Single System 2 – Design with Variability

Figure D.4: Single system 2 with points of variability
D.3.2 Single System 2 – Design without Variability

Figure D.5: Single system 2 without points of variability
D.4 Single System 3 – Requirements Specification

1. The system shall regulate two independent liquid tanks.
2. A closed feedback on/off control loop shall be used to regulate both tanks.
3. Each tank shall be controlled individually.
4. Measured liquid levels shall be logged.

4.1 Tank 1
4.1.1 Liquid tank 1 may be controlled through a serial port or a parallel port.
4.1.2 The parallel port shall be used to communicate with tank 1
4.2 The readings for tank 1 shall be converted from weight to volume measurements.
4.3 The initial set point shall be 10 litres.
4.4 A new set point shall be acquired remotely.
4.5 If the measured liquid level exceeds the set point, the inlet valve shall be switched off.
4.6 If the measured liquid level is equal to or below the set point, the inlet valve shall be opened.

5. Tank 2
5.1 Liquid tank 2 may be controlled through a serial port or a parallel port.
5.1.1 The serial port shall be used to communicate with tank 2
5.3 The initial set point shall be 10 litres.
5.4 A new set point shall be acquired remotely.
5.5 If the measured liquid level exceeds the set point, the inlet valve shall be switched off.
5.6 If the measured liquid level is equal to or below the set point, the inlet valve shall be opened.
5.7 10 measurements taken from tank 2 shall be averaged before logging.
6. Measured liquid levels shall be logged.
6.1 They shall be stored to hard disk.
6.1.1 Values shall also be stored on a secondary backup device.
6.2 They shall be spooled to a printer.
Figure D.6: Single system 3 with points of variability
D.4.2 Single System 3 – Design without Variability

Figure D.7: Single system 3 without points of variability
D.5 Single System 4 – Requirements Specification

1 The system shall regulate two independent liquid tanks.
2 A closed feedback on/off control loop shall be used to regulate both tanks.
3 Each tank shall be controlled individually.
3.1 Measured liquid levels shall be logged.
4 Tank 1
4.1 SA Liquid tank 1 may be controlled through a serial port or a parallel port.
4.1.1 The serial port shall be used to communicate with tank 1
4.2 OP The readings for tank 1 shall be converted from weight to volume measurements.
4.3 The initial set point shall be 10 litres.
4.5 If the measured liquid level exceeds the set point, the inlet valve shall be switched off.
4.6 If the measured liquid level is equal to or below the set point, the inlet valve shall be opened.
4.7 OP 10 measurements taken from tank 1 shall be averaged before logging.
5 Tank 2
5.1 SA Liquid tank 2 may be controlled through a serial port or a parallel port.
5.1.2 The parallel port shall be used to communicate with tank 2
5.2 OP The readings for tank 2 shall be converted from weight to volume measurements.
5.3 The initial set point shall be 10 litres.
5.4 OP A new set point shall be acquired remotely.
5.5 If the measured liquid level exceeds the set point, the inlet valve shall be switched off.
5.6 If the measured liquid level is equal to or below the set point, the inlet valve shall be opened.
5.7 OP 10 measurements taken from tank 2 shall be averaged before logging.
6 MA Measured liquid levels shall be logged.
6.2 They shall be spooled to a printer.
D.5.1 Single System 4 – Design with Variability

**Figure D.8:** Single system 4 with points of variability
D.5.2 Single System 4 – Design without Variability

Figure D.9: Single system 4 without points of variability
D.6 Single System 5 – Requirements Specification

1 The system shall regulate two independent liquid tanks.
2 A closed feedback on/off control loop shall be used to regulate both tanks.
3 Each tank shall be controlled individually.
3.1 Measured liquid levels shall be logged.
4 Tank 1
4.1 SA Liquid tank 1 may be controlled through a serial port or a parallel port.
4.1.1 The serial port shall be used to communicate with tank 1
4.2 OP The readings for tank 1 shall be converted from weight to volume measurements.
4.3 The initial set point shall be 10 litres.
4.4 OP A new set point shall be acquired remotely.
4.5 If the measured liquid level exceeds the set point, the inlet valve shall be switched off.
4.6 If the measured liquid level is equal to or below the set point, the inlet valve shall be opened.
4.7 OP 10 measurements taken from tank 1 shall be averaged before logging.
5 Tank 2
5.1 SA Liquid tank 2 may be controlled through a serial port or a parallel port.
5.1.1 The serial port shall be used to communicate with tank 2
5.2 OP The readings for tank 2 shall be converted from weight to volume measurements.
5.3 The initial set point shall be 10 litres.
5.4 OP A new set point shall be acquired remotely.
5.5 If the measured liquid level exceeds the set point, the inlet valve shall be switched off.
5.6 If the measured liquid level is equal to or below the set point, the inlet valve shall be opened.
5.7 OP 10 measurements taken from tank 2 shall be averaged before logging.
6 MA Measured liquid levels shall be logged.
6.1 They shall be stored to hard disk.
6.1.1 OP Values shall also be stored on a secondary backup device.
6.2 They shall be spooled to a printer.
D.6.1 Single System 5 – Design with Variability

Figure D.10: Single system 5 with points of variability
D.6.2 Single System 5 – Design without Variability

Figure D.11: Single system 5 without points of variability
Appendix E

Event-Based Case Study

E.1 Application Family Model of Requirements

1  The system shall control onboard instruments.
2  All onboard instrument can enter up to 5 states.
2.1 A standby state.
2.2 An initialising state.
2.3 A stabilising state.
2.4 A measuring state.
2.5 A shutdown/refuse state.
3  An onboard device shall switch state for two event types.
3.1 It shall switch state due to a remote macro-command.
3.2 It shall switch state due to its own internal instrument setting or request.
4  In standby the following events shall be handled.
4.1 The following processes shall be completed.
4.1.1 The device shall switch to its initialising state.
4.1.2 The device’s MVR shall be switched off.
4.1.3 The device’s USO shall be switched on.
4.2 The following processes shall be completed.
4.2.1 The device shall switch to its stabilising state.
4.2.2 The device’s MVR shall be switched off.
4.2.3 The device’s USO shall be switched on.
4.2.4 Internal USO triggers a STANDBY_TO_STABILISING event.
4.2.5 The following processes shall be completed.
4.2.2.1.1 The device shall switch to its stabilising state.
4.2.2.1.2 The device’s MVR shall be switched off.
4.2.2.1.3 The device’s USO shall be switched on.
4.3 OP Macro command MCMD_STANDBY_TO_MEASURING
4.2.1 MA The following processes shall be completed.
4.2.1.1 The device shall switch to its measuring state.
4.2.1.2 The device’s MVR shall be switched on.
4.2.1.3 The device’s USO shall be switched on.
5 In initialising the following events shall be handled.
5.1 OP Macro command MCMD_INITIALISING_TO_STANDBY.
5.1.1 MA The following processes shall be completed.
5.1.1.1 The device shall switch to its standby state.
5.1.1.2 The device’s MVR shall be switched off.
5.1.1.3 The device’s USO shall be switched off.
5.2 OP Macro command MCMD_INITIALISING_TO_STABILISING.
5.2.1 MA The following processes shall be completed.
5.2.1.1 The device shall switch to its stabilising state.
5.2.1.2 The device’s MVR shall be switched off.
5.2.1.3 The device’s USO shall be switched on.
5.3 OP Macro command MCMD_INITIALISING_TO_MEASURING.
5.3.1 MA The following processes shall be completed.
5.3.1.1 The device shall switch to its measuring state.
5.3.1.2 The device’s MVR shall be switched on.
5.3.1.3 The device’s USO shall be switched on.
6 In stabilising state the following events shall be handled.
6.1 OP Macro command MCMD_STABILISING_TO_STANDBY.
6.1.1 MA The following processes shall be completed.
6.1.1.1 The device shall switch to its standby state.
6.1.1.2 The device’s MVR shall be switched off.
6.1.1.3 The device’s USO shall be switched off.
6.2 OP Macro command MCMD_STABILISING_TO_INITIALISING.
6.2.1 MA The following processes shall be completed.
6.2.1.1 The device shall switch to its initialising state.
6.2.1.2 The device’s MVR shall be switched off.
6.2.1.3 The device’s USO shall be switched on.
6.3 OP Macro command MCMD_STABILISING_TO_MEASURING.
6.3.1 MA The following processes shall be completed.
6.3.1.1 The device shall switch to its measuring state.
6.3.1.2 The device’s MVR shall be switched on.
6.3.1.3 The device’s USO shall be switched on.
6.3.2 OP Internal MVR triggers a STABILISING_TO_MEASURING event.
6.3.2.1 The following processes shall be completed.
6.3.2.1.1 The device shall switch to its measuring state.
6.3.2.1.2 The device’s MVR shall be switched on.
6.3.2.1.3 The device’s USO shall be switched on.
7 In measuring state the following events shall be handled.
7.1 OP Macro command MCM_MASURING_TO_STANDBY.
7.1.1 MA The following processes shall be completed.
7.1.1.1 The device shall switch to its standby state.
7.1.1.2 The device’s MVR shall be switched off.
7.1.1.3 The device’s USO shall be switched off.
7.2 OP Macro command MCM_MEASURING_TO_INITIALISING.
7.2.1 MA The following processes shall be completed.
7.2.1.1 The device shall switch to its initialising state.
7.2.1.2 The device’s MVR shall be switched off.
7.2.1.3 The device’s USO shall be switched on.
7.3 OP Macro command MCM_MEASURING_TO_STABILISING.
7.3.1 MA The following processes shall be completed.
7.3.1.1 The device shall switch to its stabilising state.
7.3.1.2 The device’s MVR shall be switched off.
7.3.1.3 The device’s USO shall be switched on.
8 OP At any point both the macro command handler and the internal
     instruments may raise a SHUTDOWN event.
8.1 SA On SHUTDOWN event the following processes shall be completed.
8.1.1 The device shall be switched to the standby/refuse state.
8.1.2 The device shall be switched to the standby/refuse state and its MVR
     and USO shall be switched off.
9 In standby/refuse state the following events shall be handled.
9.1 OP Macro command RESET.
9.1.1 SA On RESET event the following processes shall be completed.
9.1.1.1 The device shall be switched to the standby state.
9.1.1.2 The device shall be switched to the standby state and its MVR and USO
     shall be switched off.
E.1.1 Application Family Design

Figure E.1: Events from Standby
Figure E.2: Events from Initialising

Figure E.3: Events from Stabilising
Figure E.4: Events from Measuring

Figure E.5: Events from Shutdown/Refuse

Figure E.6: Nested events from CI Standby
**Figure E.7**: Nested events from CI Initialising

**Figure E.8**: Nested events from CI Stabilising

**Figure E.9**: Nested events from CI Measuring
E.2 Single System 1 – Requirements Specification

1. The system shall control onboard instruments.
2. All onboard instrument can enter up to 5 states.
   2.1 A standby state.
   2.2 An initialising state.
   2.3 A stabilising state.
   2.4 A measuring state.
   2.5 A shutdown/refuse state.
3. An onboard device shall switch state for two event types.
   3.1 It shall switch state due to a remote macro-command.
   3.2 It shall switch state due to its own internal instrument setting or request.
4. In standby the following events shall be handled.
   4.3 $op$ Macro command MCMD_STANDBY_TO_MEASURING
   4.2.1 $MA$ The following processes shall be completed.
   4.2.1.1 The device shall switch to its measuring state.
5. In initialising the following events shall be handled.
   (none)
6. In stabilising state the following events shall be handled.
   (none)
7. In measuring state the following events shall be handled.
   (none)
9. In standby/refuse state the following events shall be handled.
   (none)
E.2.1 Single System 1 – Design with Variability

![Event diagram 1](image1)

**Figure E.11:** Events from Standby

![Event diagram 2](image2)

**Figure E.12:** Nested events from CI Measuring

E.2.2 Single System 1 – Design without Variability

![Event diagram 3](image3)

**Figure E.13:** Events from Standby

E.3 Single System 2 – Requirements Specification

1. The system shall control onboard instruments.
2. All onboard instrument can enter up to 5 states.
2.1 A standby state.
2.2 An initialising state.
2.3 A stabilising state.
2.4 A measuring state.
2.5 A shutdown/refuse state.
3. An onboard device shall switch state for two event types.
3.1 It shall switch state due to a remote macro-command.
3.2 It shall switch state due to its own internal instrument setting or request.
4 In standby the following events shall be handled.
4.1 OP Macro command MCM_D_STANDBY_TO_INITIALISING.
4.1.1 MA The following processes shall be completed.
4.1.1.1 The device shall switch to its initialising state.
5 In initialising the following events shall be handled.
5.1 OP Macro command MCM_INITIALISING_TO_STANDBY.
5.1.1 MA The following processes shall be completed.
5.1.1.1 The device shall switch to its standby state.
5.3 OP Macro command MCM_INITIALISING_TO_MEASURING.
5.3.1 MA The following processes shall be completed.
5.3.1.1 The device shall switch to its measuring state.
6 In stabilising state the following events shall be handled.
   (none)
7 In measuring state the following events shall be handled.
   (none)
8 OP At any point both the macro command handler and the internal
   instruments may raise a SHUTDOWN event.
8.1 SA On SHUTDOWN event the following processes shall be completed.
8.1.1 The device shall be switched to the standby/refuse state.
9 In standby/refuse state the following events shall be handled.
9.1 OP Macro command RESET.
9.1.1 SA On RESET event the following processes shall be completed.
9.1.1.1 The device shall be switched to the standby state.

E.3.1 Single System 2 – Design with Variability

![Diagram](https://example.com/diagram.png)

**Figure E.14:** Events from Standby
Figure E.15: Events from Initialising

Figure E.16: Events from Shutdown/Refuse

Figure E.17: Nested events from CI Standby

Figure E.18: Nested events from CI Initialising
**E.3.2 Single System 2 – Design without Variability**

![Diagram of CI Measuring](image1)

**Figure E.19**: Nested events from CI Measuring

![Diagram of CI Shutdown](image2)

**Figure E.20**: Nested events from CI Shutdown

![Diagram of Standby](image3)

**Figure E.21**: Events from Standby
E.4 Single System 3 – Requirements Specification

1. The system shall control onboard instruments.
2. All onboard instrument can enter up to 5 states.
   2.1 A standby state.
   2.2 An initialising state.
   2.3 A stabilising state.
   2.4 A measuring state.
   2.5 A shutdown/refuse state.
3. An onboard device shall switch state for two event types.
   3.1 It shall switch state due to a remote macro-command.
   3.2 It shall switch state due to its own internal instrument setting or request.
4. In standby the following events shall be handled.
   4.2 \textit{OP} Macro command MCMD\_STANDBY\_TO\_STABILISING
   4.2.1 \textit{MA} The following processes shall be completed.
   4.2.1.1 The device shall switch to its stabilising state.
   4.2.1.2 The device’s MVR shall be switched off.
   4.2.1.3 The device’s USO shall be switched on.
4.2.2 \textit{op} \quad \text{Internal USO triggers a STANDBY\_TO\_STABILISING event.}

4.2.2.1 \quad \text{The following processes shall be completed.}

4.2.2.1.1 \quad \text{The device shall switch to its stabilising state.}

4.2.2.1.2 \quad \text{The device’s MVR shall be switched off.}

4.2.2.1.3 \quad \text{The device’s USO shall be switched on.}

5 \quad \text{In initialising the following events shall be handled.}

\hspace{1em} (\text{none})

6 \quad \text{In stabilising state the following events shall be handled.}

6.1 \textit{op} \quad \text{Macro command MCMD\_STABILISING\_TO\_STANDBY.}

6.1.1 \textit{ma} \quad \text{The following processes shall be completed.}

6.1.1.1 \quad \text{The device shall switch to its standby state.}

6.1.1.2 \quad \text{The device’s MVR shall be switched off.}

6.1.1.3 \quad \text{The device’s USO shall be switched off.}

6.3 \textit{op} \quad \text{Macro command MCMD\_STABILISING\_TO\_MEASURING.}

6.3.1 \textit{ma} \quad \text{The following processes shall be completed.}

6.3.1.1 \quad \text{The device shall switch to its measuring state.}

6.3.1.2 \quad \text{The device’s MVR shall be switched on.}

6.3.1.3 \quad \text{The device’s USO shall be switched on.}

6.3.2 \textit{op} \quad \text{Internal MVR triggers a STABILISING\_TO\_MEASURING event.}

6.3.2.1 \quad \text{The following processes shall be completed.}

6.3.2.1.1 \quad \text{The device shall switch to its measuring state.}

6.3.2.1.2 \quad \text{The device’s MVR shall be switched on.}

6.3.2.1.3 \quad \text{The device’s USO shall be switched on.}

7 \quad \text{In measuring state the following events shall be handled.}

7.1 \textit{op} \quad \text{Macro command MCMD\_MEASURING\_TO\_STANDBY.}

7.1.1 \textit{ma} \quad \text{The following processes shall be completed.}

7.1.1.1 \quad \text{The device shall switch to its standby state.}

7.1.1.2 \quad \text{The device’s MVR shall be switched off.}

7.1.1.3 \quad \text{The device’s USO shall be switched off.}

7.3 \textit{op} \quad \text{Macro command MCMD\_MEASURING\_TO\_STABILISING.}

7.3.1 \textit{ma} \quad \text{The following processes shall be completed.}

7.3.1.1 \quad \text{The device shall switch to its stabilising state.}

7.3.1.2 \quad \text{The device’s MVR shall be switched off.}

7.3.1.3 \quad \text{The device’s USO shall be switched on.}

9 \quad \text{In standby/refuse state the following events shall be handled.}

\hspace{1em} (\text{none})
E.4.1 Single System 3 – Design with Variability

**Figure E.24**: Events from Standby

**Figure E.25**: Events from Stabilising

**Figure E.26**: Events from Measuring
Figure E.27: Nested events from CI Standby

Figure E.28: Nested events from CI Stabilising

Figure E.29: Nested events from CI Measuring
E.4.2 Single System 3 – Design without Variability

**Figure E.30:** Events from Standby

**Figure E.31:** Events from Stabilising

**Figure E.32:** Events from Measuring
E.5 Single System 4 – Requirements Specification

1  The system shall control onboard instruments.
2  All onboard instrument can enter up to 5 states.
2.1  A standby state.
2.2  An initialising state.
2.3  A stabilising state.
2.4  A measuring state.
2.5  A shutdown/refuse state.
3  An onboard device shall switch state for two event types.
3.1  It shall switch state due to a remote macro-command.
3.2  It shall switch state due to its own internal instrument setting or request.
4  In standby the following events shall be handled.
4.1  Macro command MCMD_STANDBY_TO_INITIALISING.
4.1.1  The following processes shall be completed.
4.1.1.1  The device shall switch to its initialising state.
5  In initialising the following events shall be handled.
5.1  Macro command MCMD_INITIALISING_TO_STANDBY.
5.1.1  The following processes shall be completed.
5.1.1.1  The device shall switch to its standby state.
5.2  Macro command MCMD_INITIALISING_TO_STABILISING.
5.2.1  The following processes shall be completed.
5.2.1.1  The device shall switch to its stabilising state.
6  In stabilising state the following events shall be handled.
6.1  Macro command MCMD_STABILISING_TO_STANDBY.
6.1.1  The following processes shall be completed.
6.1.1.1  The device shall switch to its standby state.
6.2  Macro command MCMD_STABILISING_TO_INITIALISING.
6.2.1  The following processes shall be completed.
6.2.1.1  The device shall switch to its initialising state.
6.3  Macro command MCMD_STABILISING_TO_MEASURING.
6.3.1  The following processes shall be completed.
6.3.1.1  The device shall switch to its measuring state.
7  In measuring state the following events shall be handled.
7.1  Macro command MCMD_MEASURING_TO_STANDBY.
7.1.1  The following processes shall be completed.
7.1.1.1  The device shall switch to its standby state.
7.3  Macro command MCMD_MEASURING_TO_STABILISING.
7.3.1  The following processes shall be completed.
7.3.1.1  The device shall switch to its stabilising state.
8  At any point both the macro command handler and the internal instruments may raise a SHUTDOWN event.
8.1 SA On SHUTDOWN event the following processes shall be completed.

8.1.1 The device shall be switched to the standby/refuse state.

9 In standby/refuse state the following events shall be handled.

9.1 OP Macro command RESET.

9.1.1 SA On RESET event the following processes shall be completed.

9.1.1.1 The device shall be switched to the standby state.

E.5.1 Single System 4 – Design with Variability

![Diagram](image)

**Figure E.33:** Events from Standby

![Diagram](image)

**Figure E.34:** Events from Initialising
Figure E.35: Events from Stabilising

Figure E.36: Events from Measuring

Figure E.37: Events from Shutdown/Refuse
Figure E.38: Nested events from CI Standby

Figure E.39: Nested events from CI Initialising

Figure E.40: Nested events from CI Stabilising

Figure E.41: Nested events from CI Measuring
Figure E.42: Nested events from CI Shutdown

E.5.2 Single System 4 – Design without Variability

Figure E.43: Events from Standby

Figure E.44: Events from Initialising
Figure E.45: Events from Stabilising

Figure E.46: Events from Measuring

Figure E.47: Events from Shutdown/Refuse
1 The system shall control onboard instruments.
2 All onboard instrument can enter up to 5 states.
2.1 A standby state.
2.2 An initialising state.
2.3 A stabilising state.
2.4 A measuring state.
2.5 A shutdown/refuse state.
3 An onboard device shall switch state for two event types.
3.1 It shall switch state due to a remote macro-command.
3.2 It shall switch state due to its own internal instrument setting or request.
4 In standby the following events shall be handled.
4.1 \textit{OP} Macro command MCMD\_STANDBY\_TO\_INITIALISING.
4.1.1 \textit{MA} The following processes shall be completed.
4.1.1.1 The device shall switch to its initialising state.
4.1.1.2 The device’s MVR shall be switched off.
4.1.1.3 The device’s USO shall be switched on.
4.2 \textit{OP} Macro command MCMD\_STANDBY\_TO\_STABILISING
4.2.1 \textit{MA} The following processes shall be completed.
4.2.1.1 The device shall switch to its stabilising state.
4.2.1.2 The device’s MVR shall be switched off.
4.2.1.3 The device’s USO shall be switched on.
4.2.2 \textit{OP} Internal USO triggers a STANDBY\_TO\_STABILISING event.
4.2.2.1 The following processes shall be completed.
4.2.2.1.1 The device shall switch to its stabilising state.
4.2.2.1.2 The device’s MVR shall be switched off.
4.2.2.1.3 The device’s USO shall be switched on.
4.3 \textit{OP} Macro command MCMD\_STANDBY\_TO\_MEASURING
4.2.1 \textit{MA} The following processes shall be completed.
4.2.1.1 The device shall switch to its measuring state.
4.2.1.2 The device’s MVR shall be switched on.
4.2.1.3 The device’s USO shall be switched on.
5 In initialising the following events shall be handled.
5.1 \textit{OP} Macro command MCMD\_INITIALISING\_TO\_STANDBY.
5.1.1 \textit{MA} The following processes shall be completed.
5.1.1.1 The device shall switch to its standby state.
5.1.1.2 The device’s MVR shall be switched off.
5.1.1.3 The device’s USO shall be switched off.
5.2 \textit{OP} Macro command MCMD\_INITIALISING\_TO\_STABILISING.
5.2.1 \textit{MA} The following processes shall be completed.
5.2.1.1 The device shall switch to its stabilising state.
5.2.1.2 The device’s MVR shall be switched off.
5.2.1.3 The device’s USO shall be switched on.
5.3 \textit{OP} Macro command MCMD\_INITIALISING\_TO\_MEASURING.
5.3.1 \textit{MA} The following processes shall be completed.
5.3.1.1 The device shall switch to its measuring state.
5.3.1.2 The device’s MVR shall be switched on.
5.3.1.3 The device’s USO shall be switched on.
6 In stabilising state the following events shall be handled.
6.1 \textit{OP} Macro command MCMD\_STABILISING\_TO\_STANDBY.
6.1.1 \textit{MA} The following processes shall be completed.
6.1.1.1 The device shall switch to its standby state.
6.1.1.2 The device’s MVR shall be switched off.
6.1.1.3 The device’s USO shall be switched off.
6.2 \textit{OP} Macro command MCMD\_STABILISING\_TO\_INITIALISING.
6.2.1 \textit{MA} The following processes shall be completed.
6.2.1.1 The device shall switch to its initialising state.
6.2.1.2 The device’s MVR shall be switched off.
6.2.1.3 The device’s USO shall be switched on.
6.3 \textit{OP} Macro command MCMD\_STABILISING\_TO\_MEASURING.
6.3.1 \textit{MA} The following processes shall be completed.
6.3.1.1 The device shall switch to its measuring state.
6.3.1.2 The device’s MVR shall be switched on.
6.3.1.3 The device’s USO shall be switched on.
6.3.2 \textit{OP} Internal MVR triggers a STABILISING\_TO\_MEASURING event.
6.3.2.1 The following processes shall be completed.
6.3.2.1.1 The device shall switch to its measuring state.
6.3.2.1.2 The device’s MVR shall be switched on.
6.3.2.1.3 The device’s USO shall be switched on.
7 In measuring state the following events shall be handled.
7.1 \textit{OP} Macro command MCMD\_MEASURING\_TO\_STANDBY.
7.1.1 \textit{MA} The following processes shall be completed.
7.1.1.1 The device shall switch to its standby state.
7.1.1.2 The device’s MVR shall be switched off.
7.1.1.3 The device’s USO shall be switched off.
7.2 \textit{OP} Macro command MCMD\_MEASURING\_TO\_INITIALISING.
7.2.1 \textit{MA} The following processes shall be completed.
7.2.1.1 The device shall switch to its initialising state.
7.2.1.2 The device’s MVR shall be switched off.
7.2.1.3 The device’s USO shall be switched on.
7.3 \textit{OP} Macro command MCMD\_MEASURING\_TO\_STABILISING.
7.3.1 \textit{MA} The following processes shall be completed.
7.3.1.1 The device shall switch to its stabilising state.
7.3.1.2 The device’s MVR shall be switched off.
7.3.1.3 The device’s USO shall be switched on.
8 OP At any point both the macro command handler and the internal instruments may raise a SHUTDOWN event.
8.1 SA On SHUTDOWN event the following processes shall be completed.
8.1.2 The device shall be switched to the standby/refuse state and its MVR and USO shall be switched off.
9 OP In standby/refuse state the following events shall be handled.
9.1 SA On SHUTDOWN event the following processes shall be completed.
9.1.1.2 The device shall be switched to the standby state and its MVR and USE shall be switched off.

E.6.1 Single System 5 – Design with Variability

Figure E.48: Events from Standby
Figure E.49: Events from Initialising

Figure E.50: Events from Stabilising
**Figure E.51**: Events from Measuring

**Figure E.52**: Events from Shutdown/Refuse

**Figure E.53**: Nested events from CI Standby
Figure E.54: Nested events from CI Initialising

Figure E.55: Nested events from CI Stabilising

Figure E.56: Nested events from CI Measuring

Figure E.57: Nested events from CI Shutdown
E.6.2 Single System 5 – Design without Variability

**Figure E.58:** Events from Standby

**Figure E.59:** Events from Initialising
Figure E.60: Events from Stabilising

Figure E.61: Events from Measuring
**Figure E.62:** Events from Shutdown/Refuse
### Repository Case Study

#### F.1 Application Family Model of Requirements

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The system shall pre-process weather satellite images before they are transmitted to the ground station for analysis.</td>
</tr>
<tr>
<td>1.1</td>
<td>The system shall acquire a digital satellite photograph from the satellite’s onboard camera.</td>
</tr>
<tr>
<td>1.1</td>
<td>Each image shall have a resolution of 373 x 523 with 256 colours.</td>
</tr>
<tr>
<td>2</td>
<td>A colour spectrum analysis shall be performed on the image.</td>
</tr>
<tr>
<td>2.1</td>
<td>The analysis shall identify cloud regions by their colour intensity:</td>
</tr>
<tr>
<td>2.1.1</td>
<td>The initial image shall be smoothed to intensify large areas of clouds.</td>
</tr>
<tr>
<td>2.1.2</td>
<td>The smoothed image shall have the number of colours reduced to 10 red, 1 green and 1 blue colour steps.</td>
</tr>
<tr>
<td>2.1.3</td>
<td>The final image shall be negated.</td>
</tr>
<tr>
<td>2.1.4</td>
<td>The resulting highlighted cloud regions shall be outlined.</td>
</tr>
<tr>
<td>2.2</td>
<td>The analysis shall identify the most intense cloud regions on the image.</td>
</tr>
<tr>
<td>2.2.1</td>
<td>The initial image shall be smoothed to intensify large areas of clouds.</td>
</tr>
<tr>
<td>2.2.2</td>
<td>The smoothed image shall have the number of colours reduced to 10 red, 1 green and 1 blue colour steps.</td>
</tr>
<tr>
<td>2.2.3</td>
<td>The image shall be negated.</td>
</tr>
<tr>
<td>2.2.4</td>
<td>All except the darkest area shall be deleted.</td>
</tr>
<tr>
<td>3</td>
<td>The original image shall be merged into the analysed image.</td>
</tr>
<tr>
<td>4</td>
<td>Further cartographic information shall be added.</td>
</tr>
<tr>
<td>4.1</td>
<td>Map outlines of the specific landmasses shall be added.</td>
</tr>
<tr>
<td>4.2</td>
<td>Longitudinal and latitudinal lines shall be added to the image.</td>
</tr>
<tr>
<td>4.3</td>
<td>Major cities and landmarks shall be added to the image.</td>
</tr>
</tbody>
</table>
5 \textit{op} \quad \text{A border shall be added to the processed image.}
5.1 \quad \text{The border shall be 10 pixels wide for the left, right and bottom border.}
5.2 \quad \text{The top border shall have a width of 36 pixels.}
5.3 \textit{op} \quad \text{The image shall be time-stamped and labelled within its top border.}
6 \textit{sa} \quad \text{After processing the final image shall be released for further processing.}
6.1 \quad \text{The image shall be stored to an internal drive for further analysis and processing by other sub-systems.}
6.2 \quad \text{The image shall be transmitted to the ground station.}
F.1.1 Application Family Design

Figure F.1: Application family design
F.2 Single System 1 – Requirements Specification

1 The system shall pre-process weather satellite images before they are transmitted to the ground station for analysis.
1.1 The system shall acquire a digital satellite photograph from the satellite’s onboard camera.
1.1 Each image shall have a resolution of 373 x 523 with 256 colours.
2 A colour spectrum analysis shall be performed on the image.
2.1 The analysis shall identify cloud regions by their colour intensity:
2.1.1 The initial image shall be smoothed to intensify large areas of clouds.
2.1.2 The smoothed image shall have the number of colours reduced to 10 red, 1 green and 1 blue colour steps.
2.1.3 The final image shall be negated.
4 Further cartographic information shall be added.
4.1 Map outlines of the specific landmasses shall be added.
4.2 Longitudinal and latitudinal lines shall be added to the image.
4.3 Major cities and landmarks shall be added to the image.
6 After processing the final image shall be released for further processing.
6.1 The image shall be stored to an internal drive for further analysis and processing by other sub-systems.
Figure F.2: Single system 1 with points of variability
F.2.2 Single System 1 – Design without Variability

Figure F.3: Single system 1 without points of variability

F.3 Single System 2 – Requirements Specification

1 The system shall pre-process weather satellite images before they are transmitted to the ground station for analysis.
1.1 The system shall acquire a digital satellite photograph from the satellite’s onboard camera.
2.1 Each image shall have a resolution of 373 x 523 with 256 colours.
2 SA A colour spectrum analysis shall be performed on the image.
2.2 The analysis shall identify the most intense cloud regions on the image.
2.2.1 The initial image shall be smoothed to intensify large areas of clouds.
2.2.2 The smoothed image shall have the number of colours reduced to 10 red, 1 green and 1 blue colour steps.
2.2.3 The image shall be negated.
2.2.4 All except the darkest area shall be deleted.
3 OP The original image shall be merged into the analysed image.
4 MA Further cartographic information shall be added.
4.3 Major cities and landmarks shall be added to the image.
6.1 After processing the final image shall be released for further processing.
6.2 The image shall be transmitted to the ground station.

F.3.1 Single System 2 – Design with Variability

Figure F.4: Single system 2 with points of variability
F.3.2 Single System 2 – Design without Variability

**Figure F.5:** Single system 2 without points of variability
F.4 Single System 3 – Requirements Specification

1 The system shall pre-process weather satellite images before they are transmitted to the ground station for analysis.

1.1 The system shall acquire a digital satellite photograph from the satellite’s onboard camera.

1.1 Each image shall have a resolution of 373 x 523 with 256 colours.

2 SA A colour spectrum analysis shall be performed on the image.

2.1 The analysis shall identify cloud regions by their colour intensity:

2.1.1 The initial image shall be smoothed to intensify large areas of clouds.

2.1.2 The smoothed image shall have the number of colours reduced to 10 red, 1 green and 1 blue colour steps.

2.1.3 The final image shall be negated.

2.1.4 OP The resulting highlighted cloud regions shall be outlined.

3 OP The original image shall be merged into the analysed image.

4 MA Further cartographic information shall be added.

4.1 Map outlines of the specific landmasses shall be added.

4.3 Major cities and landmarks shall be added to the image.

5 OP A border shall be added to the processed image.

5.1 The border shall be 10 pixels wide for the left, right and bottom border.

5.2 The top border shall have a width of 36 pixels.

6 SA After processing the final image shall be released for further processing.

6.1 The image shall be stored to an internal drive for further analysis and processing by other sub-systems.
F.4.1 Single System 3 – Design with Variability

Figure F.6: Single system 3 with points of variability
F.4.2 Single System 3 – Design without Variability

Figure F.7: Single system 3 without points of variability
F.5 Single System 4 – Requirements Specification

1 The system shall pre-process weather satellite images before they are transmitted to the ground station for analysis.
1.1 The system shall acquire a digital satellite photograph from the satellite’s onboard camera.
1.1 Each image shall have a resolution of 373 x 523 with 256 colours.
2 SA A colour spectrum analysis shall be performed on the image.
2.2 The analysis shall identify the most intense cloud regions on the image.
2.2.1 The initial image shall be smoothed to intensify large areas of clouds.
2.2.2 The smoothed image shall have the number of colours reduced to 10 red, 1 green and 1 blue colour steps.
2.2.3 The image shall be negated.
2.2.4 All except the darkest area shall be deleted.
3 OP The original image shall be merged into the analysed image.
4 MA Further cartographic information shall be added.
4.1 Map outlines of the specific landmasses shall be added.
4.2 Longitudinal and latitudinal lines shall be added to the image.
5 OP A border shall be added to the processed image.
5.1 The border shall be 10 pixels wide for the left, right and bottom border.
5.2 The top border shall have a width of 36 pixels.
5.3 OP The image shall be time-stamped and labelled within its top border.
6 SA After processing the final image shall be released for further processing.
6.1 The image shall be stored to an internal drive for further analysis and processing by other sub-systems.
F.5.1 Single System 4 – Design with Variability

Figure F.8: Single system 4 with points of variability
F.5.2 Single System 4 – Design without Variability

Figure F.9: Single system 4 without points of variability
F.6 Single System 5 – Requirements Specification

1 The system shall pre-process weather satellite images before they are transmitted to the ground station for analysis.

1.1 The system shall acquire a digital satellite photograph from the satellite’s onboard camera.

1.1 Each image shall have a resolution of 373 x 523 with 256 colours.

2 A colour spectrum analysis shall be performed on the image.

2.1 The analysis shall identify cloud regions by their colour intensity:

2.1.1 The initial image shall be smoothed to intensify large areas of clouds.

2.1.2 The smoothed image shall have the number of colours reduced to 10 red, 1 green and 1 blue colour steps.

2.1.3 The final image shall be negated.

2.1.4 The resulting highlighted cloud regions shall be outlined.

3 The original image shall be merged into the analysed image.

4 Further cartographic information shall be added.

4.1 Map outlines of the specific landmasses shall be added.

4.2 Longitudinal and latitudinal lines shall be added to the image.

4.3 Major cities and landmarks shall be added to the image.

5 A border shall be added to the processed image.

5.1 The border shall be 10 pixels wide for the left, right and bottom border.

5.2 The top border shall have a width of 36 pixels.

5.3 The image shall be time-stamped and labelled within its top border.

6 After processing the final image shall be released for further processing.

6.1 The image shall be stored to an internal drive for further analysis and processing by other sub-systems.
F.6.1 Single System 5 – Design with Variability

**Figure F.10:** Single system 5 with points of variability
F.6.2 Single System 5 – Design without Variability

Figure F.11: Single system 5 without points of variability
Appendix G

Data Analysis Results

G.1 Main/Subroutine Experiments

<table>
<thead>
<tr>
<th></th>
<th>Execution Time Overhead</th>
<th>Memory Space Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Residual error</td>
<td>0.0075</td>
<td>0.0066</td>
</tr>
<tr>
<td># sets of bivariate date</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Degrees of freedom</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>t-stat</td>
<td>47.49</td>
<td>50.44</td>
</tr>
<tr>
<td>t-distribution for 99.9%</td>
<td>12.92</td>
<td>12.92</td>
</tr>
</tbody>
</table>

**Table G.1:** Results of data analysis for main/subroutine experiments

<table>
<thead>
<tr>
<th>Observation</th>
<th>Execution Time Residuals</th>
<th>Memory Space Residual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.287667264</td>
<td>0.134246</td>
</tr>
<tr>
<td>2</td>
<td>-0.047331921</td>
<td>0.181713</td>
</tr>
<tr>
<td>3</td>
<td>0.432640134</td>
<td>0.311881</td>
</tr>
<tr>
<td>4</td>
<td>-0.234791126</td>
<td>0.187828</td>
</tr>
<tr>
<td>5</td>
<td>-0.014552061</td>
<td>-0.32673</td>
</tr>
</tbody>
</table>

**Table G.2:** Residuals for main/subroutine experiments
**Figure G.1**: Plot of residuals of execution time overhead for main/subroutine experiments

**Figure G.2**: Plot of residuals of memory space overhead for main/subroutine experiments

### G.2 Object-Oriented Experiments

<table>
<thead>
<tr>
<th></th>
<th>Execution Time Overhead</th>
<th>Memory Space Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Residual error</td>
<td>0.0106</td>
<td>0.0271</td>
</tr>
<tr>
<td># sets of bivariate data</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Degrees of freedom</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>t-stat</td>
<td>51.37</td>
<td>31.73</td>
</tr>
<tr>
<td>s-distribution for 99.9%</td>
<td>12.92</td>
<td>12.92</td>
</tr>
</tbody>
</table>

**Table G.3**: Results of data analysis for object-oriented experiments
<table>
<thead>
<tr>
<th>Observation</th>
<th>Execution Time Residuals</th>
<th>Memory Space Residual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.17083</td>
<td>0.072451</td>
</tr>
<tr>
<td>2</td>
<td>-0.19695</td>
<td>0.064139</td>
</tr>
<tr>
<td>3</td>
<td>-0.06925</td>
<td>-0.70042</td>
</tr>
<tr>
<td>4</td>
<td>-0.16932</td>
<td>0.108206</td>
</tr>
<tr>
<td>5</td>
<td>0.352141</td>
<td>0.279037</td>
</tr>
</tbody>
</table>

**Table G.4:** Residuals for object-oriented experiments

**Figure G.3:** Plot of residuals of execution time overhead for object-oriented experiments

**Figure G.4:** Plot of residuals of memory space overhead for object-oriented experiments
G.3 Pipe and Filter Experiments

<table>
<thead>
<tr>
<th>Residual error</th>
<th>Execution Time Overhead$^1$</th>
<th>Memory Space Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0044</td>
<td>0.0009</td>
</tr>
<tr>
<td># sets of bivariate data</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Degrees of freedom</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>t-stat</td>
<td>24.96</td>
<td>193.84</td>
</tr>
<tr>
<td>t-distribution for 99.9%</td>
<td>12.92</td>
<td>12.92</td>
</tr>
</tbody>
</table>

Table G.5: Results of data analysis for pipe and filter experiments

<table>
<thead>
<tr>
<th>Observation</th>
<th>Execution Time Residuals</th>
<th>Memory Space Residual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.6899</td>
<td>-0.06042</td>
</tr>
<tr>
<td>2</td>
<td>-0.76787</td>
<td>0.103325</td>
</tr>
<tr>
<td>3</td>
<td>0.001989</td>
<td>0.042903</td>
</tr>
<tr>
<td>4</td>
<td>-0.02673</td>
<td>0.042903</td>
</tr>
<tr>
<td>5</td>
<td>0.777829</td>
<td>-0.09939</td>
</tr>
</tbody>
</table>

Table G.6: Residuals for pipe and filter experiments

Figure G.5: Plot of residuals of execution time overhead for pipe and filter experiments

$^1$ Regression analysis was performed on square root of fitted quadratic equation.
Figure G.6: Plot of residuals of memory space overhead for pipe and filter experiments

G.4 Event-Based / Implicit Invocation Experiments

<table>
<thead>
<tr>
<th></th>
<th>Execution Time Overhead</th>
<th>Memory Space Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Residual error</td>
<td>0.0002</td>
<td>0.0034</td>
</tr>
<tr>
<td># sets of bivariate data</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Degrees of freedom</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>t-stat</td>
<td>73.96</td>
<td>65.54</td>
</tr>
<tr>
<td>t-distribution for 99.9%</td>
<td>12.92</td>
<td>12.92</td>
</tr>
</tbody>
</table>

Table G.7: Results of data analysis for event-based experiments

<table>
<thead>
<tr>
<th>Observation</th>
<th>Execution Time Residuals</th>
<th>Memory Space Residual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.8751</td>
<td>-0.30863</td>
</tr>
<tr>
<td>2</td>
<td>-0.36062</td>
<td>-1.10339</td>
</tr>
<tr>
<td>3</td>
<td>-0.6313</td>
<td>0.127064</td>
</tr>
<tr>
<td>4</td>
<td>0.165026</td>
<td>0.101839</td>
</tr>
<tr>
<td>5</td>
<td>0.345413</td>
<td>0.31766</td>
</tr>
</tbody>
</table>

Table G.8: Residuals for event-based experiments

2 Regression analysis was performed on square root of fitted quadratic equation.
Figure G.7: Plot of residuals of execution time overhead for event-based experiments

Figure G.8: Plot of residuals of memory space overhead for event-based experiments

G.5 Repository Experiments

<table>
<thead>
<tr>
<th></th>
<th>Execution Time Overhead</th>
<th>Memory Space Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Residual error</td>
<td>0.0062</td>
<td>0.0002</td>
</tr>
<tr>
<td># sets of bivariate</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Degrees of freedom</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>t-stat</td>
<td>88.62</td>
<td>474.59</td>
</tr>
<tr>
<td>t-distribution for 99%</td>
<td>12.92</td>
<td>12.92</td>
</tr>
</tbody>
</table>

Table G.9: Results of data analysis for repository experiments
<table>
<thead>
<tr>
<th>Observation</th>
<th>Execution Time Residuals</th>
<th>Memory Space Residual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.091738</td>
<td>-0.02079</td>
</tr>
<tr>
<td>2</td>
<td>0.151731</td>
<td>-0.01949</td>
</tr>
<tr>
<td>3</td>
<td>0.120593</td>
<td>0.040053</td>
</tr>
<tr>
<td>4</td>
<td>-0.12891</td>
<td>-0.00077</td>
</tr>
<tr>
<td>5</td>
<td>-0.10827</td>
<td>-0.01325</td>
</tr>
</tbody>
</table>

Table G.10: Residuals for repository experiments

![Figure G.9: Plot of residuals of execution time overhead for repository experiments](image)

**Figure G.9:** Plot of residuals of execution time overhead for repository experiments

![Figure G.10: Plot of residuals of memory space overhead for repository experiments](image)

**Figure G.10:** Plot of residuals of memory space overhead for repository experiments