Trader: Reliability of high-volume consumer products

A collaborative research project on the reliability of complex embedded systems

Embedded Systems Institute (The Netherlands)

Trader: Reliability of high-volume consumer products

A collaborative research project on the reliability of complex embedded systems

Editor:

Roland Mathijssen   Embedded Systems Institute

Publisher:

Embedded Systems Institute, Eindhoven, the Netherlands

Publisher:

Embedded Systems Institute

TU/e Campus, Den Dolech 2

P.O. Box 513, 5600 MB Eindhoven

Eindhoven, the Netherlands

Keywords:

reliability; dependability; fault-tolerance; robustness; recovery; diagnosis; error detection; embedded systems; software; modeling techniques; user perception

ISBN: 978-90-78679-04-2

Rev. 17 October 2009

The TRADER project has been carried out under the responsibility of the Embedded Systems Institute and is partially supported by the Netherlands Ministry of Economic Affairs under the BSIK03021 program.

Acknowledgements

We gratefully acknowledge the input of many people from NXP over the 5 years of the Trader project. Special thanks go to Louis Stroucken, Ben Pronk, Rob van Twist and Rob Golsteijn for sharing their TV knowledge with all of the Trader researchers. Also special thanks to Ben Pronk (again) and Piërre van de Laar (ESI) for the many days they spent reviewing every chapter of this book. Thanks also go to Bauke Sijtsma and Jaap Knoester for their careful work on the layout of this book.

Foreword

In 2001, long before the shops were filled with large flat screen digital TVs, developments at Philips Semiconductors (now NXP Semiconductors) were advancing well. Architects and managers were planning high performance products that would hit the market 3~7 years in the future. Part of that dream has, of course, come true with advanced TVs now commonplace and at modest prices. Beneath the covers of today's elegant TVs lies a dramatic increase in technical complexity. This continuous growth in complexity has been driven by customer demand for more channels, extended connectivity, higher picture quality and larger screen sizes. This has, in turn, created a major challenge for the global providers of TV technologies.

One of the important factors for the architects at Philips was to avoid these new generation TVs behaving like PCs. Typical customers would not accept strange messages on the screen from some embedded operating system or worst still a blue screen followed by a reboot, many customers would not even know what a reboot was! Maintaining robust product behavior was of paramount importance to Philips even under this sharp increase in complexity.

In today's competitive climate it is essential to deliver new and enhanced products almost every 6 months. This puts an incredible pressure on the development staff. Evolutionary improvements to design methods and tools will, at a certain stage, impose boundaries on the ability to deal with this increasing design complexity and need for shorter time-to-market. At a certain point in time revolutionary new techniques need to be introduced to keep up with this ever rising complexity at affordable costs.

With this is mind chief architects from Philips knew that the then current "Best in Class" practices being used would cope for products being planned for some years to come, but for how long? The increasing complexity not only was driving the need for more and more highly skilled design staff, it also was leading to a significant increase in product test and verification effort. To help limit this growth in resource demand Philips, together with ESI, defined the Trader project (Television Related Architecture and Design to Enhance Reliability). Trader was conceived to provide techniques to ensure that, in spite of reducing product design cycles and increasing design complexity, customers would continue to receive the highest quality of service.

The concept was to add improvements to all stages of the product creation process. To achieve this, research groups from Philips, the Technical University Delft, Technical University Eindhoven, the University of Twente, the University in Leiden and IMEC from Belgium came together to explore ways to enhance the reliability of future television and consumer products. Studies were carried out on how customers perceived the behavior, both good and bad, of state-of-the-art technology related consumer products such as digital TV. This knowledge helped to prioritize the wide range of possible features in products and understand what customers find most important. Architectures were designed which allowed system 'glitches' to be caught, handled and hidden from customers. Rigorous test and diagnostic techniques were developed to shorten the product test and verification phases. Used together, these techniques could help create advanced products that, to a certain degree, would be aware of their own behavior. Armed with a built-in strategy to correct themselves they could then ensure that the customer experience would not be compromised.

This book marks the end of the Trader project. It describes the various directions explored and results obtained. Whilst the overall vision was to combine the set of research results to create system awareness, each individual piece of work has the potential to be used standalone and still yield beneficial gains for companies such as NXP Semiconductors. Looking back we can ask ourselves what key lessons have we learnt? The main lesson is that, within an organization, it is not possible to be expert in all fields and working with external partners can bring added advantages. Trader used the domain of digital television as its focus and this is a field in which NXP Semiconductors is a world leader. Outside of our core competencies, the external research groups within the Trader project brought us significant benefits by exploring new architectures, development methodologies and furthering our understanding of customer behavior in their use of consumer electronics. These factors will help shape the way future consumer products are conceived and brought effectively to market.

NXP Semiconductors is happy with the results of this project. Trader has demonstrated how collaborative research can be used to bring innovative ways of thinking into our organization and achieve breakthrough results. We expect, where appropriate, to continue this way of working as an excellent way of bringing benefits to our organization and the technology ecosystem of which we are part.

René Penning de Vries
Senior Vice President and Chief Technology Officer
NXP Semiconductors
Eindhoven, the Netherlands
September 2009

Preface

It has now become the tradition that every four-year project led by the Embedded Systems Institute is concluded with a book that contains an overview of the obtained results. This book is the fourth volume in a series that started in November 2006. It is also the first book describing a project in the series of six Bsik projects (Besluit Subsidies Investeringen Kennis-infrastructuur) that ESI started in 2004.

This book describing the Trader project addresses the challenges of maintaining or even further improving the reliability of high-volume consumer products in the face of increasing complexity. The Trader project was carried out by the Embedded Systems Institute, NXP Home, NXP Research, Delft University of Technology, Eindhoven University of Technology, University of Twente, Leiden University, IMEC and TASS. The project started in 2004 and finished mid 2009.

Concerns about system reliability are present both in society and in industry. The omnipresence of embedded systems will be accepted by society only if a wide belief develops that embedded systems can genuinely be relied upon. Manufacturers and users of embedded systems realize that they can only survive if the products are inherently reliable. It is a shame to see that in many market segments products become less reliable when the level of embedded software increases. This is why reliability is one of the central themes of the ESI Research Agenda. Following the ESI approach to applied research, the Trader project was organized as an industry-as-laboratory project, where industry provides the experimental platform to develop and validate new methods, techniques and tools. ESI is unique in the application of this research format to the problems of embedded systems engineering. It has proven to be most successful in producing substantial results leading both to industrial innovation and high-quality academic output.

Trader is the first project for which we have been fortunate enough to have NXP as the carrying industrial partner. Investigating new reliability methods and testing them in the fast changing world of digital television, has put an extra challenge on consolidation and transfer of results. It also gave the project participants a valuable challenge to constantly validate whether the new concepts and methods were applicable to this dynamic marketplace. All participants have shown great commitment and their contributions have led to the success of the Trader project, for which I would like to thank them. NXP and the Dutch Ministry of Economic Affairs provided the financial basis for carrying out the project, and their support is gratefully acknowledged. We hope that with this book we can share the most important results and insights with a wider industrial and scientific community.

Prof. dr. ir. Boudewijn Haverkort
Scientific Director & Chair
Embedded Systems Institute
the Netherlands
September 2009

Contents

Chapter 1

Introduction

Author: Jozef Hooman

Abstract: The reliability of high-volume products, such as consumer electronic devices, is threatened by the combination of increasing complexity, decreasing time-to-market, and strong cost constraints. The Trader project addresses these issues by developing methods and techniques to optimize reliability. These include improvements at development time, but also techniques to maintain a high level of reliability after product release. We present a runtime awareness concept, which aims to minimize any user exposure to product-internal technical errors, thereby improving user-perceived reliability. This chapter provides an overview of the main concepts and the project results.

1.1 Context

In the Trader project, academic and industrial partners collaborated to optimize the dependability of high-volume products, such as consumer electronic devices. The project partners involved were: NXP Semiconductors, NXP Research, Embedded Systems Institute (ESI), TASS, IMEC (Belgium), University of Twente, Delft University of Technology, University of Leiden, and Design Technology Institute (DTI) at the Eindhoven University of Technology. The project started in September 2004, with a duration of five years, and included seven PhD students and two postdocs. The so-called Carrying Industrial Partner (CIP) of this project was NXP Semiconductors, providing the project with a focus on multimedia products. NXP provided the problem statement and proposed relevant case studies, which in the case of Trader were mainly taken from the TV domain.

The problem statement of Trader is based on the observation that the combination of increasing complexity of consumer electronic products and decreasing time-to-market pressures will make it extremely difficult to produce totally reliable devices that meet the dependability expectations of customers.

A current high-end TV is already a very complex device which can receive analog and digital input from many possible sources, using many different coding standards. It can be connected to various types of recording devices and includes many features such as picture-in-picture, Teletext, sleep timer, child lock, TV ratings, emergency alerts, TV guide, and advanced image processing. Moreover, there is a growing demand for features that are shared with other domains, such as photo browsing, MP3 playing, USB, games, databases, and networking. As a consequence, the amount of software in TVs has seen an exponential increase from 1 KB in 1980 to more than 20 MB in current high-end TVs. The hardware complexity is also increasing rapidly, for instance, to support real-time decoding and processing of high-definition images for large screens and multiple tuners. To meet the hard real-time requirements, a TV is designed as a system-on-chip with multiple processor cores, various types of memory, and dedicated hardware accelerators.

At the same time, there is a strong pressure to decrease time-to-market. To be able to realize products with many new features quickly, components developed by others have to be incorporated quickly into an existing architecture. This includes so-called third-party components, e.g., for audio and video decoding. Moreover, there is a clear trend towards the use of downloadable components to increase product flexibility and to allow new business opportunities (selling new features, games, etc.). Given the large number of possible user settings and types of input, exhaustive testing is impossible. Also, the product must be able to tolerate certain faults in the input stream. Customers expect, for instance, that products can cope gracefully with deviations from coding standards or bad input quality.

Although companies invest a lot of attention and effort to avoid faults in released products, it is expected that without additional measures both internal and external faults are serious threats to product dependability. The cost of non-quality, however, is high; it leads to many returned products, damaged brand image, and reduced market share.

1.1.1 Trader goal

The main goal of the Trader project is to improve the user-perceived dependability of high-volume products. The aim is to develop techniques that can compensate and mask faults in released products, such that they satisfy user expectations. The main challenge is to realize this without increasing development time and, given the domain of high-volume products, with minimal additional hardware costs and without degrading performance. Hence, classical fault-tolerance techniques that rely heavily on redundancy (e.g., duplication or even triplication of hardware and software) and require many additional resources are not suitable for this domain.

1.1.2 Terminology

In this book, the terminology of [Avizienis, 2004] is adopted. A *failure* of a system with respect to an external specification is an event that occurs when a state change leads to a run that no longer satisfies the external specification. An *error* is the part of the system state that may lead to a failure. For instance, an error can be a wrong memory value or a wrong message in a queue. A *fault* is the adjudged or hypothesized cause of an error which is not part of the system state. Examples of faults are programming mistakes (e.g., divide by zero) or unexpected input. *Reliability* is the probability that a system satisfies an external specification, i.e. does not lead to any failure, during a certain time period under certain operating conditions.

1.2 Approach

In this section we describe our approach, starting with a short explanation of the industry-as-laboratory approach and the main case studies in Section 1.2.1. Research on the user perception of reliability is described in Section 1.2.2. An overview of work on reliability improvements during the development process can be found in Section 1.2.3. The Trader vision on runtime awareness is explained in Section 1.2.4. Related work can be found in Section 1.3.

1.2.1 Industry-as-lab & Trader case studies

Similar to other ESI projects[1], we have followed in Trader an *industry-as-laboratory* approach [Potts, 1993]. This means that there is a frequent interaction between industrial problem owners and solution providers to avoid that researchers are solving problems that have already evolved in industry. Research results are studied in the intended industrial context as soon as possible to investigate the applicability and the scalability of solution strategies under the relevant practical constraints. Successful solutions immediately get industrial credibility which makes a transfer into industry easier. Moreover, the industry-as-lab approach enables academic researchers to collect realistic experimental data on their approach. Industrial application of their research usually leads to new insights, e.g., on unrealistic assumptions, scalability, or exceptional cases, and this often stimulates new research.

In the Trader project, most of the case studies were taken from the TV domain as provided by NXP Semiconductors. To become familiar with this domain and to analyze the possibilities of dealing with failures, we started with an existing analog TV in which a few systematically reproducible faults were injected. A number of input scenarios, i.e., sequences of key presses on a remote control, led to lock-up failures in Teletext. The project team investigated a number of solutions to detect and to correct these failures. During later phases of the project, techniques have been applied to more complex digital TVs that were under development at NXP. To experiment with long-term recovery techniques that would require architectural changes and code modifications, we also used the open source media player MPlayer [MPlayer, 2007]. This allows code adaptations, fault-injections and runtime experiments without the need of dedicated development environments and special hardware.

1.2.2 User perception

The aim of the work on user perception of reliability is to capture user perceived failure severity, to get an indication of the level of user-irritation caused by a product failure. The research has investigated the influence of failure and user characteristics using consumer experiments in a TV laboratory and a web-based experiment. During the laboratory experiments on function importance, it turned out that also failure attribution has a significant impact. That is, in case of a failure it makes a difference whether a user blames the TV or assumes there are external causes for the failure. For instance, in an experiment users ranked image quality as important and a motorized swivel, which can be used to turn the TV, as much less important. When injecting failures in these functions, however, users often turn out to be very tolerant concerning bad image quality (which is attributed to external sources), but they get very irritated if the swivel does not work correctly. An overview of this research can be found in Chapter 2.

1.2.3 Design-time improvements

We give an overview of Trader research that aims at reliability improvements during the development of the system, starting with the work described in this book:

- Chapter 3 discusses robustness strategies to deal with variations in the interpretation of the Teletext standard by broadcasters and TV set manufacturers. These different interpretations are caused by omissions in the Teletext standard, especially concerning the dynamic behavior, i.e., dealing with changes in the content of pages.
- Chapter 4 presents results on the use of a tool like QA-C which checks whether a piece of C code conforms to a certain coding standard such as MISRA. Since these code analysis tools

[1] See http://www.esi.nl/projects/

often produce a large number of non-conformance warnings, it is important to identify the most important violations and the most relevant coding rules. To this end, historical data at NXP has been analyzed to determine the correlation between faults in the code and rule violations.
- Chapter 5 describes the application of stress testing in the TV domain. The main idea is to take away shared resources (such as CPU cycles or bus bandwidth) artificially, to simulate the occurrence of errors or the addition of an additional resource user. The study of the effect of such overload situations on the system behavior and its fault-tolerant mechanisms has shown to be very useful in the TV domain. In addition, this chapter also presents visualizations of CPU usage and bus bandwidth that provide useful insight in system behavior.
- Chapter 6 discusses reliability improvements by the use of aspect orientation to prevent scattering and tangling of concerns, such as exception handling, initialization, and thread usage, in the software. To deal with aspect-orientation in a resource-constrained environment and to conform to the Koala component model of NXP, a new language called AspectKoala has been defined.
- Chapter 7 presents a method for the early detection and removal of requirements errors, which are an important source of unreliability and project failures. The approach is based on executable models of dynamic system behavior and can also be used include the high-level system architecture, which makes it possible to check consistency and conformance to the requirements.
- Chapter 9 describes the spectrum-based fault localization technique which can be used to improve the efficiency of the debugging process. This technique indicates for a particular problem which part of the code most likely contains the error. This technique has been applied successfully to a number of case studies, including a number of problem reports at NXP.

Finally, we mention two other Trader activities that are not represented in this book, but have been published elsewhere:

- An approach for early reliability analysis of software architectures has been defined [Sözer, 2007-a].
- An introduction to software fault tolerance has been written, describing the main concepts and a large number of design patterns for error detection and error recovery [Deckers, 2005].

1.2.4 Runtime awareness

Looking at a number of failures of consumer electronic devices, it is often the case that a user can immediately observe that something is wrong, whereas the system itself is completely unaware of the problem. Systems are often realized in a way that corresponds to the open-loop approach in control theory; for a certain input, the required actions are executed, but it is never checked whether these actions have the desired effect on the system and whether the system is still in a healthy state.

The long term vision of the Trader project is to “close the loop” and to add a kind of feedback control to products. By monitoring the system and comparing system observations with a model of the desired behavior at runtime, the system gets a form of runtime awareness which makes it possible to detect that its customer-perceived behavior is (or is likely to become) erroneous. In addition, the aim is to provide the system with a strategy to correct itself. The main ingredients of such a runtime awareness and correction approach are depicted in Figure 1.1.

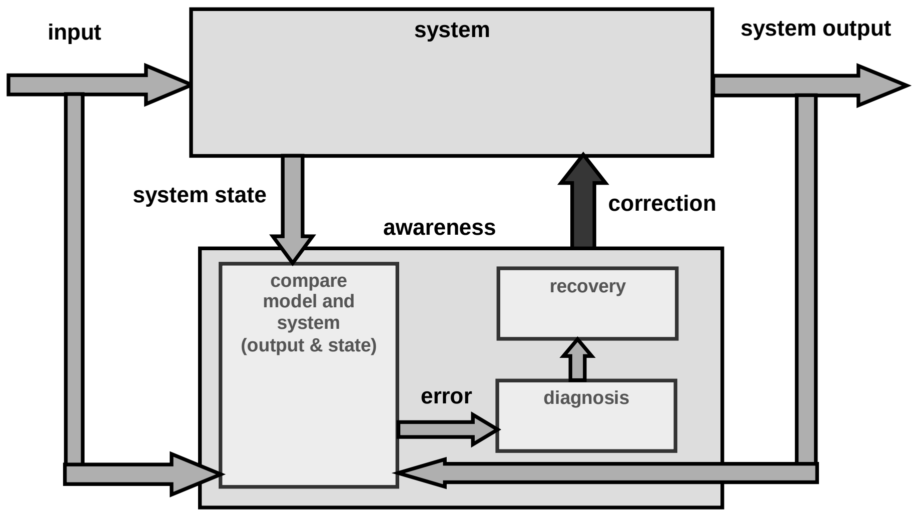


Figure 1.1 *Adding awareness at runtime*

The four main parts of this approach are:

- *Observation:* observe relevant inputs, outputs and internal system states. For instance, for a TV we may want to observe keys presses from the remote control, internal modes of components (dual/single screen, menu, mute/unmute, etc), load of processors and busses, buffers, function calls to audio/video output, sound level, etc.
- *Error detection:* detect errors, based on observations of the system.
- *Diagnosis:* in case of an error, find the most likely cause of the error.
- *Recovery:* correct erroneous behavior, based on the diagnosis results and information about the expected impact on the user.

Note that for complex systems it will not be feasible to include a complete model of the desired system behavior, but the approach allows the use of partial models, concentrating on what is most relevant for the user. Moreover, we can apply this approach hierarchically and incrementally to parts of the system, e.g., to third-party components. Typically, there will be several awareness monitors in a complex system, for different components, different aspects, and different kinds of faults.

In the next sections, we give an overview of the Trader research on the main parts of the runtime awareness concept.

Observation

A number of techniques to observe relevant aspects of the system have been investigated. This includes:

- Monitoring and visualization of CPU usage and bus bandwidth, as described in Chapter 5.
- Instrumentation of software with observation code, for instance, by means of aspect-oriented techniques as presented in Chapter 6 or based on the cfront parser for ANSI C as mentioned in Chapter 10.

Error Detection

Within Trader, a number of techniques for runtime error detection have been developed:

- A method to detect runtime errors using models of desired behavior is described in Chapter 8. An experimental Linux-based framework has been developed in which a software application and a (partial) specification of its desired behavior can be inserted.
- Fault screeners, which are generic invariants that are checked at runtime, are presented in Chapter 9 as a simple error detection method with a low overhead.
- An approach which checks the consistency of internal modes of components turned out to be effective in detecting a number of injected faults in the Teletext part of an analog TV [Sözer, 2007-b].

Diagnosis

A technique for runtime diagnoses based on program spectra is explained in Chapter 9. This technique can be implemented with a low overhead in terms of time and space. Chapter 9 also compares spectra-based diagnosis with model-based techniques.

Recovery

Two recovery techniques have been studied in Trader:

- Chapter 11 presents a method for resource management which allows runtime decisions between resource usage and user-perceived quality. For instance, it has been demonstrated that in case of overload situations (e.g., due to intensive error correction on a bad input signal), an image processing task can be migrated from one processor to another, leading to improved image quality.
- Chapter 12 defines a framework for local recovery which makes it possible to recover separate units of a system in isolation. Moreover, a systematic approach for the decomposition into recoverable units is proposed. The framework has been applied successfully to the MPlayer.

1.3 Related work

Traditional fault-tolerance techniques such as Triple Modular Redundancy and N-version programming are not applicable in our application domain of high-volume products, because of the cost of the required redundancy. Related work that also takes cost limitations into account can be found in the research on fault-tolerance of large-scale embedded systems [Neema, 2004]. They apply the autonomic computing paradigm to systems with many processors to obtain a healing network. Similar to our approach is the use of a kind of controller-plant feedback loop. Related work on adding a control loop to an existing system is described in the middleware approach of [Parekh, 2006] where components are coupled via a publish-subscribe mechanism. A method to wrap COTS components and monitor them using specifications expressed as a UML state diagrams is presented in [Shin, 2006]. The analogy between self-controlling software and control theory has already been observed in [Kokar, 1999]. Garlan et al. have developed an adaptation framework where system monitoring might invoke architectural changes [Garlan, 2003]. Using performance monitoring, this framework has been applied to the self-repair of web-based client-server systems. Our approach is also inspired by other application domains, such as the success of helicopter health and usage monitoring [Cronkhite, 1993].

1.4 References

[Avizienis, 2004] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. *Basic concepts and taxonomy of dependable and secure computing*. IEEE Transactions on Dependable and Secure Computing, 1(1): pp. 11–33, 2004

[Cronkhite, 1993] J.D. Cronkhite. *Practical application of health and usage monitoring (HUMS) to helicopter rotor, engine, and drive systems*. AHS, Proc. 49th Annual Forum. Volume 2. pp. 1445-1455, 1993

[Deckers, 2005] R. Deckers, P. Janson, F. Ogg, and P. van de Laar. *Introduction to Software Fault Tolerance: Concepts and Design Patterns*. Philips Technical Report PR-TN 2005/00451, Philips Research Eindhoven, 2005

[Garlan, 2003] D. Garlan, S. Cheng, and B. Schmerl. *Increasing system dependability through architecture-based self-repair*. In Architecting Dependable Systems, volume 2677 of LNCS, pp. 61-89. Springer-Verlag, 2003

[Kokar, 1999] M. M. Kokar, K. Baclawski, and Y. A. Eracar. *Control theory-based foundations of self-controlling software*. IEEE Intelligent Software, pp. 37–45, 1999

[MPlayer, 2007] MPlayer: *Open source media player*. http://www.mplayerhq.hu/ , 2007

[Neema, 2004] S. Neema, T. Bapty, S. Shetty, and S. Nordstrom. *Autonomic fault mitigation in embedded systems*. Engineering Applications of Artificial Intelligence, 17: pp. 711–725, 2004

[Parekh, 2006] J. Parekh, G. Kaiser, P. Gross, and G. Valetto. *Retrofitting autonomic capabilities onto legacy systems*. Cluster Computing, 9(2): pp. 141–159, 2006

[Potts, 1993] C. Potts. *Software-engineering research revisited*. IEEE Software. Volume 10, Issue 5, Sept. 1993 pp. 19-28

[Shin, 2006] M.E. Shin, F. Paniagua. *Self-management of COTS component-based systems using wrappers*. Computer Software and Applications Conference (COMPSAC 2006), IEEE Computer Society, pp. 33-36, 2006

[Sözer, 2007-a] H. Sözer, B. Tekinerdoğan, and M. Akşit. *Extending failure modes and effects analysis approach for reliability analysis at the software architecture design level*. In Architecting Dependable Systems IV, volume 4615 of LNCS, pp. 409-433. Springer-Verlag, 2007

[Sözer, 2007-b] H. Sözer, C. Hofmann, B. Tekinerdoğan, M. Akşit. *Detecting mode inconsistencies in component-based embedded software*. DSN Workshop on Architecting Dependable Systems, 2007

Chapter 2

User Perception of Product Failures

Authors: Jeroen Keijzers, Ilse Luyk

Abstract: Currently, the field of consumer electronics is one of the most challenging environments with respect to product design. Due to the combination of the continuous influx of new technology and the economic effects of globalization, it is now possible to create products with a functionality that was unimaginable even one product generation ago against a price level that opens huge markets on a global level in a very short time. However, the combination of technically sophisticated products and global markets is no guarantee for customer satisfaction.

The complexity of current software-based consumer electronics and increasing customer expectations result in increasing numbers of consumer complaints on new products in the consumer electronics industry [Den Ouden, 2006]. Analysis of these complaints indicates that to an increasing degree the root-cause of the complaint cannot be retrieved by the consumer electronics companies [Brombacher, 2005]. Current product defects do not only represent violations of the product specifications, but also unconsidered customer requirements and unexpected product behavior. Consequently, software-based consumer electronics nowadays require a design oriented New Product Development (NPD) process which focuses on the user of the product.

This chapter deals with the research on the possibilities to incorporate the user perspective into the development process of consumer electronics. The first Section (2.1) describes the need for this user focus in the development process in more detail. In Section 2.2, the overall research model is presented. Subsequently, in the third and fourth section the approaches and results of two different research focus areas are presented. Lastly, in Section 2.5 the implications of this research for the consumer electronics industry are presented together with some directions for further research.

2.1 The user perspective in consumer electronics

2.1.1 Trends in consumer electronics industry

Not too long ago, the field of consumer electronics was far more predictable. The development process of the simple and analog hardware-based consumer electronics products of the 1980's/1990's was mainly driven by static product roadmaps that were cost oriented and had a strong focus on product manufacturing optimization. The main goal was to make products that complied with technical specifications.

Therefore, in this period, product defects were mainly specification violations and the customer dissatisfaction level could be defined in terms of the required number of product repairs in the after-market. However, the complexity of current software-based consumer electronics and the increasing customer expectations result in increasing numbers of consumer complaints on new products in the consumer electronics industry [Den Ouden, 2006]. Analysis of these complaints indicates that to an increasing degree the cause of these complaints cannot be retrieved (also referred to as 'No-Failure-Found' (NFF) [Brombacher, 2005], see Figure 2.1.

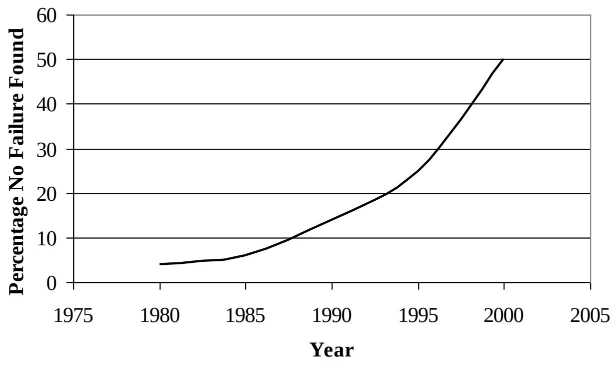


Figure 2.1 *Percentage NFF of total product failures in modern high-volume consumer electronics [Brombacher, 2005]*

Current product defects do not only represent violations of the product specifications, but also unconsidered customer requirements and unexpected product behavior. Consequently, consumer electronics nowadays require a design oriented development process focusing on the user of the product. The focus in the product development process has shifted from a specification focused manufacturing approach to a user focused design approach.

This shift in focus of the product development process has, as a logic consequence, resulted in a simultaneous increase in the level of market uncertainty over the nature and extent of customer's need for new products. This higher market uncertainty results in increased information requirements of the current product development processes. Literature review has been conducted to investigate the potential contribution of existing quality methods on market uncertainty reduction in the development process of consumer electronics products [De Visser, 2008]. It was revealed these methods are incapable of dealing with market uncertainty due to the combination of lacking user-orientation and confined completeness and specificity. Particularly, the unknown impact of quality problems on user dissatisfaction limits the potential contribution of these methods to the quality improvement decision making process. For a more elaborate discussion on the limitations of existing quality methods in consumer electronics, please refer to [De Visser, 2008].

Consequently, in order to deal with the increased market uncertainty for consumer electronics, existing quality methods from literature and practice should be complemented with a user-oriented impact assessment of the identified quality problems (and the accompanying quality improvement decisions). The combination of the existing quality methods with an accompanying user-oriented impact assessment would lower the market uncertainty in the design process [De Visser, 2008]. Eventually, this approach contributes to a decreasing number of consumer complaints in consumer electronics.

This analysis combined with the general aim of this research, results in the formulation of the following research question:

> *How to predict the impact of (potential) product quality problems on customer dissatisfaction in* consumer *electronics industry?*

2.1.2 User perceived failure severity

Based on insights presented in consumer behavior literature, the following concept is defined that captures the user impact of quality problems in consumer electronics [De Visser, 2008]:

> *User Perceived Failure Severity (UPFS) is the level of irritation experienced by the user caused by a product failure*

The UPFS concept expresses the impact of product quality problems on customer dissatisfaction in consumer electronics. Insight into the expected/predicted UPFS resulting from a certain design decision would reduce uncertainty in the product development decision making process and decrease the number of product complaints. UPFS is an emotion measurement and therefore depends on a certain stimulus (product failure) and individual- and environmental characteristics. UPFS can be measured with a combination of existing measurement scales from emotional- and behavioral research [De Visser, 2008].

The proposed user-oriented approach for impact assessment of the identified quality problems will be based on the formulation of a theory-based UPFS prediction model. The combination of the existing quality methods with an accompanying UPFS assessment will lower the market uncertainty in the design process. Eventually, this approach contributes to a decreasing number of consumer complaints in consumer electronics. For a more elaborate discussion on these scientific contributions, please refer to [De Visser, 2008].

2.2 Research model

This section focuses on developing a research model for UPFS prediction in consumer electronics. This model is based on the results of a first exploratory consumer experiment, literature from different scientific fields, and an expert validation session with people from academia and consumer electronics industry [De Visser, 2008]. The starting point for this research model is a high-level UPFS model based on emotion theory. Subsequently, this high-level model is developed into two more detailed UPFS prediction models, resulting in the so-called theory-based UPFS prediction models.

2.2.1 The theory-based UPFS prediction model

This section describes the development of an UPFS prediction model in consumer electronics. As indicated earlier, the adjusted UPFS model is based on the analysis of the results of a UPFS consumer experiment, literature from different scientific fields (e.g. quality and reliability, medicine, marketing and safety) and a validation session with people from academia and industry with a background in product reliability, usability, product development and testing. The starting point for this model is the emotional response model from consumer behavior theory, as the original definition of UPFS is based on this model. This model is shown in Figure 2.2 below.

In the previous section, UPFS is defined as the level of irritation experienced by the user as caused by a product failure. In this definition, UPFS is characterized as an emotion measurement and therefore depends on a certain stimulus (product failure) and individual and environmental characteristics [Chaudhuri, 2006].

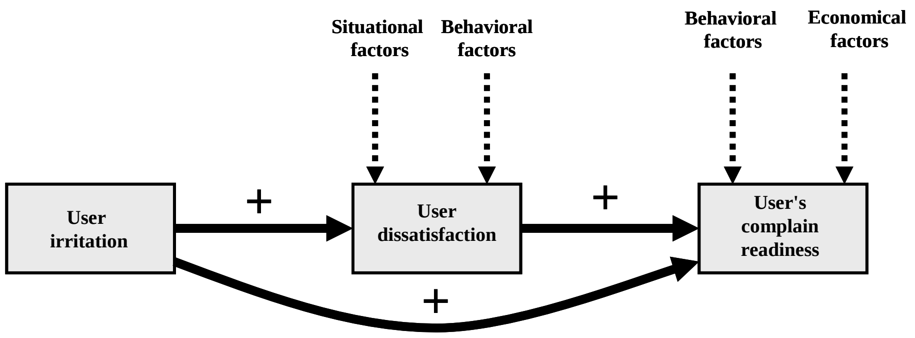


***Figure 2.2** Emotional response model from consumer behavior theory [Manrai, 1991].*

The direct translation of the emotional response model from consumer behavior theory into a high-level UPFS model is illustrated in Figure 2.3. In this high-level model, a product failure is indicated as the stimulus of the emotional response (UPFS). Consequently, the characteristics of this failure directly influence the intensity of this emotional response. Besides the stimulus, several individual and environmental characteristics influence the emotional response as well. In the context of this UPFS model, individual characteristics correspond to the personal traits of the product user. In other words, the characteristics of a user directly influence the level of UPFS. In this high-level UPFS model, the environmental characteristics correspond to the use conditions in which a user operates the product. These use conditions also directly influence the level of UPFS.

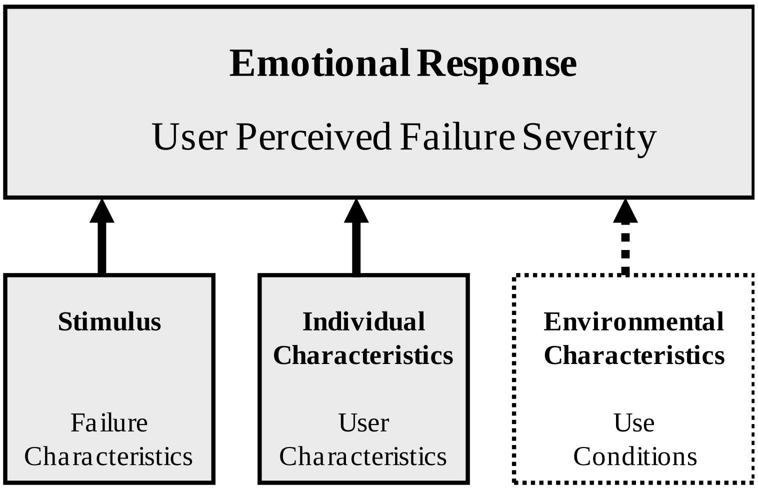

Emotional Response
User Perceived Failure Severity
Stimulus
Failure Characteristics
Individual Characteristics
User Characteristics
Environmental Characteristics
Use Conditions

***Figure 2.3** High-level research model for UPFS prediction.*

The scope of this high-level UPFS model is rather broad. This research mainly focuses on two areas:

1. The investigation of the influence of the stimulus (failure characteristics) on UPFS.
2. The investigation of the influence of the user characteristics on UPFS.

The environmental characteristics (use conditions) are treated as extraneous variables. An extraneous variable is a variable that is not of interest in the study, but which might have influence on the relationships being studied [Stangor, 1998], [Goodwin, 2005]. In order to restrict the influence of these extraneous variables on UPFS in this research, these variables should be kept as constant as possible. This research focus is also represented in Figure 2.3 by the solid arrows from failure characteristics and user characteristics to UPFS and the dotted arrow from the use conditions to UPFS.

This high-level model describes the expected influence of the failure characteristics on UPFS in general. However, in order to predict the UPFS resulting from a certain product failure, the relationships between the individual failure characteristics and UPFS should be explicated. The exploration of these relationships is presented in the next section.

The two above mentioned focus areas, failure characteristics (focus area 1) and user characteristics (focus area 2)**,** are also the key research areas of two different Ph.D. projects. These projects are conducted partly in parallel. However, the research objectives of both projects contribute to the overall understanding of UPFS in consumer electronics. The next two sections describe the different research approaches and findings for both focus areas. Subsequently, in Section 2.5 these findings are combined into general research implications and suggestions for further research.

2.3 Focus area 1: Failure characteristics

This section describes the research in focus area 1, failure characteristics. In the first section, a more detailed research model is presented to investigate the influence of failure characteristics on UPFS. Subsequently, in the second section, the research methodology of this focus area is explicated. Lastly, in Section 2.3.3, the major findings of the research within this focus area are presented.

2.3.1 Detailed research model focus area 1

In order to develop a more detailed research model for focus area 1, the potentially influential failure characteristics should be determined. The combination of different information sources (related literature, an explorative consumer experiment and an expert validation session) resulted in the identification of eight hypothetical failure characteristics [De Visser, 2008]. Table 2.1 gives an overview of these failure characteristics together with a description and their expected relation with UPFS.

A positive influence is expected of Failure Frequency, Failure Impact, Failure Solvability and Function Importance on UPFS. In other words, a higher level of these failure characteristics is expected to result in a higher level of UPFS. A negative influence is expected of Failure Reproducibility and Failure Workaround on UPFS. This means that a higher level of these failure characteristics is expected to result in a lower level of UPFS. For Failure Moment in Use process, the hypothetical relation implicates that a failure earlier in the use-life of the product results in a higher level of UPFS. The direction of the expected relation between Failure Attribution and UPFS is unknown at this moment. For a more elaborate discussion on the expected influence of the failure characteristics on UPFS, please refer to [De Visser, 2008].

Failure Characteristic	**Description**	**Hypothesized relation with UPFS**
Failure Attribution	The cause to which users attribute the failure	Positive[1]
Failure Frequency	Number of failures per time unit under standardized use conditions	Positive
Failure Impact	The percentage loss of the product use life in which the failure occurs	Positive
Failure Moment in Use Process	The degree of repeatability of a failure by the user	Negative[2]
Failure Reproducibility	The degree of repeatability of a failure by the user	Negative
Failure Solvability	The required effort a user should take after failure occurrence to normal functioning of the product (excluding the failed part of the function)	Positive
Failure Work Around	The degree in which the failure occurrence can be prevented by the user by operating the product differently	Negative
Function Importance	The relative importance of the function affected by the failure	Positive

[1] In this case a positive relationship implies: A failure that is internally attributed results in a higher level of UPFS than a externally attributed failure

[2] In this case a negative relationship implies: A failure earlier in the use life of a product results in a higher level of UPFS

Table 2.1 *Overview of the identified failure characteristics.*

The combination of the high-level research model and the identified failure characteristics results in the detailed research model shown in Figure 2.4. As mentioned before, given the time and resource constraints of this research project, the complete validation of the theory-based UPFS model within this project is unfeasible. Furthermore, the validation of the complete model requires a gradual approach in which the (combined) added values of all failure characteristics to the model are evaluated.

However, this research starts small-scale with investigating the influence of two of these failure characteristics on UPFS. In this research, Function Importance (FUI) and Failure Attribution (FA) are selected to be evaluated because the influence of these variables on UPFS is expected to be substantial. In other words, the influence of FUI and FA as independent variables on UPFS as dependent variables is investigated. For a more elaborate discussion on the selection of these two failure characteristics, please refer to [De Visser, 2008].

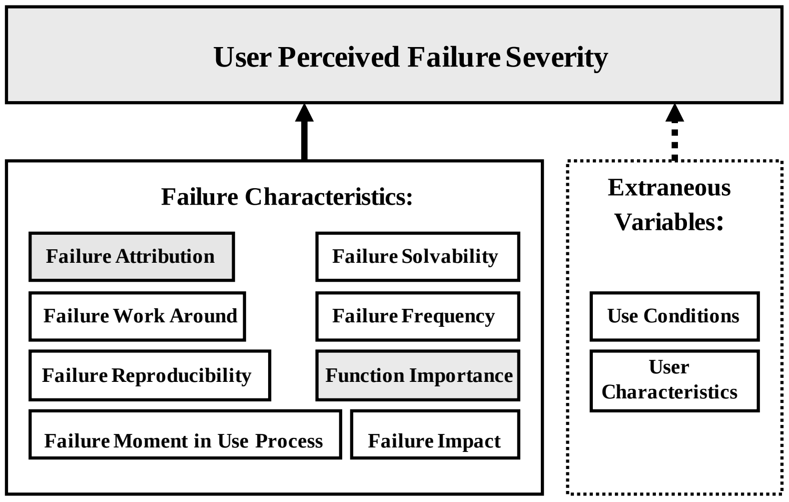


Figure 2.4 *Detailed research model for research focus area 1.*

2.3.2 Research methodology focus area 1

In order to investigate the influence of these failure characteristics on UPFS, two consumer experiments are executed:

1. Function Importance (FUI) Experiment: concentrates on investigating the influence of FUI on UPFS.
2. Failure Attribution[1] (FA) Experiment: concentrates on investigating the influence of FA and FUI on UPFS

For both consumer experiments a between-subject design is used to investigate the influence of the independent variables on UPFS. For the measurement of the experimental variables (FUI, FA and UPFS) different validated measurement scales are applied [De Visser, 2008]. Both experiments are performed in a controlled environment according to a strict experimental protocol in which participants are confronted with different failure scenarios. Between these different experimental failure scenarios, the levels of the independent variables are varied. Subsequently, the emotional response (UPFS) of the test participants to these failure scenarios is measured. Based on these measurements, the influence of FA and FUI on UPFS can be statistically determined [De Visser, 2008].

Only university students are selected as test subjects for both experiments. Students are a relatively homogenous group, which minimizes variability within the conditions of the experiment. Participants are selected using convenience samples of students from the Eindhoven University of Technology. For the FUI experiment, 25 students participated in the experiment. For the FA experiment, 149 students completed the experiment.

Different professionals from the consumer electronics industry contributed to the definition and implementation of the failure scenarios within the consumer experiments. The contribution of these professionals resulted in realistic failure scenarios with different levels of FUI and FA. For more

[1] The name "Failure Attribution Experiment" may be a bit confusing, since both Failure Attribution (FA) and Function Importance (FUI) are selected as independent variables for this experiment. However, the name of this second experiment refers to the addition of the FA variable in this experiment.

information about the research methodology and measurement scales that are used in this research, refer to [De Visser, 2008].

2.3.3 Research findings focus area 1

The FUI consumer experiment resulted in the following research results:

- Within the context of this first consumer experiment, the reliability of the UPFS measurement scale is established. The most common, and the best index for reliability is known as Cronbach's alpha (symbolized as α) [Stangor, 1998]. The Cronbach's alpha is a measure that assesses the consistency of the entire measurement scale and ranges from $\alpha = 0.00$ (indicating that the measure is entirely error) to $\alpha = 1.00$ (indicating that the measure has no error). A general rule of thumb indicates that $\alpha = 0.70$ is considered the minimum required Cronbach's alpha value for a measurement scale to be considered reliable [Stangor, 1998]. The UPFS measurement scale has a Cronbach's α value of 0.83, which makes it a valid measurement scale for UPFS research [De Visser, 2008]. The validation of this measurement scale makes it possible to perform research into the perceived irritation level that is caused by different product failures. Moreover, the UPFS measurement results of this experiment can now be used to investigate the relation between FUI and UPFS.
- Based on the experimental results, the hypothesis "There is no significant difference in UPFS between failures with different levels of function importance (FUI)" should be rejected. Thus, the positive relationship between FUI and UPFS as presented in Table 2.1 is demonstrated. However, it is important to mention that the difference in UPFS level is not very significant (> 0.01). Although the FUI of a failure influences the UPFS, the significance of this effect is rather low.
- One remarkable additional result of this experiment relates to the assumed causes of the failures (FA) by the test participants. The influence of FA on UPFS was not examined in this consumer experiment. Therefore, in order to limit the influence of FA on UPFS, the failure attribution was controlled. In the design of the failure scenarios, both failure scenarios were designed to be internally attributed (cause of the failure was the product not its environment). Within the debriefing phase of the experiment, test participants were questioned about the cause of the experienced failure scenario. The results of this debriefing phase indicate that there is a difference in experienced failure attribution among the failure scenarios. In other words, the perceived cause by test participants was different for the two failure scenarios. This difference in perceived FA between the failure scenarios could have contributed to the relative small effect size of FUI on UPFS. Therefore, the investigation of the influence of FUI on FUI requires another consumer experiment in which the influence of FA is distinguished and investigated.

Based on the research findings of this first experiment, the second (FA) experiment is performed. This FA consumer experiment resulted in the following research results:

- The results of this FA experiment demonstrate that the influence of FUI on UPFS is not significant. Consequently, the hypothesis "There is no significant difference in UPFS between failures with different levels of function importance (FUI)" cannot be rejected. The first FUI consumer experiment did identify a significant influence of FUI on UPFS. However, in this FA experiment this significant influence is not confirmed. A possible explanation for this absence of this signification relation between FUI and UPFS in this FA experiment is the difference in experimental approach [De Visser, 2008].

- The results of this FA experiment confirm that the influence of FA on UPFS is significant. Consequently, the hypothesis "There is no significant difference in UPFS caused by internally or externally attributed failures" should be rejected.
- The experimental results indicate that the interaction effect between FUI and FA is also not significant. Consequently, this insignificance of the interaction effect between FA and FUI in this experiment validates the significance of the relation between FUI and UPFS in the first (FUI) experiment. That is, the absence of this interaction effect implies that the lack of FA control in the previous experiment did not bias the experimental outcomes and the relation between FUI and UPFS.

The combination of both experiments contributed to the partial validation of the detailed research model for research focus area 1 (Figure 2.4). Moreover, based on the results of the two consumer experiments a validated UPFS research approach has been developed. For a more elaborate discussion on this research approach, please refer to [De Visser, 2008].

2.4 Focus area 2: User characteristics

This section describes the research in focus area 2, user characteristics. In the first section, a more detailed research model is presented to investigate the influence of user characteristics on UPFS. Subsequently, in the second section, the research methodology of this focus area is explicated. Lastly, in Section 2.4.3, the major findings of the research within this focus area are presented.

2.4.1 Detailed research model focus area 2

The research conducted in focus area 2, focuses on the investigation of the relation between user characteristics and UPFS. Because a fault in the product development process does not automatically lead to a consumer complaint for each individual user or user group, this research considers different stages between (potential) product development faults and UPFS, as shown in the detailed research model in Figure 2.5 below (derived from [Oliver, 1996], [Kanis, 2005] and [Verbeek, 2006]). In other words, product failures only have a meaning when they occur and are perceived by a user during user-product interaction in a product usage context. It should be noted that the extraneous variables shown in this model include both the failure characteristics and use conditions as shown in Figure 2.3 in Section 2.2.1. Due to time constraints this Ph.D. project mainly focused on the investigation of the relation between certain user characteristics and the occurrence of user-product interaction problems and user-perceived failures.

Research has shown that because of the growing diversity of user groups the most common differentiation of users and users groups on demographic and market segmentation profiles is no longer sufficient for product design evaluation [Kujala, 2006] [Shih, 2004]. Consequently, as discussed by Dillon et al. [Dillon, 1996], this research uses a differentiation of users on a deeper level of user characteristics which is taken from the research field of consumer behavior. Such characteristics are more universal across user groups and could therefore enable a more reliable prediction of user perception of product failures. Although also on this level many different potential differentiators can be found [Kujala, 2006], an explicit choice was made for this research project to mainly focus on consumer knowledge. In this context consumer knowledge is a concept which refers to *both the number of product related experiences which are stored by a user and subsequently the ability to perform product-related tasks successfully* [Alba, 1987]. This differentiation is interesting for analyzing user-perceived failures because research shows that a differentiation on consumer knowledge accounts for differences in product usage and cognitive structure and analysis [Alba, 1987] [Brucks, 1985] [Shih, 2004].

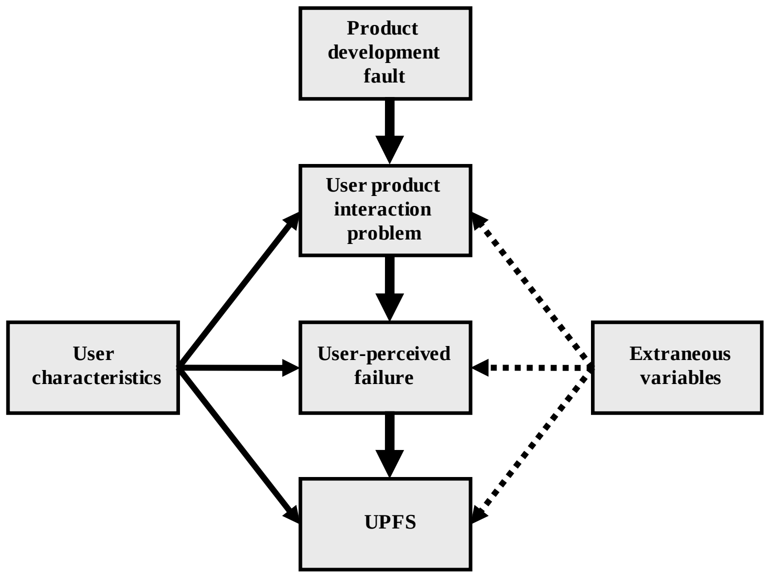


Figure 2.5 *Detailed research model for research focus area 2*

2.4.2 Research methodology focus area 2

To investigate the effect of consumer knowledge on the propagation of product development faults to user-perceived failures, two experiments are conducted:

- A laboratory experiment to investigate the effect of consumer knowledge on the occurrence of user-product interaction problems.
- A web-based experiment to investigate the effect of consumer knowledge on FA (which is a manifestation of a user-perceived failure).

For both experiments a between-subject design approach is used. In the laboratory experiment the effect of consumer knowledge on the occurrence of user-product interaction problems is investigated by asking users to perform three different tasks with varying levels of complexity on an innovative LCD television. Subsequently, usability measurements, usage patterns and user-perceived failures were recorded. In the web-based experiment, the effect of consumer knowledge on FA is investigated by confronting users with simulated product failures of an innovative LCD television in a video-based scenario. Subsequently, several measurements of FA and control variables were recorded. It is important to note that, similar to the research performed in focus area 1, all the product-related tasks and failure scenarios were carefully selected and designed based on input and reviews from industry experts. More information on the design and use of failure scenarios for evaluating user perception of product failures and the design and use of failure attribution measurements can be found in [Keijzers, 2009-a] and [Keijzers, 2009-b].

Since the goal of this research focus area is to account for diversity in different levels of consumer knowledge on complex consumer electronics, for both experiments a very diverse group of participants was selected (e.g. differences in age, educational background, experience with using technology etc.) (see also [Keijzers, 2009-c]). In the laboratory experiment, 29 participants took part in the experiment. Subsequently, in the web-based experiment, out of the 407 completed questionnaires 363 were usable for further analysis[2].

2.4.3 Research findings focus area 2

The main findings of the laboratory experiment used for analyzing the effect of consumer knowledge on the occurrence of user-product interaction problems are (based on a multivariate analysis of variance):

- Overall, for two out of the three tasks, a main effect of consumer knowledge on effectiveness and efficiency (i.e. time, number of steps and number of returns to a higher level of the menu for each task) is observed ($p < 0.05$) [Keijzers, 2009-c]. In other words, users with a higher level of knowledge were able to complete more tasks and completed the tasks faster with fewer problems.
- In-depth analysis of the usage patterns revealed that consumers with a lower level of knowledge use more irrelevant steps, use more loops and show a larger diversity of usage patterns, which does not contribute to completing a certain task. Consequently, they encountered more as well as different user-product interaction problems than consumers with a higher level of knowledge [Keijzers, 2009-c]. These results indicate that consumer knowledge can be a relevant additional contrast factor for selecting of participants in consumer tests.

The main findings of the web-based experiment used for analyzing the effect of consumer knowledge on FA are (based on a multivariate analysis of variance):

- First of all, the results show that there is a significant effect of consumer knowledge on how users attribute the product failures shown in the failure scenarios ($p < 0.001$). Although users with a higher level of knowledge are not necessarily more correct in terms of attributing the product failure to the correct physical cause, they are more extreme in their attributional response and are less satisfied with product quality than users with a lower level of knowledge.
- Secondly, the results also confirmed a significant effect of age on FA on ($p < 0.05$), but this effect is considerably less strong than the effect of consumer knowledge and only present for perceived picture quality (i.e. consumers with a higher age are more satisfied with the picture quality than consumers with a lower age).
- Finally, the results show that users attributed the failures shown in the scenarios differently than designers did. Since most television users are not familiar with the presence and properties of software in modern TVs, they attribute product failures to causes which are in accordance with their expectations and mental model of the product [Keijzers, 2009-b]. In other words, this validates the use of an experimental approach using failure attribution to investigate how users perceive product failures and to help designers better understand and diagnose user-perceived failures and complaints from a user point of view.

The combination of both experiments helps to gain more insight into the effect of consumer knowledge on the propagation of product development faults to user-perceived failures as shown in Figure 2.5. More information on the measurement and in-depth analysis of the effect of consumer knowledge and information on follow-up experiments can be found in [Keijzers, 2009-c].

[2] 44 questionnaires were excluded from further analysis due to either missing answers or not meeting the defined review criteria such as not having watched the failure scenarios completely.

2.5 Research implications and further research

Based on the results presented in the previous sections, this section is concerned with general research contributions and recommendations for further research. This section is organized as follows. Section 2.5.1 gives an overview of the major research contributions and implications. Subsequently, recommendations for further research are given in Section 2.5.2.

2.5.1 Research contributions and implications

In the beginning of this chapter it was explained that in order to deal with the increased market uncertainty for consumer electronics, existing quality methods from literature and practice should be complemented with a user-oriented impact assessment of the identified quality problems (and the accompanying quality improvement decisions). The first step into predicting the user-oriented impact of product quality problems was the definition of the UPFS concepts. Subsequently, a high-level research model was formulated that was the starting point for two research focus areas: failure characteristics and user characteristics. The research in both areas has resulted in some overall research contributions and implications about the prediction of UPFS for consumer electronics.

The UPFS prediction model

The UPFS prediction model is introduced to gain insight into the impact of quality problems on user dissatisfaction in order to reduce the number of consumer complaints after product introduction. Literature review related to consumer behavior and emotion provided the basis for the UPFS model. In this model, user dissatisfaction (UPFS) is modeled as a function of not only product related characteristics (failure characteristics) but also of user related characteristics (use conditions and user characteristics). This model was partially validated by conducting four consumer experiments. The application of these parts of the model in the NPD process of consumer electronics can potentially reduce market uncertainty in the design decision making process.

UPFS research methodology

First of all, the results of both experiments conducted in focus area 1 indicate that it is possible to measure the level of user perceived irritation that is caused by a product failure in consumer electronics in a reliable way.

Moreover, based on the results of the experiments in both focus areas, a UPFS experimental approach has been developed. By the application of this approach, future UPFS research can be performed reliably. The building blocks of this approach are:

- The UPFS measurement approach; experimental results confirmed the reliability and validity of UPFS measurement scale as a tool to measure the dependent UPFS variable in future UPFS experiments.
- The FUI and FA measurement approaches; experimental results demonstrated the ability of the FUI and FA measurement scales to measure these variables in a reliable and valid way.
- Measurement and differentiation of users with respect to their level of knowledge of consumer electronics; experimental results demonstrated the reliability and validity of consumer knowledge measurement scales to give more insight beyond differentiation of consumers on demographics.
- The scenario-based approach to analyze user perception of product failures; experimental results demonstrate both the effectiveness and efficiency of analyzing consumer perception of product failures with the use of specifically designed failure scenarios together with industry experts. These scenarios can be used in both laboratory and web-based experiments and

complement existing consumer tests which mostly do not cover user perception of product failures.

- The control for extraneous variables; based on literature and analysis of the different experimental results, this research identified different (groups of) extraneous variables in UPFS research and approaches to control them.

Valorization

Both the UPFS prediction model and the UPFS experimental approach can add value to the product development process of companies within the consumer electronics industry. Although valorization of the research results is not the primary goal of this research, the business context provided by the Trader project required constant consideration for both the business and practical perspective in this research. In the near future, the implementation of parts of this research into consumer electronics practice is therefore considered to be attainable.

2.5.2 Recommendations for further research

The previous section summarized the main results of this user perception research. Notwithstanding the important contributions of this work, the research results also trigger the formulation of new research directions. Therefore, in the following, recommendations for future research are discussed.

Further validation of the UPFS model variables

The validation of the complete UPFS model requires a gradual approach in which the (combined) added values of all failure and user characteristics to the model are evaluated. This research has investigated the influence of two of these failure characteristics (function importance and failure attribution) on UPFS. Furthermore, this research investigated the influence of user characteristics (i.e. consumer knowledge and age) on different stages of the propagation of product development faults to (potential) consumer complaints. However, in future research the influence of other user and failure characteristics on UPFS should be investigated together with the interaction effects between these user and failure characteristics. This validated UPFS model can then be used to predict the UPFS for different user groups that is caused by specific product failures. In other words, product design decisions can then be taken by predicting its influence on the UPFS levels combined for all target customer groups.

Formalization of the UPFS experimental protocol

Based on the results of these consumer experiments a UPFS research approach has been developed. This research approach consists of validated tools to measure UPFS, function importance, failure attribution, relevant user characteristics and extraneous variables. Currently, the formalization of this research approach is improved by generating a ready-for-use experimental protocol. Such a protocol should consist of a practical experimental handbook complemented with the required variable measurement tools and (straightforward) variable analysis tools.

This ongoing further formalization of the UPFS approach into a full experimental protocol makes it easier to implement in actual product development processes. Moreover, this ready-to-use protocol should lower the threshold for product designers to evaluate the influence of certain design decisions on the occurrence of user-perceived failures and subsequently on the user dissatisfaction level. Eventually, the application of this protocol in different consumer electronics companies should result in a large knowledge base on the impact of different failure- and user characteristics on UPFS.

Implementation of the UPFS model in the NPD process

The above mentioned results illustrate that market uncertainty can be potentially reduced with the application of the UPFS prediction model in the NPD process of consumer electronics products.

However, the actual implementation of the UPFS model and the accompanying experimental approach into the NPD process of consumer electronics requires some additional research.

The first version of this UPFS prediction model is mainly theory based. The previous section suggested the improvement of the practical applicability of the model and the experimental approach: the UPFS experimental protocol. But the actual implementation of the UPFS model in the NPD process requires several additional research steps. Therefore, in future UPFS research, the following questions should be answered:

- In what phase of the NPD process of consumer electronics products is the application of UPFS model most useful?
- Which people involved in the NPD process of consumer electronics products should adopt the UPFS model in their daily practice?
- Who should be hold responsible for the incorporation of the user perspective into the development process?
- How can the UPFS knowledge base be made accessible for all these involved people?

These theoretical and organizational issues should be dealt with in future UPFS research.

2.6 References

[Alba, 1987] J.W. Alba, J.W. Hutchinson. *Dimensions of Consumer Expertise*. Journal of Consumer Research, 13(4), pp. 411-454, 1987

[Bagozzi, 1999] R.P. Bagozzi, M. Gopinath, U.N. Prashanth. *The Role of Emotions in Marketing*. Journal of the Academy of Marketing Science, 27, pp. 204-206, 1999

[Bearden, 1983] W.O. Bearden, J.E. Teel. *Selected Determinants of Consumer Satisfaction and Complaint Reports*. Journal of Marketing Research, 20, pp. 21-28, 1983

[Brombacher, 2005] A.C. Brombacher, P.C. Sander, P.J.M. Sonnemans, J.L. Rouvroye. *Managing product reliability in business processes 'under pressure'*. Reliability Engineering and System Safety, Vol. 88, Issue 2, pp. 137-146, 2005

[Brucks, 1985] M. Brucks. *The Effects of Product Class Knowledge on Information Search Behavior*. The Journal of Consumer Research, 12(1), pp. 1-16, 1985

[Chaudhuri, 2006] A. Chaudhuri. *Emotion and Reason in Consumer Behavior*, Elsevier (Amsterdam), 2006

[De Visser, 2008] I.M. de Visser. *Analyzing User Perceived Failure Severity in Consumer Electronics Products – Incorporating the User Perspective into the Development Process*. Doctoral dissertation, Eindhoven University of Technology, the Netherlands, 2008

[Den Ouden, 2006] E. den Ouden. *Development of a Design Analysis Model for Consumer Complaints*. Doctoral dissertation, Eindhoven University of Technology, the Netherlands, 2006

[Dillon, 1996] A. Dillon, C. Watson. *User Analysis in HCI – the Historical Lessons from Individual Differences Research*. International Journal of Human-Computer Studies, 45, pp. 619-637, 1996

[Dolen, 2001] W. van Dolen, L. Lemmink, J. Matsson, I. Rhoen. *Affective Consumer Responses in Service Encounters: The Emotional Content in Narratives of Critical Incidents*. Journal of Economic Psychology, 3, pp. 359-376, 2001

[Fournier, 1999] S. Fournier, D.G. Mick. *Rediscovering Satisfaction*. Journal of Marketing, *63*, pp. 5-23, 1999

[Frijda, 1986] N.H. Frijda. *The Emotions*. Cambridge University Press (Cambridge), 1986

[Gilly, 1982] M.C. Gilly, B.D. Gelb. *Post-Purchase Consumer Processes and the Complaining Consumer*. The Journal of Consumer Research, 9, pp. 323-328, 1982

[Goodwin, 2005] C.J. Goodwin. *Research in Psychology – Methods and Design.* John Wiley and Sons Inc., 2005

[Kanis, 2005] H. Kanis, M.J. Rooden. *Observation as Design Tool.* Delft University of Technology, Faculty of Industrial Design Engineering, Section Applied Ergonomics and Design, 2005

[Keijzers, 2009-a] J. Keijzers, L. Scholten, Y. Lu, E. den Ouden. *Scenario-Based Evaluation of Perception of Picture Quality Failures in LCD Televisions*. Proceedings of the 19th CIRP Design Conference, pp. 497-503, Cranfield University Press (Cranfield), 2009

[Keijzers, 2009-b] J. Keijzers, E. den Ouden, Y. Lu. *Understanding Consumer Perception of Technological Product Failures: An Attributional Approach*. Proceedings of the 27th Conference Extended Abstracts on Human Factors in Computing Systems, pp. 4057-4062, ACM Press (New York), 2009

[Keijzers, 2009-c] J. Keijzers. *Do You Really Know Your Consumers – Analyzing the Effect of Consumer Knowledge on Product Use and Failure Evaluation of Consumer Electronics*. Under preparation, Eindhoven University of Technology, the Netherlands, 2009

[Kujala, 2006] S. Kujala, M. Kauppinen. *Identifying and Selecting Users for User-Centered Design*. Proceedings of the Third Nordic Conference on Human-Computer Interaction, pp. 297-303, 2006

[Manrai, 1991] L.A. Manrai, M.P. Gardner. *The Influence of Affects on Attributions for Product Failure*. Advances in Consumer Research, 18, pp. 249-254, 1991

[Mulcahy, 1998] L. Mulcahy, J.Q. Tritter. *Pathways, Pyramids, and Icebergs? Mapping the Linking Between Dissatisfaction and Complaints*. Sociology of Health and Illness, 20, pp. 825-847, 1998

[Oliver, 1996] R.L. Oliver. *Satisfaction: A behavioral Perspective on the Consumer*. Irwin McGraw-Hill (Boston), 1996

[Shih, 2004] C-F. Shih, A. Venkatesh. *Beyond adoption: Development and Application of a Use-Diffusion Model*. Journal of Marketing, 68, pp. 59-72, 2004

[Singh, 1991] J. Singh, S. Pandya. *Exploring the Effects of Consumers' Dissatisfaction Level on Complaint Behaviours*. European Journal of Marketing, 25, pp. 7-21, 1991

[Stangor, 1998] C. Stangor. *Research Methods for the Behavioral Sciences*. Houghton Mifflin Company (Boston), 1998

[Verbeek, 2006] P.-P. Verbeek, A. Slob. *Analyzing the Relations Between Technologies and User Behavior: Towards a Conceptual Framework*. P.-P. Verbeek & A. Slob (Eds.), User Behavior and Technology Development: Shaping Relations Between Consumers and Technologies, pp. 385–399, Springer (Dordrecht), 2006

[Webb, 2005] K. Webb. *Consumer Behaviour*. McGraw-Hill (Australia), 2005

[Westbrook, 1991] R.A. Westbrook, R.L. Oliver. *The Dimensionality of Consumption Emotion Patterns and Consumer Satisfaction*. Journal of Consumer Research, 18, pp. 84-91, 1991

Chapter 3

End-to-end reliability of Teletext

Authors: Piërre van de Laar, Teun Hendriks, Koen van Langen, Mathijs Opdam

Abstract: Teletext was developed in the UK in the early 1970's to communicate textual information from TV broadcasters to their viewers, for display on demand. As a low-cost technology, with the information carried inside a TV signal, it has seen tremendous adoption throughout Europe and Asia. The vast majority of TV sets sold in these areas by various suppliers can display Teletext information by virtue of a built-in Teletext decoder.

In this chapter, we examine the Teletext standard, and the impact on end-to-end reliability of a major technology evolution halfway through the life of this standard, namely the introduction of local storage memories in the TV sets. This evolution changed the Teletext information access paradigm from 'view when transmitted', to 'view cached information'. We review a specific aspect of the Teletext standard which became more relevant due to this paradigm shift, and compare a number of local robustness strategies to safeguard the end-to-end reliability as perceived by the viewer.

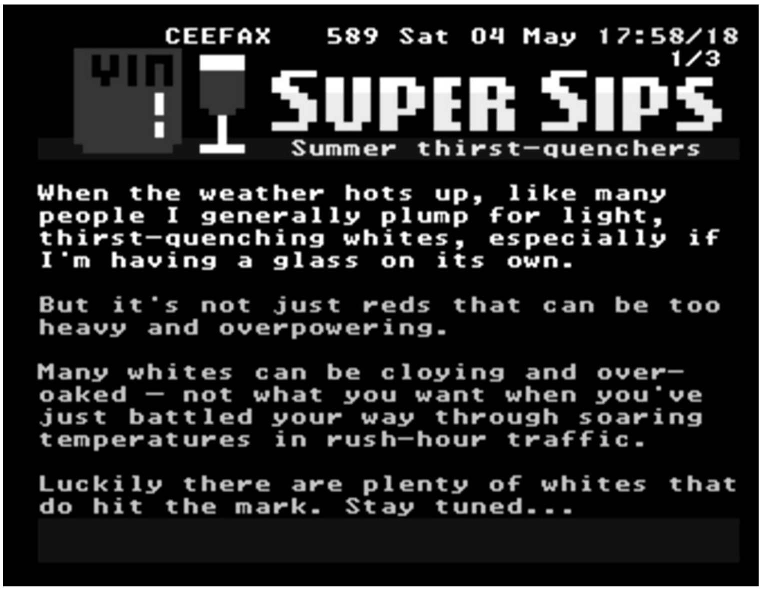
3.1 Introduction

The radio and television broadcast industry is one of the longest running example of a system of systems. Millions of television sets of various set makers can receive TV signals from numerous broadcasters serving almost all countries throughout the world. The Teletext standard [BBC, 1976] [ETSI, 2003] was an interesting development: its definition meant a significant step in the information flexibility of connecting systems. Whereas the analog TV broadcast format standard definition (be it PAL, SECAM, or NTSC) is purely a matter of a fixed sequence of image lines on a display, the Teletext standard defines a structure in which information is partitioned in content magazines, pages and sub-pages, i.e. a logical data model. In Teletext, broadcasters have considerable freedom to choose which pages to transmit, in which order, and with which repetition rate.

This chapter presents a brief overview of the Teletext standard. Taking the perspective of a TV viewer, we report on our experimental verification of error occurrence in real-life Teletext broadcasts and their impact on various, commercially available TV sets. Based on the observed failures in end-to-end communication, we compare different TV-local robustness strategies and analyze them to show their impact on the end-to-end reliability as perceived by the TV user. This chapter ends with conclusions.

3.2 Teletext

Teletext was developed in the UK in the early 1970’s to broadcast any sort of information to its audience. It has seen tremendous adoption throughout Europe and Asia. In those parts of the world, a large majority of TV broadcasters provide a Teletext service: e.g., news, weather, sports, and TV guide information. Currently all but the utmost low-end televisions sold in Europe and Asia can display Teletext pages. Teletext has proven to be a reliable text news service during events such as the September 11, 2001 terrorist attacks, during which the web pages of major news sites became inaccessible due to unexpected demand [Wikipedia, 2008].



CEEFAX 589 Sat 04 May 17:58/18
1/3

SUPER SIPS

Summer thirst-quenchers

When the weather hots up, like many people I generally plump for light, thirst-quenching whites, especially if I'm having a glass on its own.

But it's not just reds that can be too heavy and overpowering.

Many whites can be cloying and over-oaked - not what you want when you've just battled your way through soaring temperatures in rush-hour traffic.

Luckily there are plenty of whites that do hit the mark. Stay tuned...

Figure 3.1 *A Teletext page according to the original standard (a.k.a. level 1)*

The original Teletext standard (a.k.a. level 1) [BBC, 1976] supported Teletext pages consisting of 24 rows of 40 characters, chosen from a limited set of characters, in a fixed color palette. Figure 3.1 shows such a Teletext page. Since then, Teletext has evolved considerably. Later versions of the standard, amongst others, extended the character repertoire (level 1.5), increased the color palette, provided side panels for additional text and graphics (level 2.5), and introduced different font styles and proportional spacing (level 3.5) [ETSI, 2003].

3.2.1 The Teletext conceptual information model

Magazines and pages

Teletext partitions information into eight magazines. Examples of magazines are TV Guide, news, sporting news, economic news, and weather forecasts. A magazine contains a number of pages. For example, a sporting news magazine typically contains pages for basketball, soccer, baseball, and hockey. A Teletext page is identified by a number consisting out of three digits of which the most significant digit specifies the magazine number. A page number is typically fixed to a subtopic within a magazine. For example, page 201 typically gives the TV schedule of today, and page 888 provides subtitles.

Pages and sub-pages

When the information of a subtopic doesn't fit on a single screen, the information is distributed over a number of sub-pages. A page with sub-pages is also called a rotating page. The viewer is typically notified by the broadcaster of the available information by using the notation *x*/*y*, meaning x^{th} sub-page of *y* sub-pages in total. And, the television often presents a sub-page navigation bar when multiple sub-pages are available. The amount of available information on a news item is generally not fixed but may change over time. For example, the traffic information report may contain no reports at all during the night. At the onset of the morning rush hour the first reports start to fill a first page, to grow to multiple sub-page as the morning rush hour reaches its peak, finally to reduce again to a single page with a few or even no reports during midday. A traffic information report page can thus change from a single page to a page with sub-pages, and again back to a single page.

3.2.2 Teletext performance

Pages in a magazine are broadcasted after one another. The order of the pages is not specified by the Teletext standard, but left open to the broadcaster. Since the broadcaster is aware of the content, performance can be optimized [Ammar, 1987-a] [Ammar, 1987-b] [Vaidya, 1999]. For this reason, the Teletext standard provides a number of features to support a broadcaster in achieving the desired performance [ETSI, 2003]. A broadcaster must organize its transmission such that TV's can provide an optimal performance to their viewers given their capabilities, e.g. with and without local storage memory. One aspect of the performance is the access time of the different Teletext pages. To control the access time, both objective and subjective, Teletext provides, amongst others, the following concepts:

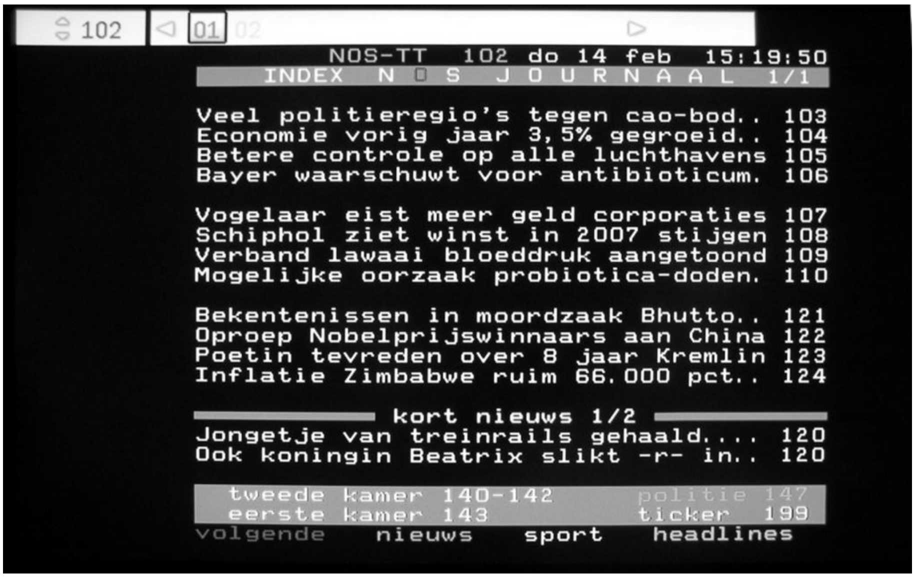
- Rolling headers inform the users about the pages being received, and positively influence the subjective access time. Rolling headers require a sequence in the numbers of the broadcasted pages.
- Out of sequence broadcasts enable that popular pages, like the start page, can be broadcasted more frequently than the other pages without interfering with the rolling headers. Out of sequence broadcasts reduce the objective access time for the popular pages.

Other aspects of the performance include the time available to read the information and robustness for bad reception. These aspects are, for example, addressed by the Teletext standard, by allowing a sub-page of a rotating page to be repeated. In other words, three sub-pages can be broadcasted as either 1,2,3, or 1,1,2,2,3,3, or 1,1,1,2,2,2,3,3,3, etc.

3.2.3 Teletext navigation and presentation

The amount of information that can fit on one Teletext screen is limited. When the amount of information cannot be presented on one screen, some form of navigation is necessary. On the Internet, we see two approaches. On the one hand, web browsers provide sliders for large web pages that enable the users to change the part of the information that is visible. On the other hand, most website builders divide the information of their website in sufficiently small parts, which do nicely fit onto a single screen, while providing additional navigational pages and links.

The lay-out of a Teletext page is fixed. The first line contains the header information: typically the name of the broadcaster, the current page number, the date, and the clock-time. Lines 2 till 24 contain plain text. When present, line 25 contains four differently colored, human-readable keywords representing hyperlinks. The hyperlinks can be followed by pressing the associated colored button on the remote control. Like hyperlinks on the internet, Teletext does not guarantee that the target exists, and that the content is reflected by the keyword used.



```
102  ◁ 01 02                              ▷
          NOS-TT   102 do 14 feb   15:19:50
   INDEX  N O S   J O U R N A A L   1/1

Veel politieregio's tegen cao-bod.. 103
Economie vorig jaar 3,5% gegroeid.. 104
Betere controle op alle luchthavens 105
Bayer waarschuwt voor antibioticum. 106

Vogelaar eist meer geld corporaties 107
Schiphol ziet winst in 2007 stijgen 108
Verband lawaai bloeddruk aangetoond 109
Mogelijke oorzaak probiotica-doden. 110

Bekentenissen in moordzaak Bhutto.. 121
Oproep Nobelprijswinnaars aan China 122
Poetin tevreden over 8 jaar Kremlin 123
Inflatie Zimbabwe ruim 66.000 pct.. 124

──────────── kort nieuws 1/2 ────────────
Jongetje van treinrails gehaald.... 120
Ook koningin Beatrix slikt -r- in.. 120

  tweede kamer 140-142       politie 147
  eerste kamer 143           ticker  199
volgende    nieuws    sport    headlines
```

Figure 3.2 *Screen showing a combination of information*

The information presented on one screen typically is a mixture of different kinds of information, see Figure 3.2. A part of the screen is used for navigational purposes. Another part of the screen might be reserved for commercial advertisements. A third part of the screen contains the information the user is interested in. Note that this part is sometimes even built up out of different parts.

3.2.4 A major technology evolution: impact on Teletext decoders

Since the introduction of Teletext in the early 1970's, the price of memory has considerably dropped. This has enabled a new kind of Teletext decoders that stores broadcasted pages. These so-called page-collecting decoders were already envisioned early on [Chew, 1977] [Tanton, 1979], and provide the Teletext viewer a requested page directly from memory instead of having to wait for the page to be transmitted. This improvement in access time to Teletext pages has a negative side effect: the possibility that obsolete information is presented to the Teletext viewer. Hence, such decoders must maintain consistency between broadcasted and stored information, i.e. a decoder needs to implement a page obsolescence policy. Note that the change in the information access paradigm from 'view when transmitted' to 'view cached information' made the dynamics of Teletext content more relevant. Whereas originally the Teletext receiver had to keep only the Teletext page being watched up-to-date, now all Teletext pages in the cache must be kept up-to-date.

In the context of a standards process, typically one observes that the possibilities of such a technology change are exploited as a competitive product advantage first. Companies do not wait for an evolution of the standard. Net consequence often is that such a change is never standardized anymore. After various products with proprietary technology have hit the market, each exhibiting a slightly different behavior, a common standards extension often is not feasible anymore.

3.3 Experimental study of Teletext decoders

This section reports on a study we did to chart the variety of interpretations of the standard by broadcasters and TV set manufacturers. Our focus is the variety of realizations of Teletext pages local storage in TV sets, and their information obsolescence handling policies. Here we report our findings concerning the handling of Teletext pages with and without subpages. The purpose of this study was to evaluate the differences in handling policies, to study broadcaster practices (including potential standards interpretation issues), and to get the requirements for a robust local strategy that has best empirical quality with respect to information availability and obsolescence.

3.3.1 Experimental setup

For this experimental study, we used a number of televisions from various brands. On the one hand, we looked for inconsistencies caused by different interpretations of the Teletext standard. On the other hand, we looked for failures in keeping the broadcasted and stored information identical. For the latter case, we looked at the Teletext pages available to the TV viewer when the television was tuned to the same channel for several hours. This amount of time is sufficiently large for the broadcasted information to have undergone changes. And this amount of time is still realistic in a user context, since watching Teletext after a movie (including commercial breaks) would give similar experiences. We compared the Teletext pages available to the TV viewer with the pages present in the actual broadcast. To determine the currently broadcasted pages, we either used a TV without memory to store Teletext pages, or we erased the stored Teletext pages from the television’s memory either by zapping to another channel and back; or by turning the TV off and on again (power cycling).

We took pictures of the screens of those TVs when we observed a problem. Note that we highlighted areas of interest in those pictures using a simple drawing tool.

3.3.2 States of a Teletext page

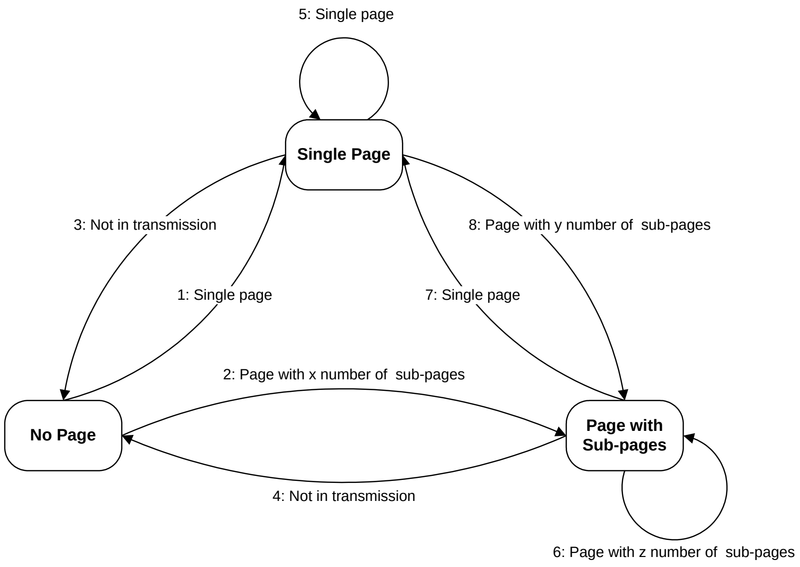


Figure 3.3 *Transitions and states of a Teletext page.*

According to the Teletext standard [ETSI, 2003], a Teletext page can be in one of the three following states:

1. Page not in transmission
2. Page with no sub-pages associated (coded 00)
3. Page with sub-pages (coded 01 till maximally 79)

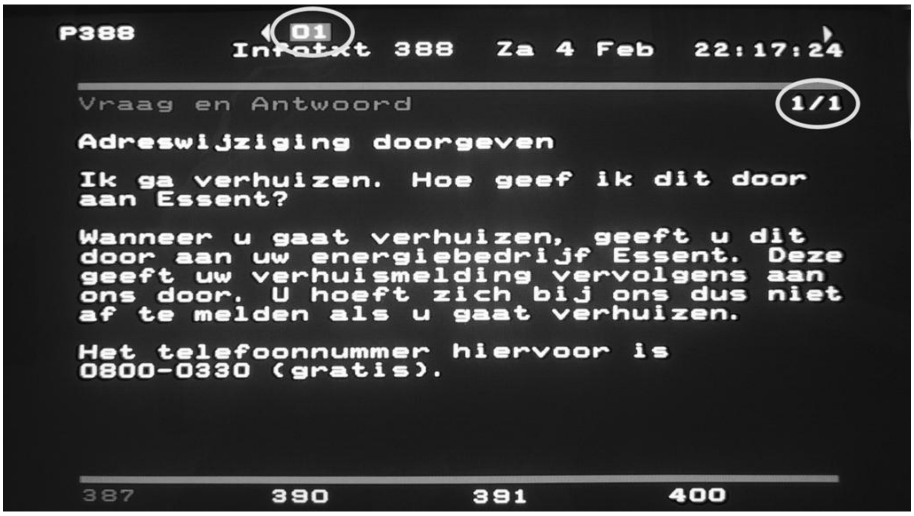
In Figure 3.3, we have captured the dynamics of a Teletext page using these three states. Note that neither such a picture, nor the explanation of dynamics, is contained in any of the Teletext standard documents. Clearly in terms of ease of comprehension of this concept, the standard is poor on this aspect.

As already described earlier, the viewer expects sub-page navigation only in the case of a page with sub-pages, and especially, on a page-collecting decoder.

3.3.3 Difference between pages with and without sub-pages

Observations

We observed in our tests that on most TVs sub-page navigation was present on some pages with no sub-pages associated. See Figure 3.4. The sub-page navigation bar (topmost line is this figure) is under the control of the TV and indicates presence of subpages. The page content as transmitted by the broadcaster signals a single page through the indication '1/1'. We also observed pages with sub-pages whose code started at 00 instead of 01. See Figure 3.5.


P388 01
Infotxt 388 Za 4 Feb 22:17:24

Vraag en Antwoord 1/1

Adreswijziging doorgeven

Ik ga verhuizen. Hoe geef ik dit door aan Essent?

Wanneer u gaat verhuizen, geeft u dit door aan uw energiebedrijf Essent. Deze geeft uw verhuismelding vervolgens aan ons door. U hoeft zich bij ons dus niet af te melden als u gaat verhuizen.

Het telefoonnummer hiervoor is 0800-0330 (gratis).

387 390 391 400

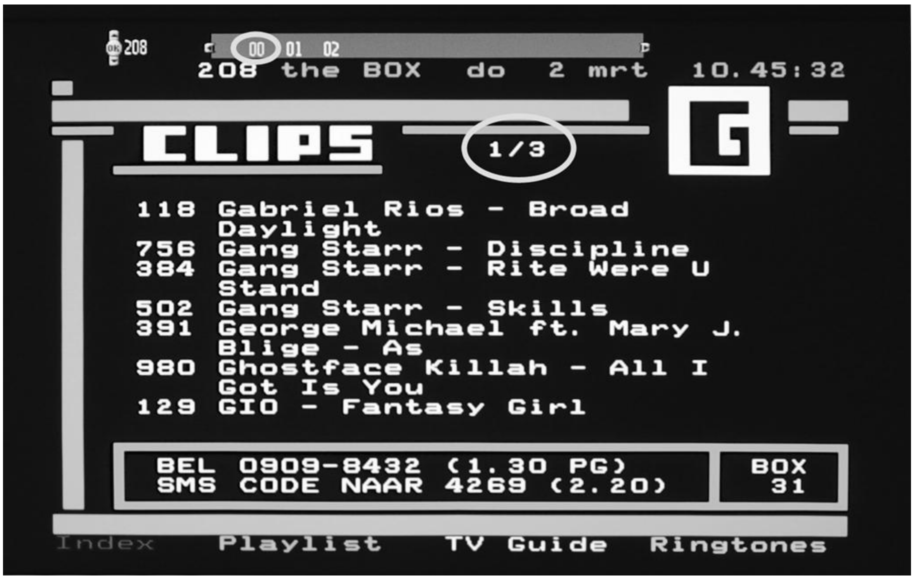
Figure 3.4 *Broadcaster which uses 01 for pages with no sub-pages. This results in a useless sub-page navigation bar.*

Explanation

According to the Teletext standard [ETSI, 2003], a Teletext page can be associated either with no sub-pages (coded 00) or with sub-pages (coded 01 till maximally 79). The syntactic difference [Lindland, 1994] between pages with and without sub-pages is thus small: it only depends on the actual value of the sub-code. We hypothesize that this small syntactic difference is the cause of the violations of the Teletext standard as shown in Figure 3.4 and Figure 3.5. The violations of the standard as depicted in Figure 3.4 and Figure 3.5 are understandable, since multiple models of counting exist. For example, for many computer scientists counting always starts at zero [Kaldewaij, 2003], not only for array indices but also for chapters, exercises, and proofs.

Proposed solution

The Teletext standard would have benefited from a more explicit distinction between pages with and without sub-pages. One possibility is the inclusion of the number of sub-pages in the meta-data of a Teletext page in the standard. This would not really have increased the bandwidth usage, since currently this information is typically encoded in the Teletext page content, e.g. as the last part of 1/1 and 1/3 in Figure 3.4 and Figure 3.5, respectively.



```
208        00  01  02
      208 the BOX  do  2 mrt  10.45:32

 CLIPS            1/3              G

     118 Gabriel Rios - Broad
         Daylight
     756 Gang Starr - Discipline
     384 Gang Starr - Rite Were U
         Stand
     502 Gang Starr - Skills
     391 George Michael ft. Mary J.
         Blige - As
     980 Ghostface Killah - All I
         Got Is You
     129 GIO - Fantasy Girl

   BEL 0909-8432 (1.30 PG)       BOX
   SMS CODE NAAR 4269 (2.20)      31

 Index    Playlist   TV Guide  Ringtones
```

Figure 3.5 *Broadcaster which uses 00 for page with sub-pages.*

3.3.4 Transitions of a page

Observations

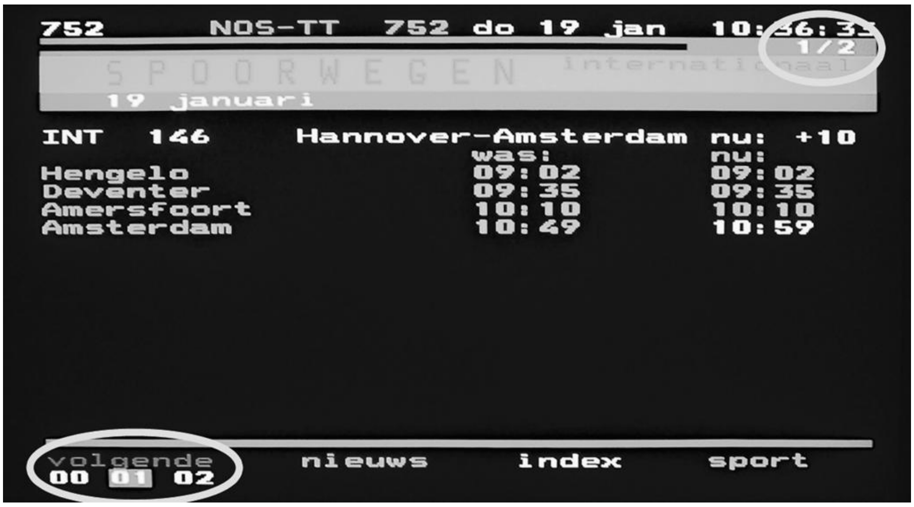
We observed that obsolete and current information was mixed after particular kinds of transitions. More precisely, after a transition from a single page to a page with sub-pages and vice versa; and after a transition of a page with sub-pages to a page with fewer sub-pages, the consistency between cached and broadcasted information is lost.

Figure 3.6 and Figure 3.7 show the results of an incorrectly handled transition from a single page to a page with sub-pages (or vice versa). Figure 3.8 and Figure 3.9 show the results of an incorrectly handled transition when the number of sub-pages of a page decreases.

Explanation

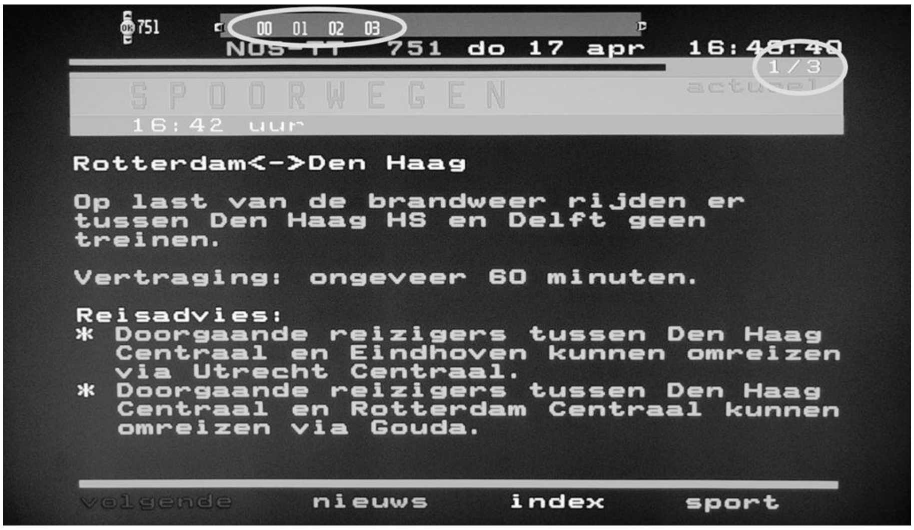
Both Figure 3.6 and Figure 3.7 are instances of the previously described traffic information report that grows over time. Initially, the traffic information report fits on a single page, and thus is coded with 00. When the report grows beyond a single page it changes into a page with sub-pages which are encoded with 01 and 02. However, the television did not remove the obsolete page with sub-code 00 from the cache on the transition to a page with sub-pages.

Both Figure 3.8 and Figure 3.9 contain the traffic information at the end of rush hour. At the end of rush hour, the number of sub-pages needed to report the traffic jams decreases. The televisions however did not maintain consistency of broadcasted and stored information, since besides the currently broadcasted sub-pages also earlier broadcasted sub-pages with higher sub-codes are still available.



```
752      NOS-TT   752 do 19 jan   10:36:35
                                      1/2
  SPOORWEGEN             internationaal
  19 januari
INT  146     Hannover-Amsterdam  nu:  +10
                   was:          nu:
Hengelo            09:02         09:02
Deventer           09:35         09:35
Amersfoort         10:10         10:10
Amsterdam          10:49         10:59

volgende
00 01 02     nieuws     index     sport
```

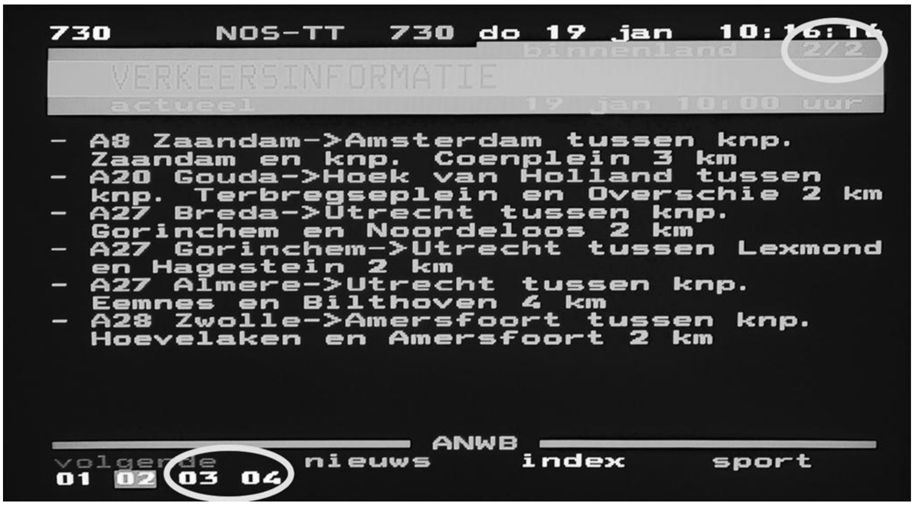
Figure 3.6 Samsung *television that has combined a page with sub-pages (codes 01 and 02) and without sub-pages (code 00) broadcasted after each other.*



```
751        00 01 02 03
  NOS-TT   751 do 17 apr   16:48:40
                              1/3
  SPOORWEGEN              actueel
  16:42 uur

Rotterdam<->Den Haag

Op last van de brandweer rijden er
tussen Den Haag HS en Delft geen
treinen.

Vertraging: ongeveer 60 minuten.

Reisadvies:
* Doorgaande reizigers tussen Den Haag
  Centraal en Eindhoven kunnen omreizen
  via Utrecht Centraal.
* Doorgaande reizigers tussen Den Haag
  Centraal en Rotterdam Centraal kunnen
  omreizen via Gouda.

volgende     nieuws     index     sport
```

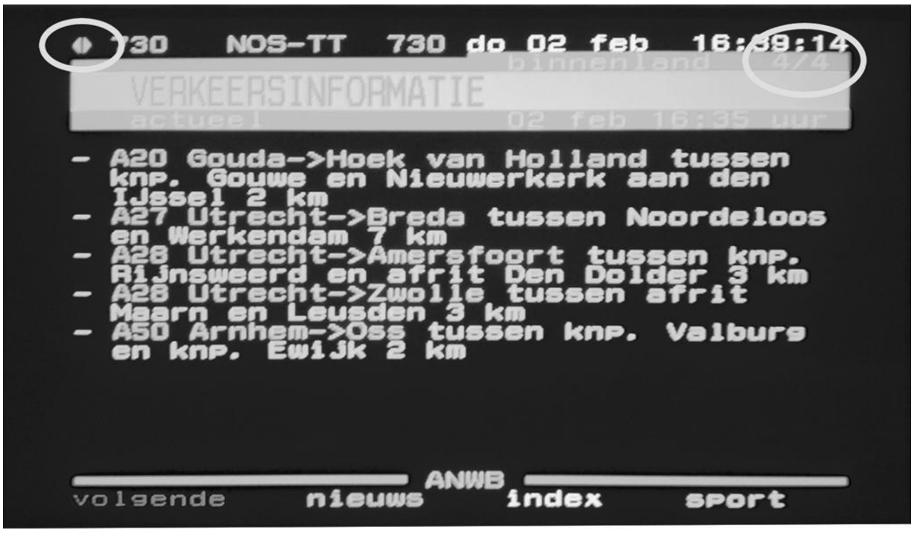
Figure 3.7 Philips *television that has combined a page with sub-pages (codes 01, 02, 03) and one without sub-pages (code 00) broadcasted after each other.*

Despite the fact that the original standard [BBC, 1976] contains a control bit to signal that a page should be not be confused with an earlier broadcasted page, we observed on a number of brands that changes in the content of a Teletext page results into inconsistencies between the currently broadcasted and stored information. We hypothesize that either the control bit is lost, it is ignored, or not applied to the page as a whole but only the currently associated sub-code. To clarify the latter case, take the example of the traffic information report, when sub-page 01 is broadcasted for the first time, its control bit flags a change, but the page collecting decoder might only remove the sub-page with code 01 from the cache instead of all codes associated with that page.



```
730        NOS-TT   730 do 19 jan   10:16:1[illegible]
                              binnenland       2/2
     VERKEERSINFORMATIE
     actueel                  19 jan 10:00 uur

-  A8 Zaandam->Amsterdam tussen knp.
   Zaandam en knp. Coenplein 3 km
-  A20 Gouda->Hoek van Holland tussen
   knp. Terbregseplein en Overschie 2 km
-  A27 Breda->Utrecht tussen knp.
   Gorinchem en Noordeloos 2 km
-  A27 Gorinchem->Utrecht tussen Lexmond
   en Hagestein 2 km
-  A27 Almere->Utrecht tussen knp.
   Eemnes en Bilthoven 4 km
-  A28 Zwolle->Amersfoort tussen knp.
   Hoevelaken en Amersfoort 2 km

                       ANWB
volgende       nieuws      index      sport
01 02 03 04
```

***Figure 3.8 Samsung** television displaying sub-page 2 out of 2, whereas storing out-of-date sub-pages 03 and 04.*



```
◂▸ 730   NOS-TT   730 do 02 feb   16:39:14
                         binnenland      4/4
    VERKEERSINFORMATIE
    actueel                02 feb 16:35 uur

-  A20 Gouda->Hoek van Holland tussen
   knp. Gouwe en Nieuwerkerk aan den
   IJssel 2 km
-  A27 Utrecht->Breda tussen Noordeloos
   en Werkendam 7 km
-  A28 Utrecht->Amersfoort tussen knp.
   Rijnsweerd en afrit Den Dolder 3 km
-  A28 Utrecht->Zwolle tussen afrit
   Maarn en Leusden 3 km
-  A50 Arnhem->Oss tussen knp. Valburg
   en knp. Ewijk 2 km

                      ANWB
volgende       nieuws     index      sport
```

***Figure 3.9 Sony** television displaying sub-page 4 out of 4, whereas storing out-of-date sub-pages with higher sub-pages numbers (indicated at top left of page by the two arrows signaling the possibility to navigate to sub-pages with lower and higher numbers).*

Proposed solution

The original standard [BBC, 1976] did not introduce the concepts of states. It only introduced the concept of transition: the erase[1] page control bit signals that the currently transmitted page "is significantly different from that in the previous transmission [...], such that the two should not be confused". The transmission of an instantaneous event over a noisy communication channel without reception acknowledgements is error-prone. Losing this single event means losing the request to erase a page from memory forever. For robustness and consistency, it would have been better if the current state was encoded into the Teletext page[2], and whenever a new state is observed the obsolete page is erased from memory. Note that due to standard violations, as shown in Figure 3.5, state change detections based on Figure 3.3 might give undesirable results for these non-adhering broadcasters. In this particular example, the out-of-specification broadcast of a rotating page triggers two state changes in every cycle: A state change from page with sub-pages to a single page on reception of 00, and a state change from a single page to a page with sub-pages on reception of 01.

The usage of time-stamps is another alternative to erase obsolete pages from memory. Although obsolete pages are eventually erased from memory, they are not instantly erased when new pages arrive.

3.4 Designing a local strategy to offset the standards gap

Given the current Teletext standard and the observed violations of the standard, we wondered what kind of algorithm for storing Teletext pages would handle best the Teletext standard as-is in combination with observed broadcasting practices. To understand the scope of this issue, we asked experts of Philips Consumer Electronics and NXP to characterize Teletext transmissions through the following questions:

1. What are the probabilities of broadcasting single pages and pages with a given number of sub-pages?
2. What are the probabilities of a Teletext broadcast to be within and out of specification?
3. What is the probability of sub-pages to be repeated?
4. What are the probabilities of a perfect transmission and transmissions with a given loss rate?
5. What are the probabilities of a Teletext page to be in a steady state and in a transition between states? Furthermore we asked the experts of Philips Consumer Electronics and NXP to give a value judgment on the relative importance of the obsolescence handling policy:
6. How important is the prevention of obsolete (sub-) pages compared to the unavailability of (sub-) pages?

The characterization and value judgment as given by the experts was used to compare the following algorithms for Teletext page obsolescence handling:

[1] The goal of the erase page control bit is to prevent confusing the content of two successively broadcasted pages. Although the name erase page bit suggests that the Teletext conceptual information model assumes the content to be news, i.e., old content must be erased when new content becomes available, its goal is valid for both news and series content.

[2] We think that a few bits are sufficient to encode the state of a Teletext page, since pages are typically repeated a large number of times before being changed.

- No sub-page aging. All information is stored to ensure that any sub-page is always available. Obsolete sub-pages are just accepted; no removal is attempted at all. Viewers always are able to view stored sub-pages even though their content may be severely outdated.
- Last sub-page only. Only the last broadcasted sub-page is stored to ensure that no obsolete pages are in the Teletext store. The unavailability of all other sub-pages than the last broadcasted one is just accepted. A viewer must wait until a sub-page is broadcast to view that particular sub-page.
- Upper-bound sub-page aging. The highest sub-code of all sub-pages is remembered. After each cycle[3], sub-pages in store with a higher number than the highest sub-code observed in that cycle are erased.
- State change detection and upper-bound sub-page aging. State changes between single pages and pages with sub-pages are detected based on the value of the sub-codes (00 versus non-00). State changes between pages with different number of sub-pages are detected by the previously described upper-bound sub-page aging algorithm.
- Upper- and lower-bound sub-page aging. The lowest and highest sub-codes of all sub-pages are remembered. After each cycle[3], sub-pages in store outside the range defined by the lowest and highest sub-code observed in that cycle are erased.

We have depicted the performance of these algorithms using the average values as given by the experts on the Teletext transmission characterization and valuation questions in Figure 3.10 and Figure 3.11. The lower the score of the algorithm (value on the vertical axis), the better it performs in response to transmissions according to the characterization and valuation of the experts.

Figure 3.10 shows a sensitivity analysis of these algorithms as function of the answer on question 5. On the vertical axis, the weighted number of ‘wrong’ pages (either obsolete pages or unavailable pages) is depicted against the ratio of steady state transmissions versus transitions. This figure shows that the upper- and lower-bound sub-page aging is the preferred algorithm independent of the probabilities of a Teletext page to be in a steady state and in a transition between states (see also Figure 3.3). This algorithm does retain by far the minimum obsolete sub-pages for display when the transmission for a page is mostly transitional (i.e. number of sub-pages rapidly changing in the broadcast). This algorithm also keeps the most sub-pages available for view, when the transmission for the page is mostly steady-state (i.e. only infrequent changes in the number of sub-pages do occur in the broadcast).

[3] A cycle is detected when the number of sub-pages exceeds a threshold. The value of the threshold depends on the highest (and lowest) sub-code observed since the beginning of the cycle. Furthermore, the threshold contains a multiplication factor that reflects that sub-pages can be repeated or be missed due to bad reception.

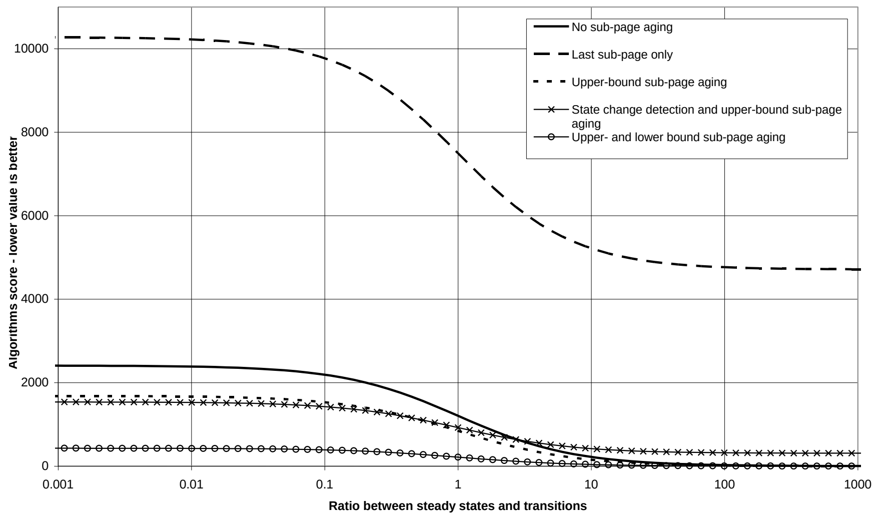


Figure 3.10 *Score of various algorithms based on the expert judgment on the likelihood of Teletext broadcasts except for the ratio between steady state and transitions.*

Figure 3.11 shows the performance as function of the answer on question 6. Again, we have depicted the performance of the previously described algorithms using the average values as given by the experts on the Teletext transmission characterization questions. The lower the score of the algorithm (value on the vertical axis), the better it performs in response to transmissions according to the characterization of the experts. In this figure, we see that 'last sub-page only' is preferable when preventing obsolete sub-pages is far more important than unavailability of sub-pages. In all other cases, upper- and lower-bound sub-page aging is the preferred algorithm.

Interesting to note is that the best performing algorithm is also the algorithm that used the least number of assumptions on the use of the standard as possible (sub-pages are broadcasted in order). This observation is in line with the claim of defensive programming: using as few assumptions as possible yields the most secure and reliable code.

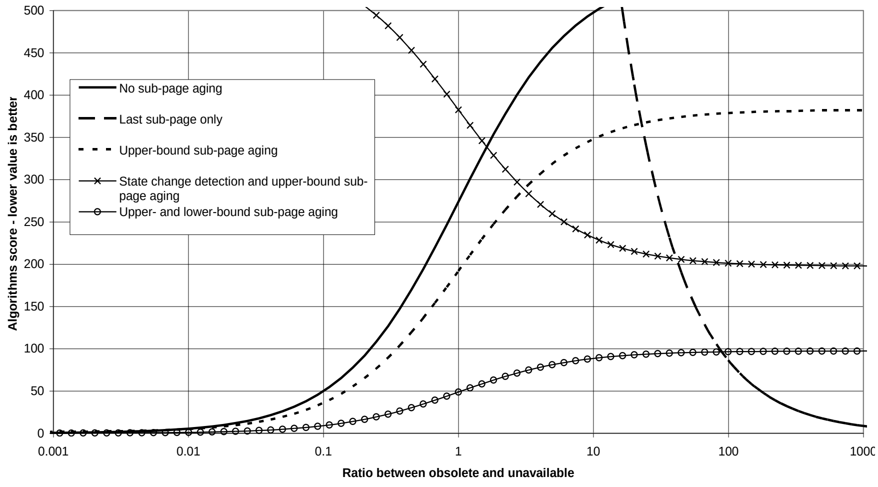


Figure 3.11 *Score of various algorithms based on the expert judgment on the likelihood of Teletext broadcasts except for the ratio between obsolete and unavailable pages.*

3.5 Conclusions

In this chapter, we have reported on a study to improve the end-to-end reliability of Teletext by building robustness strategies into Teletext enabled consumer electronics products. For this study, we have described part of the Teletext standard, and given empirical evidence of problems in terms of quality issues and gaps in the Teletext specification. These problems were exacerbated by a major technology evolution halfway through the life of the Teletext standard: the introduction of local storage memories in the TV sets. For these so-called page collecting decoders, a balanced local strategy was presented to compensate for the lack of definition of Teletext dynamic behavior.

Overall, the Teletext standard has seen tremendous success. Its operational use spans already thirty years with almost universal adoption rate for TVs sold in Europe and Asia. The standards definition was sufficiently flexible to support several evolutions. In retrospect however, the omission of specification of dynamic behavior, i.e., the change in content on a page, did cause significant problems for Teletext decoders. This study has shown that such specification gaps can be largely overcome by local robustness strategies. Thus, even in the presence of external errors, products based on the Teletext standard will appear reliable to TV viewers.

3.6 References

[Ammar, 1987-a] M.H. Ammar, J.W. Wong. *On the Optimality of Cyclic Transmission in Teletext Systems*. IEEE Transactions on Communications 1987; 35 (1): pp. 68-73

[Ammar, 1987-b] M.H. Ammar. *Response Time in a Teletext System: An Individual User's Perspective*. IEEE Transactions on Communications 1987; 35 (11): pp. 1159-1170

[BBC, 1976] *Broadcast Teletext Specification*. British Broadcasting Corporation, Independent Broadcasting Authority, and British Radio Equipment Manufacturers' Association, September 1976

[Chew, 1977] J.R. Chew. *CEEFAX: Evolution And Potential*. BBC Research Department 1977/26, August 1977

[ETSI, 2003] *Enhanced Teletext Specification*. ETSI EN 300 706 v1.2.1 (2003-04), European Telecommunication Standards Institute (ETSI), 2003

[Kaldewaij, 2003] A. Kaldewaij. *Programming: The Derivation of Algorithms*. 1990

[Lindland, 1994] O.I. Lindland, G. Sindre, A. Sølvberg. *Understanding Quality in Conceptual Modelling*. IEEE Software 1994; 11 (2): pp. 42-49

[Tanton, 1979] N.E. Tanton. *UK Teletext-Evolution and Potential*. IEEE Transactions on Consumer Electronics 1979; 25 (3): pp. 246-250

[Vaidya, 1999] N.H. Vaidya, S. Hameed. *Scheduling data broadcast in asymmetric communication environments*. Wireless Networks 1999; 5 (3): pp. 171-182

[Wikipedia, 2008] *Wikipedia page on Teletext*: http://en.wikipedia.org/wiki/Teletext, 2008

Chapter 4

Using software history to guide deployment of coding standards

Authors: Cathal Boogerd, Leon Moonen

Abstract: In spite of the widespread use of coding standards and tools enforcing their rules, there is little empirical evidence supporting the intuition that they prevent the introduction of faults in software. Therefore, we propose to use information from software and issue archives to link standard violations to known bugs. In this chapter we introduce such an approach and apply it to three industrial case studies. Furthermore, we discuss how to use the historical data to address two practical issues in using a coding standard: which rules to adhere to, and how to rank violations of those rules.

4.1 Introduction

Coding standards have become increasingly popular as a means to ensure software quality throughout the development process. They typically ensure a common style of programming, which increases maintainability, and prevent the use of potentially problematic constructs, thereby increasing reliability. The rules in such standards are usually based on expert opinion, gained by years of experience with a certain language in various contexts. Over the years various tools have become available that automate the checking of rules in a standard, helping developers to locate potentially difficult or problematic areas in the code. These include commercial offerings (e.g., QA-C[1], K7[2], CodeSonar[3]) as well as academic solutions (e.g., [Johnson, 1978] [Engler, 2000] [Flanagan, 2002]). Such tools generally come with their own sets of rules, but can often be adapted such that also custom standards can be checked automatically. In a recent investigation of bug characteristics, Li et al. argued that early automated checking has contributed to the sharp decline in memory errors present in software [Li, 2006]. However, in spite of the availability of appropriate standards and tools, there are several issues hindering adoption.

[1] www.programmingresearch.com

[2] www.klocwork.com

[3] www.grammatech.com

Automated inspection tools are notorious for producing an overload of non-conformance warnings (referred to as violations in this chapter). For instance, 30% of the lines of one of the projects used in this study contained such a violation. Violations may be by-products of the underlying static analysis, which cannot always determine whether code violates a certain check or not. Kremenek et al. observed that all tools suffer from such false positives, with rates ranging from 30-100% [Kremenek, 2004]. Furthermore, rules may not always be appropriate for all contexts, and many of their violations can be considered false positives. For instance, we find that one single rule is responsible for 83% of all violations in one of the analyzed projects, which is unlikely to only point out true problems. As a result, manual inspection of all violating locations adds a significant overhead without clear benefit.

Although coding standards can be used for a variety of reasons, such as maintainability or portability, in this chapter we will focus on their use for fault prevention. From this viewpoint, there is an even more ironic aspect to enforcing ineffective rules. Any modification of the software has a non-zero probability of introducing a new fault, and if this probability exceeds the reduction achieved by fixing the violation, the net result is an increased probability of faults in the software [Adams, 1984].

Therefore, in previous work [Boogerd, 2008-a], [Boogerd, 2009] we investigated the link between violations and known faults in two software archives using the MISRA standard [MISRA, 2004], a widely adopted industrial standard for C. We found that a small subset of the rules can be linked to known faults, but that this set differs between the two cases investigated. In this chapter, we expand the previous case studies and show how to use the same approach to address a number of practical challenges.

4.1.1 Challenges

Given the fact that interaction of code and coding standard can vary so dramatically, and that it has a great impact on the coding habits of developers, managing a coding standard for a software project is not a trivial task. Specifically, we identify two challenges:

1. Which rules to use from a standard? A standard may be widely adopted, but still contain rules that, at first sight, do not look appropriate for a certain project. In addition, when comparing several different coding standards an explicit evaluation of individual rules can help in selecting the right standard.
2. How to prioritize a list of violations? Although a standard may be carefully crafted and customized for a project, developers may still be faced with too many violations too fix given the limited time. To handle this problem most efficiently, we need to prioritize the list of violations, and define a threshold to determine which ones have to be addressed, and which ones may be skipped.

In other words, we define a rule selection criterion and a violation ranking strategy. We discuss our approach in Section 4.2, introduce our cases in Section 4.3 and discuss results of the approach in Section 4.4. We evaluate the results and describe how to meet the challenges in Section 4.5. Finally, we compare with related work in Section 4.6 and summarize our findings in Section 4.7.

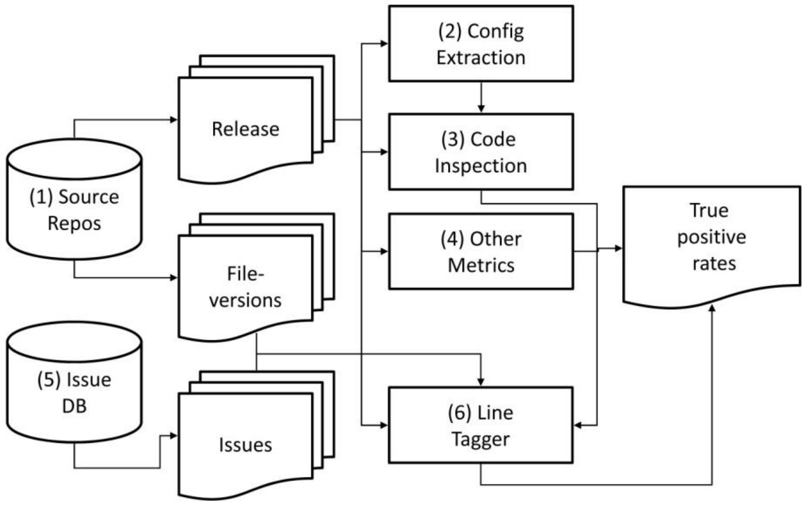


Figure 4.1 *Measurement process overview*

4.2 Approach

The approach uses a Software Configuration Management (SCM) system and its accompanying issue database to link violations to faults. Such an issue database contains entries describing observed problems or change requests, and these issues can be linked to the source code modifications made to solve it. Using this system impacts the definition of our measures: we define a *violation* to be a signal of non-conformance of a source code location to any of the rules in a coding standard; a *bug* to be an issue in the database for which corrective modifications have been made to the source code (the *fix*); those original faulty lines constitute the *fault*.

Measuring violations Figure 4.1 illustrates the steps involved in the measurement process. First we select the range of releases relevant to the study, i.e., the ones part of the selected project. We iterate over all the releases, retrieved from the source repository (1), performing a number of operations for each. From a release, we extract the configuration information (2) necessary to run the automatic code inspection (3), which in turn measures the number of violations. This configuration information includes the set of files that are to be part of the measurements, determined by the project’s build target. We record this set of files and take some additional measurements (4), including the number of lines of code in the release.

4.2.1 Matching faults and violations

Matching requires information on the precise location(s) of a fault. We start gathering this information by checking which file versions are associated with bugs present in the issue database (5). We proceed to compute a difference with previous revisions, indicating which changes were made in order to solve the issue, and marking the modified lines. When all issues have been processed, and all lines are marked accordingly, the complete file history of the project is traversed to propagate violations and determine which ones were involved in a fix. All this takes place in the Line Tagger (6), described below.

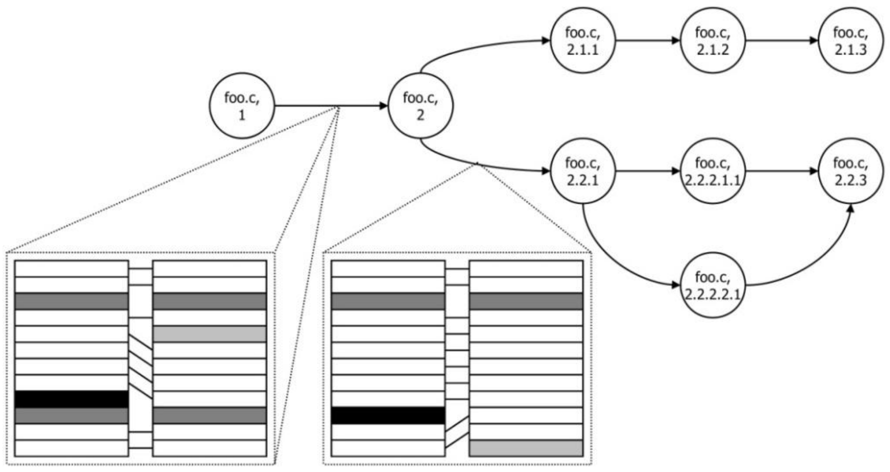


Figure 4.2 *File-version and annotation graphs*

Using the set of files recorded in (2) the Line Tagger constructs a file version graph, retrieving missing intermediate file versions where necessary. For every file-version node in the graph, we record the difference with its neighbors as annotation graphs. An example is displayed in Figure 4.2. The round nodes denote file versions, each edge in that graph contains an annotation graph, representing the difference between the two adjoining nodes. Lines in the annotation graph can be either new (light grey), deleted (black) or modified (dark grey, pair of deleted and new line).

Using the file-version graph, matching faulty and violating lines becomes straightforward. To compute the true positive ratio, we also need the total number of lines, defined as the number of unique lines over the whole project history. What we understand by unique lines is also illustrated in Figure 4.2: if a line is modified, the modified version is considered a new unique line. This makes sense, as it can be considered a new candidate for the code inspection. In addition, it means that violations on a line present in multiple versions of a file are only counted once. Our line tagging is similar to the tracking of fix lines in [Kim, 2007-b], although we track violations instead.

4.2.2 Determining significance of matchings

Dividing the number of hits for a certain rule (violations on faulty lines) by the total number of its violations results in the desired true positive rate. But it does not give us a means to assess its significance. After all, if the code inspection flags a violation on every line of the project, it would certainly result in a true positive rate greater than zero, but would not be very precise or efficient. In fact, any random predictor, marking random lines as faulty, will, with a sufficient number of attempts, end up around the ratio of faulty lines in the project. Therefore, assessing significance of the true positive rate means determining whether this rate is significantly higher than the faulty line ratio. This will give us an intuition as to whether the matchings are simply chance or not.

We model this by viewing the project as a large repository of lines, with a certain percentage p of those lines being fault-related. A rule analysis marks n lines as violating, or in other words, selects these lines from the repository. A certain number of these (r) is a successful fault-prediction. This is compared with a random predictor, which selects n lines randomly from the repository. Since the number of lines in the history is sufficiently large and the number of violations comparably small, p remains constant after selection, and we can model the random predictor as a Bernoulli process (with p = p and n trials). The number of correctly predicted lines r has a binomial distribution; using the cumulative distribution function (CDF) we can compute the significance of a true positive rate (r/n).

Because we know that only files that were actually changed contain faults, we only consider lines from these files for the analysis. This is a stricter requirement for significance than including also non-changed files.

4.2.3 Requirements

There are two important requirements for the approach that should be considered when replicating this study. The first is that of the strict definition of which files are part of the analysis as well as what build parameters are used, as both may influence the lines included in the subsequent analysis, and thus the number of faults and violations measured. The second requirement is a linked software version repository and issue database. The link may be (partially) reconstructed by looking at the content of commit log messages, or be supported by the database systems themselves, as in our case. Many studies have successfully used such a link before [Li, 2006] [Sliwerski, 2005] [Kim, 2006-b] [Weiß, 2007] [Kim, 2007-c] [Williams, 2005] [Kim, 2006-a].

4.3 Three case studies

4.3.1 Projects

For this study we selected three projects from NXP [4], denoted TVoM, TVC1 and TVC2. The first project was a part of our pilot study [Boogerd, 2008-b], the second was previously studied in [Boogerd, 2009], the third one has been added in this chapter. We shortly describe each of them below.

TVoM is a typical embedded software project, consisting of the driver software for a small SD-card-like device. When inserted into the memory slot of a phone or PDA, this device enables one to receive and play video streams broadcasted using the Digital Video Broadcast standard. The complete source tree of the latest release contains 148KLoC of C code, 93KLoC C++, and approximately 23KLoC of configuration items in Perl and shell script (all reported numbers are non-commented lines of code). The real area of interest is the C code of the actual driver, which totals approximately 91KLoC.

TVC1 and TVC2 are software components of the NXP TV platform (referred to as TVC, for TV component). Both are part of a larger archive, structured as a product line, primarily containing embedded C code. This product line has been under development for five years and most of the code to be reused in new projects is quite mature. We selected from this archive the development for two different TV platforms: TVC1 comprises 40 releases from June 2006 until April 2007; TVC2 has 50 releases between August 2007 and November 2008.

In all these projects, no coding standard or inspection tool was actively used. This allows us to actually relate rule violations to fault-fixing changes; if the developers would have conformed to the standard we are trying to assess, they would have been forced to immediately remove all these violations. The issues we selected for these projects fulfilled the following conditions: (1) classified as ‘problem’ (thus excluding feature implementation); (2) involved with C code; and (3) had status ‘concluded’ by the end date of the project.

[4] www.nxp.com

Coding standard

The standard we chose for this study is the MISRA standard, first defined by a consortium of automotive companies (The Motor Industry Software Reliability Association) in 1998. Acknowledging the widespread use of C in safety-related systems, the intent was to promote the use of a safe subset of C, given the unsafe nature of some of its constructs [Hatton, 1995]. The standard became quite popular, and was also widely adopted outside the automotive industry. In 2004 a revised version was published, attempting to prune unnecessary rules and to strengthen existing ones.

Implementation

The study was implemented using Qmore and CMSynergy. Qmore is NXPs own front-end to QA-C version 7, using the latter to detect MISRA rule violations. Configuration information required by Qmore (e.g., preprocessor directives, include directories) is extracted from the configuration files (Makefiles driving the daily build) that also reside in the source tree. For the measurements, all C and header files that were part of the daily build were taken into account. The SCM system in use at NXP is CMSynergy, featuring a built-in issue database. All modifications in the SCM are grouped using tasks, which in turn are linked to issues. This mechanism enables precise extraction of the source lines involved in a fix.

4.4 Results

Three tables list the results for each of the three cases: TVoM in Table 4.1, TVC1 in Table 4.2, and TVC2 in Table 4.3. In these tables we display detailed results for all the significant rules, that is, the ones that outperformed a random predictor with significance level $\alpha = 0.05$. Also, we include three categories of aggregated results: all rules, non-significant rules, and significant rules. For each of these, the second and third columns hold the total number of violations and the number of those that could be linked to fixes (i.e., true positives). The fourth and fifth hold the true positive ratio and its significance. The last column displays the number of issues for which at least one involved line contained a violation of the rule in question.

4.4.1 Case comparison

Noticeable are the differences in the set of significant rules between the three cases. All cases agree on only one rule, 8.1, which states:

> *“Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.”*

A possible explanation of this is that some functionality may be implemented at first without too much attention for quality, so a function prototype is forgotten. Since it is new code, it will likely be changed a number of times before it is mature, and as a result is easily linked to a bug fix.

The agreement is (as expected) higher between the two cases from the TVC archive, which have eight rules in common: 1.1, 5.1, 8.1, 10.1, 10.2, 12.5, 14.10, and 19.5. Especially interesting is the concentration of rules in chapter 10 of the standard concerning arithmetic type conversions. For instance, rule 10.1 states that the value of an integer expression should not be implicitly converted to another underlying type (under some conditions). The reason for these rules to show up so prominently in our results is that the core code of TCV projects consists of data stream handling and manipulation, which is also the most complex (and fault-prone) part of the code. This leads to a concentration of fixes and these particular violations in that part of the code. Also in an informal discussion with an architect of the TVC project, the chapter 10 rules were identified as potentially related to known faults. In fact, the development team was making an effort to study and correct these violations for the current release.

MISRA rule	Total violations	True positives (abs)	True positives (ratio)	Significance	Issues covered
5.3	2	1	0,50	0,97	1
8.1	56	15	0.27	0.99	2
8.5	2	1	0.50	0.97	1
9.2	2	1	0.50	0.97	1
11.1	38	10	0.26	0.97	4
12.7	91	22	0.24	0.98	5
12.13	36	10	0.28	0.98	1
14.2	258	68	0.26	1.00	9
16.9	2	1	0.50	0.97	1
All rules	9898	572	0.06	n/a	25
Non-significant rules	9411	443	0.05	n/a	25
Significant rules	487	129	0.26	n/a	16

***Table 4.1** Summary of rules for TVoM*

These results suggest the importance of tailoring a coding standard to a specific domain, as the observed violation severity differs between projects. These differences are clearly smaller within a single archive, showing that it is feasible to employ this approach in longer-running projects, with regular updates to assess whether rules should be included or excluded from the set of adhered rules.

4.4.2 Limitations

This section points out some limitations of the approach, how we deal with them, and to what extent they influence the results of the approach.

Measurement correctness

Some residual faults may be present in the software at the end of development that remain invisible to the approach. However, Fenton et al. found in their case study that the number of faults found in prerelease testing was an order of magnitude greater than during 12 months of operational use [Fenton, 2000]. Furthermore, the development only ends after the product has been integrated into the clients' products, and therefore all major issues will have been removed. Also, it is possible that some violations, removed in non-fix modifications, pointed to latent faults. However, this category is typically small. For instance, in TVC1 this accounts for only 3% of the total number of violations, and is therefore unlikely to change the existing results significantly.

MISRA rule	Total violations	True positives (abs)	True positives (ratio)	Significance	Issues covered
1.1	7	2	0.29	0.99	2
3.4	22	4	0.18	0.98	1
5.1	65	25	0.38	1.00	1
8.1	35	5	0.14	0.97	3
10.1	1081	179	0.17	1.00	35
10.2	15	6	0.40	1.00	2
11.3	532	58	0.11	1.00	13
12.5	148	21	0.14	1.00	8
12.13	15	4	0.27	1.00	1
14.8	88	10	0.11	0.96	4
14.10	63	10	0.16	1.00	5
19.5	25	4	0.16	0.97	1
All rules	51529	1743	0.03	n/a	121
Non-significant rules	49433	1415	0.03	n/a	118
Significant rules	2096	328	0.16	n/a	52

Table 4.2 *Summary of rules for TVC1*

Finally, the matching between violations and faults may be an underestimation. Some fault-fixes only introduce new code, such as the addition of a previously forgotten check on parameter values. Overestimation is less likely, although not all lines that are part of a fix may be directly related to the fault (for instance, moving a piece of code). Even so, violations on such lines still point out the area in which the fault occurred. In addition, by computing significance rates we eliminated rules with coincidental matchings.

Generalizing results

Since results differ significantly between projects, it is difficult to generalize them. They are consistent in the sense that there is a small subset of rules performing well, while no relation can be found for the other (non-significant) rules.

However, the rules themselves differ. There are a number of important factors that play a role in this difference. In discussing this, we must separate the cases by archive rather than project.

MISRA rule	**Total violations**	**True positives (abs)**	**True positives (ratio)**	**Significance**	**Issues covered**
1.1	116	8	0.07	0.96	8
1.2	73	6	0.08	0.97	7
5.1	169	11	0.07	0.96	4
5.2	27	3	0.11	0.98	2
6.1	5	1	0.20	0.99	1
6.2	8	3	0.38	1.00	2
8.1	160	15	0.09	1.00	8
9.2	15	8	0.53	1.00	1
10.1	4006	204	0.05	1.00	44
10.2	40	10	0.25	1.00	5
10.6	190	13	0.07	0.98	7
12.1	2698	210	0.08	1.00	45
12.4	91	9	0.10	1.00	8
12.5	584	116	0.20	1.00	28
12.6	302	27	0.09	1.00	9
12.7	1653	108	0.07	1.00	19
13.1	4	2	0.50	1.00	1
13.2	1256	119	0.09	1.00	34
13.3	265	43	0.16	1.00	11
14.6	4	1	0.25	0.99	2
14.10	135	22	0.16	1.00	14
15.2	4	2	0.50	1.00	1
16.10	1620	133	0.08	1.00	30
17.4	935	72	0.08	1.00	8
19.5	27	4	0.15	1.00	2
20.10	4	1	0.25	0.99	1
All rules	77158	3143	0.04	n/a	96
Non-significant rules	62767	1992	0.03	n/a	86
Significant rules	14391	1151	0.08	n/a	77

Table 4.3 *Summary of rules for TVC2*

Since results differ significantly between projects, it is difficult to generalize them. They are consistent in the sense that there is a small subset of rules performing well, while no relation can be found for the other (non-significant) rules. However, the rules themselves differ. There are a number of important factors that play a role in this difference. In discussing this, we must first separate the cases by archive rather than project, since these factors differ more between archives than within archives.

One difference between the TVoM archive and the TVC archive is that the former is a single, new project, whereas the latter is an archive containing five years of development. To counter influences of maturity of code, we only analyzed the new and edited code in TVC. There are two further major factors that were not under our control: (1) the application domain; and (2) the development team. TVC contains a significant number of domain-specific algorithms and procedures, affecting the type of code written and violations introduced, and requiring specialized developers.

When comparing projects within the TVC archive, these two factors also play a role. Although the team is mostly the same, developers join and leave over the course of the two projects, that together span almost 2.5 years. The type of TV component developed is the same in both projects, but the hardware platform for which the software is written is different, also impacting the type of violations introduced.

Finally, note that the set of rules analyzed for these cases is always a subset of all the rules in the MISRA standard (typically 50-60%), as in none of the cases violations were found for all rules. However, the analyzed rules cover almost all of the topics (i.e., chapters) of the standard. Only rules from chapters 4 (two rules on character sets) and 7 (one rule on constants) were not present.

4.5 Discussion

In this section, we will discuss how to use the results of our approach to meet the two challenges as identified in Section 4.1: rule selection and ranking.

4.5.1 Rule selection

There are three criteria involved in rule selection: the true positive rate, its significance, and the number of issues covered. The first tells us how likely a violation is to point to a faulty location, the second indicates whether this number is due to chance, and the third expresses the effectiveness of the rule. In this chapter, we have used the significance as a cutoff measure to eliminate rules that have a relatively high true positive rate simply because they occur so often. Using only the significant rules reduces the number of violations to inspect by 95% in TVoM, while still covering 64% of the issues covered by all the rules. For TVC1 and TVC2 reduction is 96% and 81%, with 43% and 80% of the total issues covered, respectively. The number of rules can be further reduced by setting a threshold for the true positive rate until the desired balance between number of violations and issues covered has been reached. For instance, setting a threshold of 0.10 for the true positive rate in TVC2 induces a reduction of 98% with a coverage of 41%. This explicit tradeoff between number of violations and number of covered issues makes the approach especially useful when introducing a coding standard in a legacy project with many violations.

4.5.2 Rule ranking

While rule selection is performed when choosing a coding standard, rule ranking is the problem of presenting the most relevant violations to a developer at compile time. From a fault prevention point of view, the most logical ranking criterion is the true positive rate. However, there is the added problem of the number of violations to inspect. Even with a customized standard, an inspection run may result in too many violations to inspect in a single sitting. In this case, one may define a maximum number of violations to present, but it is also possible to set a maximum on the number of *issues* to inspect. Using the true positive ratio attached to each violation we can compute the expected number of issues covered in a ranking. A contrived example would be finding two violations of rule 13.1 (TP = 0.50) and four of 20.10 (TP = 0.25) in TVC2: in this case we expect to find two issues (2 * 0.5 + 4 * 0.25). Since we may expect solving real issues to require significantly more time than inspecting violations, limiting inspection effort based on this number can be useful.

4.6 Related work

In recent years, many approaches have been proposed that benefit from the combination of data present in SCM systems and issue databases. Applications range from an examination of bug characteristics [Li, 2006], techniques for automatic identification of bug-introducing changes [Sliwerski, 2005] [Kim, 2006-b], bug-solving effort estimation [Weiß, 2007], prioritizing software inspection warnings [Kim, 2007-a] [Kim, 2007-b], prediction of fault-prone locations in the source [Kim, 2007-c], and identification of project-specific bug-patterns, to be used in static bug detection tools [Williams, 2005] [Kim, 2006-a].

Software inspection (or defect detection) tools have also been studied widely. Rutar et al. studied the correlation and overlap between warnings generated by the ESC/Java, FindBugs, JLint, and PMD static analysis tools [Rutar, 2004]. They did not evaluate individual warnings nor did they try to relate them to actual faults. Zitser et al. evaluated several open source static analyzers with respect to their ability to find known exploitable buffer overflows in open source code [Zitser, 2004]. Engler et al. evaluate the warnings of their defect detection technique in [Engler, 2001]. Heckman et al. proposed a benchmark and procedures for the evaluation of software inspection prioritization and classification techniques [Heckman, 2008]. Unfortunately, the benchmark is focused at Java programs.

Wagner et al. compared results of defect detection tools with those of code reviews and software testing [Wagner, 2005]. Their main finding was that bug detection tools mostly find different types of defects than testing, but find a subset of the types found by code reviews. Warning types detected by a tool are analyzed more thoroughly than in code reviews. Li et al. analyze and classify fault characteristics in two large, representative open-source projects based on the data in their SCM systems [Li, 2006]. Rather than using software inspection results they interpret log messages in the SCM.

More similar to the work presented in this paper is the study of Basalaj [Basalaj, 2006]. While our study focuses on a sequence of releases from a single project, Basalaj takes an alternative viewpoint and studies single versions from 18 different projects.

These are used to compute two rankings, one based on warnings generated by QA C++, and one based on known fault data. For 12 warning types, a positive rank correlation between the two can be observed (reportedly, nearly 900 warning types were involved in the study). Wagner et al. evaluated two Java defect detection tools on two different software projects [Wagner, 2008]. Similar to our study, they investigated whether inspection tools were able to detect defects occurring in the field. Their study could not confirm this possibility for their two projects. Apart from these two studies, we are not aware of any other work that reports on measured relations between coding rules and actual faults. There is little work published that evaluates the validity of defects identified by automated software inspection tools, especially for commercial tools. One reason is that some license agreements explicitly forbid such evaluations, another may be the high costs associated with those tools.

The idea of a safer subset of a language, the precept on which the MISRA coding standard is based, was promoted by Hatton [Hatton, 1995]. In [Hatton, 2004] he assesses a number of coding standards, introducing the signal to noise ratio for coding standards, based on the difference between measured violation rates and known average fault rates. He assessed MISRA C 2004 in [Hatton, 2007], arguing that the update was no real improvement over the original standard, and "both versions of the MISRA C standard are too noisy to be of any real use". This study complements these assessments by providing new empirical data and by investigating opportunities for selecting an effective non-noisy subset of the standard.

4.7 Conclusions

In this chapter, we have discussed an approach that uses historical software data to customize a coding standard. The results from our three case studies indicate that rule performance in fault prevention differs significantly between projects, stressing that such customization is not a luxury but a must. After all, adhering to rules that are not related to faults may increase, rather than decrease, the probability of faults in the software.

We return to the challenges as stated in the introduction and summarize how our approach can assist in meeting them:

- *Which rules to use from a standard?* In a customization step, rules can be selected based on the measured true positive rate and the number of issues covered by each rule. The true positive rate expresses the likelihood of pointing to actual faults, and includes a mechanism to exclude accidental matches.
- *How to prioritize a list of violations?* Violations can be ranked using the measured true positive rate. In addition, this measure can be used to compute the expected number of issues covered, which is useful in defining a threshold in the number of violations that need to be inspected.

Of course, fault prevention is only one of the reasons for choosing a coding standard, so also other criteria (such as maintainability and portability) should be considered in rule selection and ranking. Nevertheless, the historical approach allows us to quantify the fault prevention performance of rules and makes the tradeoff of rule adherence with regard to this aspect explicit.

Acknowledgements: The authors wish to thank the people of NXP for their support in this investigation.

4.8 References

[Adams, 1984] E.N. Adams. *Optimizing Preventive Service of Software Products*. IBM J. of Research and Development, 28(1): pp. 2-14, 1984

[Basalaj, 2006] W. Basalaj. *Correlation between coding standards compliance and software quality.* In White paper, Programming Research Ltd., 2006

[Boogerd, 2008-a] C. Boogerd and L. Moonen. *Assessing the Value of Coding Standards: An Empirical Study*. Proc. 24th IEEE Int. Conf. on Software Maintenance, pp. 277-286. IEEE, 2008

[Boogerd, 2008-b] C. Boogerd and L. Moonen. *Assessing the Value of Coding Standards: An Empirical Study*. Technical Report TUD-SERG-2008-017, Delft University of Technology, 2008

[Boogerd, 2009] C. Boogerd and L. Moonen. *Evaluating the relation between coding standard violations and faults within and across versions*. Proceedings of the Sixth IEEE Working Conference on Mining Software Repositories (MSR). IEEE. To Appear

[Engler, 2000] D. Engler, B. Chelf, A. Chou, and S. Hallem. *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*. Proc. 4th Symp. on Operating Systems Design and Implementation, pp. 1-16, 2000

[Engler, 2001] D. Engler, D.Y. Chen, S. Hallem, A. Chou, and B. Chelf. *Bugs as deviant behavior: a general approach to inferring errors in systems code*. Symp. on Operating Systems Principles, pp. 57-72, 2001

[Fenton, 2000] N.E. Fenton and N. Ohlsson. *Quantitative Analysis of Faults and Failures in a Complex Software System*. IEEE Trans. Softw. Eng., 26(8). (2000)

[Flanagan, 2002] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. *Extended static checking for java*. Proc. ACM Conf. on Programming Language Design and Implementation (PLDI), pp. 234-245. ACM, 2002

[Hatton, 1995] L. Hatton. *Safer C: Developing Software in High-integrity and Safety-critical Systems*. McGraw-Hill, New York, 1995

[Hatton, 2004] L. Hatton. *Safer language subsets: an overview and a case history, MISRA C.* Information & Software Technology, 46(7): pp. 465-472, 2004

[Hatton, 2007] L. Hatton L. *Language subsetting in an industrial context: A comparison of MISRA C 1998 and MISRA C 2004*. Information & Software Technology, 49(5): pp. 475-482, 2007

[Heckman, 2008] S. Heckman and L. Williams. *On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques*. ESEM ’08: Proc. 2nd ACM-IEEE Int. Symp. on Empirical Software Eng. and Measurement, pp. 41-50. ACM., 2008

[Johnson, 1978] S.C. Johnson. *Lint, a C program checker*. Unix Programmer’s Manual, volume 2A, chapter 15, pp. 292-303. Bell Laboratories, 1978

[Kim, 2006-a] S. Kim, T. Zimmermann, K. Pan, and E.J. Whitehead Jr. *Automatic Identification of Bug- Introducing Changes*. Proc. 21st IEEE/ACM Int. Conf. on Automated Software Eng. (ASE), pp. 81-90. IEEE, 2006

[Kim, 2006-b] S. Kim, K. Pan, and E.J. Whitehead Jr. *Memories of bug fixes.* Proc. 14th ACM SIGSOFT Int. Symp. on Foundations of Software Eng. (FSE), pp. 35-45. ACM, 2006

[Kim, 2007-a] S. Kim and M.D. Ernst. *Prioritizing Warning Categories by Analyzing Software History*. Proc. 4th Int. Workshop on Mining Software Repositories (MSR), page 27. IEEE, 2007

[Kim, 2007-b] S. Kim and M.D. Ernst. *Which warnings should I fix first?* Proc. 6th joint meeting of the European Software Eng. Conf. and the ACM SIGSOFT Int. Symp. on Foundations of Software Eng., pp. 45-54. ACM, 2007

[Kim, 2007-c] S. Kim, T. Zimmermann, E.J. Whitehead Jr., and A. Zeller. *Predicting Faults from Cached History*. Proc. 29th Int. Conf. on Software Eng. (ICSE), pp. 489-498. IEEE, 2007

[Kremenek, 2004] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler. *Correlation Exploitation in Error Ranking*. Proc. 12th ACM SIGSOFT Int. Symp. on Foundations of Software Eng. (FSE), pp. 83-93. ACM, 2004

[Li, 2006] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. *Have things changed now?: an empirical study of bug characteristics in modern open source software.* Proc. 1st Workshop on Architectural and System Support for Improving Software Dependability (ASID), pp. 25-33. ACM, 2006

[MISRA, 2004] MISRA (2004). *Guidelines for the Use of the C Language in Critical Systems*. MIRA Ltd., http://www.misra.org.uk/ , ISBN 0-9524156-2-3

[Rutar, 2004] N. Rutar and C.B. Almazan. *A comparison of bug finding tools for java.* ISSRE'04: Proc. 15th Int. Symp. on Software Reliability Engineering, pp. 245-256. IEEE, 2004

[Sliwerski, 2005] J. Sliwerski, T. Zimmermann, and A. Zeller. *When do changes induce fixes?* Proc. Int. Workshop on Mining Software Repositories (MSR). ACM, 2005

[Wagner, 208] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb. *An evaluation of two bug pattern tools for java*. 1st Int. Conf. on Software Testing, Verification, and Validation, pp. 248-257. IEEE, 2008

[Wagner, 2005] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger. *Comparing bug finding tools with reviews and tests*. Proc. 17th Int. Conf. on Testing of Communicating Systems (Test-Com'05), volume 3502 of LNCS, pp. 40-55. Springer, 2005

[Weiß, 2007] C. Weiß, R. Premraj, T. Zimmermann, and A. Zeller. *How Long Will It Take to Fix This Bug?* Proc. 4th Int. Workshop on Mining Software Repositories (MSR), page 1. IEEE, 2007

[Williams, 2005] C.C. Williams and J.K. Hollingsworth. *Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques*. IEEE Trans. Software Eng., 31(6): pp.466-480, 2005

[Zitser, 2004] M. Zitser, R. Lippmann, and T. Leek. *Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code.* Proc. 12th ACM SIGSOFT Int. Symp. on Foundations of Software Eng., pp. 97-106. ACM, 2004

Chapter 5

Stress testing and observing its effects using real-time monitoring

Authors: Iulian Nitescu, Manvi Agarwal, André Nieuwland, Rob Golsteijn, Roland Mathijssen

Abstract: This chapter describes the design and usage of SW-based stress testing techniques for dependability and design evaluation and the added value of visualization of the usage of resources. The concepts and ideas of stress testing a system beyond its normal limits are quite common in the mechanical and hardware field but are less common in software development. Here, methods applied in hardware and mechanics are transferred to, and applied in, the field of software testing. Applying stress on software and observing its behavior helps in showing potential weak points. Also stress testing indicates where resource usage is close to critical values. This is especially important for systems not designed for worst-case inputs, such as many consumer products. Moreover, adding artificial resource usage through stressing can help in early integration. By creating more realistic behavior of software components which are still under construction, the final system load is more realistically approximated. Being able to monitor in real-time the usage of, possibly stressed, resources can also help in finding the cause of potential problems.

5.1 Introduction

Modern systems use shared resources to maximize the performance while minimizing the overall system costs. In this context we define shared resources to be any type of resource that can be used concurrently by different tasks or devices. Typical shared resources are: input and output peripherals, CPU cycles, memory, buses and other OS or system resources (such as the file system handlers, semaphores, etc). What happens though if these shared resources are not available at the right moment in the right quantity? Is the system robust against these types of overload situations? Due to cost constraints consumer products can use only a limited set of shared resources. This often implies that true worst-case situations –as far as they can be defined– cannot be properly handled. In these cases the system is specified as “best effort”, i.e. “do not crash” or “reduced quality”.

Another aspect is if one can add features that use these shared resources without affecting the existing functionality? Especially when designing a component or subsystem that is further applied by an OEM customer, one needs to be able to reliably specify how many resources are still available for customer applications. At the same time one must be able to show that the own software is still behaving correctly or as specified in the presence of the added software.

In the next sections we first look at the concept of dependability (Section 5.1.1). Next software stress testing is related to traditional stress testing methods (Section 5.1.2). In the next sections of this chapter we will look at a TV architecture of a TV52x system (Section 5.2) that is used for implementing the CPU stressor (Section 5.3) and the Memory stressor (Section 5.4). The TV architecture of the TV52x is used as example throughout this chapter. Section 5.5 discusses an example of using the stressor. Monitoring in real-time is further discussed in Section 5.6.

5.1.1 Dependability evaluation

For embedded software applications, especially consumer products, the user has an expectation that the system is reliable or robust (see also Chapter 2 on User Perception). Robustness can be defined as “capable of performing without failure under a wide range of conditions <robust software>” [Merriam-Webster, 2009-a]. Depending on the application and the user perception, it is allowable for the system to behave in a degraded fashion (e.g. slowing down, dropping commands or missing input data) while under stress. The degradation must be graceful however and, once the stress is gone, the system must be recoverable (recover automatically). Of course, missed input cannot be recovered and even if it could, often it should not be recovered to prevent unexpected behavior afterwards. A lack of robustness would lead to unacceptable behavior both from the specified functionality and user perception perspective (e.g. missing deadlines and unexpectedly resetting the processor creating loss of data or displaying distorted data).

The embedded software should also demonstrate elasticity, defined as the “capability of a strained body to recover its size and shape after deformation; the quality of being adaptable” [Merriam-Webster, 2009-b]. In a software context this means that, after the stress has been relieved, the system should leave the degraded mode and return to its normal operating state, with full functionality restored (without side effects).

The combination of these two properties means that the system should be able to “bend” gracefully while under stress and return to normal after the stress is removed. Testing a system in order to validate its robustness (can it bend) and elasticity (can it recover) has the potential to expose design flaws and implementation defects which are difficult or even impossible to discover using traditional testing approaches. As with humans, stress brings out the best or the worst in software systems. Problems which are dormant in a normal operation mode become visible during or immediately after a stressed operation mode. In such cases, it may take hours or even days or weeks before the fault becomes a failure (waiting for a “natural” occurrence of the stressed operation mode). The stress test magnifies these inherent problems allowing them to be detected earlier and more easily, to be root caused, and to be corrected before the software is deployed. Another big issue in testing is the reproducibility of failures; when the fault is related to lack of resources in the system, stress testing can help in making a problem reproducible in a much shorter time than waiting for the system to reach that critical state.

Figure 5.1 and Figure 5.2 illustrate how stress testing may speed up finding faults. The first figure shows the (simulated) CPU usage, where only after about 183 hours (more than one week) a high load is observed during which a fault can become an observed failure. Increasing CPU usage can bring this flaw to the surface much earlier. Here, stress testing has its analogy in accelerated lifetime testing where e.g. the environmental temperature or the supply voltage are increased to observe potential system flaws earlier.

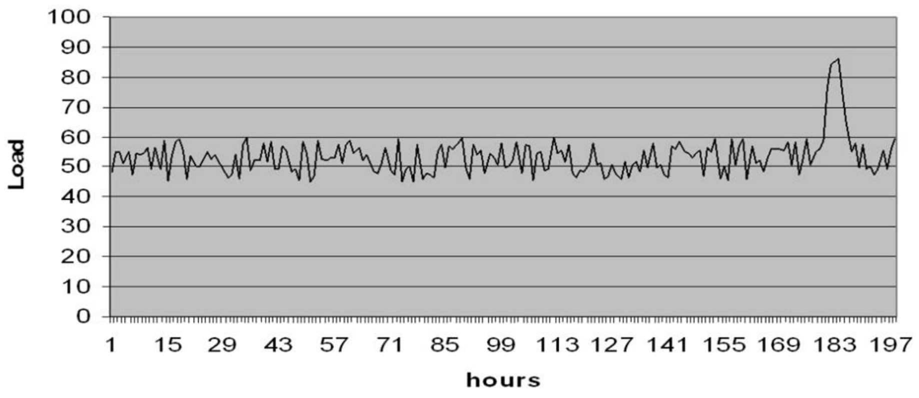


Figure 5.1 *Simulated CPU usage. Resource problem after 183 hours*

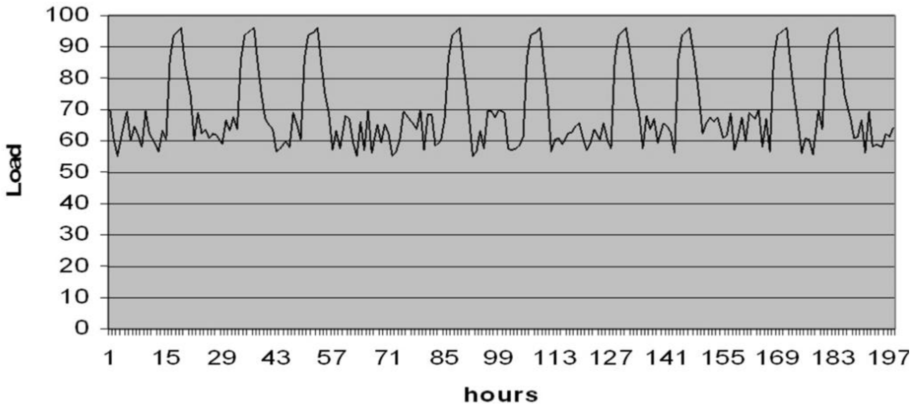


Figure 5.2 *Simulated CPU usage with stressor. Resource problem occurs after only 15 hours and is reproducible roughly every 15 hours subsequently*

5.1.2 Load testing / stress testing vs. traditional testing methods

Traditional software testing is a standard phase in nearly every software development methodology. During this phase test engineers develop and execute test cases that are meant to validate the software requirements as defined during earlier stages of the development process. These test cases have specific initial conditions and well-defined expected results for specific inputs. Even though some test methodologies (e.g. boundary analysis) generate test cases that exercise the software outside the limits defined in the requirements, the test cases do not cover the situations where the resources of the system are exercised outside the design bounds of the software [Graham, 2008].

Other forms of testing are used to investigate the behavior of the system outside the comfort zone as defined by the requirements. These forms of testing are usually deployed to probe the behavior of the system in case more resources are used; either due to extra internal or external load (see Table 5.1)

In the testing literature, the term “load testing” is usually defined as the process of exercising the system under test by feeding it with different tasks it needs to handle, but still within the design requirements. Load testing assesses how a system performs under a certain load and deals with subjecting a system to a statistically representative load. The two main reasons for using such loads are in support of software reliability testing and in performance testing. Load testing is sometimes called *volume testing*, or *longevity/endurance testing*. The load is generated internally by exposing the system to inputs close to design limits. In the case of a TV system, load testing is performed by providing input streams with different properties (such as video resolution, encoding method, etc).

Stress testing requires subjecting a system to an exceptionally high load with the intention of breaking it. The system is not expected to process the overload without adequate resources, but to behave (e.g., fail) in a decent manner (e.g., not corrupting or losing data). The software is intentionally exercised "outside the box". Known weaknesses, vulnerabilities or limitations in the software design may be specifically exploited. Degraded performance of a system under stress may be deemed perfectly acceptable, thus, the interpretation of test results and the definition of pass / fail criteria can be more subjective, though criteria such as "may not crash", "no distortion in sound" and "max. 40 milliseconds extra latency on commands" are still objective and measurable. Furthermore, a test that stresses one aspect or component of the system may lead to undesirable side effects in another area of the system (e.g. multiprocessor systems). This implies that the entire system behavior as a whole must be evaluated to properly analyze the results.

Contrary to the usual load testing in acceptance or performance tests where inputs that generate high loads are used, stress testing induces load either by exposing the system to distorted inputs or by internal methods that generated this extra load. The advantages of the internal methods over the distorted inputs are twofold. Firstly, internal load is easier to quantify and control and secondly, it can generate the extra load in more parts of the system. In this chapter we investigate systems that have to deal with extra load, regardless of the origin of this load.

Traditional Acceptance Testing	**Software Stress Testing**
Black Box Testing. Software internals are not taken into consideration	White Box Testing. Known vulnerabilities or limitations of the software are taken into consideration
Test cases are mainly designed to verify that the software meets requirements	Test cases specifically designed to "break" the software
Test cases exercise software within the limits defined by the requirements	Test cases intentionally violate constraints to stress software
Pass / Fail criteria are clearly defined	Pass / Fail criteria are more subjective

Table 5.1 *Acceptance testing vs. stress testing*

Both load and stress testing are important for validating the robustness and elasticity of the system under test. For example, the TV system has requirements about the maximum CPU cycles and the bus bandwidth that can be used by the TV platform. The work described in this chapter can be used for both purposes, because the internal load generating method is scalable: both loads within and outside the requirements can be generated.

We used two different methods to stress resources: diminish the available CPU cycles and diminish the available bus bandwidth. By being able to stress test CPU cycles and memory bandwidth one gets a good overview about how many of these resources can be safely used by OEM customers; a figure often very hard to simulate or compute upfront. Of course, the stress testing concept can be extended to stress other resources as well, such as available memory.

5.2 TV system overview

In these sections, we briefly describe the TV system used for the implementation of the stressors, firstly as a general architecture (Section 5.2.1) and then from the point of view of the two resources that are to be stressed (Section 5.2.2 and 5.2.3).

5.2.1 TV system general architecture

NXP's TV52x television platform consists of a MIPS processor, a TriMedia processor (TM) and several hardware units for data capturing and display of pixels (see Figure 5.3). All these components use a shared, off chip, memory to pass data between them.

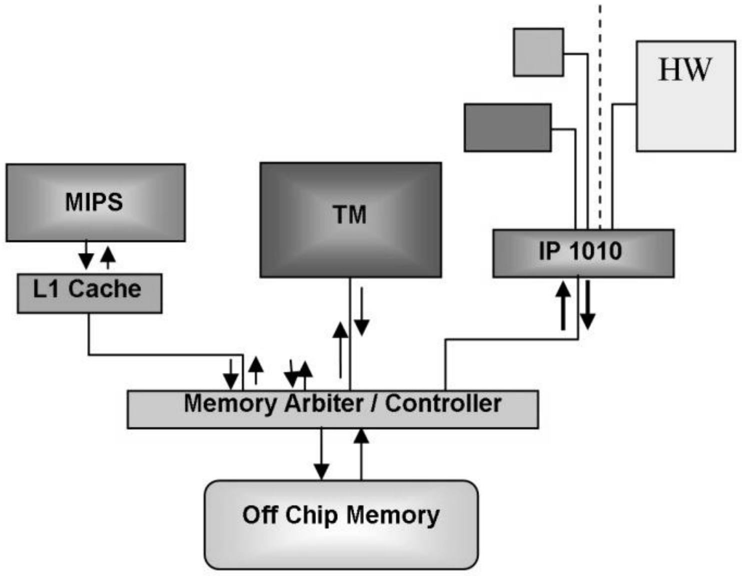


Figure 5.3 *Schematic view of the TV 52x television platform*

5.2.2 CPU architecture

The TV52x software uses the two processors in the core for different purposes:

1. The MIPS is mainly used for control processing (e.g. UI). Application SW runs on top of the Linux operating system. The processes running on the MIPS have soft real-time deadlines;
2. The TriMedia is mainly used for streaming processing such as audio/video decoding and rendering. Application SW runs on top of the pSoS real time kernel. The tasks in the TriMedia software have hard real-time deadlines, missing those results in poor audio & video quality.

The two software stacks communicate using an inter-processor communication mechanism with hard real-time deadlines and keep-alive mechanisms. Although the two software stacks are running independently, unresponsiveness on one side, e.g. due to extra load, influences the other side.

5.2.3 Memory architecture

Most of the hardware components have typically block-based direct memory access (DMA) to the off-chip memory (DRAM). Hardware devices typically fail if they have to wait too long for memory access, hence their traffic characterizes as "hard real-time" (HRT); a response is required within a certain time limit. Figure 5.3 depicts how this DMA type traffic from the different hardware units is merged behind a first level arbiter (depicted by IP 1010, an NXP proprietary arbiter). After the IP1010 this hard real-time traffic is merged with the low-latency (LL) traffic from MIPS and TriMedia by the IP2032 memory arbiter/controller (NXP proprietary as well). The LL traffic must be handled faster than the HRT traffic from the hardware devices. The task for the memory arbiter/controller is to combine all the memory traffic in such a way that the hard real-time bandwidth for the (hardware) devices is guaranteed, while the access latencies for both the TriMedia and MIPS are minimized. A detailed analysis of memory access for HRT and LL traffic is given in [Steffens, 2008].

5.3 CPU stressor

In these sections we will look at an actual stressor that is implemented in a TV system. First the concept of the stressor that eats away CPU cycles is defined in Section 5.3.1, while the next sections describe the actual implementation of the stressor and discuss several design choices related to the stressor (Section 5.3.2 and Section 5.3.3). Finally, a CPU load visualization method implemented to validate the stressor is described in Section 5.3.4.

5.3.1 CPU stressor concepts

Increasing the CPU load is an obvious way to place the software under stress. This is especially effective when stress testing pre-emptive multitasking systems. Software designers spend considerable effort assigning task priorities in a pre-emptive system to ensure that all real-time deadlines are met. Techniques such as rate-monotonic scheduling always guarantee a feasible scheduling when the CPU load is kept just under 70% of real-time tasks [Liu, 1973]. Experience, however, shows that even when tasks schedules are poorly designed, task deadlines will usually be met when the overall CPU load is less than about 88%. In other words, having 12% overall free CPU load often provides enough margin to mask flaws regarding task priority assignments [Lehoczky, 1989]. So to expose flaws by stress testing one must at least be able to set a total target CPU load of 90% or more.

5.3.2 CPU stressor implementation

To actually apply and test the concept of stressing a CPU cycle budget of a system, a so-called "cycle eater" was implemented in the software of a digital TV system. This software simulates a task using more cycles than envisaged. A cycle eater is created as a system task on MIPS or TriMedia or both (see Figure 5.3). Such a task can be started and stopped from outside the system at will. A cycle eater task runs in a periodic fashion, with a certain priority for a number of times. Apart from consuming CPU cycles, it also logs its activity (average CPU load taken, etc).

To control the task of consuming CPU cycles, a number of parameters can be set as follows:

- Processor: consume cycles on the MIPS, the TriMedia or both
- Priority of the task
- Period of the task (see below)
- Budget (see below)
- Number of runs

The task is given a period and a budget. This means that one can set how much time is consumed in a given period. E.g. one can choose to consume 5 milliseconds (budget) in a period of 10 milliseconds (period), or consume 50 milliseconds in a period of 100 milliseconds. Even though both settings consume 50% of CPU load, the first presents a more strict constraint than the second, which leaves the system more flexibility when to spend the CPU cycles.

The following information is logged:

- Average generated load
- Missed periods

Currently this cycle eater is part of the SysLog[1] component of the TV software. This eater –when started– periodically eats CPU cycles. The settings of the eater can be controlled via a Perl script running on a test PC. To enable even more flexible control over the CPU stressor, for every CPU more than one stress task can be started in parallel, with different priorities, periods, and budgets. The eaters communicate via the EJTAG probe with the SysLog component. In Figure 5.4, the message sequence chart is shown while in the next sections the calls are briefly described.

5.3.3 Realization

To create a CPU eater task, the function `islctrl_create_cpu_eater` must be called. This function creates an eater thread on the specified processor with the parameters given in the function call. The main thread locks a `pthread mutex` that doesn't allow the eater thread to start until an explicit call to `islctrl_start_cpu_eater` is performed. The eater thread will run the `eater_task` internal function after its creation.

The parameters related to an eater are stored in a structure in the main thread (for later settings changes and control) and the eater thread is using them directly (as a form of inter-thread communication).

To start the CPU eater, the function `islctrl_start_cpu_eater` must be called. An eater can be specified using the `eaterid` returned by `create_cpu_eater`. The mutex that blocks the execution of the eater thread is released and the eater thread can start executing.

To stop the CPU eater, the function `islctrl_stop_cpu_eater` must be called. This function calls `pthread_cancel` on the eater thread in order to stop it. The eater thread tests if a cancel signal was sent at the beginning of each eating loop.

[1] SysLog is a component in the TV52x architecture for diagnostic purposes during SW development. The component polls a register twice a second for new (parameterized) commands and executes them. It has no return channel. Commands are typically given by a PC writing the command register via EJTAG probe (in a running system).

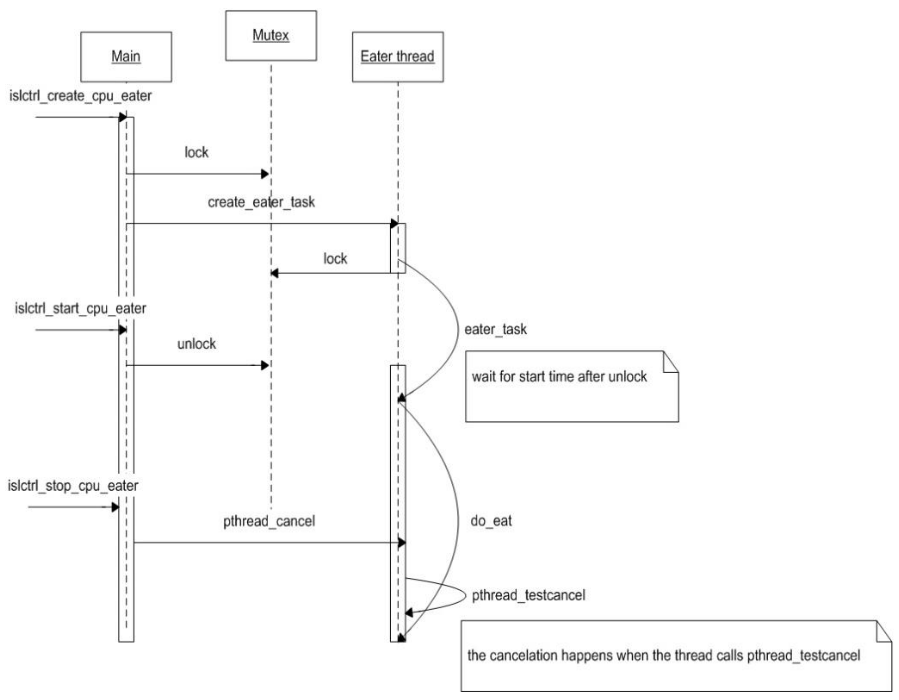


Figure 5.4 *Message sequence chart of the CPU stressor*

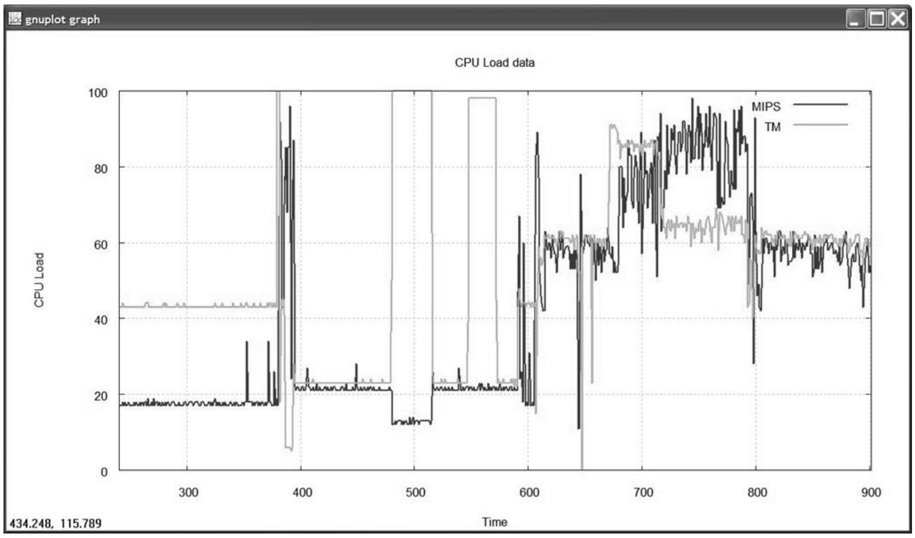
The eating of cycles happens in the function `do_eat()`. This function runs two `while` loops, the exterior one to check if there are any more runs required, and an internal one that checks the consumed budget and the arrival of the next period. Within the internal loop, the thread reads the cycles from the Time-Stamp Unit (TSU) clock and every time a change in the number of ticks is detected, the consumed budget is updated. If the consumed budget reaches the allocated budget, the eater is stopped for the current period and the eater thread is set to sleep until the next period. Since this loop reads from the TSU, one needs to be aware that higher priority tasks and interrupts can interfere with this eater task. The results from the TSU must therefore be handled with some care in order to get a reliable eater.

Furthermore a tick of the TSU is assigned entirely to either the eater or the running program. Since normal task switches and interrupts have a low frequency compared to the TSU frequency the measurement error they create can be neglected and will typically cancel out. Note that reading the TSU also increases bus load; CPU eating is not completely orthogonal to other resource usage. Using interrupts to indicate the end of a period is not feasible since periods are typically in the order of magnitude of the interrupt latency and interrupt handling adds a significant amount of additional CPU cycles. Furthermore the impact of interrupts differs per CPU and per OS. The OS (in this case Linux) of the TV platform did not provide a timer with sufficient granularity or a known fixed amount of CPU cycles spent by the OS for handling the end of period notification.

5.3.4 CPU cycle visualization

Together with implementing the cycle eater in the television, reading out CPU usage and visualizing it in real-time was implemented which was needed for verification. The actual eater has time information which reflects CPU usage. This kind of visualization was already being developed for real time monitoring of memory bandwidth usage described later in the chapter. To this visualization tool, the plotting of CPU cycle consumption was added. Some results of displaying the CPU usage are plotted in Figure 5.5, where the CPU usage of both the MIPS and the TriMedia (TM) are shown without additional SW stress added. It can be seen that on some moments the TriMedia uses its entire CPU budget and that at the same time the MIPS CPU usage diminishes. This indicates an issue with memory bandwidth or communication between both processors. Observing such events in real-time can help in finding correlations between certain behaviors of the system with resource usage and input signals and where to locate this behavior in the software.

Visualization is very useful since it also gives a quick condensed overview over a large period of time. Before visualization was available load measurements were used on an infrequent basis only (every few releases or on request of customers). These measurements had a 1-second average in a steady state. It was not possible to measure load reliably during e.g. zapping (changing channels) or other short actions. The real-time visualization described here can be used whenever desired and does give reliable information during actions such as zapping (switching TV channels).



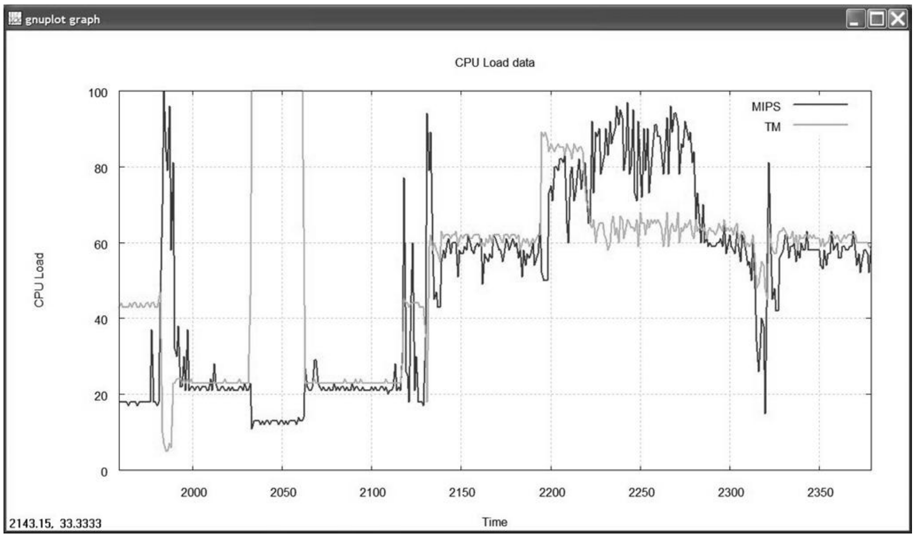


Figure 5.5 *Some examples of CPU usage monitoring*

5.4 Memory bus stressor

Similar to consuming CPU cycles, one can also consume memory bus cycles or memory bandwidth. There are three basic techniques to consume memory bandwidth without applying hardware modifications, as follows:

- The first method is to simply use a software loop that reads from or writes to memory. Though this method is simple, it also influences CPU usage. Since one wants to know what actually causes potential stress problems one wants to separate using memory from using CPU time. Therefore this method is usually not recommended for single processor systems. In a multi-processor system as the TV52x architecture is, one can use this simple method with some caution.
- The second method is to change memory controller settings to limit the possible amount of memory accesses. Though this method is non-intrusive for other resources, due to the complex policies and priority schemes and hierarchical configuration of most arbiters it is hard to determine correct new settings without affecting "normal" behavior.
- The third method uses a spare DMA device to generate memory reads/writes; assuming that a spare DMA device is available. This has the advantage that it creates little overhead for other resources. A disadvantage can be that this is a use case dependent implementation. The set of all DMA devices can have a fixed priority with respect to the processors and other devices priority in accessing memory. Changing these settings will also change the behavior of the "normal" system. Priority amongst DMA devices can be changed. Care must be taken that such an implementation does not read from cache, as this heavily influences the actual memory usage.

Depending on the actual system and the desired measurement one can choose the method that has least impact. For an optimal bus stressor one may even need to add one or more DMA devices in the silicon design of the system, merely for creating memory loads. With this set of “dummy” DMAs one can make an optimal mix to create a memory bandwidth stressor that has least impact on other system resources.

In the TV52x case a bus stressor was implemented using the first method (a SW loop). In general a SW method is not considered advisory, especially in single-processor systems. In this case though, it was acceptable, since the MIPS generated the bus load, while observing the resulting artifacts caused by limited bus access of the TriMedia. The software of both CPUs is sufficiently independent and the CPU load of the MIPS even with bus-stressor is sufficiently low to opt for this method. Having knowledge about the actual architecture and implementation made it possible to achieve a usable bus stressor with a relatively low implementation effort.

5.5 Example of using a stressor on a television

To test the concepts of the CPU stressor described above, we used a digital television at the maturity stage of integration tests. First the CPU eater was used to stress the CPU in a highly used scenario (everyday analog and digital TV viewing). The stressor was started and stopped at regular intervals, bringing the system up to 100% CPU (TriMedia) utilization and back again to normal utilization. As one would expect, when the load was increased to 100%, the system became slower. Furthermore, it no longer responded to inputs from the remote control within an acceptable short time, even though the user interface runs on the other processor. However the television did not crash and video and audio kept on streaming, even while video and audio processing was done by the stressed CPU. As soon as the eater was stopped, the television worked again as specified and no residual effects could be found. This is a desired scenario for such a product. If for any reason the CPU cannot temporarily cope with high loads, the systems keeps operating in a reliable way.

In another experiment the CPU eater was used to stress a so-called sensitive shop scenario (playing an Mpeg2 video from an USB stick). This mode of displaying a movie is intended to be used in shops, to continuously show a (commercial) video and show the features of the television (the so-called SuperShopDemo). After the stressor was started, bringing the system to 95% CPU utilization, one suddenly saw that the television no longer worked as under normal conditions. The video got hampered and the audio got out of sync and disappeared. In itself this may be allowed behavior; the system is stretched beyond its limits. When the stressor was stopped again, the video recovered after a few seconds. The audio however did not come back when the stressor was stopped. Only after a random time period, usually minutes long, or when the video was restarted (it plays in a loop) the audio returned. This is a typical case where a temporary CPU overload causes an error that does not recover properly. Where the temporary disturbances might be acceptable, it is not acceptable that the system needs some sort of restart to recover from the temporary overload.

This erroneous behavior was not observed during earlier integration tests and hinted to a problem that was not expected in the execution architecture. Use of the CPU stressor helped discovering this potential field problem.

5.6 Real-time bandwidth monitoring

As explained earlier in the chapter, resources are shared to maximize the performance and minimize the system costs. Shared memory is one of the critical resources as the cost of the system increases substantially with the increase in the number of the pins to interface with memory. Also, as the embedded software evolves, more use cases need to be supported and the applications are extended. This may lead to an increase in the memory bandwidth demand in an unpredictable way. Real time memory bandwidth monitoring tool aids in the integration process showing the consumption of bandwidths by different ports and an early detection of faulty behavior, and the head room available for future application modifications.

The implementation of bandwidth stressor could also be verified by this monitoring tool. Since a visualization of data leads to faster interpretation of data, a real time visualization tool was developed to display the data being read by the memory monitor.

The goal of bandwidth monitoring is to be able to optimally adjust the memory controller since it is difficult to accurately estimate the net available bandwidth at design time. An accurate estimate can only be based on measurements after the application with nearly full functionality in critical use cases is brought-up.

5.6.1 Controlling bandwidth and real-time monitoring

The IP 2032 is a memory arbiter/controller which intelligently merges (DMA type) hard real-time traffic from the hardware devices with low-latency traffic from the processors (see Figure 5.3). It uses a budget-and-accounting scheme on the gross bandwidth per port, combined with (variable) priority settings. The IP 2032 memory arbiter/controller has four data ports: one for hard real-time bandwidth traffic, and three for low-latency traffic. As shown in Figure 5.3, only one constant bandwidth port and two low-latency ports are used by the arbiter in the TV 52x platform. Furthermore, the arbiter has internal status registers to track various parameters (e.g. memory idle time, amount of memory control cycles, etc.) as well.

In the design phase of the system, the schedules for different streams accessing the memory cannot be made such that the utilization of the memory bandwidth is 100%. The main reasons are the following:

- Consecutive memory accesses from the system are not always on consecutive memory addresses. This leads to so called “bank-conflicts”, which implies that the memory access takes additional time and during that time the memory is not available. Especially for the hard-real time traffic, which is the combined traffic from different hardware devices, there will be a large fraction of miss-aligned memory accesses, hence large fraction of bank-conflicts. This causes the gross bandwidth of a port to be significantly larger than the net bandwidth.
- Memory banks have to be prepared for the read accesses. As reads may follow writes, the read has to be stalled until the memory bank has been prepared after the write access. This read-after-write conflict further reduces the maximum memory utilization.
- The available memory bandwidth is also lowered due to the time required for the DRAM-refreshes.

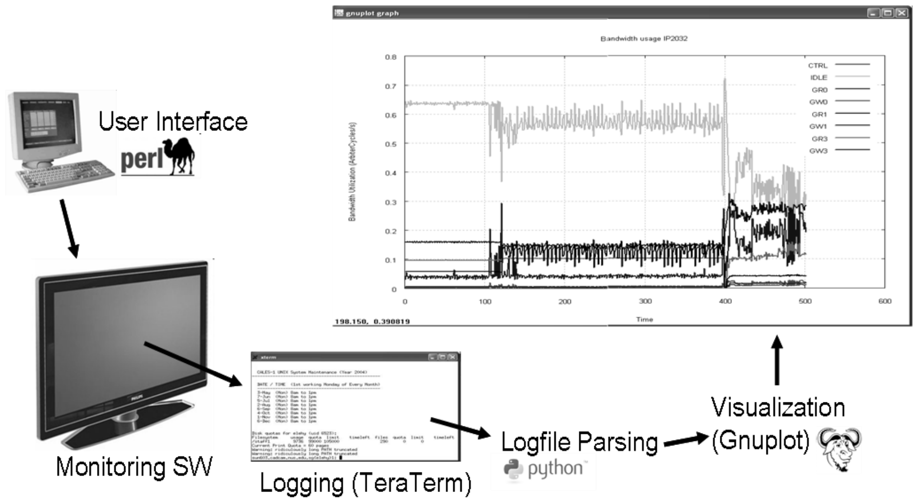
These reasons are termed as inefficiencies of the system. In the IP 2032 memory arbiter/controller, these inefficiencies are contributed to the port accessing the memory before the conflict occurs. As the gross bandwidth is the bandwidth utilized for the memory accesses including the time for the inefficiencies, the gross bandwidth of a port is always larger than the net bandwidth, which is the bandwidth needed for purely the data transport [Agarwal, 2008].

It is difficult to accurately estimate the net available bandwidth at design time, as the amount of memory conflicts and related memory inefficiencies depend on the actual scheduling of data transactions between the ports. An accurate estimate can only be based on measurements after the application is brought-up. Moreover, the IP 2032 arbiter/controller is programmed using the gross bandwidth rather than the net bandwidth. Because the gross bandwidth can vary largely due to the interference between the memory traffic of different ports, an additional problem is caused: the gross bandwidth assigned to a device (or group of devices) may be too low to serve the net bandwidth requirements under all circumstances!

The net bandwidth utilization may fall short with respect to the required bandwidth due to the interference and inefficiencies mentioned above. If so, there will be a reliability risk. To properly analyze the effective usage of the memory bandwidth and deal with these problems, it is essential to measure the bandwidth utilization and adapt the parameters of the memory arbiter such that the bandwidth requirements and real-time constraints are met for each stream connected to the arbiter/controller. Real-time monitoring of bandwidth usage helps in uncovering these reliability issues.

5.6.2 Implementation of bandwidth monitoring

Real-time bandwidth monitoring requires the access of the internal status registers of each port of the memory arbiter. These internal registers keep track of the memory cycles used by each port. In case of the IP2032, these registers can be accessed by several methods: (i) with the help of a flag which is set whenever a value of the registers is changed and (ii) polling and reading the register values at regular time intervals. The values obtained can be stored in the on-chip memory-buffer, or printed on a hyper terminal on the PC screen via UART, or they can be displayed on the TV screen using on screen display functionality (OSD). Traces can also be stored in the off-chip memory (DRAM). However, the strong point of real-time feedback to the designer is lost in this case. The core program performing the internal read-out of data runs on the programmable device of the architecture, in case of TV 52x either on MIPS or the TriMedia.



```
 CALES-1 UNIX System Maintenance (Year 2004)
------------------------------------------------
 DATE / TIME  (1st working Monday of Every Month)
------------------------------------------------
 3-May  (Mon) 8am to 1pm
 7-Jun  (Mon) 8am to 1pm
 5-Jul  (Mon) 8am to 1pm
 2-Aug  (Mon) 8am to 1pm
 6-Sep  (Mon) 8am to 1pm
 4-Oct  (Mon) 8am to 1pm
 1-Nov  (Mon) 8am to 1pm
 6-Dec  (Mon) 8am to 1pm

Disk quotas for elehy (uid 6523):
Filesystem     usage  quota  limit    timeleft  files  quota  limit    timeleft
/staff1         9736  99000 105000               290      0      0
Current Print Quota = 60 pages
Warning: ridiculously long PATH truncated
Warning: ridiculously long PATH truncated
sun6XX.cadcam.nus.edu.sg[elehy]1: 
```

Figure 5.6 *Visualization set-up*

The data read from the registers needs to be provided as an input to the visualization tool (see Figure 5.6). This is done with a post processor which takes as input the data read from the arbiter registers. The post processor prepares the data to be used as an input to any visualization tool. Applying this method lead to the detection of unpredictable behavior of the system: e.g. in TV520, idle port EXT2 consumed considerable higher bandwidth through DMA agents as compared to idle ports EXT1 and EXT3. This was brought about only through the help of real-time bandwidth monitoring tool.

5.7 Conclusions

CPU stress testing has proven to be a valuable extra testing technique. The CPU stress testing method described in this chapter is implemented in the TV52x production software and can be used during the integration phase. The systematic application of stress testing requires a change in the integration process though, which seems to be a somewhat bigger step to take.

To create the stressor itself one needs to be aware of internal measurement challenges. The process that eats CPU cycles will also use CPU cycles, and can be influenced by other tasks. Thus it can have impact not only on the targeted shared resource, but also other resources. Thus, measurements could be slightly off when higher priority tasks interfere.

Shared resource stressing can be useful for both reliability and design evaluation. It provides the means for a systematic approach that –once implemented and added to the process and build flow– can be applied early on to quickly check the system robustness against shared resources problems.

The visualization of the CPU usage and memory bandwidth usage was quickly adopted inside NXP development and appreciated as a valuable tool to tune software, and to search for potential problems.

The real-time view on the memory traffic and CPU load is a powerful enhancement for design and test engineers. It eases the burden of system integration and validation. The overhead on the system is only marginal; the small program which runs on the embedded processor, and which captures, formats and logs the data to the outside world causes only a marginal additional load on the system. Via an externally connected PC, monitoring settings can be set and modified at runtime, e.g. the selection of the interval with which the data capturing takes place; which memory arbiter data needs to be logged. The visualization then will automatically adapt to the selection made. And finally, with real-time data on memory traffic available, next steps can be taken to automatically detect and correct memory-overload conditions, potentially even for systems that are deployed in the field.

Acknowledgements: We gratefully acknowledge the input from Ben Pronk, Rob van Twist, Liesbeth Steffens and the execution architecture team of NXP Home.

5.8 References

[Agarwal, 2008] M. Agarwal, A.K. Nieuwland. *Real Time Bandwidth Monitoring: IP2032 A Case Study*. Technical Note NXP-R-TN 2008/00086, NXP Semiconductors

[Graham, 2008] D. Graham, E. van Veenendaal, I. Evans, R. Black. *Foundations of Software Testing: ISTQB Certification*. Edition updated for ISTQB Foundation, 2nd Edition. 2008. Cengage Learning, London. ISBN-10: 1-84480-989-7

[Liu, 1973] C.L. Liu, J. Layland. *Scheduling algorithms for multiprogramming in a hard real-time environment*. Journal of the ACM 20 (1): pp. 46-61, 1973

[Lehoczky, 1989] J. Lehoczky, L. Sha, Y. Ding. *The rate monotonic scheduling algorithm: exact characterization and average case behavior*. IEEE Real-Time Systems Symposium 1989, pp. 166-171

[Merriam-Webster, 2009-a] *Robust*. Merriam-Webster Online Dictionary. Retrieved April 21, 2009, from http://www.merriam-webster.com/dictionary/robust

[Merriam-Webster, 2009-b] *Elasticity*. In Merriam-Webster Online Dictionary. Retrieved April 21, 2009, from http://www.merriam-webster.com/dictionary/elasticity

[Steffens, 2008] L. Steffens, M. Agarwal, P. van der Wolf. *Real-Time Analysis for Memory Access in Media Processing SoCs – A Practical Approach.* Proceedings of Euromicro Conference on Real-Time Systems, pp. 255-265; 2008

Chapter 6

Aspect orientation and resource constrained environments

Authors: Piërre van de Laar, Rob Golsteijn

Abstract: Separation of concerns [Dijkstra, 1976] is a well-known principle to deal with complexity. Unfortunately, the mainstream development languages and tools do not support separation of concerns in the design and implementation of software. This negatively influences the reliability of the resulting software and system. Aspect orientation improves the support for separation of concerns in the design and implementation. However, aspect orientation needs reflective information for identification purposes. Yet, the overhead in the current available approaches is too high for resource constrained environments. In this chapter, we describe AspectKoala, a framework for aspect orientation in a resource constrained environment. We focus on the impact of aspect orientation on reliability and on the realization of an efficient design for providing the necessary reflective information in a resource constrained environment.

6.1 Introduction

Many consumer products contain a significant amount of software. The amount of software in these products is exponentially growing according to Moore's law. With the increase of software also the complexity, as measured by the necessary software staff, increases. Figure 6.1 shows this trend for televisions. Despite the increase in complexity, products should remain reliable. Separation of concerns, i.e., the ability to deal with the difficulties, the obligations, the desires, and the constraints one by one [Dijkstra, 1976], is a well known principle to cope with this growing complexity. Unfortunately, the currently existing design and development languages and tools do not support separation of concerns. Consequently, we observe scattering and tangling of concerns in the current software. Figure 6.2 shows the implementation of a concern that handles access before initialization of software components. Figure 6.2 clearly shows that this concern is not localized in one software file but is scattered throughout the whole software. The scattering of concerns reduces the reliability. For example, when making a change, the likelihood of introducing errors is increased, since a part of the scattered code of the concern might be forgotten. Figure 6.3 shows a software component in which multiple concerns are tangled: a programmer has manually combined multiple concerns (handle access before initialization, precondition checking, core functionality, and notifications of change to observers) in a single function. The tangling of concerns also reduces the reliability. Due to tangling, existing code is harder to understand which increases the chances of introducing errors during maintenance and evolution.

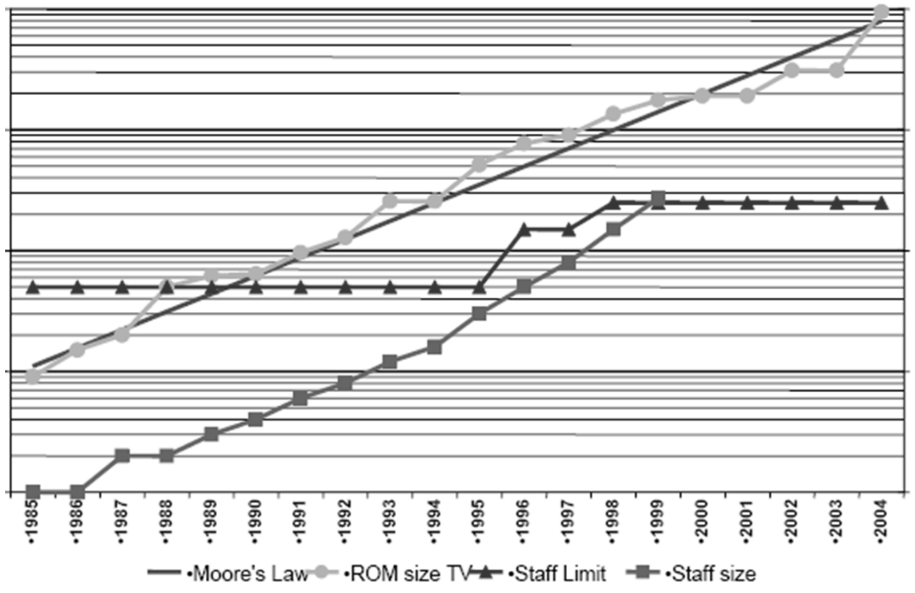


Figure 6.1 *The amount of software in a television is exponentially growing according to Moore's law. With the increase of software also the complexity, as measured by the necessary software staff, increases.*

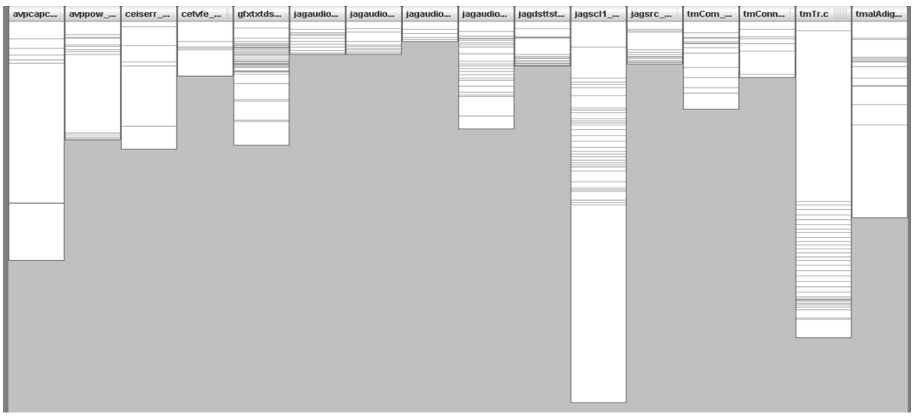


Figure 6.2 *The scattering of a concern over multiple files is visualized using AspectBrowser [Griswold, 1999]. Each source file is depicted as a block. Each source line dealing with this concern is grey and horizontal.*

```
void asf_SetBlackBarDetectedFormat(int aspectratio, int vershift)
{
    /* handle access to component before initialization concern */
    ASSERT( m_initialized );

    /* precondition checking concern */
    ASSERT( aspectratio>0 );
    ASSERT( (vershift >= asf_MinVerticalShift) &&
   (vershift <= asf_MaxVerticalShift) );

    /* core functionality concern */
    m_bbd_aspect_ratio = aspectratio;
    m_bbd_vertical_shift = vershift;

    /* notifications of change to observers concern */
    sclN_OnBlackBarDetectedFormatChanged(aspectratio, vershift);
}
```

Figure 6.3 *Tangling of multiple concerns in a single function. Function is taken from the Nexperia Home software. The function is written in the C language. We have annotated the function using comments to describe the different tangled concerns.*

Preventing scattering and tangling of concerns, in general, improves the reliability of the software. But reliability profits even more, since many reliability concerns are currently not localized in a single file but are tangled with and scattered over the whole software. Some examples of reliability concerns are

- The handling of access to software components before initialization. This concern has already been discussed and is visualized in Figure 6.2.
- Exception handling.
- Ensuring correct usage of software designed for single threaded access only.
- Checking specified resource usage during actual execution.
- Ensuring correct usage of state-based components.

The Trader project aims to improve the reliability of consumer products, such as televisions, hard-disk DVD-recorders, and hi-fis. In this market, producers compete on consumer price. Due to the high volumes, a low bill of material is crucial. As a result, resources, such as memory (both ROM and RAM), CPU cycles, and bandwidth, are constrained. Aspect orientation [Kiczales, 1997] [Laddad, 2003] improves the separation of concerns in the design and implementation, which positively influences reliability. However, aspect orientation has not yet been applied in a resource constrained environment. Therefore, in the Trader project, we developed AspectKoala [van de Laar, 2006], a framework for aspect orientation in a resource constrained environment.

In the remainder of this article, we will first give a simplified picture of aspect orientation and describe a particular challenge to enable aspect orientation in a resource constrained environment. Second, we describe the architectural decisions we took to address this challenge. Third, we describe the actual design related to our challenge in AspectKoala. We show that aspect orientation in a resource constrained environment is possible and highlight the positive impact on reliability we experienced. We end with conclusions.

6.2 What is aspect orientation?

Aspect orientation introduces join points: identifiable points in the execution of a program. Aspect orientation enables interaction between different concerns at these join points. To give an example, join points around instructions that change items in a database for a user, enable that:

- Before the instructions are executed, the access to the database is logged;
- The instructions are only executed when the user has the rights to modify the items in the database;
- After execution of the instructions, all observers of the database are notified to ensure accurate visualizations.

An aspect implements a concern[1] and contains pointcuts and advices. A pointcut specifies where an aspect crosscuts other aspects by selecting join points. Join points can be selected based on program language constructs, such as methods, variable accesses, and exceptions, and the types, attributes, and names used therein. An advice specifies in a method-like construct what behavior to exhibit at the selected join points. For the implementation of an advice, an aspect may require functionality of other aspects, access meta-data of parts of the software, and introduce variables.

Making a product from aspects is called weaving: joining the aspects at the selected join points. Weaving can occur at different points in time, for instance, before compile time by code weaving, at load-time by the class loader, or at runtime by the virtual machine. A single program might even contain multiple aspects weaved together at different points in time.

For illustration, we will describe how the concern of tracing method calls can be implemented using aspect orientation. The pieces of code of this example are written in the well-known languages Java [Arnold, 1996] and AspectJ [AspectJ]. The pieces of code interact only at the join points. The exposed set of join points is a choice of the aspect language. AspectJ introduces, amongst others, the following join points: method execution and call, field read and write access, and object initialization. We want to trace method calls in the following Java class (see Figure 6.4):

```java
public class MyClass
{
    public void f() { /* some code */ }

    public void g() { f(); /* and some more code */ }

    public static void main(String[] args) {
        MyClass mc = new MyClass();
        mc.f();
        mc.g();
    }
}
```

Figure 6.4 *An example of a Java class*

[1] Functionality is just one of the concerns.

For tracing, we write the following AspectJ aspect (see Figure 6.5):

```
1 public aspect MyAspect {
2   pointcut MyClassMethodCall(): call (* MyClass.* (..));
3   before(): MyClassMethodCall() {
4     System.out.println(thisEnclosingJoinPointStaticPart
5                             + " calls " + thisJoinPoint);
6   }
7 }
```

Figure 6.5 *An aspect that traces the Java class `MyClass`.*

We limit our explanation to this example. For more details on AspectJ, we refer to [AspectJ]. On line 1 of Figure 6.5, an aspect called `MyAspect` is declared. It contains two elements called pointcut and before advice. Line 2 defines the pointcut, i.e., a selection of join points. This pointcut, named `MyClassMethodCall`, selects all method calls to any method in `MyClass`. Lines 3 to 6 define the before advice. This before advice specifies that before any of the join points selected by the pointcut `MyClassMethodCall`, we want to execute the code at lines 4 and 5. The before advice accesses reflective information via the two predefined objects `thisEnclosingJoinPointStaticPart` and `thisJoinPoint`. This reflective information serves two purposes. First, a single method can be called from multiple locations. The object `thisEnclosingJoinPointStaticPart` gives information about the current calling context. Second, a pointcut can use wildcards and hence match multiple join points. The object `thisJoinPoint` contains the information about the currently matched join point.

Compiling and running the class and aspect produces the following output:

```
execution(void MyClass.main(String[])) calls call(void MyClass.f())
execution(void MyClass.main(String[])) calls call(void MyClass.g())
execution(void MyClass.g()) calls call(void MyClass.f())
```

Figure 6.6 *The output of running the Java class `MyClass` with the tracing aspect `MyAspect`.*

The aspect consists of only seven lines and is independent of the number of methods in the traced class. Its output is comprehensible and effective thanks to the reflective information. As this example demonstrates, reflective information is crucial for tracing using aspect orientation. In fact, when aspect orientation is used, reflective information is always needed for identification, e.g., for error isolation, caller localization, or debugging purposes.

Reflection adds an amount of overhead in performance and resources. The overhead in performance, such as CPU consumption, has received considerable attention (see e.g., [Cazzola, 2004] and [Chiba, 1997]). The overhead in resources, such as the consumption of ROM and RAM memory, has so far not received much attention. For consumer products, both kinds of overhead are important.

All approaches [Lämmel, 2005] [Rajan, 2005] [AspectJ] [JBoss] currently used in aspect orientation to handle reflective information consume more resources than available in a resource constrained environment, such as consumer products. Therefore, in this chapter we address the challenge of developing a resource effective methodology for reflection on join points to enable aspect orientation in consumer products.

6.3 Architectural decisions

In this section, we answer three questions necessary to add reflective information on join points in the context of consumer products. The answers are mainly driven by reliability and overhead requirements for this context.

6.3.1 How to pass reflective information?

How should we pass reflective information on a join point to the advice? Two alternatives exist: explicit and implicit reflection parameters. In case of explicit reflection parameters, an advice can only access reflective information when its signature contains a specific reflection parameter. In case of implicit reflection parameters, an advice can always access reflective information. Advices don't have to request the reflective information: the aspect-oriented framework just always provides this information[2]. Eos [Rajan, 2005] and JBoss AOP [JBoss] are examples of aspect-oriented frameworks with explicit reflection parameters, and AspectJ [AspectJ] is an example of implicit reflection parameters. See also Figure 6.7. The two alternatives behave different with respect to overhead, reliability, and conceptual consistency.

```
Explicit

abstract aspect MyAspectX
{
  abstract pointcut MyPointCut();

  [before MyPointCut]
  void f (JoinPoint thisJoinPoint){
     …
  }
}

aspect MyAspectY:MyAspectX
{
  …
  [before MyPointCut]
  void f (JoinPoint thisJoinPoint){
     Debug.WriteLine (thisJoinPoint);
  }
}

aspect MyAspectZ:MyAspectX
{
  …
  [before MyPointCut]
  void f (JoinPoint thisJoinPoint){
     …
  }
}
```

```
Implicit

abstract aspect MyAspectX
{
  abstract pointcut MyPointCut();

  [before MyPointCut]
  void f (){
     …
  }
}

aspect MyAspectY:MyAspectX
{
  …
  [before MyPointCut]
  void f (){
     Debug.WriteLine(thisJoinPoint);
  }
}

aspect MyAspectZ:MyAspectX
{
  …
  [before MyPointCut]
  void f (){
     …
  }
}
```

Figure 6.7 *Explicit versus implicit reflective parameters.*

The overhead of passing a reflection parameter includes stack space, CPU cycles, and ROM footprint. With explicit reflection parameters, the overhead is only incurred when it is actually used

[2] To improve performance and resource usage, the aspect-oriented framework can analyze the advice and exclude reflective information from the implicit signature when not used by that advice.

by an aspect in an inheritance hierarchy. For example, in Figure 6.7 `MyAspectX`, `MyAspectY`, and `MyAspectZ`, must have an explicit reflection parameter, since reflection is used in their inheritance hierarchy by `MyAspectY`. With implicit reflection parameters, the overhead will be always incurred in general. Since, in general, aspects are required to be replaceable (e.g., to enable dynamic weaving or dynamic linking) and the system can consist out of multiple compilation units, therefore any sub-aspect, defined in another compilation unit, could use the reflection parameter, and the advice's implicit signature must thus contain the reflection parameter to ensure compatibility at the binary level. See also Figure 6.7: `MyAspectX`'s implicit signature must include the reflection parameter, since otherwise subtype `MyAspectY`, whose implicit signature includes the reflection parameter, will be incompatible at the binary level. However, by limiting a system to a single compilation unit, a weaver can analyze all aspects for the actual usage of the reflection parameter and make the overhead identical to the overhead with explicit reflection parameters. If aspects are also not required to be replaceable, as in AspectJ [AspectJ], the overhead can be made even smaller than with explicit reflection parameters. We will illustrate this by using the example aspects of Figure 6.7. Given that aspects are not required to be replaceable, the aspect `MyAspectZ` (and its base aspect) can be analyzed in isolation. The method `f` of aspect `MyAspectZ` implements the before advice for all join points in the pointcut `MyPointCut`. Since this method `f` does not use reflection, it will not have a reflection parameter in the implicit case, yet it will have one in the explicit case. Of course, in all cases the method `f` of `MyAspectY` will have a reflection parameter.

Reliability benefits when the intentions can be verified too. With implicit reflection parameters, the intention to use reflection cannot be expressed, while the presence of an explicit reflection parameter in a method's signature specifies the intention to use reflection[3]. Verification covers both illegal usage of a reflection parameter and a reflection parameter not being used[4]. These kinds of verification for parameters are present in all modern compilers. Hence, an explicit reflection parameter allows for verification of intention to use and thus increases reliability.

For conceptual consistency, no difference between aspects and classes should exist [Rajan, 2005]. Advices and methods become identical when implicit reflection parameters of an advice are made explicit.

An advantage of explicit over implicit reflection parameters is that only for explicit reflection parameters their type can be put under the control of the user. Whether this advantage is useful will be discussed later on in this section.

Based on these differences, we prefer explicit reflection parameters to implicit reflection parameters for consumer products. In other words, our conclusion is that reflective information on join points should be passed explicitly. Yet, what information should be passed to an advice via these explicit reflection parameters?

6.3.2 What reflective information to pass?

The reflective information on the join points can range from general-purpose, i.e. for all aspects and all join points identical, to dedicated, i.e. specific for a particular set of aspects and join points. AspectJ [AspectJ], for example, uses instead of one general-purpose reflection type, three more dedicated reflection types: `thisJoinPoint`, `thisJoinPointStaticPart`, and `thisEnclosingJoinPointStaticPart`. AspectJ analyses the advices to determine the necessity of

[3] The reflection parameter can be used by the aspect itself or by aspects in its inheritance hierarchy.

[4] Reflection parameter not being used is only informative when the complete inheritance hierarchy is taken into consideration.

these three reflection parameters. Based on the result of the analysis, AspectJ limits the overhead to roughly the necessary reflective information. The overhead is still considerable, since reflection allows extracting information in the verbose human readable form, like in Figure 6.6.

We compare general-purpose and dedicated reflective information with respect to overhead, reliability, and evolve-ability.

A dedicated reflection parameter needs only to satisfy a single purpose, while a general-purpose reflection parameter must serve all possible purposes. A dedicated parameter can leave out information that is not needed for its purpose and apply optimizations that are possible given its specific purpose. Hence, the overhead of memory, CPU cycles, and ROM footprint for a dedicated reflection parameter will never be larger than for a general-purpose reflection parameter. In particular, many consumer products have only the requirement to support off-line or post-mortem manual analysis. To satisfy this requirement, only a unique identifier per join point, comparable to IDREF in AspectCobol [Lämmel, 2005], is sufficient at runtime. Off-line, the unique identifier can be linked via a look-up table to any static reflective information, including information in verbose human readable form.

A general-purpose reflection parameter specifies the intention to use reflective information. A dedicated reflection parameter also specifies the particular part of the reflective information that is intended to be used. Hence, a dedicated reflection parameter compared to a general-purpose parameter allows for more verification and thus increases reliability.

A dedicated reflection parameter only evolves when the related system requirements change. A general-purpose reflection parameter also evolves when the environment changes. It evolves, for example, due to changes in the language and compiler, such as the addition of custom attributes, and due to changes in the reflection type based on requirements of other systems.

Based on this comparison, we prefer dedicated reflection parameters to general-purpose reflection parameters in the context of consumer products. Hence, our conclusion can only be that reflective information on join points should be passed explicitly using dedicated reflection parameters. Yet, who controls the types of these dedicated reflection parameters?

6.3.3 How to control reflective information?

The type of reflective information is under control of either the user or the aspect-oriented framework, e.g., via predefined or configurable types. In the first case, the user must not only specify the type but also the value of the reflection parameter as function of the join point when the advice is called, since the framework is not knowledgeable of the reflective information. Of course, a library can support the user herein with for example string compression, bit encoding for collections, and bits concatenation. In the case that the type of reflective information is under control of a generic aspect-oriented framework[5], domain knowledge cannot be exploited to efficiently encode the reflective information.

Examples of exploiting domain specific knowledge which is possible when the user is in control but impossible otherwise are:

- A user can tune the size of the reflective type based on global product knowledge, such as the number of join points.

[5] For example, AspectC++ tailors down the reflective information according to the individual requirements of the actual advice [Spinczyk, 2005] by using a super-type of the generic reflection type.

- A user can let the reflection type contain Boolean fields that indicate:
    - The begin or end of an interval delimited by a method pair, like begin/end critical section, enter/leave menu, open/close file, and acquire/release semaphore; or
    - Whether the join point is in the application or platform part of the software.

Since, for consumer products, the reduction of overhead outweighs the additional user effort, we decided to pass reflective information in our framework explicitly using dedicated reflection parameters which types are controlled by the user. Yet, how do we design that?

6.4 AspectKoala

In this section, we describe the design of reflection on join points in AspectKoala. To understand the design, we briefly introduce Koala and the high-level design of AspectKoala. Then, we apply AspectKoala to measure in a real-world example the overhead of different alternatives for reflection on join points. We end with our experiences in using AspectKoala to handle a reliability concern.

6.4.1 Koala

The software in some televisions that contain NXP chips is currently modularized using the component model Koala [van Ommering, 2004]. Koala has many similarities with other component models, such as OMG's CORBA Component Model [Siegel, 2000] and Microsoft's COM [Rogerson, 1997]. In Koala:

- Interfaces and components are specified in an architecture description language.
- Components can only interact with each other via interfaces.
- A component provides functionality for which it may require functionality from its environment.
- A component can provide and require multiple interfaces of the same type, since a component refers to its interfaces by instance name and not by type.
- A third party instantiates components and connects their interfaces

Unfortunately, the separation of concerns cannot always be reflected in the modularization of the software into components, as can be observed in the current software and was already shown in Figure 6.2 and Figure 6.3:

- Many (non-functional) concerns are not localized in one component but are scattered throughout the software; and
- Multiple concerns are tangled in one component.

To better support the modularization of the reasoning, NXP and Philips have investigated more effective modularization techniques for software, such as aspect orientation. For this purpose, an aspect-oriented framework on top of Koala [van Ommering, 2004] has been developed: AspectKoala [van de Laar, 2006].

6.4.2 High-level design

AspectKoala is built on top of the abstract syntax tree of Koala, much like SourceWeave.Net [Jackson, 2004] is built on top of CodeDOM: the abstract syntax tree of .Net. The architectural description of all components and interfaces is contained in the abstract syntax tree of Koala. AspectKoala weaves aspects into the product based on these architectural descriptions at compile time, and it is independent of the programming languages used to implement the components. The join points of AspectKoala are the execution of and calls to Koala interface functions. Aspects in AspectKoala are compiled .Net classes that implement the `IComponentAspect` interface. This interface contains methods to specify the pointcuts, to implement the advices in the C programming language, and to declare dependencies, such as usage interfaces [Lieberherr, 1999]. Hence, unlike AspectJ that provides regular expressions to specify pointcuts, in AspectKoala methods of the `IComponentAspect` interface must be implemented for this purpose. Of course, these methods can use .Net's regular expressions to specify pointcuts.

Currently, adding an aspect to Koala's component-based software consists of the following steps:

1. Write and compile an aspect[6], i.e., a class in .Net that implements the `IComponentAspect` interface.
2. Run AspectKoala to weave the aspect with a particular Koala configuration. This results in a new Koala configuration where the aspect is added.
3. Compile this new Koala configuration as usual.

6.4.3 Design of Reflection on Join Points

We describe the part of the `IComponentAspect` interface relevant for user-controlled reflection on join points, see also Figure 6.8. The method `JoinPointHasKindAdvice` specifies that a join point is part of a pointcut and has a particular `Kind` of advice, i.e., before, around, or after advice. If a join point has a particular kind of advice, the method `KindAdvice_UsesReflection` explicitly specifies the presence of a reflection parameter at that join point for that kind of advice. When a reflection parameter is present, `KindAdvice_TypeOfReflectionParameter` specifies its user-defined type. Since the user defines the type of the reflection parameter, it also becomes his responsibility to provide an argument for it. Due to a limitation of the C programming language, i.e., C does not accept struct literals as arguments, the interface contains three methods[7] to specify the argument to the reflection parameter for a kind of advice: The method `KindAdvice_IsSetupNeededForArgumentToReflectionParameter` specifies that a setup (for a struct literal) is needed; the method `KindAdvice_SetupOfArgumentToReflectionParameter` specifies that setup when needed; and the method `KindAdvice_ArgumentToReflectionParameter` provides the actual reflection argument, which can refer to the elements introduced in the setup when present. The advice is implemented by the method `KindAdvice_Code`. The advice can access the reflective information, since its parameter called `reflection` contains the instance name of the reflection parameter in C.

Aspects programmed in AspectKoala usually do not implement interface `IComponentAspect` directly, but inherit from the abstract `ComponentAspect` class. This class returns `false` on all Boolean methods of `IComponentAspect`. As a result, only the features that are used by an aspect must be implemented by overriding the appropriate methods. This considerably simplifies the implementation of aspects.

[6] Our framework is not limited to one aspect, since an aspect can instantiate multiple other aspects and control their precedence.

[7] Since our framework is targeted at a resource limited environment, the `IComponentAspect` interface also contains two methods to factor out the commonalities in the setups of the reflection argument for the before, around, and after advices: `IsCommonSetupNeededForArgumentToReflectionParameter` and `CommonSetupOfArgumentToReflectionParameter`.

```
interface IComponentAspect
{
//PreCondition: true
bool JoinPointHasBeforeAdvice(JoinPoint jp);

//PreCondition: JoinPointHasBeforeAdvice(jp)
bool BeforeAdvice_UsesReflection(JoinPoint jp);

//PreCondition: BeforeAdvice_UsesReflection(jp)
Type BeforeAdvice_TypeOfReflectionParameter(JoinPoint jp);

//PreCondition: BeforeAdvice_UsesReflection(jp)
bool BeforeAdvice_IsSetupNeededForArgumentToReflectionParameter(JoinPoint jp);

//PreCondition: BeforeAdvice_IsSetupNeededForArgumentToReflectionParameter(jp)
Code BeforeAdvice_SetupOfArgumentToReflectionParameter(JoinPoint jp);

//PreCondition: BeforeAdvice_UsesReflection(jp)
Code BeforeAdvice_ArgumentToReflectionParameter(JoinPoint jp);

//PreCondition: JoinPointHasBeforeAdvice(jp)
Code BeforeAdvice_Code(JoinPoint jp, string reflection);

//similar for Around and After Advice
...

//Factor out commonality of Before, Around, and After advice to save
resources
//PreCondition: BeforeAdvice_UsesReflection(jp)||
//              AroundAdvice_UsesReflection(jp)||
//              AfterAdvice_UsesReflection(jp)
bool IsCommonSetupNeededForArgumentToReflectionParameter (JoinPoint jp);

//PreCondition: IsCommonSetupNeededForArgumentToReflectionParameter(jp)
Code CommonSetupOfArgumentToReflectionParameter(JoinPoint jp);
}
```

Figure 6.8 *The part of the* `IComponentAspect` *interface relevant for reflection.*

```
public class MyAspect : ComponentAspect
{
public override bool JoinPointHasBeforeAdvice(JoinPoint jp)
{ return true; }

public override bool BeforeAdvice_UsesReflection(JoinPoint jp)
{ return true; }

public override Type BeforeAdvice_TypeOfReflectionParameter (JoinPoint jp)
{ return typeof(int); }

public override Code BeforeAdvice_ArgumentToReflectionParameter (JoinPoint jp)
{ return Code.IntValue(LookUpTableJP2UID(jp)); }

public override Code BeforeAdvice_Code (JoinPoint jp, string reflection)
{ return Code.Skip; }

//identical for After Advice
...
}
```

Figure 6.9 *Example aspect in AspectKoala that uses a unique identifier per join point as reflective information.*

Figure 6.9 contains an example aspect written in AspectKoala. This aspect specifies that all join points have a before advice in `JoinPointHasBeforeAdvice`. In `BeforeAdvice_UsesReflection` and `BeforeAdvice_TypeOfReflectionParameter` it specifies that reflection is used in the before advices of all join points and that the type of the reflection parameter is always integer, respectively. The implementation of `BeforeAdvice_ArgumentToReflectionParameter` specifies that the argument to the reflection parameters depends on the particular join point: The integer value as generated by the lookup-table method that assigns a unique identifier to each join point. Note that this lookup table does not end up in the target code. The table is only needed off-line or post-mortem to translate the integer values used on the target into join point information in the verbose human readable form needed for manual analysis. Since the implementation of `AfterAdvice_ArgumentToReflectionParameter` is identical, the before and after advices associated with the same join point will be called with the same integer value of the reflection parameter. Finally, the aspect specifies in `BeforeAdvice_Code` that the before advice of all join points is equal to the skip-statement.

6.4.4 Overhead

AspectKoala enables us to experiment not only with the presence of a reflection parameter but also with the type of this parameter. We wrote four aspects to add tracing[8] before and after the execution of every join point in an existing configuration. The first aspect adds tracing without reflection and acts as our baseline. The second aspect adds tracing with a unique identifier per join point and is (partly) shown in Figure 6.9. The third aspect adds tracing with only the function, interface, and component name per join point. The fourth aspect adds tracing with a general-purpose reflection type parameter comparable to AspectJ's `thisJoinPointStaticPart`.

We spent considerable effort to minimize the overhead of these aspects. For example, for the last two aspects as mentioned in the previous section, we used look-up tables to factor out commonalities and thus to save resources. However, we only looked at one specific but representative product, a Philips television, in this experiment. And, we did not include custom attributes in the last aspect. The numbers we present are thus not hard values, but mere indications.

reflection type	**increase**	
	CPU load	**ROM footprint**
unique identifier	4.7 %	1.8 %
names	5.5 %	10 %
general-purpose	5.5 %	36 %

Table 6.1 *The percentages of increase in CPU load and ROM footprint due to different types of reflection parameters.*

With these aspects, we measured the increase in CPU load and ROM footprint (i.e., the size of the binary code and the constant/static data) due to reflection. The results are presented in Table 6.1. Note that this increase is on top of the increase already introduced by tracing for a release build as measured for the baseline, our first aspect. Since we consider an increase above 15% in ROM footprint as not acceptable for consumer products, we cannot apply a general-purpose reflection parameter in this resource constrained environment. Furthermore, as shown in Table 6.1, the

[8] Only the infrastructure needed to trace is added, no tracing information is collected. I.e., the advice contains no statements.

difference between “unique identifier” and “names” is considerable. This justifies that we put the user in control of the reflective information such that the user can incorporate domain knowledge to save memory, CPU cycles, and bandwidth for his particular application.

6.4.5 Reliability

How to handle access before initialization is a reliability concern that affects all components. Although one can easily describe how to handle access before initialization in general, it is currently handled per component. Even worse, this handling even differs between components. With aspect orientation, we were able to write three different strategies to handle access before initialization. The first strategy asserts that a component is not accessed before initialization; the second strategy ignores accesses when the component is not yet initialized; and the third strategy calls the initialization code when the component is accessed but not initialized before. With these strategies:

1. We could ensure that all components handle access before initialization identically.
2. We could separate the initialization implementation from the functional implementation. This not only reduces the lines of code by 2% (the lines in Figure 6.2), but also makes reuse more likely. Reuse becomes more likely since the reuse environment has only to match either the initialization requirement (to reuse one of the three initialization aspects) or the functional requirement (to reuse one of the components), but not both.
3. We could postpone the decision for an initialization strategy from implementation to integration. In other words, weaving of the initialization strategy and the components’ functionality is changed from manually work of the programmer to an automatic process executed by the aspect-oriented framework.

6.5 Conclusions

In this chapter, we indicated, for the context of consumer products, that resources are constrained and that aspect orientation improves the reliability of such products. We also illustrated that reflection on join points is crucial for aspect orientation. Unfortunately, all approaches [Lämmel, 2005] [Rajan, 2005] [AspectJ] [JBoss] currently used in aspect orientation to handle reflective information consume more resources than are affordable for consumer products. Since we wanted to apply aspect orientation in this resource constrained environment, we had the following objective: develop a resource effective design for reflection on join points. We achieved our objective by explicitly passing dedicated reflection parameters. Furthermore, we enabled the user to control the reflective information, such that the user can incorporate domain knowledge to save memory, CPU cycles, and bandwidth for his particular application. We showed that, with user-controlled reflection on join points, we are able, for a given application, to reduce resource usage to an affordable level without sacrificing the necessary expressive power. Putting the user in control of the reflective information is a fundamental difference between AspectKoala and all other aspect-oriented frameworks. Our experience with a reliability concern implemented using AspectKoala shows that reliability indeed profits from aspect orientation.

Note: This chapter contains earlier published parts from Pierre van de Laar, *Aspect Orientation Enables Differentiation with Software*, Philips Software Conference, 2006, and Pierre van de Laar and Rob Golsteijn, *User-Controlled Reflection on Join Points*, Journal of Software, Vol. 2, No. 3, pp. 1-8, September 2007.

6.6 References

[Cazzola, 2004] W. Cazzola. *SmartReflection: Efficient Introspection in Java*. Journal of Object Technology, pp. 117-132, Vol. 3 (11), 2004

[Chiba, 1997] S. Chiba. *Implementation Techniques for Efficient Reflective Languages*. University of Tokyo, Technical Report 97-06, 1997

[Dijkstra, 1976] E.W. Dijkstra. *A Discipline of Programming*. ISBN 0613924118, 1976

[Jackson, 2004] A. Jackson and S. Clarke. *SourceWeave.NET: Cross-Language Aspect-Oriented Programming*. Generative Programming and Component Engineering (Vancouver, Canada), pp. 115-135, 2004

[Lämmel, 2005] R. Lämmel and K. De Schutter. *What does aspect-oriented programming mean to Cobol?* Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05) (Chicago, Illinois, USA), pp. 99-110, 2005

[Lieberherr, 1999] K. Lieberherr, D. Lorenz, and M. Mezini. *Programming with Aspectual Components*, Technical Report, NU-CSS-99-01, March 1999
Available at http://www.ccs.neu.edu/research/demeter/biblio/aspectual-comps.html

[Rajan, 2005] H. Rajan and K.J. Sullivan. *Classpects: Unifying Aspect- and Object-Oriented Language Design*. Proceedings of the 27th International Conference on Software Engineering **(**St. Louis, Missouri, USA), 2005, pp. 59-68

[Rogerson, 1997] D. Rogerson. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, ISBN 1-572-31349-8, 1997

[Siegel, 2000] J. Siegel. *CORBA 3: Fundamentals and Programming*. OMG Press, ISBN 0-471-29518-3, 2000

[van de Laar, 2006] P. van de Laar. *Combining component-based and aspect-oriented software development in a resource constrained environment*. Philips, Technical Note PR-TN 2006/00648, 2006. Available at http://www.extra.research.philips.com/publ/rep/nl-ur/PR-TN2006-00648.pdf

[van Ommering, 2004] R. van Ommering. *Building Product Populations with Software Components*. PhD Thesis, University of Groningen, the Netherlands, ISBN 90-74445-64-0, 2004. Available at http://irs.ub.rug.nl/ppn/27516956

[Arnold, 1996] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, ISBN 0201634554, 1996

[AspectJ] *AspectJ Home Page*. http://www.aspectj.org

[Griswold, 1999] W. Griswold, Y. Kato, and J. Yuan. *AspectBrowser: Tool Support for Managing Dispersed Aspects*. CS1999-0640, 1999
Available at http://citeseer.ist.psu.edu/griswold99aspect.html

[JBoss] *JBoss AOP Home Page*. http://labs.jboss.com/jbossaop

[Spinczyk, 2005] O. Spinczyk, D. Lohmann, and M. Urban. *Advances in AOP with AspectC++*. Software Methodologies, Tools and Techniques (SoMeT 2005) (Tokyo, Japan), pp. 33-53, 2005

Chapter 7

Behavior modeling

Author: Jozef Hooman

Abstract: We present a method to capture the desired dynamic behavior of an embedded system. The method is based on the use of structured state machines to obtain scalable and executable models. The modeling approach turns out to be useful for the early detection and removal of requirements errors, which are an important source of unreliability and project failures. Requirements can be validated easily by means of a visualization of user-perceived behavior. Furthermore, models can be refined to include the high-level system architecture which makes it possible to check consistency and conformance to the requirements. We show that also performance aspects can be modeled. The method is illustrated by examples of a TV and a car entertainment system.

7.1 Introduction

This chapter describes an approach to model desired system behavior. Starting point of this work was the aim to use a model of desired user-perceived behavior for runtime error detection, as described in Chapter 8. For most industrial systems such a model is not available. Neither it is easy to derive such a model from existing requirements descriptions which, in common industrial practice, are often only partially available and distributed over many documents and databases. Moreover, these documents often focus on technical implementation details and hardly describe the user-perceived behavior which is central in the Trader awareness concept. Hence, part of Trader work was aimed at capturing this behavior in a model.

The importance of a proper requirements description for system reliability is well known from the literature. Studies indicate that requirements errors are the greatest source of defects and quality problems [Nakajo, 1991] [Standish, 1995]. It is also often observed that errors made in the requirements phase are extremely expensive to repair. On top of Boehm’s “Software Defect Reduction Top 10 List” is the observation that repairing software problems after product release is often 100 times more expensive than finding and fixing them during the requirements and design phase [Boehm, 2001]. The conclusion of the field study reported in [Hofmann, 2001] states that getting requirements right might be the single most important and difficult part of a software project. Hence, our work on requirements modeling is not only beneficial for runtime error detection, but should lead to a reduction of requirements errors, thereby improving both reliability and project effectiveness.

Since the TV domain is our source of inspiration and the focus is on user-perceived reliability, the first aim was to make a model that captures the user view of a particular type of TV in development. The model should capture the relation between user input, via the remote control and buttons on the TV, and output, via images on the screen and sound.

A few first experiments with the specification of the control behavior of a TV indicated that the use of state machines is quite intuitive and close to informal descriptions. Since flat state machines would quickly lead to scalability problems, structuring mechanisms such as hierarchy and parallelism are indispensable. Our experiments also revealed that it was very easy to make modeling errors. Constructing a correct model was more difficult than expected. Getting all the information was not easy, and many interactions were possible between features. Examples are relations between dual screen, Teletext and various types of on-screen displays that remove or suppress each other. Hence, our approach is based on executable models to allow quick feedback on the user-perceived behavior and to increase the confidence in the fidelity of the model. Model execution allows early experiments with the system being designed, as already observed in [Selic, 2003].

Besides the control behavior, a TV also has a complex streaming part with a lot of audio and video processing. Typically, this gets most attention in the requirements documentation. We would like to model this on a more abstract level, with emphasis on the relation with the control part. These considerations led to the use of Matlab/Simulink [Mathworks, 2009]. The Stateflow toolbox of Simulink is used for the control part and the Image and Video Processing toolbox for the streaming part.

The Stateflow language contains the required structuring mechanisms to allow the specification of complex systems. In addition, it provides convenient possibilities to view state transitions during simulation. During simulation the external input-output behavior can be visualized in parallel with an animation of the state transitions of the specification, which is very convenient when validating and debugging the requirements. In this way, requirements can be improved early in the development process and at the appropriate level of abstraction, instead of doing this late at the code level. A related approach can be found in the Statemate system which allows the animation of externally visible behavior of specifications expressed in Statecharts [Harel, 1990]. A similar method based on the formal notation of Labeled Transition Systems has been described in [Magee, 2000].

In general, our models consist of the following ingredients:

- A visualization of the system input which can be used to insert events and data during model simulation. This input is coded into numbers and stored in a buffer to decouple real-time user input and the execution of the model using simulation time.
- A Stateflow diagram which periodically reads the input buffer and generates appropriate events and data for the main specification.
- A main Stateflow diagram which specifies the control of the system and generates data for other parts of the model.
- A Simulink block which models physical aspects of the system, based on control data. It may also send (sensor) data to the control diagram.
- A visualization of the system output.

Note that the requirements model is intended to describe the desired relation between external input and output of the system only. The internal structure of the model does not reflect the architecture of the system, it is just one possible way to describe the input-output relation. It is, however, possible to make a refined model which includes the main components of the system and their behavior. Then the conformance of the architectural design with the system requirements can be tested.

The main benefits of our modeling approach are:

1. Early detection of incomplete, ambiguous, unnecessary, and inconsistent requirements.
2. Quick feedback on new functionality and feature interaction.
3. Localization of requirements in a single model.

Moreover, systems tests can be derived from the model. In the remainder of this chapter we present our requirements modeling approach in the TV domain in Section 7.2. Section 7.3 describes the application to a car entertainment system, including a model of the architecture and a more detailed performance model. Note that the models presented here are slightly adapted and sometimes simplified for confidentiality reasons. Concluding remarks can be found in Section 7.4.

7.2 Modeling user-perceived TV behavior

When applying our approach in the TV domain, we concentrated on the global control of a TV and the user-perceived behavior, that is, the relation between input via a remote control (and/or TV buttons) and output via a TV screen and speakers. We ignored many aspects such as the installation procedure, image and sound quality, view modes, and external devices, but still observed that it was far from easy to obtain a correct model. To illustrate this, we show an intermediate model which might seem reasonable at first sight, but contains several errors.

A screenshot of the model during simulation is shown in Figure 7.1.

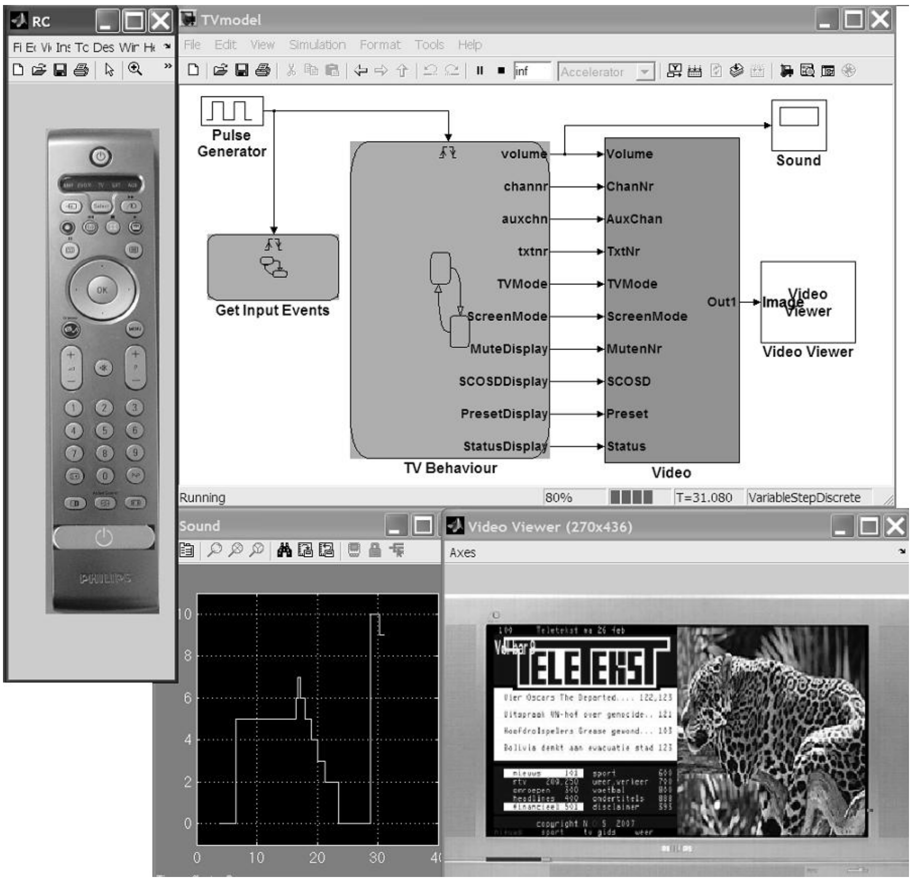


Figure 7.1 *TV model during simulation*

The Simulink model, depicted in the upper part of Figure 7.1, contains a Stateflow block *Get Input Events* which controls model input. It starts by calling a Matlab function which displays the photograph of the remote control shown on the left-hand side of Figure 7.1. For convenience, we have inserted a picture of a TV button on the lower part of this photograph. Next, this Matlab function reads the position of the cursor when a mouse click occurs. This position is translated into a number which codes the key on the remote control. Key codes are put into a buffer which is read by the *Get Input Events* block. On each pulse generated by the *Pulse Generator* block (which is part of the Simulink library), block *Get Input Events* tries to read a key code from the buffer. If the buffer is not empty, an event corresponding to the code is generated and Stateflow block *TV Behavior* checks whether it can execute transitions. For instance, code 12 leads to event *VolumeUp*, code 13 to *VolumeDown*, and code 100 to *MainsOnOff*.

Based on the received events, block *TV Behavior* sets a number of variables, such as the volume level, the channel number, the TV mode (menu, Teletext or TV) and the screen mode (off, standby, single screen or dual screen). These variables are used by block *Video* which displays a TV screen with the resulting video output, as shown in the lower right corner of Figure 7.1. The volume level is also fed into a so-called scope block which graphically shows this level (see the bottom left part of Figure 7.1). Block *Video* is constructed by means of the Video and Image Processing Blockset. The top-level decomposition of this block is shown in Figure 7.2.

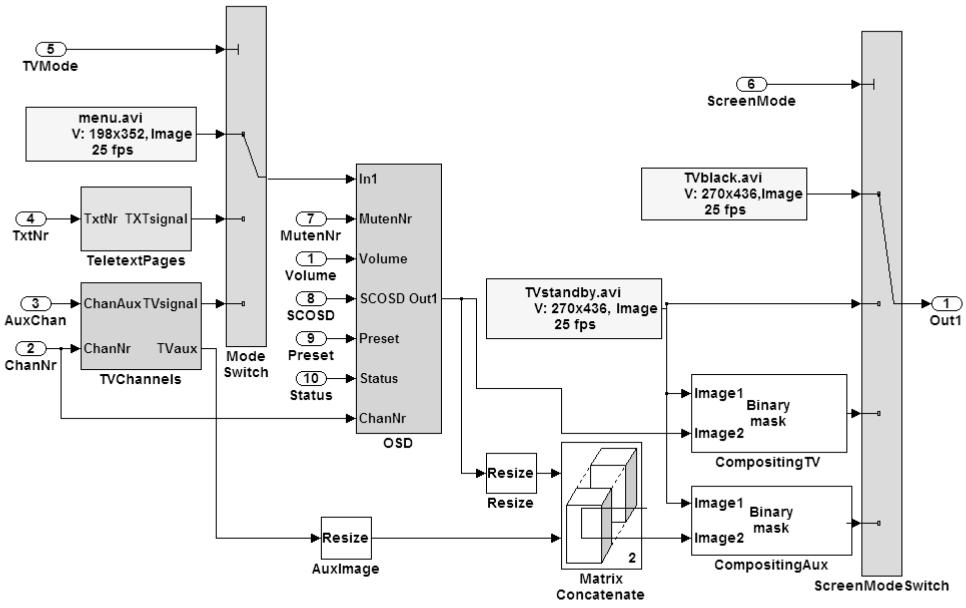


Figure 7.2 *Video part of TV model*

Next we describe Stateflow block *TV Behavior* in more detail. The top-level state diagram is shown in Figure 7.3. Note that state *On* consists of two parallel states called *InputTransform* and *TVModes*, as indicated by the dashed lines around these states. In state *InputTransform* certain internal events are generated based on external input events. For instance, external events that indicate a volume or channel change lead to an internal *WakeUp* event.

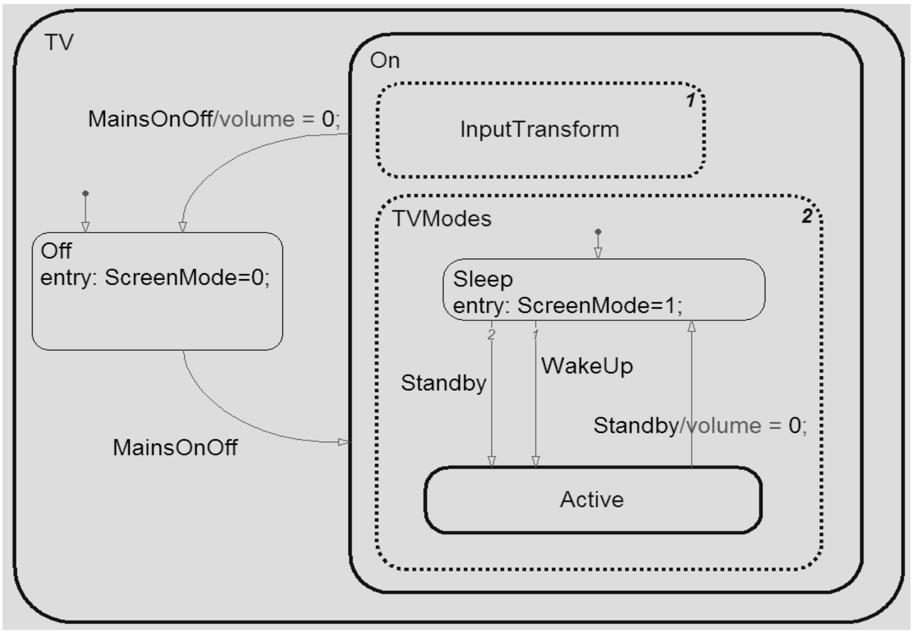


Figure 7.3 *Top-level state diagram of TV behavior*

State *TVModes* has a substate *Active* which is decomposed into three parallel states, as shown in Figure 7.4.

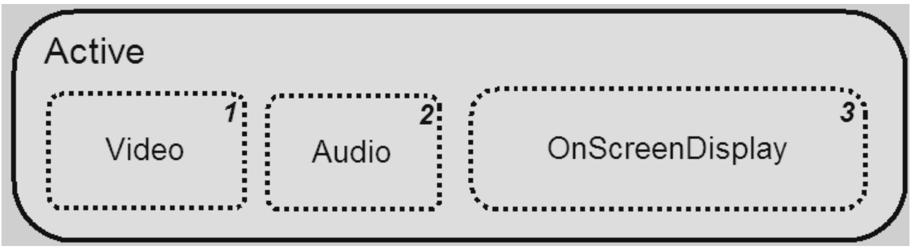


Figure 7.4 *Parallel decomposition of state Active*

Figure 7.5 shows the details of substate *Video* of state *Active*. This diagram is a translation of a large table in a TV requirements document. Our diagram is more concise than the table, but the main advantage is the possibility to execute the state machine, to experiment with different sequences of input events, and to obtain immediate feedback from our visualization of the output. For instance, only during model simulation it became clear that if the TV is in dual screen mode with TV on both screens, it is not easy to obtain Teletext in one of the screens. The user first has to go to single screen mode, switch there to Teletext, and then go back to dual screen mode. This interaction between dual screen mode and Teletext is difficult to detect when looking at the table in the requirements document.

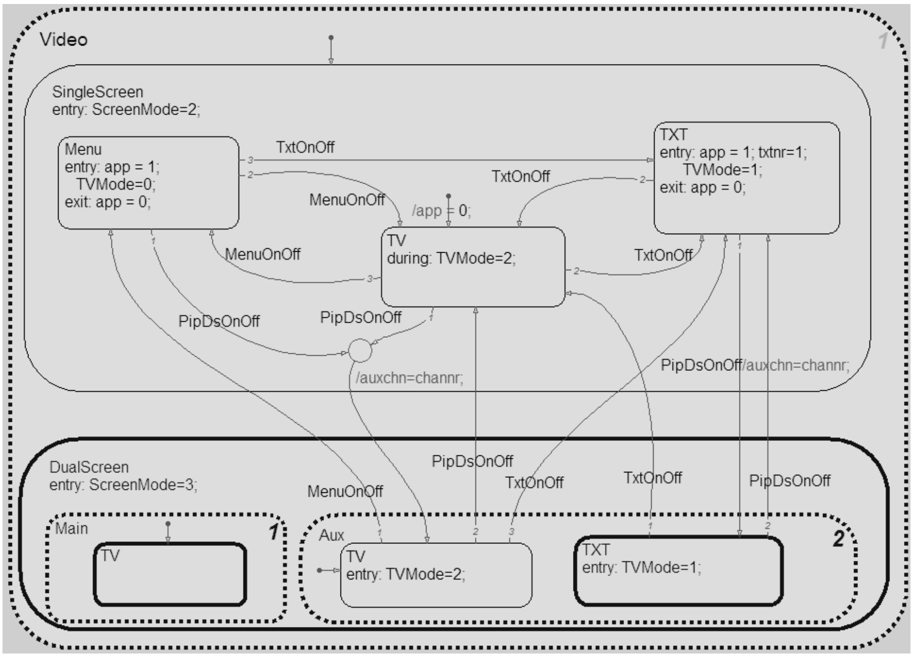


Figure 7.5 *Detailed view of state Video*

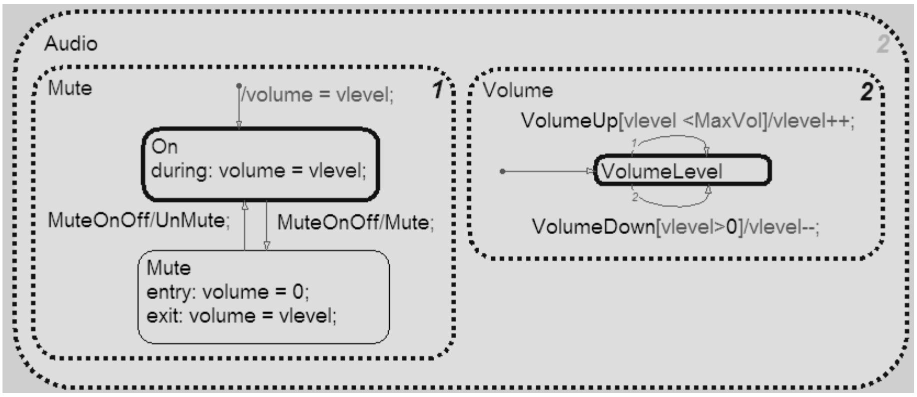


Figure 7.6 *State diagram of audio part*

Since the documentation of the audio part of the TV was not available, we used own knowledge to model substate Audio of state Active, as depicted in Figure 7.6.

The audio part is decomposed into two parallel states that deal with mute behavior and volume changes, respectively. Simulation revealed that this leads to undesirable behavior; if volume is muted, the user might press the volume up button, assuming this also unmutes the volume. According to the model shown here, there is no implicit unmute but the volume up button increases some internal volume value. This might lead to the behavior depicted in the volume scope in the lower-left part of Figure 7.1, where we mute with a low volume level, try to unmute by pressing the volume up button many times, and then unmute, leading to a very large volume level. Clearly, such errors are easy to repair in the model, whereas a correction in the final implementation would be much more costly.

The state diagram for the *OnScreenDisplay* state is not shown here. It is rather large because there are many parallel possibilities for displaying text and icons on the screen, such as mute/unmute icons, volume and channel status, and preset information. The rules for the on screen display were specified in much detail, but when making the model a number of conflicting sentences in the requirements document were detected. This concerned, e.g., the priority of displays, expressing when one display should suppress another. To resolve such conflicts, designers and testers may make different choices, leading to problems in the test phase.

7.3 Modeling a car entertainment system

In this section, we apply our modeling approach to the domain of car entertainment systems where multiple media devices (CD player, iPod, MP3 player, SD card, radio), navigation system, microphone, telephone, and car-specific chimes (e.g., given warnings such as “door open”) are connected to car speakers and headphones. A straightforward adaptation of our TV models to the requirements modeling of such systems is described in Section 7.3.1. In Section 7.3.2 we show how this requirements model can be refined to include the system architecture. Finally, Section 7.3.3 describes a further refinement to analyze performance aspects.

7.3.1 Requirements model of a car entertainment system

The high-level TV model of the previous section has been adapted to deal with a car entertainment system. The main differences are:

- The user input has been visualized by means of a small GUI which represents media selection of multiple users (e.g. on front and back seats of the car), the presence of phone calls, navigation voice, chimes, etc.
- The Stateflow model, called *Car Entertainment Spec* here, especially concentrates on limitations concerning the concurrency of media streams. For instance, the number of available tuners leads to constraints on the number of parallel available analog or digital channels and the availability of traffic information.
- The output has been visualized by a picture of a car with textual indications about the output from speakers and headphones. For the moment, the display is used to show restrictions on the possible input combinations.

A snapshot of the model during simulation is shown in Figure 7.7.

As already observed in the TV domain, the execution of such a requirements model leads to a number of questions about missing or unclear requirements. In this case, the requirements that were captured in DOORS described many requirements on individual media and data streams, but the interaction of these streams and the constraints on the allowed concurrency were unclear. An executable model makes this more explicit and improves the communication with stakeholders. For instance, the executable model of car entertainment triggered discussions about the requirements on use case switching, e.g., how to switch from one media stream to another; an aspect that was absent in the requirements captured in DOORS.

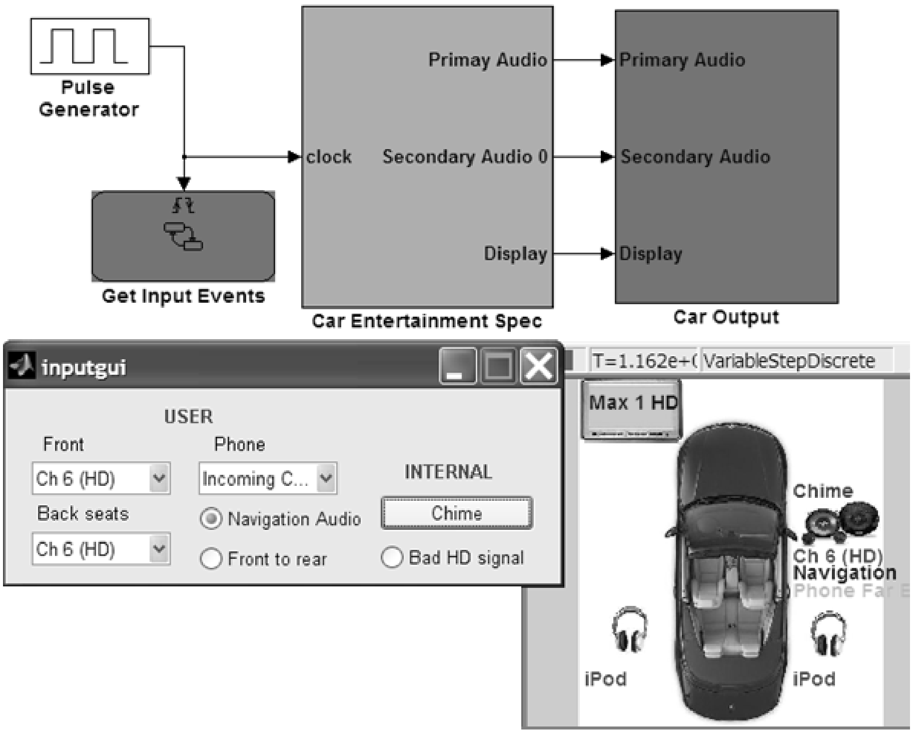


Figure 7.7 *Requirements model car entertainment*

7.3.2 Design model

Next the requirements model is refined to include the high-level architecture of the system. The input and output visualization remain unchanged, and the aim is to maintain the input – output relation as specified by the requirements model. The main difference is that the media sources, such as iPod and MP3 player are represented explicitly as sources that transmit a characteristic value. This leads to the top-level view depicted in Figure 7.8.

Depending on the usage scenario selected in the input GUI, the characteristic values of input sources are selected and routed through the architecture, possibly mixed, and finally forwarded to the output visualization. Simulink block *Car Entertainment* contains a high level view of the architecture, as shown in Figure 7.9; all components depicted there are further refined to the level of data flow diagrams.

A clear advantage of such an executable model is that it immediately shows inconsistencies between component interfaces and errors in data flow models. Typically, various architectural aspects are distributed over several documents and it is very difficult to detect errors by inspection and to keep all information consistent under a continuous stream of changes. This is much easier when all aspects are combined in a single executable model that shows the impact of changes on the user-perceived behavior.

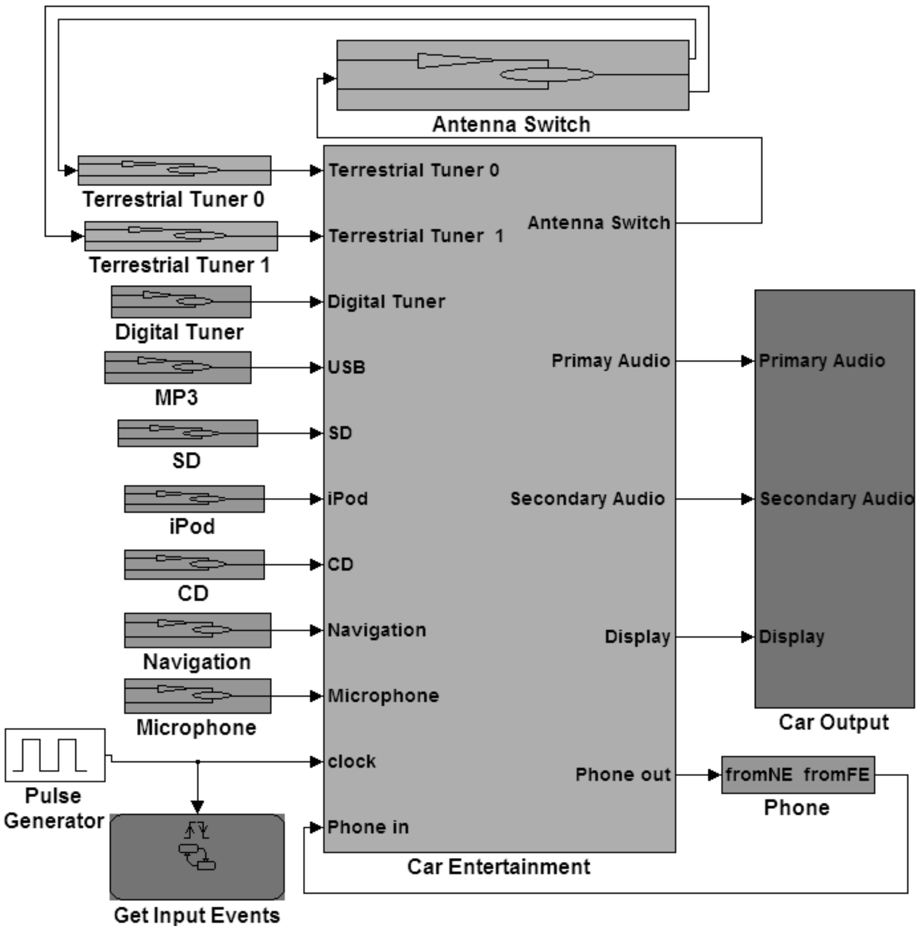


Figure 7.8 *Top-level view of refined car entertainment model*

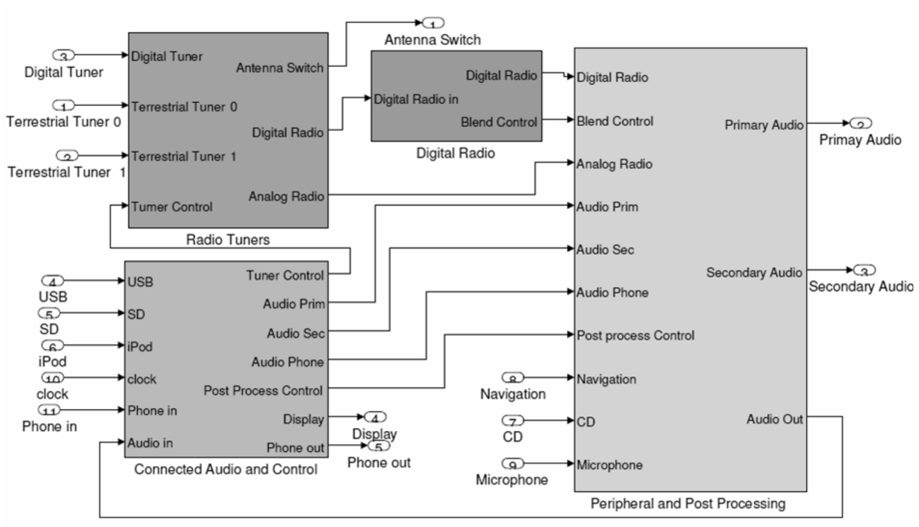


Figure 7.9 *Architecture car entertainment system*

7.3.3 Modeling performance aspects

The next refinement of the model aims at the evaluation of performance aspects. The main goal is to investigate the bandwidth of the bus to external memory and the schedulability of certain tasks, such as echo cancellation and a number of MP3 decoders, on a particular processor for different usage scenarios. To this end, we used the TrueTime package [Lund, 2007] which provides a network block, which can be used to model a bus to external memory, and a node block which can communicate with a network block and other Simulink blocks. On the node block a number of tasks can be programmed and one can specify the scheduling policy, priorities, etc. During simulation, it is easy to visualize the utilization of the node and the traffic on the memory bus, as depicted in the lower part of Figure 7.10.

Such a dynamic model, including use case switches and concurrency, improves current practice which is usually based on Excel sheets for static situations only.

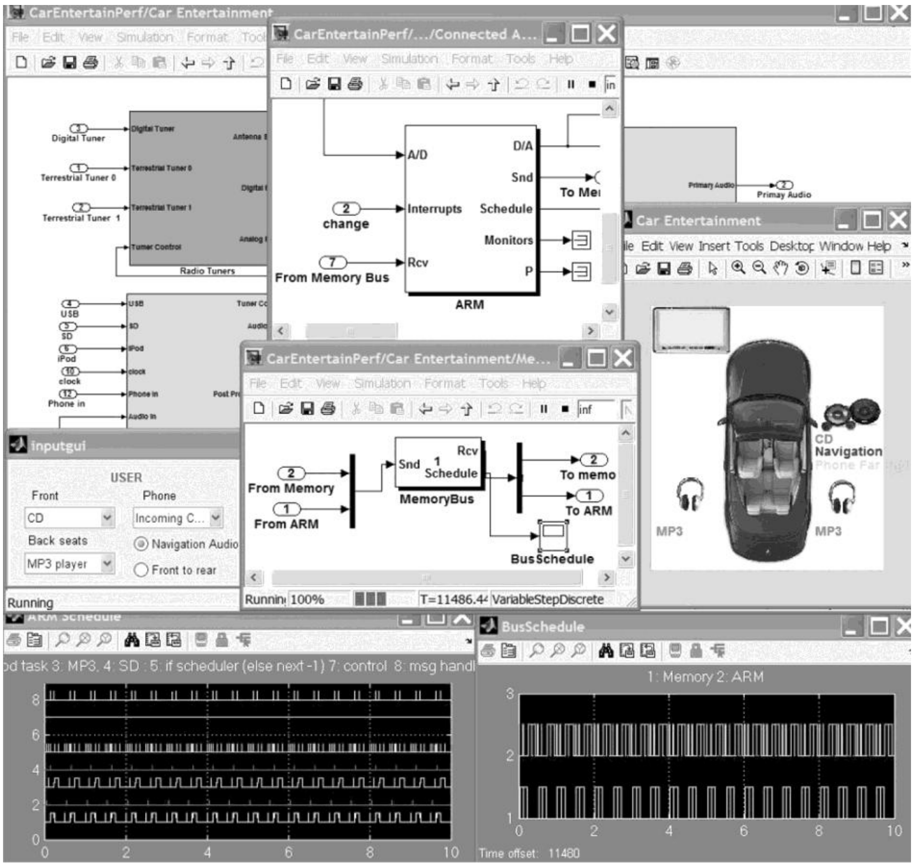


Figure 7.10 *Performance model of car entertainment system*

7.4 Concluding remarks

We have described an approach to create an executable requirements model of an embedded system and to visualize the external, user-perceived system behavior. The first application to the TV domain showed that making such a model was more difficult than expected, not only because of gaps in the existing documentation, but also because the large number of decisions that had to be taken on detailed functionality. Making an executable model raises many questions about details and subtle interactions between features that did not turn up when reading the documentation.

The approach has also been applied to obtain a high-level model for the MPlayer [MPlayer, 2007]. This model has been used to experiment with model-based error detection, as reported in Chapter 8. With the increasing use of models, also the quality of these models becomes increasingly important. Since models change frequently, we have used our MPlayer models to experiment with the tool Reactis to generate test scripts to check conformance after model changes [Reactis, 2009]. This tool can also be used to validate model properties. Related functionality is provided by the Simulink Design Verifier, which also allows the formal proof of properties. Unfortunately, the current version of this toolbox does allow checking of Stateflow models.

The modeling of the car entertainment system showed that it was easy to apply the approach quickly to a slightly different domain. The models presented here were developed during six weeks of part-time work and they turned out to be an effective means to capture domain knowledge and to identify weak spots in requirements and design. Current work includes the application of the modeling technique to an electron microscope in the context of ESI-project Condor [Condor 2009].

In general, the visualization of the user view on input and output of the model turned out to be very useful to detect faults in requirements as early as possible. This includes undesired feature interactions and missing, ambiguous or even contradictory requirements. As we have shown in the car entertainment case, the approach allows refinements of the model to include the global system architecture and performance aspects. This makes it possible to check conformance of the design with respect to the requirements. The requirements model may also be used to generate system tests for the final implementation.

7.5 References

[Boehm, 2001] B. W. Boehm, V. R. Basili. *Software Defect Reduction Top 10 List*. IEEE Computer, Vol. 34, No. 1, pp. 135-137, 2001

[Condor 2009] *The Condor project: System performance and evolvability*. Embedded Systems Institute, http://www.esi.nl/condor/, 2009

[Harel, 1990] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sheman, A. Shtul-Trauring, and M. Trakhtenbrot. *STATEMATE A Working Environment for the Development of Complex Reactive Systems*. IEEE Transactions on Software Engineering, Vol. 16, pp. 403-414, 1990

[Hofmann, 2001] H. F. Hofmann, F. Lehner. *Requirements Engineering as a Success Factor in Software Projects*. IEEE Software, Vol. 18, No. 4, pp. 58-66, 2001

[Lund, 2007] Lund University. *TrueTime: Simulation of Networked and Embedded Control Systems*. http://www.control.lth.se/truetime/, 2007

[Magee, 2000] J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer. *Graphical Animation of Behavior Models*. Proc. 22nd Int'l. Conf. Software Engineering (ICSE 2000), pp. 499-508, 2000

[Mathworks, 2009] *The Mathworks: Matlab/Simulink*. http://www.mathworks.com/, 2009

[MPlayer, 2007] MPlayer: *Open source media player*. http://www.mplayerhq.hu/ , 2007

[Nakajo, 1991] T. Nakajo, H. Kume. *A Case History Analysis of Software Error Cause-Effect Relationships*. IEEE Transactions on Software Engineering, Vol. 17, No. 8, pp. 830-838, 1991

[Reactis, 2009] *Reactive Systems: Model-Based Testing and Validation with Reactis*. http://www.reactive-systems.com/, 2009

[Selic, 2003] B. Selic. *The Pragmatics of Model-Driven Development*. IEEE Software, Vol. 20, No. 5, pp. 19-25, 2003

[Standish, 1995] The Standish Group. *Chaos*. http://www.standishgroup.com/, 1995

Chapter 8

Model-based error detection

Authors: Jozef Hooman, Somayeh Malakuti

Abstract: We present a method to detect runtime errors using models of desired behavior. An experimental Linux-based framework has been developed in which a software application and a (partial) specification of its desired behavior can be inserted. Specifications are expressed as structured state diagrams. The framework allows both time-triggered and event-triggered comparison of actual and desired behavior.

8.1 Introduction

Runtime error detection is part of the runtime awareness concept introduced in Section 1.2.4. The aim is to check during system execution whether the actual behavior of the system complies with the specified behavior and to report an error when an undesirable deviation is detected. This is often called *runtime verification*. To use a particular runtime verification technique on an industrial system, an engineer should take the following steps:

Step 1: Identify properties

Decide which important aspects of the system have to be monitored and informally describe the desired properties that should be checked. For a complex price-sensitive device such as a TV, it is cost-inhibitive to check the complete system behavior at runtime. Hence, a choice has to be made based on the likelihood of errors, the impact of errors on the user, and the costs of correcting those errors at runtime.

Step 2: Specify properties formally

Become familiar with the used specification language (e.g., temporal logic formalisms, process algebra, regular expressions, Boolean assertions, diagrams) and express the properties in this language. Specifications should be easy to write and to inspect by industrial engineers. In the context of embedded systems, desired properties may refer to timing or resource usage, so it should be possible to express such properties in a convenient way.

Step 3: Validate specification

Validate the correctness of the specification. To avoid false errors, it is important that the specification is consistent and captures the intended behavior. Note that for runtime verification there is no need to specify full system behavior; the specification may cover only an essential part of the system behavior or focus on a particular component.

Step 4: Add system observations

Investigate which information is needed during system execution to evaluate the specified properties (e.g., values of variables, occurrence of events, or memory usage). Next, instrument the system such that this information becomes observable. Typically, the software is instrumented at certain monitoring points with code that provides observations.

Step 5: Add property evaluation

Add monitoring code that checks whether the specified properties hold at runtime and which reports an error in case of a deviation from the specification.

Depending on the technique used, some of these steps may be automated. Note that runtime verification techniques are not limited to software, but the principles can be applied to a complex system with both hardware and software parts.

The approach described in this chapter starts from specifications expressed in Stateflow of Matlab/Simulink [Mathworks, 2009], a widely used graphical notation, especially to describe the discrete control of continuous-time systems. To support scalability, Stateflow includes all structuring mechanisms of the classical Statecharts language, such as hierarchy, history connectors, and parallelism with event-based communication. We use the Stateflow notation to obtain executable specifications of the desired behavior of a certain system.

To get insight in model-based error detection, first this concept itself has been modeled in Simulink, as will be described in Section 8.2. To allow quick experimentation with model-based error detection and the awareness concept of Trader, we present in Section 8.3 the design and implementation of a monitoring framework. The application of this framework to the open source media player MPlayer is described in Section 8.4. Related work can be found in Section 8.5. Section 8.6 contains concluding remarks.

8.2 Simulating the main concepts

In our approach of model-based error detection, a particular System Under Observation (SUO) is monitored and its behavior is compared to a Stateflow model of its desired behavior. Before implementing this concept, it has been modeled in Matlab/Simulink to investigate the main concepts. A high-level view is depicted in Figure 8.1, illustrating the comparison of the volume level of a TV.

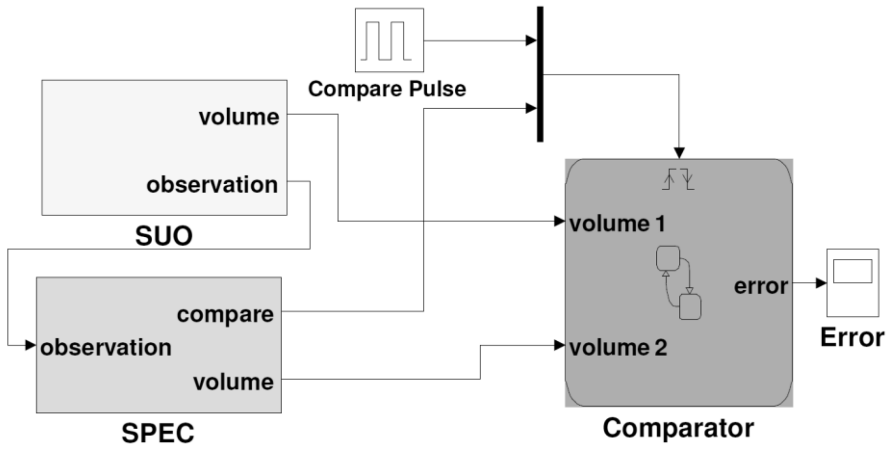


Figure 8.1 *Model of model-based error detection*

The SPEC block contains a Stateflow model of the desired behavior, according to the modeling approach described in Chapter 7. This model is a hierarchical state machine which describes the main states of the TV (whether it is on, off, muted, etc.) as far as they affect the volume level. Relevant system observations, such as input events that affect the volume level, are forwarded from the SUO to the SPEC. In the TV case this concerns, for instance, a press on remote control buttons for on, off, mute, volume up or volume down.

To experiment with the comparison strategy, we also made a Stateflow model for the SUO. This is a more detailed architectural model with timing delays to simulate the execution time of internal actions. This model also includes user input (e.g., via a remote control) and output to the user (here via a TV screen and speakers). The SUO sends the actual volume level to the Comparator, whereas the SPEC sends the desired level. Basically, the Comparator compares the two values periodically, triggered by a timer – the Compare Pulse in Figure 8.1. Besides this time-triggered comparison, the SPEC can start and stop the comparison by the *compare* signal.

Simulations with this model of error detection clarified a number of issues:

- Our initial specification models had to be adapted to include best-case and worst-case execution times in the SPEC block. To capture non-determinism in the SUO behavior (e.g., due to timing variations), we extended the SPEC model with intermediate states that represent situations where the system is in transition from one mode to another. To avoid that worst case execution times accumulate, these intermediate states in the SPEC, that often model worst case execution times, can be exited when the SPEC receives new observations from the SUO that indicate the end of the previous action.
- Part of the comparison strategy is included in the specification model, to be able to use domain knowledge about processing delays and intermediate states. Comparison should only take place when the system is in a stable state. For instance, comparison is usually stopped during mode transitions that take time.
- The comparator should not be too eager to report errors. For instance, small communication delays easily lead to differences during a short amount of time. Hence, the comparator should only report an error if differences persist during a certain amount of time. A trade-off has to be made between taking more time to avoid false errors and reporting errors fast to allow quick repair. This also influences the frequency with which comparisons take place (modeled by the Compare Pulse in Figure 8.1).
- The observations that are sent from SUO to SPEC and Comparator need not be restricted to the external interface of the SUO. It might be very useful to include information about the internal state of the SUO in these observations and in the SPEC model. This allows early detection of internal errors and, hence, improves the possibilities to correct these errors before they lead to failures visible by the user.

8.3 Error detection framework

Since in our project context the control software of a TV is often implemented on top of Linux, we have developed a Linux-based framework for runtime error detection. A particular SUO can be inserted in this framework. The SUO has to be adapted to provide certain observations to the so-called awareness monitor. The specification model of the desired system behavior is included in this monitor by using the possibilities of Simulink to generate code from Stateflow models. Hence, it is easy to experiment with different specification models. The awareness part also contains a comparator that can be adapted to include different comparison and detection strategies.

The design of the error detection framework is shown in Figure 8.2. The Stateflow Coder of Simulink is used to generate C-code from a Stateflow model of the desired behavior. This code is included in the *Stateflow Model Implementation* component and executed by the *Model Executor*. The SUO and the awareness monitor are separate processes communicating via Unix domain sockets. The SUO sends messages with relevant observations to *Input* and *Output Observers*. Information about the identification of observations is stored in the *Configuration* component.

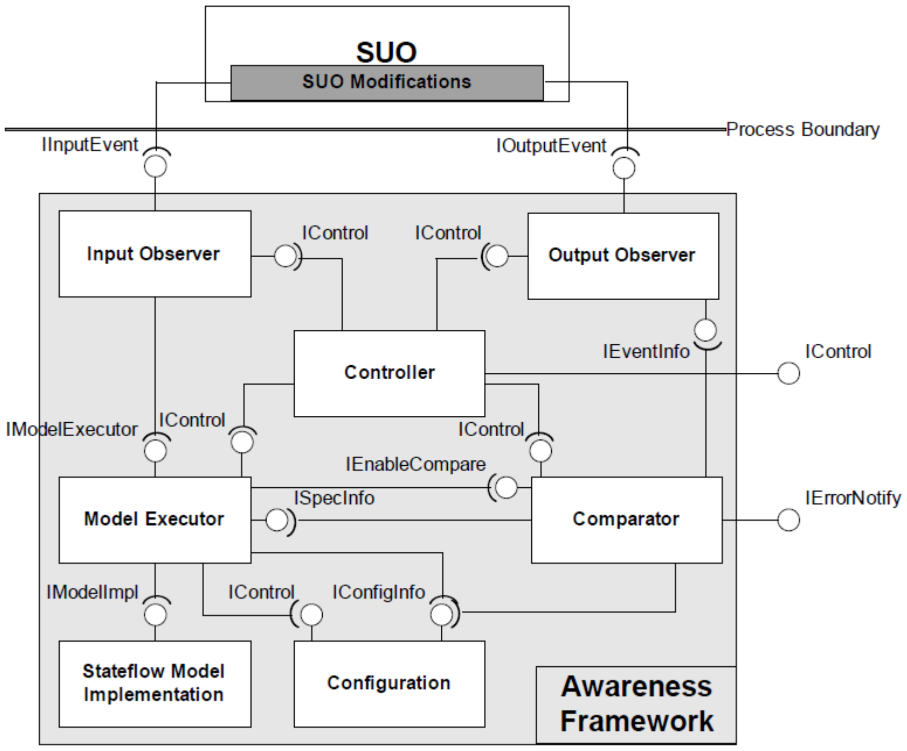


***Figure 8.2** Design of error detection framework in Linux*

The *Input Observer* receives information is that relevant for the execution of the Stateflow specification model. This information is used by the *Model Executor* to provide input to the code of the model. The *Model Executor* also receives output from the model and makes this available to the *Comparator*. The *Comparator* compares model output, which specifies desired behavior, with the actual system observations from the SUO as provided by the *Output Observer*. Note that the distinction between “Input” and “Output” observations is related to the use of this information in the Stateflow specification model. It might very well be different from the external input and output of the SUO and both observers might receive information about internal states of the SUO.

For each observation that has to be compared by the *Comparator*, the user of the framework can specify (1) a threshold for the allowed maximal deviation between specified and actual value, and (2) a limit on the number of consecutive deviations that are allowed before an error will be reported. Another parameter is the frequency with which time-based comparison takes place. This can be combined with event-based comparison by specifying in the specification model when comparison should take place and when not (e.g., when the SUO is expected to be unstable between certain modes). The *Model Executor* obtains this information from executing the implementation of the model and uses it to start and stop the *Comparator*. The *Controller* initiates and controls all components, except for the *Configuration* component which is controlled by the *Model Executor*.

8.4 Application of steps to MPlayer

We describe the application of our framework to the open source media player MPlayer [MPlayer, 2007], following the five steps described in Section 8.1:

Step 1: Identify properties

As an example, we consider the volume of the MPlayer, i.e., we want to verify that the volume level is consistent with user interactions such as volume changes and mute. Because of space limitations, the example has been simplified and we do not consider other user actions (such as pause, play and stop) and on-screen display (e.g., showing the mute/unmute icon).

Step 2: Specify properties formally

The relation between the external events VolumeUp, VolumeDown, and MuteOnOff, and the value of external variable volume is defined by the Stateflow model of Figure 8.3.

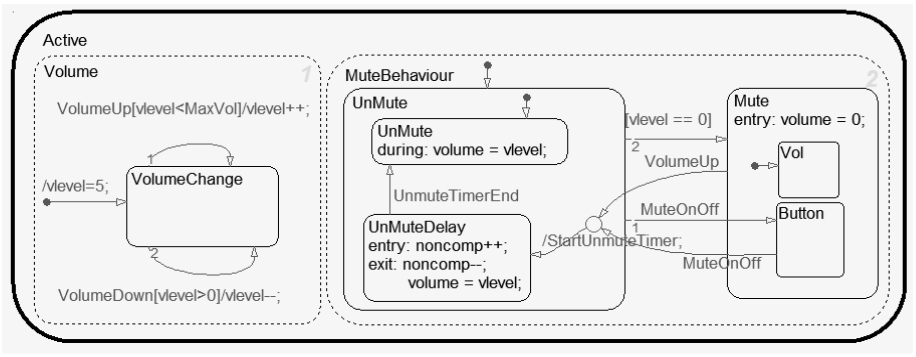


Figure 8.3 Stateflow specification model

State *Active* consists of two parallel states *Volume* and *MuteBehavior* (as indicated by the dashed borders of these two states). Note that state *Volume* uses an internal variable *vlevel*. As an example, we specify that it may take some time before an unmute command (going from state *Mute* to state *UnMute*) becomes effective. The maximal delay is modeled by the start (*StartUnmuteTimer*) and the expiration (*UnmuteTimerEnd*) of a timer. This timer is modeled in a parallel chart (not shown here), using the simulation time of Simulink. State *UnMuteDelay* represents this possible delay. In this state variable *noncomp* is used to enforce that no comparison with the SUO takes place in this state. This avoids false errors when the actual delay in the SUO is shorter than the worst case, leading to a situation where the SUO is already unmuted but the specification still has volume 0. The *Comparator* component of the error detection framework (Figure 8.2) does not compare if *noncomp* is positive (initially it has value 0). Note that in general there can be several parallel states that might change *noncomp*.

Step 3: Validate specification

There are various ways to validate that the specification model indeed captures the intended behavior. First of all, the Stateflow model can be executed and we have made a basic GUI which allows the insertion of events (such as mute and volume changes) and the visualization of external output (e.g., volume level and mute icon).

This allows extensive simulation of the model. Moreover, there are various tools connected to Stateflow that allow the generation of test scripts and conformance checks after model changes. Finally, also formal model-checking techniques can be applied to Stateflow models. See also Chapter 7.

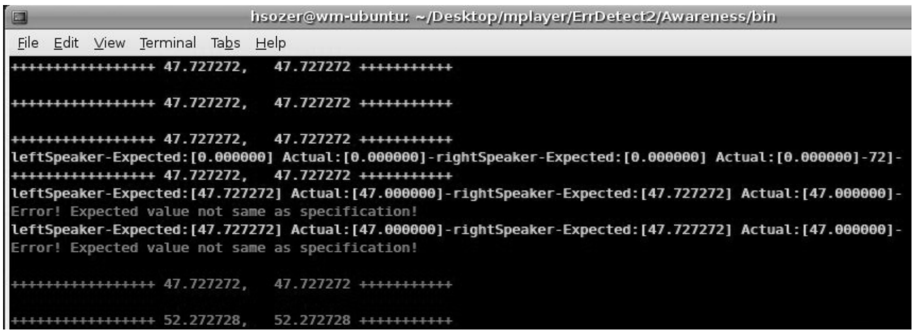
Step 4: Add system observations

The SUO is instrumented manually to observe relevant system behavior. In our example, the external input events that correspond to VolumeUp, VolumeDown, and MuteOnOff are sent to the Input Observer of the error detection framework. Similarly, changes of the volume level are sent to the Output Observer. In fact, the Output Observer might also be used to observe internal states and variables, which is often useful to detect errors early. Observed events and variables are distinguished by means of an identifier and the error detection framework is configured to relate model elements to these identifiers. Note that it is often far from trivial to insert observation code in a large piece of software that is maintained by many people and lacks detailed documentation (as in the case of the MPlayer).

Step 5: Add property evaluation

The Stateflow Coder of Simulink is used to generate C-code from the Stateflow model of the desired behavior. No code is generated for the parts that model timers; instead, timing events are coupled manually to Linux timers. The resulting code is included in component *Stateflow Model Implementation* (see Figure 8.2). At runtime, the *Model Executor* provides input to the code of the model based on event notifications from the *Input Observer* and it sends the desired volume level, as given by the specification model, to the *Comparator*. The *Comparator* compares this value with the actual volume level of the SUO, as obtained from the *Output Observe*r. The *Model Executor* starts and stops the *Comparator* based on the value of the *noncomp* variable.

To monitor the behavior of the *Comparator* closely, we have implemented a version where it writes expected (according to the specification) and actual values of the left and the right speaker to the screen. An example of a screen shot is depicted in Figure 8.4, showing the detection of small difference of volume values (using a small threshold on the allowed deviation).


hsozer@wm-ubuntu: ~/Desktop/mplayer/ErrDetect2/Awareness/bin
File Edit View Terminal Tabs Help

```
++++++++++++++++ 47.727272,   47.727272 +++++++++++

++++++++++++++++ 47.727272,   47.727272 +++++++++++

++++++++++++++++ 47.727272,   47.727272 +++++++++++
leftSpeaker-Expected:[0.000000] Actual:[0.000000]-rightSpeaker-Expected:[0.000000] Actual:[0.000000]-72]-
++++++++++++++++ 47.727272,   47.727272 +++++++++++
leftSpeaker-Expected:[47.727272] Actual:[47.000000]-rightSpeaker-Expected:[47.727272] Actual:[47.000000]-
Error! Expected value not same as specification!
leftSpeaker-Expected:[47.727272] Actual:[47.000000]-rightSpeaker-Expected:[47.727272] Actual:[47.000000]-
Error! Expected value not same as specification!

++++++++++++++++ 47.727272,   47.727272 +++++++++++

++++++++++++++++ 52.272728,   52.272728 +++++++++++
```

***Figure 8.4** Screen shot of Comparator output*

8.5 Related work

Related work consists of assertion-based approaches such as runtime verification [Colin, 2005]. For instance, monitor-oriented programming [Chen, 2004] supports runtime monitoring by integrating specifications in the program via logical annotations. In our approach, we aim at a small adaptation of the software of the system only, to be able to deal with third-party software and legacy code.

An overview of a large number of runtime monitoring techniques can be found in [Delgado, 2004]. Many of these methods use a form of logic to specify properties. An exception is the specification language TLCharts [Drusinsky, 2005] that combines the visual and intuitive appeal of non-deterministic state charts with formal specifications written in temporal logic. In general, hardly any method uses automata to specify properties, whereas in [Delgado, 2004] it is already observed that visual automata-based specifications might be better understandable for people with limited training in formal methods.

Also in more recent runtime verification techniques, the use of automata is very limited. Trace-based monitors such as Tracematches [Allan, 2005] and Tracechecks [Bodden, 2006] use regular expression or Linear Temporal Logic (LTL). Similar to many recent methods, aspect-oriented techniques (in this case AspectJ) are used to weave monitoring code into the original program. The use of these methods requires background in aspect-orientation technology. Related to our use of state machines is a method to monitor COTS components using specifications expressed by means of UML state diagrams [Shin, 2006].

Unlike our method, the runtime verification techniques mentioned in the previous sections are not suitable to detect timing violations. Early work on timing is based on Real-Time Logic (RTL) [Chodrow, 1991], whereas a timed version of temporal logic is used in the MaC-RT system [Sammapun, 2005]. Main difference with our approach is the use of a timed version of Linear Temporal Logic to express requirements specifications, whereas we use executable timed state machines to promote industrial acceptance and validation. Other work on runtime verification of real-time systems, such as [Pohlack, 2006], concentrates on observing the system with minimal intrusiveness and ensuring a predictable disturbance.

Our approach to model-based error detection is also related to on-the-fly testing techniques which combine test generation and test execution [Larsen, 2005] [van Weelden, 2005]. The main difference is that these testing techniques generate input to the system based on the model, whereas we consider normal input during system operation and forward this input to the awareness component. Hence, our approach is more related to so-called passive testing. An additional difference is that testing methods concentrate on testing the input/output interface, whereas our focus is on fast error detection (preferably before output failures occur) which often leads to the monitoring of internal implementation details such as internal variables or load.

8.6 Concluding remarks

We have designed and implemented an error-detection framework which allows the runtime comparison of a particular system under observation (SUO) with a model of its desired behavior. According to the taxonomy presented in [Delgado, 2004], our framework can be characterized an asynchronous outline runtime verification technique, since the evaluation code is executed as a separate process and the SUO need not wait for the completion of this evaluation code. This has been done to have minimal disturbance of the SUO and to allow hardware observations. The approach has been applied to the MPlayer to check data-oriented properties, such as the volume level. In another, small-scale, experiment we have used it to check memory usage. Other possibilities are, for instance, the detection of deadlock situations and timing violations.

Starting point of our research was the aim to use intuitive specifications that are based on state machines and which allow easy validation. The Stateflow notation is widely used in industry and own experiments in the multimedia domain showed that it is rather easy and intuitive to specify the control behavior in Stateflow. More details about the modeling approach can be found in Chapter 7.

The simulation possibilities of Simulink/Stateflow are very convenient to validate that the model captures the intended behavior. To deal with large models with many possible input events, leading to an enormous number of possible executions, tools to automate event generation according to certain coverage criteria are very useful. Because Stateflow has a well-defined deterministic semantics, also formal model-checking is possible, although scalability is expected to be limited.

Although the first experiments are encouraging, there are a number of points that need improvement and are topics of future work:

- *Instrumentation of SUO.* Experiments with the MPlayer showed that manual addition of observation code is often very difficult because information might be distributed and adequate documentation is often not available.
- *Low costs and low overhead.* Current experiments with the MPlayer show that the normal functionality of the application is not affected by adding our techniques for a limited set of properties. But overhead reduction was not our focus and clearly more work is needed to quantify the overhead and to investigate the possibilities for optimization.
- *Correctness of error detection.* Besides detecting errors fast, it is also very important to avoid false errors, i.e., avoid alarms when there is no error. This depends, for instance, on the correctness of the specification which should also include timing information to avoid that a slow system response leads to an error because the model is evaluated fast. Also the correctness of the error detection framework itself is crucial. Hence, more research is needed to be able to guarantee the absence of false errors.
- *Integration of awareness techniques.* We have done some first experiments with the integration of the techniques developed within Trader. We injected a fault in the MPlayer which, when activated, causes a problem with the volume slider in the GUI. This error is detected by our error detection framework and reported to the local recovery framework described in Chapter 12 which corrects the problem by restarting the GUI. We have also made a combination with the diagnoses techniques presented in Chapter 9 to find the component which most likely contains the error. It became clear that more work is needed to obtain a complete awareness framework which integrates all techniques. For instance, errors about values of variables require other recovery techniques. On the other hand, the local recovery framework also includes some error detection, e.g., concerning missing communications, making separate detection of these errors unnecessary.

Acknowledgements: Teun Hendriks and Chetan Nair are gratefully acknowledged for their contribution to the development of the awareness framework.

8.7 References

[Allan, 2005] C. Allan, P. Augustinov, A. Christensen, L. Hendren, S. Kuzins, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble. *Adding trace matching with free variables to AspectJ*. Proc. 20th Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05), pp. 345-364, 2005

[Bodden, 2006] E. Bodden, V. Stolz. *Tracechecks: Defining semantic interfaces with temporal logic*. Software Composition, LNCS 4089, Springer-Verlag, pp. 147-162, 2006

[Chen, 2004] F. Chen, M. D'Amorim, G. Rosu. *A formal monitoring-based framework for software development and analysis*. Proc. ICFEM 2004. Volume 3308 of LNCS, Springer-Verlag, pp. 357-372, 2004

[Chodrow, 1991] S. Chodrow, F. Jahanian, M. Donner. *Runtime monitoring of real-time systems*. Proc. IEEE Real-Time Systems Symposium. pp. 74-83, 1991

[Colin, 2005] S. Colin, L. Mariani. *Runtime verification*. Proc. Model-Based Testing of Reactive Systems. Volume 3472 of LNCS, Springer-Verlag, pp. 525-555, 2005

[Delgado, 2004] N. Delgado, A.Q. Gates, S. Roach. *A taxonomy and catalog of runtime software-fault monitoring tools*. IEEE Transactions on Software Engineering, Vol. 30, No. 12, pp. 859-872, 2004

[Drusinsky, 2005] D. Drusinsky. *Semantics and runtime monitoring of TLCharts: Statechart automata with temporal logic conditioned transitions*. Electronic Notes in Theoretical Computer Science 113, pp. 3-21, 2005

[Larsen, 2005] K. Larsen, M. Mikucionis, B. Nielsen, A. Skou. *Testing real-time embedded software using UPPAAL-TRON: an industrial case study*. 5th ACM Int. Conf. on Embedded Software (EMSOFT'05), ACM Press New York, pp. 299-306, 2005

[Mathworks, 2009] The Mathworks: *Matlab/Simulink*. http://www.mathworks.com/, 2009

[MPlayer, 2007] MPlayer: *Open source media player*. http://www.mplayerhq.hu/ , 2007

[Pohlack, 2006] M. Pohlack, B. Döbel, A. Lackorzyński. *Towards runtime monitoring in real-time systems*. Proc. 8th Real-Time Linux Workshop, 2006

[Sammapun, 2005] U. Sammapun, I. Lee, O. Sokolsky. *Checking correctness at runtime using real-time Java*. Proc. 3rd Workshop on Java Technologies for Real-time and Embedded Systems (JTRES'05), 2005

[Shin, 2006] M.E. Shin, F. Paniagua. *Self-management of COTS component-based systems using wrappers*. Computer Software and Applications Conference (COMPSAC 2006), IEEE Computer Society, pp. 33-36, 2006

[van Weelden, 2005] A. van Weelden, M. Oostdijk, L. Frantzen, P. Koopman, J. Tretmans. *On-the-fly formal testing of a smart card applet*. Int. Conf. on Information Security (SEC 2005), pp. 565-576, 2005

Chapter 9

Fault localization of embedded software

Authors: Rui Abreu, Peter Zoeteweij, Arjan J.C. van Gemund

Abstract: Residual faults in embedded software are a major threat to systems dependability, and automated diagnosis techniques help to counter this threat in two ways: as a means to improve the efficiency of the debugging process they reduce the fault density, and as an integral part of fault detection, isolation, and recovery mechanisms they help harnessing runtime errors caused by residual faults. In this chapter we introduce spectrum-based fault localization (SFL), a diagnosis technique based on statistical analysis of execution profiles. SFL requires no additional modeling effort and has a low CPU and memory overhead, which makes it well suited for application in existing development environments and for the embedded systems domain. For deployment-time application, we complement SFL with light-weight generic error detection mechanisms based on program invariants, thus giving rise to systems that can autonomously identify potentially erroneous states, and locate the components that are likely to be involved. Using a probabilistic model of a software system, we show that the expected diagnostic accuracy of SFL approaches that of more expensive alternatives based on automated reasoning.

9.1 Introduction

Software reliability can generally be improved through extensive testing and debugging, but this is often in conflict with market conditions: software cannot be tested exhaustively, and of the bugs that are found, only those with the highest impact on the user-perceived reliability can be solved before the release. As a result, products are typically deployed with known and unknown faults that can manifest themselves as system failures at deployment-time. Automated diagnosis techniques help to deal with this reliability threat in two ways. First, as debugging aids, they guide developers to the root cause of failures observed during testing, thus improving the efficiency of the debugging process and reducing the number of known faults. Second, the impact of residual faults can be managed via so-called fault-detection, isolation, and recovery (FDIR) strategies, where automated diagnosis is applied to identify the system components that are involved in an imminent failure. The runtime awareness mechanism outlined in Section 1.2.4 is an example of an FDIR strategy.

The subject of this chapter is a particular diagnosis technique called *spectrum-based fault localization* (SFL). Program spectra are execution profiles that identify the system components involved in a given run or transaction, and in SFL this information is used to reveal the components whose behavior shows statistical similarity to the occurrence of errors and/or failures. We will demonstrate that SFL can be implemented with low CPU and memory overheads, which makes it well-suited for fault diagnosis of embedded software. In addition, we complement SFL with light-weight, generic error detection mechanisms known as *fault screeners*. This way, we arrive at systems that can autonomously detect and diagnose potentially erroneous states, thus covering the fault detection and isolation steps of an FDIR strategy, which can be used to drive recovery mechanisms such as, for example, microrebooting [Candea, 2004].

Because of its statistical nature, the diagnostic accuracy of SFL is inherently limited, and it can only diagnose multiple faults in sequence. To gain an understanding of the practical impact of these limitations, we compare SFL to a number of alternative techniques derived from model-based diagnosis. While it is computationally more expensive, model-based diagnosis will never yield invalid explanations for observed behavior, and is inherently suited for diagnosing multiple faults. An analysis based on synthetic execution data generated for a simple model of a software system shows that SFL can be expected to have an excellent trade-off between computational overhead and diagnostic accuracy.

The remainder of this chapter is organized as follows. In Section 9.2 we introduce spectrum-based fault localization. In Section 9.3 we complement it with fault screeners, and in Section 9.4 we compare the effectiveness of SFL against that of more rigorous approaches derived from model-based diagnosis. Related work is discussed in Section 9.5. We conclude in Section 9.6. In Chapter 10 we describe our experience with applying the techniques introduced in this chapter in practice.

9.2 Spectrum-based Fault Localization

9.2.1 Terminology

In this chapter we will use the following terminology.

- *Components* are the unit of software composition at whose level of granularity a diagnosis is made.
- A *failure* is a deviation from the desired behavior of a system, and an *error* is a system state that can lead to a failure. Errors and failures are caused by *faults*.
- *Runs* are the exercised functionality that a diagnosis is based on. If a failure occurs, or an error is detected, runs are classified as *failed*, and as *passed* otherwise.

Software fault diagnosis is typically performed at the lowest possible level of granularity, so here a component can also be a statement, code block, or function. Runs can be the individual test cases of a test suite, but also, for example, usage scenarios, or fixed periods of system activity.

9.2.2 Program Spectra

A *program spectrum* [Reps, 1997] is a collection of data that provides a specific view on the dynamic behavior of a software system. This data is collected during a run of a system, and typically consists of a number of counters or flags for the different components. Various forms of program spectra exist, see [Harrold, 2000] for an overview.

SFL is based on component-hit spectra. These are binary vectors, whose length equals the number of system components, and whose elements indicate whether the corresponding component was active in a run, or not. Depending on the level of granularity, specific types of component-hit spectra can be identified, for example, statement-hit spectra, or block-hit spectra, or program spectra for more abstract notions of a component, such as data dependencies.

9.2.3 Diagnosis

The component-hit spectra for all runs underlying a diagnosis can be organized as the rows of a binary matrix, whose columns correspond to the system components. For example, Table 9.1 shows the program spectra for a system of five components, and four runs. The spectrum in the first row shows that components c_1, and c_4 are involved in the first run, and the spectrum in the second row shows that components c_1, c_4, and c_5 are involved in the second run, etc. The information on whether a run passed or failed constitutes another column vector, which is usually referred to as the *error vector*. In Table 9.1, the error vector indicates that runs 2, 3, and 4 failed. Now spectrum-based fault localization entails identifying the component whose column vector resembles the error vector most.

	c_1	c_2	c_3	c_4	c_5	**error**
run 1	1	0	0	1	0	0
run 2	1	0	0	1	1	1
run 3	0	1	1	1	1	1
run 4	1	1	0	1	1	1
s_J	½	⅔	⅓	¾	1	

Table 9.1 *SFL example*

In the field of data clustering, resemblances between vectors of binary, nominally scaled data such as the columns in our matrix of program spectra, are quantified by means of similarity coefficients (see, e.g., [Jain, 1988]). As an example, the Jaccard similarity coefficient, s_J, equals the number of positions in which the vectors share an entry 1, divided by that same number, plus the number of positions in which they differ:

$$s_J = \frac{a_{11}}{a_{11} + a_{10} + a_{01}}$$

For comparing the column vector of a component to the error vector, the factors of the Jaccard coefficient have the following meaning:

a_{11} - the number of failed runs a component was involved in

a_{10} - the number of passed runs a component was involved in

a_{01} - the number of failed runs a component was not involved in

Thus calculating the similarity of the component vectors to the error vector yields a ranking of the system components according to the likelihood that they are involved in the detected errors or observed failures. Table 9.1 shows the calculated Jaccard coefficients for the five component vectors below the matrix, which identify c_5 as the most likely fault location, c_4 as the second most likely fault location, c_2 as the next, etc.

In addition to the Jaccard coefficient, which is used, for example, in the Pinpoint framework [Chen, 2002], several other coefficients are used in practice. For a study on the influence of the similarity coefficient, which includes s_J, and the coefficient used by the Tarantula system [Harrold, 2002], see [Abreu, 2007]. The outcome of this study is that the Ochiai coefficient (see Section 10.3.3), which is know from the biology domain, provides superior diagnostic accuracy when used for software fault diagnosis.

9.2.4 Efficiency

SFL can be implemented with low time and space complexity. Recording the program spectra requires a bit to be set every time a component is activated, which incurs a linear overhead in the number of system components on the execution time. Calculating the similarity coefficients is linear in the number of system components, and ranking the components on their calculated similarity coefficients involves sorting, which is O(n log n). This last step has to be performed only after an error is detected, or a failure occurs.

Regarding space complexity, in the embedded systems domain, memory is typically a scarce resource, and storing the spectra for a large number of runs to be post-processed at diagnosis time is usually not an option. Fortunately, the set of spectra that a diagnosis is based on contains much more information than needed, and can easily be compacted at runtime. In the end, the only information that is necessary for generating the ranking are the counters a_{11}, a_{10}, a_{01} for each of the components, and the space required to store them is linear in the size of the program. To avoid having to store the actual spectra, we can update the counters right after a run has finished, and the passed/failed verdict has become available:

- For a passed run, increment the a_{10} counters of the active components.
- For a failed run, increment the a_{11} counters of the active components. a_{01} can always be obtained by subtracting a_{11} from the number of failed runs.

After thus having processed the program spectrum of a passed or failed run, the spectrum itself can be discarded, and the diagnosis can be performed any time after processing at least one failed run.

9.3 SFL at Deployment Time

For development-time diagnosis, the passed/failed verdict is usually made by developers or by comparison to a reference output, for example in regression testing. The latter mode is depicted in the top half of Figure 9.1. For deployment-time diagnosis, however, error detection is typically implemented through tests, or invariants that range from *application-specific* (e.g., assertions) to *generic* (e.g., compiler-generated range checks). While application-specific invariants cover many failures anticipated by the developers, and have low false positive and false negative rates, their manual integration in the code is typically costly and error-prone. Conversely, despite their simplicity, generic invariants can be automatically generated and in many cases, their application-specific training can also be performed automatically, as an integral part of the testing process during the development phase.

In view of these attractive properties, generic invariants, often dubbed *fault screeners*, have long been a subject of study in both the software and hardware domain. They were first introduced in [Ernst, 1999], for the purpose of supporting program evolution. They are often used for fault localization directly (see, e.g., [Hangal, 2002] [Pytlik, 2003]), but here we propose to use them in combination with SFL to arrive at systems that can autonomously detect potentially erroneous states, and identify the components that are likely to be involved. This combination is depicted in the bottom half of Figure 9.1.

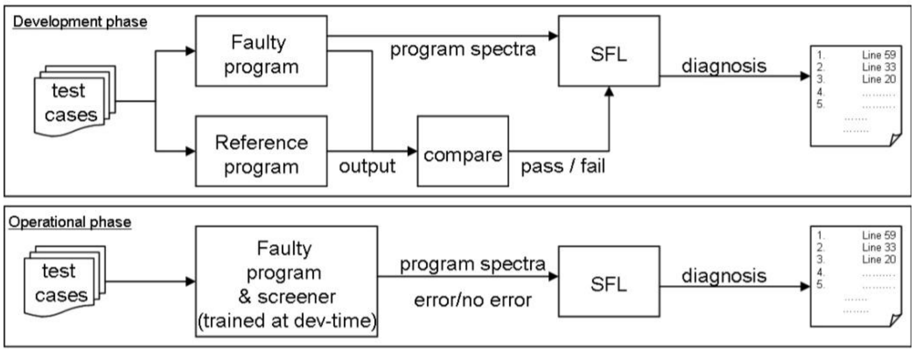


Figure 9.1 *Error detection using a reference program and using fault screeners*

As an example of a fault screener, *range invariants* (see, e.g., [Racunas, 2007]) represent the (integer or real) bounds of a program variable. Every time a new value v is observed, it is checked against the currently valid lower bound l and upper bound u according to

$violation = \neg\ (l < v < u)$

If v is outside the bounds, an error is flagged in error detection mode (deployment phase), whereas in training mode (development phase) the range is extended according to the assignments

$l := \min(l,v)$

$u := \max(l,v)$

The range invariant fault screener is illustrated in Figure 9.2.

Other examples of fault screeners with a low overhead are *bitmask invariants* [Hangal, 2002] [Racunas, 2007], representing the bits in which a variable's value may differ from its initial value, and *Bloom filters* [Bloom, 1970] [Racunas, 2007], which are space-efficient probabilistic data structures for checking if an element is a member of a set. Important characteristics of fault screeners are the rate at which they fail to detect errors (false negatives), the rate at which they report errors when no fault is activated (false positives), and the amount of training needed to attain specific rates. Chapter 3 reports some empirical results for the fault screeners mentioned above. For more information, including an analysis of the learning rates, see [Abreu, 2008-a].

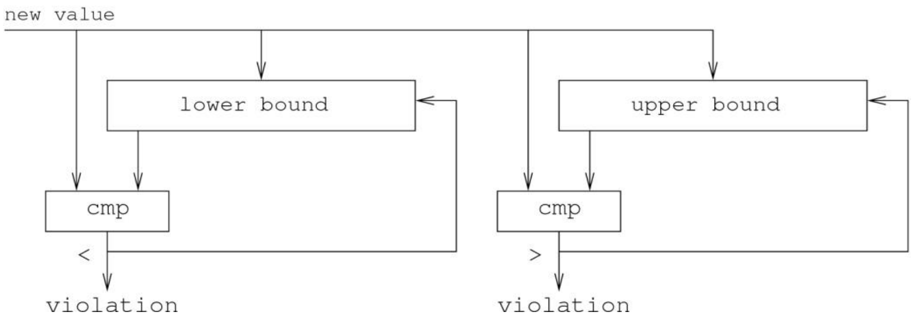


Figure 9.2 *The range invariant fault screener*

9.4 Spectrum-based Versus Model-based Techniques

Although SFL can be applied iteratively until all tests pass, it is inherently a single-fault technique. This limitation is overcome by software fault diagnosis techniques derived from *model-based diagnosis* (MBD), which are able to handle multiple faults directly. In addition, while due to SFL's statistical nature innocent components may rank high, model-based techniques always yield diagnoses that fully explain the observed behavior in terms of a model of the system. Below we outline model-based diagnosis and its adaptation to software. Next, using a simple, probabilistic model of a software system, we compare the accuracy of SFL to that of MBD.

9.4.1 Observation-based Modeling

Model-based diagnosis (MBD, [De Kleer, 1987]) originated in the artificial intelligence domain, and combines a compositional, behavioral model of a system with real-world observations to infer diagnoses. Essentially, MBD entails searching for minimal combinations of components whose failure explains the observed behavior, according to the model. It has successfully been applied to digital circuits and complex mechanical systems, but in absence of suitable behavioral models, its application to software has been troublesome (see [Mayer, 2007] for an overview).

To overcome the need for explicit modeling, in [Abreu, 2008-b] an observation-based model is derived from the program spectra used in SFL. This model states that the components involved in a failed run cannot all be healthy. For example, for the spectra in Figure 9.2 the model amounts to

$(\neg h_1 \vee \neg h_4 \vee \neg h_5) \wedge (\neg h_2 \vee \neg h_3 \vee \neg h_4 \vee \neg h_5) \wedge (\neg h_1 \vee \neg h_2 \vee \neg h_4 \vee \neg h_5)$,

where the variables h_i reflect the health of the components. In this case, the model states that components c_1, c_4, and c_5 cannot all be healthy, components c_2, c_3, c_4, and c_5 cannot al be healthy, and components c_1, c_2, c_4, and c_5 cannot al be healthy.

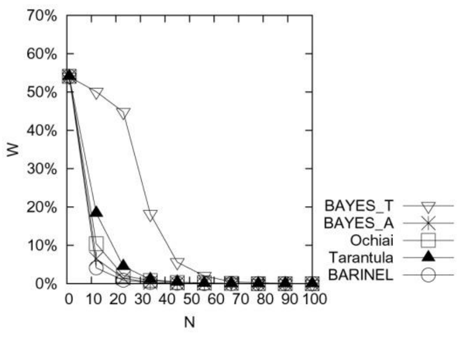
Typically, MBD yields several diagnoses. For example, the above model has four minimal solutions: either c_4 is faulty, or c_5 is faulty, or c_1 and c_2 are faulty, or c_1 and c_3 are faulty. In classical model-based diagnosis, these alternative diagnoses are ranked by calculating their conditional probabilities using Bayesian reasoning, possibly extended with an intermittent fault model [de Kleer, 2007]. The latter option entails that the conditional probabilities are calculated using a *goodness* parameter, which captures the probability that faulty components behave correctly.

Generating all minimal diagnoses is an NP-complete search problem, and to counter the computational complexity of model-based diagnosis we use a heuristic reasoning algorithm called STACCATO (STAtistiCs-direCted minimAl hiTting set algOrithm), where the SFL similarities guide the search towards those solutions that have the highest conditional probability. As a result, STACCATO returns a diagnostic report of limited size (typically 100 explanations), yet capturing all significant probability mass at dramatically reduced reasoning cost.

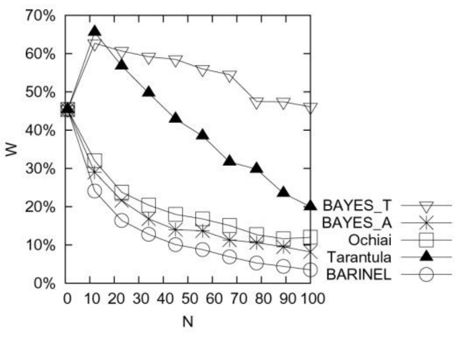
9.4.2 Evaluation

In order to compare iterative SFL to model-based diagnosis in the multiple-fault case, we generated synthetic program spectra for a probabilistic model of a software system whose main parameters are the number of faults, the probability r that a component is activated in a run, and the intermittency rate g, which represents the probability that a faulty components behaves correctly. The various diagnosis techniques are compared on the *wasted effort, W,* incurred when following the suggested order in searching for the (seeded) faults [Abreu, 2008-b].

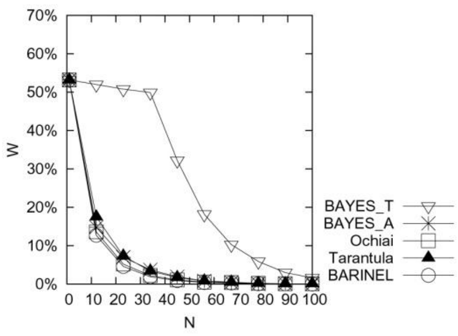
Figure 9.3 shows the outcome of the comparison for two and five faults with various intermittency rates in a system of 20 components with *r* = 0.6 (the trends for other numbers of components and values of *r* are the same). Each graph plots the wasted effort for *N* = 1,..,100 runs underlying the diagnosis. Every point in the graph is averaged over 1,000 sets of spectra, yielding a coefficient variance of approximately 0.02.



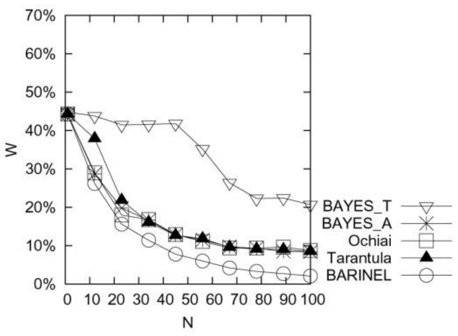
(a) 2 faults, g=0.1



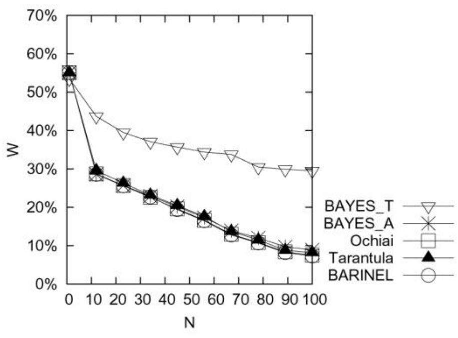
(b) 5 faults, g=0.1



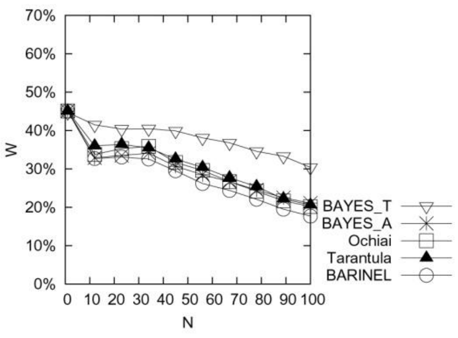
(c) 2 faults, g=0.5



(d) 5 faults, g=0.5



(e) 2 faults, g=0.9



(f) 5 faults, g=0.9

Figure 9.3 *Wasted effort for two and five faults*

The various techniques compared in the graphs are the following:

- BAYES_T: traditional MBD applied to the observation-based model.
- BAYES_A: MBD extended with the intermittency model, where the goodness parameter of every component is approximated from the spectrum matrix.
- Ochiai: SFL using the Ochiai coefficient, which is similar to the Jaccard coefficient of Section 9.2.3, but which has been shown to yield superior diagnoses [Abreu, 2007] (see also Chapter 10).
- Tarantula: SFL using the coefficient of the Tarantula system [Harrold, 2002], which is commonly used as a reference technique.
- BARINEL: MBD extended with the intermittency model, where the goodness parameters of the individual components are determined to maximize the conditional probabilities.

In all MBD variants, the STACCATO heuristic is used, and for the maximization problem inherent to the BARINEL technique we use a simple gradient ascent procedure.

From these results we conclude that apart from the classical, non-intermittent MBD approach and SFL using the Tarantula coefficient, SFL applied iteratively to systems with multiple faults can be expected to yield results that are comparable to that of the more rigorous techniques derived from model-based diagnosis. As is apparent from Figure 9.3 (b) and (d), the observation-based modeling approach, and most notably BARINEL can be expected to yield significantly more accurate diagnoses. However, although through STACCATO, the time complexity of this technique is in the same complexity class as that of SFL, the technique is computationally more expensive in practice, and BARINEL has a memory overhead that is exponential in the number of anticipated faults. While none of these are fundamental limitations, we expect that SFL will be an attractive approximation in many practical situations.

9.5 Related Work

Two tools that implement spectrum-based fault localization were already mentioned: the Tarantula tool [Harrold, 2002] provides statement level diagnosis of C programs, and Pinpoint [Chen, 2002] is a framework for root cause analysis on the J2EE platform, targeted at large, dynamic Internet services such as web-mail services and search engines. Pinpoint has been developed as a part of the Recovery Oriented Computing project (ROC, [Patterson, 2002]). For a further discussion on the relation between SFL and MBD see [Zoeteweij, 2008]. Several other approaches to software fault diagnosis exist (see [Abreu, 2007] for an overview), and the diagnostic performance of SFL compares favorably to many of these. For example, [Abreu, 2008-b] presents empirical evidence that it outperforms Sober [Liu, 2005] and Delta Debugging [Cleve, 2005]. Sober is a statistical debugging tool which analyses traces fingerprints and produces a ranking of predicates by contrasting the evaluation bias of each predicate in failing cases against those in passing cases. Delta Debugging compares the program states of a failing and a passing run, and actively searches for failure-inducing circumstances in the differences between these states.

9.6 Conclusions

In this chapter we have introduced spectrum-based fault localization, a highly efficient statistical diagnosis technique that helps to reduce the reliability threat of residual defects in embedded software in two ways:

- As a debugging aid, it guides developers to the root cause of failing tests, thus reducing the fault density by shortening the test-diagnose-repair cycle.
- As a part of a fault detection, isolation, and recovery (FDIR) mechanism, it allows the isolation, and subsequent recovery of those system components that are involved in detected errors before these errors propagate to system failures.

Specifically for the latter purpose, we proposed to complement SFL with light-weight, generic error detection mechanisms called fault screeners that can automatically be adapted to specific products via training, as an integral part of existing testing procedures. The low CPU and memory overhead of SFL and fault screeners make this combination well-suited for application in embedded software.

Finally, we have compared the diagnostic performance of SFL to that of a number of more rigorous techniques based on automated reasoning. While especially the BARINEL combination of heuristics can be expected to yield more accurate diagnoses, experiments on a high-level, probabilistic model of a software system indicate that the added accuracy is limited. Given the significant computational overhead of these techniques, we expect that for many applications SFL will have a better trade-off between efficiency and diagnostic accuracy.

Acknowledgements: We gratefully acknowledge the work of MSc. students Alberto González, Mario Plasencia, and Rafi Vayani.

9.7 References

[Abreu, 2007] R. Abreu, P. Zoeteweij, A.J.C. van Gemund. *On the accuracy of spectrum-based fault localization*. In Proc. TAIC/PART '07, pp. 89-98. IEEE Computer Society, 2007

[Abreu, 2008-a] R. Abreu, A. González, P. Zoeteweij, A.J.C. van Gemund. *On the performance of fault screeners in Software Development and Deployment*. In Proc. ENASE '08, pp. 123-130. INSTICC Press, 2008

[Abreu, 2008-b] R. Abreu, P. Zoeteweij, A.J.C. van Gemund. *An observation-based model for fault localization*. In Proc. WODA '08, pp. 64-70. ACM Press, 2008

[Bloom, 1970] B. Bloom. *Space/time trade-offs in hash coding with allowable errors.* Communications of the ACM, 13(7): pp. 422-426, 1970

[Candea, 2004] G. Candea, S. Kawamoto, Y. Fujiki, G. Fieldman, A. Fox. *Microreboot – a technique for cheap recovery*. Proc. OSDI '04, USENIX Association, 2004

[Chen, 2002] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer. *Pinpoint: problem determination in large, dynamic internet services*. Proc. DSN 2002, pp. 595-604. IEEE Computer Society

[Cleve, 2005] H. Cleve, A. Zeller. *Locating causes of program failures*. Proc. ICSE '05, pp. 342-351. ACM Press, 2005

[Ernst, 1999] M. Ernst, J. Cockrell, W. Griswold, D. Notkin. *Dynamically discovering likely program invariants to support program evolution*. Proc. ICSE '99, pp. 213-224. IEEE Computer Society, 1999

[Hangal, 2002] S. Hangal, M. Lam. *Tracking down software bugs using automatic anomaly detection*. Proc. ICSE '02, pp. 291-301. IEEE Computer Society, 2002

[Harrold, 2000] M.J. Harrold, G. Rothermel, K. Sayre, R. Wu, L. Yi. *An empirical investigation of the relationship between fault-revealing test behavior and differences in program spectra*. Journal of Software Testing, Verification, and Reliability, 10(3): pp. 171--194, 2000

[Harrold, 2002] J. A. Jones, M.J. Harrold, J. Stasko. *Visualization of test information to assist fault localization*. In Proc. ICSE '02, pp. 467-477. ACM Press, 2002

[Jain, 1988] A.K. Jain, R.C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988

[De Kleer, 1987] J. de Kleer, B.C. Williams. *Diagnosing multiple faults*. Artificial Intelligence, 32(1): pp. 97-130, 1987

[De Kleer, 2007] J. de Kleer. *Diagnosing intermittent faults*. Proc. DX'07, pp. 138-145, 2007.

[Liu, 2005] C. Liu, X. Yan, L. Fei, J. Han, S.P. Midkiff. *Sober: Statistical model-based bug localization*. Proc. ESEC/FSE '05, pp. 286-295. ACM Press, 2005

[Mayer, 2007] W. Mayer, M. Stumptner. *Models and tradeoffs in model-based debugging*. In Proc. DX'07, pp. 138-145. 2007

[Patterson, 2002] D. Patterson et al. *Recovery oriented computing: motivation, definition, techniques, and case studies*. Technical report CSD-02-1175, UC Berkeley Computer Science, 2002

[Pytlik, 2003] B. Pytlik, M. Reinieris, S. Krishnamurthi, S. Reis. *Automated fault localization using potential invariants*. In Proc. AADEBUG '03, pp. 273-276

[Racunas, 2007] P. Racunas, K. Constantinides, S. Manne, S. Mukherjee. *Perturbation-based fault screening*. In Proc. HPCA '07, pp. 169-180. IEEE Computer Society, 2007

[Reps, 1997] T. Reps, T. Ball, M. Das, J. Larus. *The use of program profiling information for software maintenance with applications to the year 2000 problem.* Proc. ESEC/FSE '97, LNCS 1301, pp. 432-449. Springer-Verlag, 1997

[Zoeteweij, 2008] P. Zoeteweij, J. Pietersma, R. Abreu, A. Feldman, A.J.C. van Gemund. *Automated fault diagnosis in embedded systems*. In Proc. SSIRI '08, pp. 103-110, IEEE Computer Society, 2008

Chapter 10

Spectrum-based fault localization in practice

Authors: Peter Zoeteweij, Rui Abreu, Rob Golsteijn, Arjan J.C. van Gemund

Abstract: In this chapter we give an account of our experience with applying the spectrum-based fault localization (SFL) technique, introduced in Chapter 9, to actual software systems. First, using a widely accepted benchmark set of software faults, we investigate the influence of the number of runs underlying the diagnosis, and of the accuracy of the error detection mechanism, both of which are key for deployment-time application in embedded systems. Second, to assess its relevance as a debugging aid in an industrial development environment, we apply SFL to a number of problems that were reported and repaired during the development of several related product lines of television sets, and evaluate how well the diagnosis resembles the one underlying the repairs. Our results indicate that SFL can successfully be applied in the resource-constrained environment of embedded software in consumer electronics products.

10.1 Introduction

As a means to improve the efficiency of the debugging process, and as an essential part of fault detection, isolation, and recovery (FDIR) mechanisms, such as the runtime awareness and correction approach of Section 1.2.4, automated diagnosis techniques constitute a significant contribution to software reliability. In Chapter 9 we introduced spectrum-based fault localization (SFL), a specific technique for software fault diagnosis based on discovering statistical similarities between the activity of components and the occurrence of errors or failures. The relatively low CPU and memory overhead make the technique well-suited for the embedded systems domain, and empirical evidence on synthetic execution data generated for a probabilistic model of a software system suggests that the accuracy of SFL diagnoses compares favorably to that of alternative, more expensive techniques based on automated reasoning. To assess its practical relevance further, in this chapter we describe our experience with applying SFL to actual software systems.

Our first test subject is a benchmark set of software faults known as the Siemens set, consisting of over 130 bugs in seven small C programs, for which the correct versions and extensive test suites are supplied. Using this benchmark set we investigate the diagnostic accuracy as a function of the rate of false positives/negatives implied by a given error detection mechanism, and the number of program spectra for passed and failed runs. Since fault screeners are inherently unreliable, and a diagnosis cannot be postponed until conclusive evidence has been gathered, the practical relevance of the mechanisms proposed in Chapter 9 will be determined by the influence of these parameters.

Second, to assess the feasibility of SFL as a debugging aid in an industrial development process, we applied it to a number of problems that were reported and repaired during the development of the control software embedded in several closely related product lines of televisions sets. In this case we evaluate how well the diagnosis resembles the one underlying the repairs made by the developers. Both sets of experiments firmly confirm our belief that SFL and fault screeners are practicable means for improving the reliability of systems with embedded software.

The remainder of this chapter is organized as follows. In Section 10.2 we recapitulate the SFL principles, and describe the tooling that we implemented for the software systems under study. In Section 10.3 we report the results of the benchmark experiments. In Section 10.4 we discuss our validation experiments on the industrial test cases. We conclude in Section 10.5.

10.2 Spectrum-Based Fault Localization

10.2.1 Principles

Spectrum-based fault localization is based on discovering statistical similarities between the activity of the components of a software system, and the occurrence of errors and failures. The components can be any unit of software composition (e.g., statements, blocks of code, functions, etc.), and their activity is laid down in so-called program spectra, which are binary vectors, indicating the components that are involved in a certain run (usage scenario, test case, or other piece of functionality that we may be interested in).

The program spectra for all runs underlying the diagnosis can be organized as the rows of a binary matrix, whose columns correspond to the system components. In addition, an extra column vector, which is usually referred to as the error vector, is used to classify the runs as passed or failed. Spectrum-based fault localization consists in identifying the component whose column vector resembles the error vector most. For this purpose we use similarity coefficients, as found in data clustering. Thus quantifying the similarity of all column vectors to the error vector yields a ranking of components with respect to the likelihood that they are involved in the detected errors and observed failures.

10.2.2 Tools

In this chapter we will mainly be working at the level of individual statements and blocks of statements, and the particular kinds of program spectra involved are statement-hit spectra and block-hit spectra. These program spectra are obtained by instrumenting the software that we want to diagnose to record the activity of the statements or blocks at runtime. In addition, we instrument programs for learning and the checking fault screeners that we introduced in Chapter 9. The following technologies are used.

Front

Our primary instrumentation tool is based on the cfront parser for ANSI C, which is distributed as a part of the Front parser generator [Augusteijn, 2002], and which is also part of the development environment of NXP Semiconductors, the carrying industrial partner of the Trader project. Using this tool, we insert a call to a logging function in every block of code. By a block of code we mean a C language statement, where we do not distinguish between the individual statements of a compound statement, but where we do distinguish between the cases of a switch statement.

LLVM

For implementing the fault screeners, we use the Low-Level Virtual Machine (LLVM) compiler infrastructure [Lattner, 2004]. LLVM consists of a virtual machine, with its own instruction set (the LLVM intermediate representation), and compilation steps for translating to and from several languages, including C. Our instrumentation tool is a so-called LLVM pass that inserts code to monitor all memory load/store instructions and function argument and return values in a program's intermediate representation. To facilitate experiments on the combination of SFL and fault screeners, we added the option to record program spectra at the level of basic blocks in the intermediate representation, which corresponds to a statement-level instrumentation of the C language. See [González, 2007] for details.

In addition to the instrumentation tools, we wrote complementary software components that implement the spectrum bookkeeping, invariant learning and checking, and spectrum-based fault localization.

10.3 Benchmark Experiments

In Chapter 9 we have already seen that SFL can be implemented with fairly low CPU and memory overhead, which makes it well-suited for runtime application in embedded systems. An important motivation for runtime application of SFL is application in FDIR strategies for improving system reliability by harnessing residual faults. However, the effectiveness of SFL for this specific purpose will not only depend on its efficiency, but also on the influence of the number of runs underlying the diagnosis: at a single failed run, all active components are equally likely to be at fault, and additional passed and failed runs are needed to come to a more meaningful diagnosis, but in FDIR applications, recovery typically cannot be postponed until conclusive evidence on the nature of the fault is available. In addition, because generic error detection mechanisms, such as the fault screeners that we proposed, are inherently inaccurate, a second important parameter is the quality of the passed/failed information.

In this section we investigate the influence of the quantity and quality of the observations on the diagnostic accuracy of SFL, using a widely accepted benchmark set of software faults. The outcome of these experiments indicates that SFL can already provide an accurate diagnosis for low numbers of runs, and with limited accuracy of the error detection information, which confirms its suitability for FDIR purposes. Finally, we empirically evaluate the proposed combination of SFL and fault screeners.

10.3.1 Benchmark Set

Our experiments are based on two sets of faults that are available from the Software-artifact Infrastructure Repository (SIR, [Do, 2005]): (1) the Siemens Set [Hutchins, 1994], which is a collection of faults in seven small C programs, and (2) a set of faults in a somewhat larger program called **space**. Table 10.1 lists details of these programs. For each of them, a correct version, and a number of faulty versions is available. The faulty versions contain a single fault, but this fault may span through multiple statements and/or functions. In addition, every program has a set of inputs (test cases) designed to provide full code coverage. For **space**, 1,000 test suites are provided, each of which consist of a selection of approximately 150 test cases that retain full code coverage, but which are more realistically sized, and easier to handle than the full set of 13,585 available test cases.

Because some faults in the Siemens Set and **space** are not bound to C statements (e.g., global variable initializations), we could not use all of them in our experiments. In total, we used 162 faulty versions: 128 out of 132 faulty versions provided by the Siemens Set, and 34 out of 38 faulty versions of **space**.

Program	versions	LOC	blocks	test cases	description
print_tokens	7	539	110	4,130	lexical analyzer
print_tokens2	10	489	105	4,115	lexical analyzer
replace	32	507	124	5,542	pattern recognition
schedule	9	397	53	2,650	priority scheduler
schedule2	10	229	60	2,710	priority scheduler
tcas	41	174	20	1,608	altitude separation
tot_info	23	398	44	1,052	information measure
space	38	5,763	777	13,585	array definition language

Table 10.1 *Benchmark set*

In our evaluation, we use the q_d metric for diagnostic quality [Abreu, 2007], which is based on the position of the known fault location in the SFL ranking. It represents the amount of code that need not be inspected when SFL is used to locate the fault, averaging if several components have the same similarity coefficient. For faults that span multiple locations, the q_d metric is based on the one location that is involved in all failed runs (a property of the benchmark set).

10.3.2 Observation Quantity Impact

To investigate the influence of the number of runs on the quality of the diagnosis, we evaluate q_d while varying the number of passed and failed runs (N_P and N_F, respectively) that are involved in the diagnosis, across the entire benchmark set. Figure 10.1 shows two examples of such evaluations, which are representative in the sense that they best show the two main effects that we observed. We plot q_d averaged over 50 randomly sampled combinations of 1 to 100 passed and failed runs, for faulty version 1 of **print_tokens2**, and version 2 of **schedule**. Apart from the apparent monotonic increase of q_d with N_F, we observe that for the former faulty version, q_d decreases when more passed runs are added, while for the latter version, it increases.

Figure 10.2 shows the influence of N_F and N_P averaged over the entire experiment, projected along the N_P axis. Overall, it shows that adding failed runs improves the quality of the diagnosis, but the effect of having more than around 10 failed runs is marginal on average. Conversely, adding passed runs has little influence on q_d averaged over the entire experiment, but can be significant for individual faults, as shown in Figure 10.1. Either way, we found that q_d stabilizes around 20 passed runs.

The actual effect of adding passed runs depends on the frequency with which the fault location is activated in failed runs, and cannot be predicted beforehand. In any case, the relatively high diagnostic quality at low numbers of failed runs confirms the suitability of SFL for runtime applications. For more information on these experiments, see [Abreu, 2007].

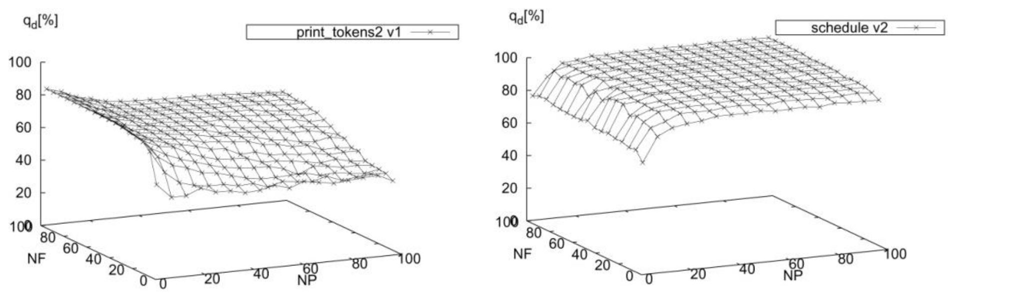


Figure 10.1 *Observation quantity impact*

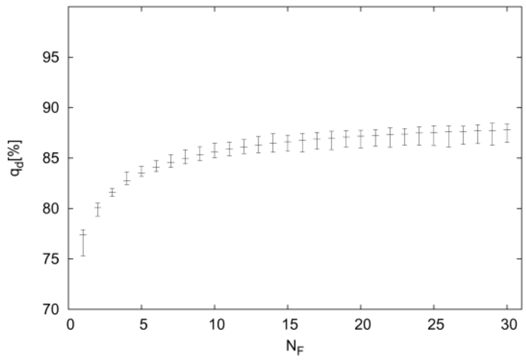


Figure 10.2 *Impact of N_F and N_P on q_d, on average; the vertical bars are marked at the minimum, average, and maximum observed for $1 \leq N_P \leq 100$*

10.3.3 Observation Quality Impact

As a model of the quality of the passed/failed information, we define q_e to be the fraction of all runs that activate the fault, for which an error is detected. All faults in our benchmark set have an intrinsic q_e value for the provided test suites, and this value can be controlled by modifying the test suite as follows.

- Excluding a passed run that activates the fault increases q_e.
- Excluding a failed run that activates the fault decreases q_e.

By randomly sampling the passed and failed runs to exclude, we were able to control q_e within a 99% confidence interval for all faulty versions in our benchmark set. Figure 10.3 shows the resulting quality of the diagnosis averaged over all versions.

The three graphs in Figure 10.3 are for three different similarity coefficients: in addition to the Jaccard coefficient, introduced in Section 9.2.3, we also included the coefficient used by a comparable system, called Tarantula [Harrold, 2002], and the Ochiai coefficient, known from the biology domain, in our experiments. Using the notation of Section 9.2.3, the latter coefficient is defined as follows.

$$s_O = \frac{a_{11}}{\sqrt{(a_{11} + a_{01}) \cdot (a_{11} + a_{10})}}$$

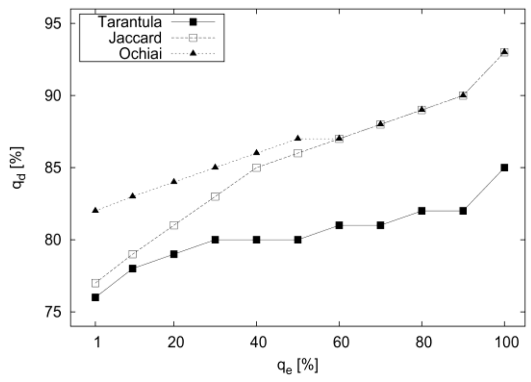


***Figure 10.3** Observation quality impact for the benchmark set*

The results indicate that the Ochiai coefficient outperforms the others, especially at low quality error detection information. See [Abreu, 2009] for an analysis of this effect. All other results reported in this chapter are for the Ochiai coefficient.

A first conclusion that can be drawn from this experiment is that SFL already reduces the debugging effort at low quality passed / failed observations. Even if only one percent of the fault activations leads to an error or a failure, more than 80% of the benchmark code need not be investigated if the developers follow the SFL ranking. Second, improving q_e improves the quality of the diagnosis, up to over 90% for perfect error detection. However, because of the high offset, the gain in diagnostic performance that is to be expected from better error detection mechanisms is limited, which suggests that cheap, but inherently inaccurate error detection mechanisms such as the fault screeners proposed in Chapter 9 are a logical choice.

Of course, having to inspect 10 – 20% of a system with millions of lines of code would still be infeasible in practice, but the percentage score is used only to normalize results across our benchmark set. As we shall see in Section 10.4, SFL is quite useful in practice, and the relative amount of code that needs to be inspected for large systems is much lower.

10.3.4 SFL and Fault Screeners

Having observed that the quality of the diagnosis is largely independent of the accuracy of the error detection mechanism, the next step is to confirm this for the combination of SFL and fault screeners proposed in Chapter 9. One discrepancy with the experiments in the previous section is that whereas we only investigated undetected activation of the fault (false negatives), without exhaustive training fault screeners will also report errors when no fault is activated (false positives). However, the corresponding terms, a_{10} and a_{01}, respectively, contribute in the same way to the Ochiai coefficient, and for this reason we expect that our results for false negatives also extend to false positives.

Figure 10.4 shows the rate at which the three fault screeners mentioned in Chapter 9 generate false positives (fp) and false negatives (fn), for different percentages of passing test cases used in training. The plots are averaged over all Siemens Set programs in our benchmark set, where we verified that the variance is negligible. From these graphs, we can conclude that the rate of false positives decreases when more test cases are used to train the screeners. In addition, it can be seen that Bloom filter screeners (bloom) learn slower than range screeners (rng), which in turn learn slower than

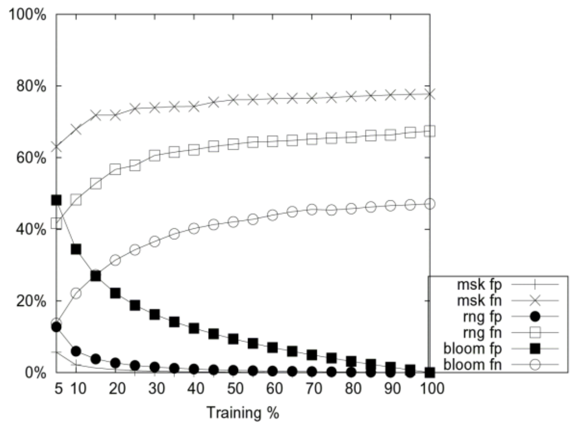


***Figure 10.4** Screeners' rate of false negatives and false positives*

bitmask screeners (msk). Furthermore, for all screeners, the rate of false negatives rapidly increases, meaning that even after minimal training, many errors are already tolerated. Finally, due to its strictness, the Bloom filter screener has on the one hand a lower false negative rate than the range screener. On the other hand, this strictness increases the false positive rate. For an analysis of these effects, see [Abreu, 2008].

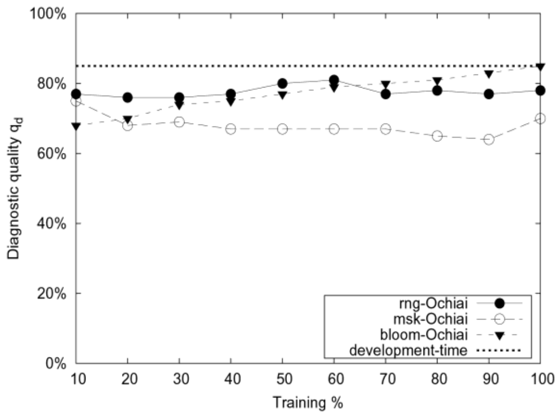


***Figure 10.5** SFL + screeners diagnostic quality*

Putting it all together, Figure 10.5 shows the quality of the SFL diagnosis when fault screeners are used for error detection, for different percentages of passing test cases used in training. From the figure we conclude that the bitmask screener performs worst. In general, the performance of the Bloom filter and range screeners is similar: the higher false negative rate of the range screener is compensated by its lower false positive rate, compared to the Bloom filter. The best diagnostic quality, 81% for the range screener is obtained around 50% training, whereas the Bloom filter steadily increases towards 85% accuracy at 100% training (at which point no passed runs are involved in the actual diagnosis, and the false positive rate is zero). The q_d = 85% line shows the diagnostic accuracy of the development-time scenario, when a reference output is used for error detection.

From these results we conclude that despite its slower learning curve, the Bloom filter screener can outperform the range screener if large amounts of data are available for training. If only a few test cases are available, it is better to use the range screener. In either case, Figure 10.5 shows that the diagnostic performance of the SFL + screeners combination approaches that of SFL with a reference output, meaning that the technique suffers little from the inherent inaccuracy of the screeners, which confirms the observations made in Section 10.3.3. Together with the low CPU and memory overheads, this makes the combination of fault screeners and SFL an attractive option for deployment-time error detection and diagnosis of embedded software.

10.4 Industrial Evaluation

While benchmark problems are well-suited for studying the influence of parameters such as the quality and quantity of the observations, they give little indication on the accuracy of spectrum-based fault localization for large-scale codes, and the kind of problems that are encountered in practice. Therefore, in this section we report our experience with applying SFL as a debugging aid for an industrial software product, namely the control software of a number of related product lines of television sets. We diagnose several problems that were encountered during the development of these sets, and compare the SFL diagnosis against the repairs made by the developers. These experiments indicate that the integration of SFL in an existing development environment is feasible, and that the technique can deliver accurate and useful diagnoses for problems that are encountered in practice.

10.4.1 Experimental Platform

One of the products of the carrying industrial partner in the Trader project, NXP Semiconductors, is the TV520 platform for building hybrid analog/digital LCD television sets, which in turn serves as the basis for televisions sets manufactured by NXP's customers. The TV520 platform comprises one or two proprietary CPUs for audio and video signal processing, plus a MIPS CPU running the control software (under Linux). More details on TV520 can be found on the NXP website [NXP]. Most of our experiments involve television sets based on TV520, but for some of our experiments we use an older, CRT-based set which we will refer to as ADOC.

All our experiments were performed on the control software of the television sets, which is responsible for tasks such as decoding the remote control input, most Teletext functionality, navigating the on-screen menu, coordinating the hardware (e.g., the tuner), and optimizing parameters for audio and video processing based on an analysis of the signals. It comprises roughly one million lines of C code (150,000 blocks) for the TV520 cases, and 450K lines of code (60,000 blocks) for ADOC. In both cases, the control software was configured from a much larger (several MLOC) code base of Koala software components [van Ommering, 2000].

In addition to instrumenting the C code at block level, as described in Section 10.2.2, we add a small Koala component which implements the spectrum bookkeeping. In both the TV520 and ADOC cases, insufficient memory is available for storing block-hit spectra for the full usage scenarios underlying the diagnoses, and on TV520 we use the mechanism described in Section 9.2.4 for performing the diagnosis online. On ADOC, the spectra are offloaded via a UART connection, and the diagnosis is performed on a PC.

10.4.2 Case descriptions

In total, we diagnosed five problems, most of which were reported and repaired during the development of the various product lines. Four of these are for TV520, and one of them is for

ADOC, but an artificial version of one of the TV520 cases was reproduced on ADOC, leading to six experiments in total. The cases are described below.

NVM Corrupted (TV520)

The TV set contains a small amount of non-volatile memory (NVM), whose contents are retained without the set being powered. In addition to storing information such as the last channel watched, and the current sound volume, the NVM contains several parameters of a TV set's configuration, for example to select a geographical region. These parameters can be set via the so-called service menu, which is not normally accessible to the user. A subset of the parameters stored in NVM is so important for the correct functioning of the set, that it has been decided to implement them with triple redundancy and majority voting. This provides a basic protection against memory corruption, since at least two copies of a value have to be corrupted to take effect. The problem that we analyze here entails that two of the three copies of redundant NVM parameters are not updated when changes are made via the service menu.

Scrolling bug (TV520)

The TV set has several viewing modes to watch content with different aspect ratios. In 16:9 viewing mode, only part of a 4:3 image is displayed on screen, and the “window” through which the image is watched can be positioned using the directional buttons on the remote control (scrolling). The problem considered here entails that after scrolling in a vertical direction, switching to dual-picture mode and back re-centers the screen. Continuing to scroll after this re-centering has occurred makes the screen jump back to the position that it had before entering dual-picture mode, and scrolling continues from that position. It should be noted that in dual-picture mode, one of the two screen halves displays the original picture, and the other half displays Teletext (a standard for broadcasting information, such as news, weather, TV guide, which is popular in Europe).

Pages Without Visible Content (TV520)

In the particular product line where this problem manifests itself, it is possible to highlight a word on a Teletext page, and then search the whole database of Teletext pages for the current channel for other occurrences of that word. However, the Teletext standard provides for pages with invisible content, through which, for example, certain control messages can be broadcast: the characters are there, but a special flag marks them invisible to the user. The problem that we investigate here entails that the word search function also finds occurrences of a word on invisible pages, and that hitting such an occurrence locks up the search functionality.

Repeated Tuner Settings (TV520)

Some broadcasters' signals contain regional information in a protocol that is recognized by many television sets, and which specifies, for example, a preferred order for the television channels. The problem that we investigate here entails that after an installation (finding all channels) is performed in presence of this regional information, tuning two times in sequence to an analog signal at the same frequency results in a black screen.

Load Problem (ADOC)

A known problem with the specific version of the ADOC control software that we had access to, is that after Teletext viewing, the CPU load when watching television is approximately 10% higher than before Teletext viewing.

State Inconsistency (ADOC)

In an early attempt to evaluate SFL, the symptoms of the TV520 invisible pages case described above were reproduced on the ADOC platform by introducing a remote-control key sequence to trigger an inconsistency in two state variables in different components, for which only specific combinations are allowed.

10.4.3 Results

To diagnose each of the problems described above, we used simple scenarios, based on the descriptions in the problem database. In general, in addition to demonstrating the problem, we would activate related functionality to exonerate as much code as possible. For the NVM corruption and the load problem, we started a new run once per second, and some extra error detection code was added to classify the runs as passed or failed automatically. For the state inconsistency, runs were delimited and classified on every remote-control key press. In the other cases, run delimiters and pass/failed characterizations were entered manually on a terminal emulator communicating with the diagnosis component in the TV set. For these cases, our scenarios typically involved around 20 runs. For a more detailed account of our approach, see [Zoeteweij, 2007] and [Abreu, 2009].

The results of the TV520 and ADOC experiments are shown in Table 10.2. Nearly perfect diagnoses are generated for the state inconsistency and the scrolling bug: if the SFL ranking is followed, respectively only two and five blocks of code have to be investigated before the location is found where the developers decided to repair these problems. For the invisible pages problem, exact information on the fault location was not available to us, so an accurate evaluation of our results is not possible. However, several code locations at the top of the SFL ranking involve statements whose execution depends on whether a page contains invisible content, or not. We expect that this could well serve as a reminder that pages can have invisible content, and that this information provides a good suggestion on the nature of a possible fix. In the ranking generated for the load problem, the logical thread that has been identified by the developers as the cause came second out of 315. In the first position was a logical thread related to Teletext, whose activation is part of the problem, so in this case we can conclude that even though the diagnosis is not perfect, the implied suggestion for investigating the problem is still useful.

platform	**case**	**inspect** 1)	**out of**
TV520	NVM corrupt	96 blocks, 10 files 2)	150K, 1.8K
TV520	scrolling bug	5 blocks	150K
TV520	invisible pages	12 blocks 3)	150K
TV520	tuner problem	2 files	1.8K
ADOC	load problem	2 logical threads	315
ADOC	state inconsistency 4)	2 blocks	60K

1) for hitting the first fault / repair location

2) data dependent problem: indirect diagnosis

3) exact information on the fault location not available

4) artificial problem, to reproduce the TV520 invisible pages case on ADOC

Table 10.2 *Results for the industrial test cases*

The repairs for the tuner problem involve modifications in 13 code blocks, all in the same file. Although none of the exact locations appears at the top of the SFL ranking, depending on the exact scenario, typically 11 other blocks are found at the highest level of similarity, 10 of which are from the file where the problem has been repaired, making this the obvious place to start debugging. Given the fact that over 1,800 C files are involved in the build, with approximately a dozen files related to low-level tuner functionality, this can be considered a reasonably accurate diagnosis.

Finally, for the NVM corruption, 96 blocks have the highest similarity to the error vector. These blocks are in ten files, one of which is part of the NVM stack. The latter files' functions access modules for normal, and redundant access to NVM, which confirms that the problem is in this area. The bug, however, resides in a routine that is called at system initialization time to populate a table describing the NVM layout. Since this routine is executed at initialization, SFL cannot associate it with the failures that occur later on, so in this case, the diagnosis is indirect at best. In general, SFL based on block-hit spectra cannot be expected to directly locate data-dependent faults, or faults in code that is always executed. However, debugging is usually an iterative process, and zooming in on the code that accesses the NVM layout table is a valuable suggestion for where to look next.

10.5 Conclusions

In this chapter we have discussed our experience with applying spectrum-based fault localization to actual software systems. First, using a widely accepted benchmark set of software faults we established that near-optimal diagnostic quality is already reached for low numbers of observations, and that the influence of the error detection mechanism's accuracy is limited. The latter property allows the application of inherently inaccurate general-purpose error detection mechanisms. Together with the low CPU and memory overheads, this makes the combination of fault screeners and SFL an attractive option for deployment-time error detection and diagnosis of residual faults in embedded software.

Second, to assess its relevance as a debugging aid in an industrial development environment, we applied SFL to a number of problems that were reported and repaired during the development of two particular product lines of television sets, and evaluated how well the diagnosis resembles the one underlying the repairs. Our results indicate that SFL can provide highly accurate suggestions on the fault location, and that application in an industrial development environment is feasible, even for resource-constrained platforms like consumer electronics devices.

Acknowledgements: We gratefully acknowledge the work of MSc. students Alberto González, Mario Plasencia, and Rafi Vayani.

10.6 References

[Abreu, 2007] R. Abreu, P. Zoeteweij, A.J.C. van Gemund. *On the accuracy of spectrum-based fault localization*. In Proc. TAIC/PART '07, pp. 89-98. IEEE Computer Society, 2007

[Abreu, 2008] R. Abreu, A. González, P. Zoeteweij, A.J.C. van Gemund. *On the performance of fault screeners in Software Development and Deployment.* In Proc. ENASE '08, pp. 123-130. INSTICC Press, 2008

[Abreu, 2009] R. Abreu, P. Zoeteweij, R. Golsteijn, A.J.C. van Gemund. *A practical evaluation of spectrum-based fault localization*. Journal of Systems and Software, 2009, doi:10.1016/j.jss.2009.06.035

[Augusteijn, 2002] L. Augusteijn. *Front: a front-end generator for Lex, Yacc and C,* Release 1.0, 2002. http://front.sourceforge.net/

[Do, 2005] H. Do, S.G. Elbaum, G. Rothermel. *Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact*. Empirical Software Engineering: An International Journal, 10(4): pp. 405--435, 2005

[González, 2007] A. González. *Automatic Error Detection Techniques Based on Dynamic Invariants*. Master's thesis. Delft University of Technology and Universidad de Valladolid, 2007

[Harrold, 2002] J. A. Jones, M.J. Harrold, J. Stasko. *Visualization of test information to assist fault localization*. In Proc. ICSE '02, pp. 467-477. ACM Press, 2002

[Hutchins, 1994] M. Hutchins, H. Foster, T. Goradia, T. Ostrand. *Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria*. In Proc. ICSE '94, pp. 191-200. IEEE Computer Society, 1994

[Lattner, 2004] C. Lattner, V. Adve. *LLVM: a compilation framework for lifelong program analysis & transformation*. In Proc. CGO'04, pp. 75-88. IEEE Computer Society, 2004

[NXP] NXP Semiconductors website. http://www.nxp.com/

[van Ommering, 2000] R. van Ommering, F. van der Linden, J. Kramer, J. Magee. *The Koala component model for consumer electronics software*. IEEE Computer, March 2000

[Zoeteweij, 2007] P. Zoeteweij, R. Abreu, R. Golsteijn, A.J.C. van Gemund. *Diagnosis of embedded software using program spectra*. In Proc. ECBS'07, pp. 213-218. IEEE Press, 2007

Chapter 11

System-level resource management for user-perceived reliability

Authors: Zhe Ma, Francky Catthoor

Abstract: This chapter presents a system-level resource management methodology that can make intelligent decisions for resource allocations at runtime. Such decisions can minimize the user perceived system failures (such as deadline misses) while maximizing the resource usage efficiency. This methodology performs a comprehensive design/runtime combined trade-off management for all simultaneously running components. In this chapter, we show that the users' subjective priorities associated with different aspects of a system can help us derive performance constraints. Such constraints can then be used during the selection of Pareto-optimal trade-offs between resources usages and system performance at runtime in order to maximize the user perceived quality. This can be achieved while keeping the hardware resource cost significantly lower than that required by the traditional over-dimensioning approaches based on the static worst-case analysis. Two case studies will be presented to illustrate and demonstrate our concept of user-perceived system reliability and how to derive resource versus performance trade-offs.

11.1 Introduction

As advanced semiconductor processing technology has enabled the fabrication of Multi-Processor System-On-a-Chip (MPSoC), the embedded software running on such MPSoC systems becomes increasingly sophisticated to provide more and better service. The increased complexity of the embedded software, together with their intrinsic real-time requirement, has made the whole system more fault-prone in the parametric sense. Moreover, in the very cost-sensitive domain of consumer electronics, a system cannot just over-allocate hardware resources to enable the conventional fault tolerance techniques (e.g. triple-module redundancy). Therefore, a methodology that can minimize the user-perceived system faults while maximizing the resource efficiency becomes necessary.

In a real-time embedded system, a user-perceived failure can be either a functional failure or a timing failure. Both failures can create disturbing artifacts even if the system can still manage to avoid a total crash. Nevertheless, these repetitive or transient artifacts can give users an impression of low reliability. The idea of user-oriented fault tolerance in an embedded system is to minimize the disturbed feeling of users due to the artifacts and thereby maximize the user-perceived system reliability.

Unfortunately for system designers, it becomes more difficult to fulfill the users' requirements on reliability. This is because today's embedded systems in the multimedia and communication domains encounter more dynamic behaviors from software applications (e.g. adaptive video codec), hardware architecture (e.g. cache misses) and the external environment (e.g. the users of a system can use more combinations of functionalities, the Internet-based contents fetching and uploading). Because of the increasingly dynamic behavior, the resource requirements of a system become very volatile. Simply over-allocating resources is prohibitively expensive, whereas insufficient resources would lead to either functional or timing faults. Therefore, it becomes a challenging problem to intelligently allocate resources to maintain the correct running of a system without the need of an expensive high-end hardware platform. In order to enable efficient resource allocations, an essential step is to identify the user-perceived importance of each individual component and to explore the best performance-resource trade-off points for each component, while postponing the decision of the overall user experiences to runtime. This requires both a thorough user-perceived quality analysis per component at design-time and an effective evaluation of the system-wide combined quality of all concurrent components at runtime.

Based on a decade of research we have developed a comprehensive design-time/runtime system-level trade-off management methodology (presented in [Ma, 2007]). This system-level trade-off methodology provides the required decision-making mechanism for the user-oriented fault tolerance. This is essentially a hierarchical decision-making at local and global levels. The local decision (component-level decision) is made at design time. The decision result is not a single configuration for a component, but a set of Pareto-optimal configurations. The global decision (system-level decision) is postponed until runtime when sufficient information is available to choose among the available Pareto-optimal configurations.

In this chapter, we describe how to use this methodology for the handling of user-perceived parametric system reliability. Specifically, through two case studies, we show that the priorities associated with different aspects of a system can help us derive performance versus resource trade-offs at design-time. Such trade-offs can then be represented by Pareto-optimal system configurations and can be selected at runtime to guarantee appropriate resource allocations and to maximize the user experiences. This can be achieved while keeping the hardware cost significantly lower than that required by the traditional worst-case design approaches.

In the rest of this chapter, we first present the structure of our system-level trade-off management methodology, together with the algorithm for runtime decision making (Section 11.2). Then we present two case studies. In the first case study (Section 11.3), we show how to trade performance with the processor cycles on a video enhancement system (Vproc system on NXP PNX8550 chip). In the second case study (Section 11.4.3), we show how to trade performance with the memory bandwidth on a video encoder (MPEG-4 encoder mapped on an MPSoC simulator). Finally, we summarize this chapter in Section 11.5.

11.2 System-level trade-off management

Our system-level trade-off management methodology has a design-time preparation phase and a runtime decision making phase. While the design-time phase typically requires component- and platform-specific information, the runtime phase is similar for all running components. We will first give an overview of our methodology; then we will use an example table to illustrate what is the result from the design-time preparation; finally we will show how to formulate the problem of making runtime decisions and what algorithms can be used to solve this problem.

11.2.1 Overview of the two-phase trade-off management

In our two-phase approach, first a design-time exploration per component leads to a set of possible operating points in a multi-dimension search space (Figure 11.1). Only points that are better than the other ones in at least one dimension are retained. They are called *Pareto points*. The resulting set of Pareto points is called the Pareto set. Typical dimensions are costs (e.g. energy consumption), constraints (e.g. performance) and used platform resources (e.g. memory usage, processors, clocks, communication bandwidth). In addition to the Pareto sets, we also need to identify the user-perceived subjective importance of each component under all scenarios. In order to generate actual executable code, each Pareto point is annotated with a code version referring to a specific implementation of the application, with specific task mapping and data transfers schedules between the shared and processor local memories of the MPSoC platform [Ykman-Couvreur, 2006]. Therefore, identifying different Pareto points actually means that the components developers to create and configure programs that can run in more than a single mode. This will increase the designer's efforts. However, in the next sections we will see some techniques that can automatically perform the analysis and pruning of such trade-offs. IMEC has even developed a prototype tool that can generate multiple versions of code in some cases (see Section 11.4).

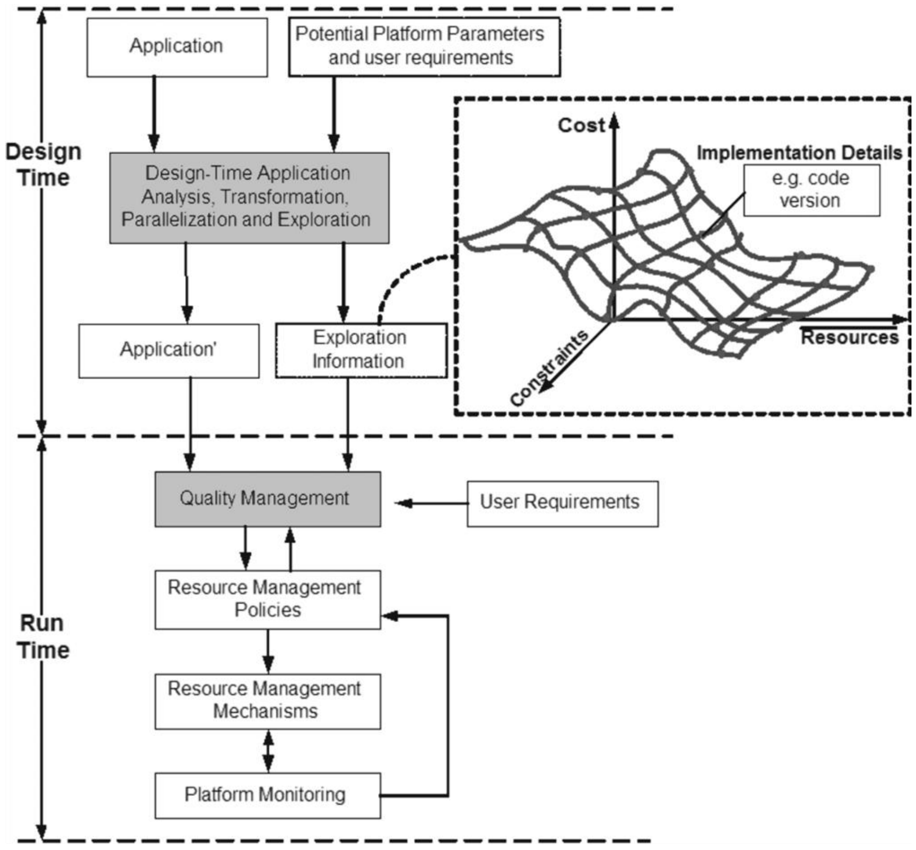

Design Time
Application
Potential Platform Parameters and user requirements
Design-Time Application Analysis, Transformation, Parallelization and Exploration
Application'
Exploration Information
Cost
Implementation Details
e.g. code version
Resources
Constraints
Run Time
Quality Management
User Requirements
Resource Management Policies
Resource Management Mechanisms
Platform Monitoring

Figure 11.1 *Two-phase trade-off management: at design time an exploration phase takes place and results are stored in a multi-dimension set; at runtime, the right operating point is chosen by an operating point selection algorithm.*

The full exploration is done at design-time, whereas the critical decisions with respect to application quality are taken during the second phase by a low-complexity runtime manager, created on top of the basic OS layer (Figure 11.1). Whenever the environment is changing (e.g., when a new application/use case starts, or when the user requirements change), for each active component, the runtime manager has to select an operating Pareto point from the multi-dimension search space. The selection is performed according to the available platform resources, in order to maximize the total amount of components' performance values, weighted by their importance, while respecting all constraints. This is illustrated in Figure 11.2, restricted to energy resource allocations. When application A starts, a first operating Pareto point is selected assigning A to three Processing Elements (PEs) with a slow clock (ck2). As soon as application B starts, a new operating point is needed to map A on only two PEs. By speeding up the clock (ck1), the application deadline is still met. After A stops, B can be spread over three PEs in order to reduce the energy consumption. This runtime optimization problem can be modeled as a classical Multi-choice Multi-dimension Knapsack Problem (MMKP), which is NP-hard.

Several different algorithms already exist for solving MMKPs: they provide either exact or near-optimal solutions. However they are still too slow for the context of multi-processor runtime management. Indeed the speed of the heuristic must be within acceptable runtime boundaries. As a reference, the time required to start a new component using the Linux OS is in the order of magnitude of 1ms to 10ms depending on the platform. This time should include not only the operating point selection, but also the required platform resource assignment.

The low-complexity runtime manager incorporated on top of the basic OS services maps the components on the MPSoC platform and performs 2 subtasks: (1) it selects the next operating Pareto point while globally optimizing costs and resources across all active applications, according to the valid constraints (e.g. performance, user requirements) and available platform resources. (2) Next to that, another part of this runtime manager performs at low cost switches between possible configurations of the component, as required by environment changes, while incorporating the switch overhead.

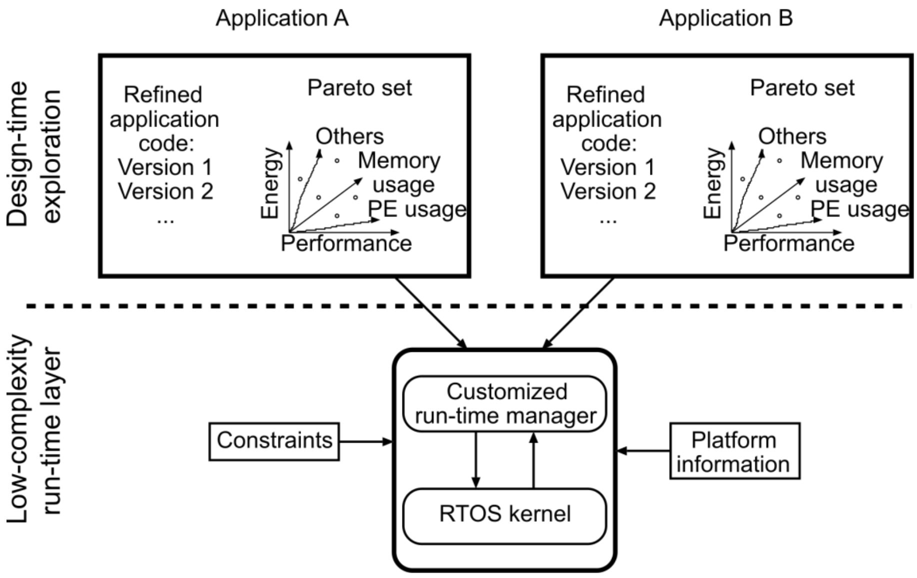


Figure 11.2 *Our runtime trade-off management*

More in particular, the low-complexity runtime manager (Figure 11.2) provides the following services:

1. when a new component is activated, our runtime manager parses its Pareto set provided by the design-time exploration;
2. when the environment is changing (e.g., when a new application/use case starts, or when the user requirements change), our runtime manager first selects in a predictable way a Pareto point from its Pareto set for each running component while maximizing the total user experience without violating constraints. Then it informs each component of the new Pareto point and assigns the platform resources, it adapts the platform parameters, and it invokes the switching of the active Pareto point in all running components.

11.2.2 Design-time preparations

Our design-time analysis should provide a weight factor and a table for each component, each row of this table shows that when a component runs at a certain percentage of its full performance, how much of each resource will be consumed.

For instance, if we need to consider the system resources trade-offs for a component mapped onto a dual-DSP hardware architecture, the component designer should provide a table such as below to characterize all Pareto optimal trade-off options of this component.

DSP1	DSP2	Bus BW	Functionalities
30%	70%	50%	100%
20%	10%	10%	70%
20%	0%	0%	30%

Table 11.1 *Pareto-optimal mapping configurations*

In Table 11.1, the designer provides 3 Pareto-optimal mappings for his component. For each mapping, this component needs an allocated resource budget for each resource and provides a certain amount of value (performance or functionalities). In general, the more constrained resource budgets a component has, the slower performance or less functionalities it can provide.

Also, we need a weight factor for this component in the final system where this component is deployed. A higher factor means a more significant user-perceived system slow-down/malfunction artifact is associated with this component.

11.2.3 Runtime decision-making

The runtime optimization should be the maximization of aggregated performance numbers weighted by their weight factors. This operating point selection can be formulated as follows. It has to select exactly one Pareto point from each active set, according to the available platform resources, in order to minimize the total resources consumption of the platform, while respecting all application deadlines.

This maximization problem is a classical Multi-choice Multi-dimension Knapsack Problem (MMKP). The MMKP is generally stated as follows. Suppose p sets of points are present. Each set i contains N_i points (for complexity analysis, in the following, N_i is assumed constant and = N). Point j in Set i has value v_{ij} (weighted by the weight factor), and it requires resources r_{ijk}, $0 <= k < m$. The amount of available resources are given by R_k, $0 <= k < m$. The MMKP is to pick exactly one point from each set in order to maximize the total value of the pick, subject to the resource constraints. Formally the MMKP is expressed as follows:

$$\begin{aligned}
&Maximize \quad \sum_{0\le i<p} \sum_{0\le j<N_i} x_{ij}v_{ij} \\
&Subject\ to \quad x_{ij} \in \{0,1\}, 0 \le i < p, 0 \le j < N_i \\
&\qquad \sum_{0\le j<N_i} x_{ij} = 1,\ 0 \le i < p \\
&\qquad \sum_{0\le i<p} \sum_{0\le j<N_i} x_{ij}r_{ijk} \le R_k,\ 0 \le k < m.
\end{aligned}$$

Finding exact solutions for the MMKP is NP-hard [Parra-Hernandez, 2005]. Algorithms such as [Armstrong, 1983], [Khan, 1997] and [Koleser, 1967] have an execution time that increases dramatically with the number of sets. They are not applicable for our operating point selection, where a solution must be provided at runtime on the MPSoC platform within 1ms. A fast and effective heuristic is required. Existing ones, suitable for real-time systems, are now summarized.

Khan's heuristic [Khan, 2002] applies the concept of aggregate resource consumption as measurement to select one point in each set. It has a worst-case complexity of $O(mp^2(n-1)^2)$, and it finds solutions with a total value on average equal to 94% of the optimum.

Moser's heuristic [Moser, 1997] is based on Lagrange multipliers and on the concept of graceful degradation. It performs on average better than [Khan, 2002] in terms of optimality. Nevertheless its rate of failure in finding a feasible solution is higher.

Akbar's heuristic [Akbar, 2006] first reduces the multi-dimension search space into a two-dimension one, and then constructs the convex hull of each set to reduce the search space. Its worst-case complexity is only $O(mpN + pN\log(pN) + pN\log(p))$, but it finds less optimal solutions than [Khan, 2002].

Hernandez's heuristic [Parra-Hernandez, 2005] relaxes the MMKP to a multi-dimension knapsack problem. Its rate of failure in finding a feasible solution is lower than in than [Khan, 2002], its solution quality is also better, but its complexity is higher.

Tabu search [Dammeyer, 1991], simulated annealing [Drexel, 1988] and genetic algorithms [Khuri, 1994] can also be applied. However they are much costlier than [Khan, 2002]: genetic algorithms have an exponential worst-case complexity, since they can explore all point combinations; tabu search and simulated annealing are based on looking at all neighboring point combinations of the current solution.

Fast greedy heuristics solving multi-choice knapsack problems (with several sets, but one resource) also exist for runtime task scheduling on embedded systems [Yang, 2003] and runtime quality-of-service management in wireless networks [Mangharam, 2005]. However for MMKPs, an important issue is first to provide an initial feasible solution. For the multi-choice knapsack problem, a feasible solution, if existing, can always be obtained by choosing the lowest-cost element of each set. For MMKPs, this is not the case and these fast greedy heuristics are not applicable.

In conclusion, the most appropriate heuristics for the operating point selection (i.e. generating a good solution, with little computational runtime effort, and low rate of failure in finding a feasible solution) are [Akbar, 2006] and [Khan, 2002]. But they are still too slow to be integrated into a multi-processor platform. Hence a faster heuristic has been developed for MPSoC runtime management (see [Ma, 2007] for all the technical details of this heuristic algorithm).

11.3 Handling of DSP overloading

This section presents a case study of the resource management with the Video Processing (vproc) component of NXP PXN8500 chip in a digital TV. This vproc component includes software running on two DSP processors as well as other hardware accelerators.

In this case study, we have first created a resource competitor (a DSP cycle eater) on one of the two DSP processors and then observed that the resource contention without a system-level resource management can lead to user-perceived faulty behaviors. Then we have tried to incorporate multiple trade-off options into both the vproc component and the resource competitor such that the runtime resource manager can dynamically re-configure both components.

11.3.1 Identification and implementations of resource allocation trade-offs

The vproc component uses both of the two DSP processors as well as a number of hardware accelerators. The vproc component has two principal software tasks: the vproc main control loop and the Software Nature Motion (SNM). The vproc component uses a master-slave cooperation scheme among its principle tasks: the vproc main running on DSP1 (TriMedia) is the master. It contains a sequence of controlling commands for all the hardware IPs as well as the SNM which resides on DSP2 (TriMedia) for a coordinated way of video processing.. Because the hardware accelerators are hardware IPs specific for vproc’s purposes, we cannot use them for other components in the system. Therefore, we focus on the resource of DSP processors, i.e., their cycle budgets.

For vproc, we have a fixed performance constraint, i.e. it must process one video frame before the next video frame arrives, and typically 25 frames would come in a second to give users a smooth video playback. Either we could run the vproc faster for each video frame (by finding a more parallel schedule for speedup) or we could run less number of tasks (loss of functionalities). Because the vproc component includes many tasks running on hardware accelerators and thus cannot be easily re-scheduled, and also because a large proportion of time is spent during a motion compensation algorithm of SNM that is hard to run in parallel processors, we chose to go for a smaller number of tasks in vproc’s trade-off options. By choosing different levels of functionalities losses, we could allow the vproc component to work with different DSP resources consumption. In particular, we have identified two levels of functionalities for vproc: the level of full functionalities and the level without SNM.

The two levels of functionalities correspond to two control flows in vproc. Once these trade-off options are identified, we add them in the component by writing additional code to dynamically switch between two control flows. This additional code provides an interface for runtime resource manager to control the behaviors of the vproc component. We have used a software profiling tool (TimeDoctor) to instrument and measure the cycles on each DSP processor for both configuration levels. This measurement provides the following two Pareto points for our modified vproc component. Please note that we measured the loss of functionalities by using the ratio between the number of remaining tasks and the number of the complete set of tasks.

DSP1	DSP2	Functionality
20%	90%	Full (100%)
20%	1%	No SNM (80%)

Table 11.2 *Mapping configurations for vproc*

Another implementation decision related to the runtime management is the runtime monitor. The DSP overload could be detected in software by monitoring the running time of the IDLE task in a given period. However, more efficient hardware monitor could be implemented for this purpose in the future.

We have added a new software component into the system as the resource competitor: a counter based on an Inverse Discrete Cosine Transformation (IDCT) benchmark. This counter runs the IDCT benchmark repeatedly, and increases its count after each round of IDCT. For this IDCT component, we have identified two trade-off options, see Table 11.3:

DSP1	DSP2	Functionality
90%	0%	Full (100%)
0%	90%	Full (100%)

Table 11.3 *Mapping configurations for IDCT*

The two trade-off options for IDCT component are actually corresponding to two ways of mapping it onto DSP processors. The important thing here is that for an existing instance of IDCT component, it should keep a consistent component status no matter which option is taken by the system-level resource manager. That is, when the system-level resource manager asks the IDCT component to switch from one option to another option, the user of the IDCT component should not observe any difference. In this simple example, it means the IDCT must resume its computation and remember how many times it has run the IDCT computation as soon as the computation is re-mapped to the other DSP processor.

11.3.2 Demonstration with the resource manager

During our demonstration, we have first only launched the vproc component, and the system-level resource manager immediately chose option 1 for it. So both DSP processors are maximally used and full functionalities are obtained. Then we have launched the IDCT component with the resource manager deactivated. We have observed very disturbing video frame dropping artifacts because of the resource contentions between the IDCT and vproc components. When the resource manager is re-activated, it selects option 2 for vproc and option 2 for IDCT component. This selection ensures that both vproc and IDCT component can get sufficient resources to run smoothly and artifacts due to the loss of SNM are significantly less disturbing than the frame dropping.

11.4 Handling of memory bandwidth overloading

This section presents an application of system-level resource management in the allocation of the bus bandwidth between processing elements.

In a scenario where multiple components need to run simultaneously on several processing elements that share a common interconnection, the system-level resource manager needs to allocate appropriate interconnection bandwidth to each component. Because of dynamic behaviors of these components or other parts of the system, this bandwidth allocation must be adjusted continuously at runtime to get the maximal performance/functionality values from the active components.

As explained earlier, for each component we need a set of Pareto optimal trade-off options. In this section, we focus on the trade-offs between performances and bandwidth consumptions of components. In order to obtain such trade-offs in the form of Pareto sets, this section will present a genetic algorithm to automatically explore the Pareto optimal solutions of the mappings for each component on the target processing elements. These generated Pareto sets can be then used by the system-level resource manager to decide the appropriate allocations at runtime.

We will first present the formulation of this mapping problem; then we will describe our genetic algorithm for the automatic exploration of Pareto optimal mappings; after that, we will show an experiment on an MPEG-4 AVC (Advanced Video Coding) encoder in an embedded software code generation tool developed at IMEC for multi-processor platforms.

11.4.1 Problem formulation of mapping explorations

In our model, each component is represented by a task graph. Each task in the task graph can be started once all its control and data dependencies are fulfilled, i.e., all its predecessor tasks are finished. We consider that this task graph can be mapped to a number of connected processing elements which share a common bus for inter-task communication. When two communicating tasks are running on the same processing element, they do not consume the bus bandwidth; otherwise, they use the bus for data communicating between them. Therefore a specific task mapping and scheduling can give a specific execution time as well as a bandwidth requirement for a component. In general, if a task mapping has more tasks running in parallel, it could lead to a faster schedule but it will typically also consume more bandwidth; on the other hand, a more sequential execution is slower but consumes less bandwidth.

The purpose of mapping exploration is to find those Pareto optimal mappings. A task mapping is considered as Pareto optimal if for all valid mappings that are faster than it, there is no mapping consumes less bandwidth and for all valid mapping that consumes less bandwidth than it, there is no mapping runs faster. Therefore this exploration is essentially pruning all the sub-optimal mappings as illustrated in Figure 11.3. The line linking all remaining results indicates the boundary of the best trade-off solutions, i.e., all points on this line are equal in terms of Pareto optimality.

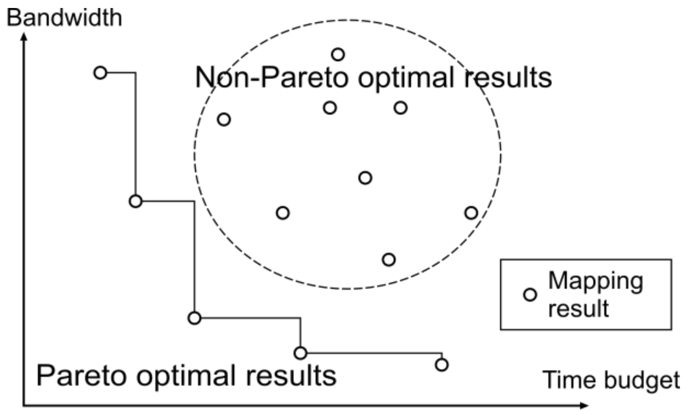


Figure 11.3 *Illustration of Pareto optimal solutions*

11.4.2 Our algorithm for automatic explorations

When a component has a small and simple task graph and the number of processing elements is small, an intuitive enumeration can be performed to identify all valid mappings and to prune the sub-optimal ones. But such intuitive and exact algorithms cannot be scaled up to larger problems due to the computation intractability of such multi-objective optimization problems. For a problem with several dozens of tasks and processing elements, to find a set of Pareto optimal solutions in a reasonable amount of time (say several minutes) will often require the help of certain heuristic algorithms (i.e. non-exact algorithms); otherwise, an exact algorithm (such as branch-and-bound) could take an extremely long time to find solutions when the problems' sizes scale up.

Many evolutionary algorithms have been proposed for solving the multi-objective optimization problem [Deb, 2001], in particular the genetic algorithms such as NSGA [Srinivas, 1994] and NSGA-II [Deb, 2002] have been reported to be effective. The basic idea of these genetic algorithms is to mimic the real-life evolutions of lives guided by the nature selection principle. This algorithm keeps a pool of valid solutions and let them to evolve. Only the good solutions (judged by a merit function) survive while others are removed from the pool. The survivors can exchange their genes and generate offspring solutions. As the selection of optimal solutions is continued for many times, the remained survivors should be close to the optimal solutions. The essential step of applying such genetic algorithm in our case is to find an effective way to represent different mapping with the gene combinations.

We use genes to represent the task allocation in our implementation. That is, for a task graph with N tasks, each mapping solution has N genes. Each gene represents an individual task. When a task is allocated to the processing element X, then the value of gene representing this task is X.

The first step of an exploration is to create a pool of valid random solutions. The size of pool should be adequate to allow further evolution.

Then for each objective of our multi-objective optimization problem, we evaluate all valid solutions. That is, we calculate the total execution time and bandwidth consumption for a task graph where all tasks are allocated and scheduled to the target processing elements. The total execution time is the make-span (i.e. from the start of the first task to the end of the last task) of the task schedule generated by a list scheduling algorithm (ASAP scheduling, where competing tasks are prioritized according to the accumulated execution times of their offspring tasks). The total bandwidth consumption is the total amount of data transfers between processing elements divided by the total execution time.

After the evaluation, only the Pareto optimal solutions are kept while other solutions are removed from the pool. An interesting alternative option here is to keep also the immediately next Pareto optimal solutions to have a larger number of survivors and hence more diversities for further evolution. That is extremely useful for Pareto point switching, especially. The survivors then randomly exchange some genes to generate valid solutions for the next generation. When sufficient new solutions are generated, we could start the evaluation of the next generation.

The evolution can go on for a pre-determined number of generations and the Pareto optimal solutions from the survivor solutions of the pool are picked up as the result.

We have implemented this genetic algorithm in `Python` code based on a previous implementation of NGSA-II [Fita, 2007]. To test our implementation, we have used an open source tool called TGFF (Task Graph For Free) to generate random task graphs and multi-processor architectures for experiments. We have done experiments with this genetic algorithm for a few dozens of random task graphs with up to 50 tasks inside each graph. The evaluation program could derive effective Pareto sets within a minute on a PC with CPU running at 2.8GHz (with 10 generations and 50 solutions per generation). Note that each run not only gives the Pareto optimal set, but also the next four immediate sub-optimal sets. Further experiments could be done in the future to study how many generations of evolutions do we need to derive good results for real-life inputs.

11.4.3 AVC experiment: how to generate code for efficient switches

IMEC has developed a prototype MPSoC tool. It can generate multiple versions of code which represent different parallelizations. By using this tool, we do not require the task migration support from the underlying OS layer. Instead we generate multiple versions of code, one version of code for each Pareto optimal trade-off solution. For instance, if our exploration would find that in a Pareto optimal solution two tasks need to be allocated to the same processing element, this tool could cluster the source code corresponding to these two tasks and generate the code of a single task running on that processing element. Therefore, an application programmer only need to instrument additional code in his source code to help this MPSoC tool to indentify the possibilities to create task-level parallelism; this tool can automatically generate the parallelized code; after profiling the generated code to get the cycles numbers and bandwidth consumptions, the programmer can use our genetic algorithm to find the Pareto optimal ways of mapping the parallelized code on a multi-processor platform; each Pareto point represents a specific version of parallelized code; finally a switch function can be implemented together with all versions of parallelized code, and chooses the proper version at runtime.

We have analyzed an AVC encoder and our exploration tool finds 3 Pareto optimal solutions out of 7 valid mappings. These 3 mapping solutions are then fed to the MPSoC tool to generate the corresponding versions of code. We then use the cycle accurate simulator (based on IMEC ADRES processor [Mei, 2005]) to measure the total execution cycles and the amount of data transfer on the interconnection. We have generated the final hybrid code based on the 3 versions of code. The dynamic switch between different versions happens at the granularity of video frames. That is, we only allow the system-level resource manager to choose which version of code to run at the beginning of encoding a video frame. The measured results for different mixes of the 3 versions of code (version 0, 1 and 2) are illustrated in the Figure 11.4. Please note that measured amount of data transfers include both the data transfer between tasks and the intrinsic data transfers of the tasks (i.e. read/write data inside a task), therefore the measurement represents the realistic usage of interconnections by the AVC encoder.

Although the switching between different versions of code takes place at video frame level, the switching overheads remain small thanks to the hardware-assisted dynamic threading. This low-overhead switching has been shown by the following experiment: we performed different numbers of switching for ten video frames and then compared their switching overheads. As illustrated in Figure 11.5: we can see that the total cycle numbers are very close for a single switching (the two left red circles) and five consecutive switching (the two right red circles). This means 5 switches between code version 1 and 2 do not incur significantly more cycles than a single switch.

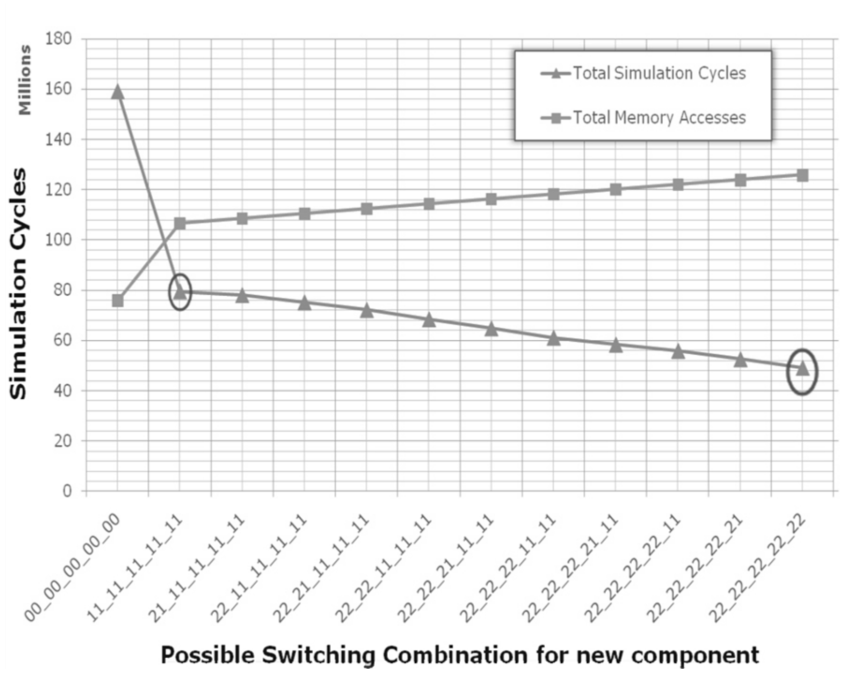


Figure 11.4 *AVC mapping trade-offs*

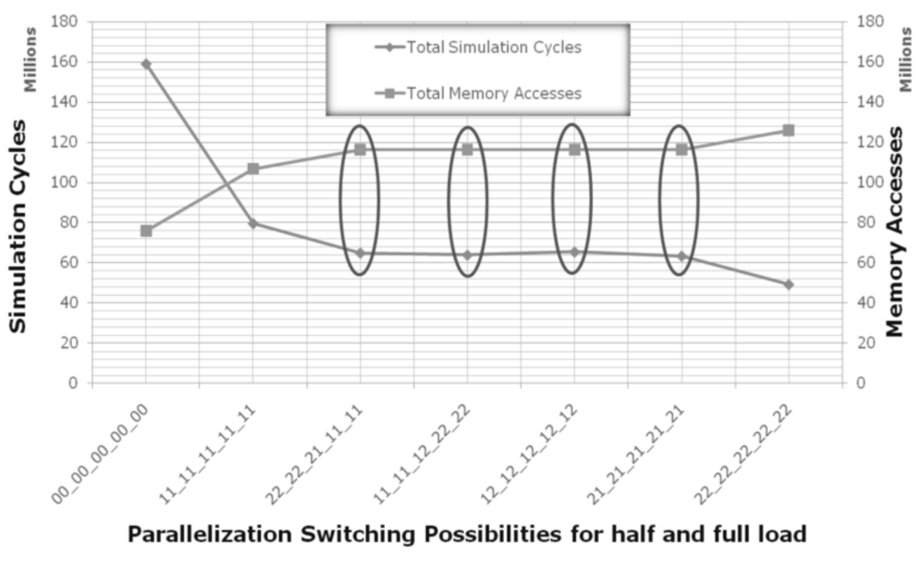


Figure 11.5 *AVC switching overheads*

11.5 Summary

In this chapter we have shown a two-phased approach to optimize the resource management for dynamic embedded systems. The design-time phase exploration of resource management trade-offs is performed for each component. This exploration requires a great amount of insights into a particular component and the platform, though some techniques can help to reduce the design-time efforts. An automatic exploration technique has been presented to speed up the performance versus bandwidth trade-offs. A fast heuristic algorithm has also been discussed to perform the runtime phase decision making. Overall this combined approach should allow a multi-component embedded system to still provide greater user-perceived reliability under dynamic runtime conditions while the provision of system resources is significantly more limited than that based on the worst case requirements.

11.6 References

[Akbar, 2006] M. Akbar, M. Rahman, M. Kaykobad, E. Manning, and G. Shoja. *Solving the multidimensional multiple-choice knapsack problem by constructing convex hulls*. Computers and Operations Research, 33(5): pp. 1259-1273, 2006

[Armstrong, 1983] R. Armstrong, D. Kung, P. Sinha, and A. Zoltners. *A computational study of a multiple choice knapsack algorithm*. ACM Transactions on Mathematical Software, 9: pp. 184-198, 1983

[Dammeyer, 1991] F. Dammeyer and S. Voss. *Dynamic tabu list management using the reverse elimination method.* Annals of Operations Research, 1991

[Deb, 2001] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons, Ltd. 2001

[Deb, 2002] Kalyanmoy Deb, A. Pratap, S. Agarwal and T. Meyarivan. *A fast and elitist multiobjective genetic algorithm: NSGA-II*. IEEE Transactions on Evolutionary Computation, 6(2): pp. 182-197, 2002

[Drexel, 1988] A. Drexel. *A simulated annealing approach to the multiconstraint zero-one knapsack problem.* Annals of Computing, 40: pp. 1-8, 1988

[Fita, 2007] M. Fita. *Implementation of NSGA-II in Python*. http://code.google.com/p/py-nsga-ii , 2007

[Khan, 1997] S. Khan, K.F. Li, and E. Manning. *The utility model for adaptive multimedia systems*. Proceedings of the International Workshop on Multimedia Modeling, pp. 111-126, 1997

[Khan, 2002] S. Khan, K. Li, E. Manning, and M. Akbar. *Solving the knapsack problem for adaptive multimedia system.* Studia Informatica Universalis, pp. 161-182, 2002

[Khuri, 1994] S. Khuri, T. Back, and J. Heitkotter. *The zero/one multiple knapsack problem and genetic algorithms.* Proceedings of the ACM Symposium of applied computation, 1994

[Kolese, 1967] P. Koleser. *A branch and bound algorithm for knapsack problem*. Management Science, 13: pp. 723-735, 1967

[Ma, 2007] Z. Ma, P. Marchal, D. Scarpazza, P. Yang, C. Wong, J. Gomez, S. Himpe, C. Ykman-Couvreur and F. Catthoor. *Systematic methodology for real-time cost-effective mapping of dynamic concurrent task-based systems on heterogeneous platforms*. Springer, 2007

[Mangharam, 2005] R. Mangharam, S. Pollin, B. Bougard, R. Rajkumar, Fr. Catthoor, L. Van der Perre, and I. Moerman. *Optimal fixed and scalable energy management for wireless networks*. Proceedings of the IEEE Conference on Computer Communications, Miami, FI, USA, March 2005

[Mei, 2005] B. Mei, A. Lambrechts, D. Verkest, J. Mignolet, R. Lauwereins. *Architecture Exploration for a Reconfigurable Architecture Template*. IEEE Design & Test of Computers 22(2): pp. 90-101 (2005)

[Moser, 1997] M. Moser, D. Jokanovic, and N. Shiratori. *An algorithm for the multidimensional multiple-choice knapsack problem.* IEICE Transactions on Fundamentals of Electronics, 80(3): pp. 582-589, 1997

[Parra-Hernandez, 2005] R. Parra-Hernandez and N. Dimopoulos. *A new heuristic for solving the multichoice multidimensional knapsack problem*. IEEE Transactions on Systems, Man, Cybernetics - Part A: Systems and Humans, 35(5): pp. 708-717, 2005

[Srinivas, 1994] N. Srinivas and Kalyanmoy Deb. *Multiobjective Optimization Using Nondominated Sorting in Genetic Algorithms*. Evolutionary Computation, 2(3): pp. 221-248, 1994

[Yang, 2003] P. Yang and F. Catthoor. *Pareto-optimization-based runtime task scheduling for embedded systems*. Proceedings of the International Symposium on System Synthesis, pp. 120-125, California, USA, October 2003

[Ykman-Couvreur, 2006] C. Ykman-Couvreur, V. Nollet, T. Marescaux, E. Brockmeyer, F. Catthoor and H. Corporaal. *Pareto-Based Application Specification for MP-SoC Customized Runtime Management.* Proceedings of 2006 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, pp. 78-84, Samos, Greece, July 2006

Chapter 12

Software architecture design for local recovery

Author: Hasan Sözer, Bedir Tekinerdoğan, Mehmet Akşit

Abstract: Local recovery is an effective fault tolerance technique to attain high system availability. For achieving local recovery the architecture needs to be decomposed into separate units that can be recovered in isolation. It appears that this required decomposition for recovery is usually not aligned with the existing decomposition of the system. We propose a systematic approach for decomposing software architecture to introduce local recovery. We have developed a set of analysis tools for evaluating decomposition alternatives and we have developed a framework to support the realization of a selected decomposition. We discuss our experiences in the application and evaluation of the approach for introducing local recovery to the open-source media player called MPlayer.

12.1 Introduction

One of the key principles in software architecture design is modularity that aims to decompose the system into separate, modular units [Parnas, 1972]. The decomposition of a system into modules is usually driven by the required quality concerns such as adaptability, reuse, and performance. Likewise, fault tolerance mechanisms for error recovery also have an impact on the decomposition of the software system.

Recovery can be applied at different levels of granularity. In case of global recovery, the system is recovered as a whole when errors are detected. For example, in case of a deadlock, restarting the whole system makes it completely unavailable until the system is in its normal operational mode again. This lack of availability can be avoided by applying local recovery in which only the erroneous parts of the system are recovered. To recover from a deadlock, for instance, only the modules that are involved in the deadlock need to be restarted, while the other parts can remain available. Local recovery has an additional benefit because it also decreases the mean time to recover [Candea, 2004]. Hence, for better availability and faster recovery, it is necessary to reduce the granularity of the parts in the system that can be recovered and as such realize local recovery. For achieving local recovery, the corresponding system needs to be separated into a set of isolated *recoverable units* (RUs) so that the propagation of errors can be prevented. However, it appears that this required decomposition for recovery is usually not aligned with the decomposition based on functional concerns.

In principle, software architecture decomposition for local recovery should be considered early in the software development life cycle. However, there exist many software systems that have been already developed without local recovery in mind. Some of these systems comprise millions of lines of code and it is not possible to develop these systems from scratch within time constraints. We have to refactor and possibly restructure them to incorporate local recovery. We propose a systematic approach for decomposing software architecture to introduce local recovery. We have developed a set of analysis tools to support the approach in selecting an optimal decomposition with respect to availability and performance. We also introduce a framework FLORA, which supports the realization of software architecture decomposition for local recovery. The framework provides reusable abstractions for defining RUs and the necessary coordination and communication protocols for their recovery. We discuss our experiences in the application and evaluation of the approach for introducing local recovery to the open-source media player called MPlayer [MPlayer, 2007].

The remainder of this chapter is organized as follows. In Section 12.2 we discuss the impact of local recovery on the architectural decomposition using a case study. In Section 12.3, we present the overall process. In Section 12.4, we introduce analysis techniques for evaluating and selecting decomposition alternatives for recovery. In Section 12.5, we introduce FLORA and illustrate its application. In Section 12.6, we evaluate our approach. We provide a summary of previous related studies in Section 12.7 and we conclude the chapter in Section 12.8.

12.2 Refactoring MPlayer for local recovery

We have applied local recovery to an open-source software, MPlayer [MPlayer, 2007]. MPlayer is a media player, which supports many input formats, codecs and output drivers. It embodies approximately 700K lines of code (LOC) and it is available under the GNU General Public License. In our case study, we have used version v1.0rc1 of this software that is compiled on Linux Platform (Ubuntu version 7.04). Figure 12.1 presents a simplified view of the MPlayer software architecture with basic implementation units and direct dependencies among them. In the following, we briefly explain the important modules, which are shown in this view. *Stream* reads the input media and provides buffering, seek and skip functions. *Demuxer* separates the input into audio and video channels. *Mplayer* connects the other modules, and maintains the synchronization of audio and video. *Libmpcodecs* embodies the set of available codecs. *Libvo* displays video frames. *Libao* controls the playing of audio. *Gui* provides the graphical user interface of MPlayer.

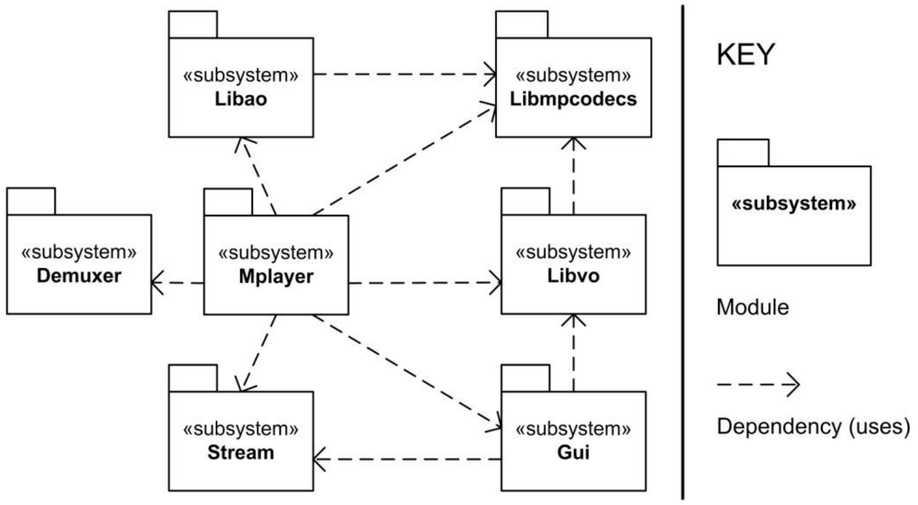


Figure 12.1 *A simplified view of the MPlayer software architecture.*

We have introduced a local recovery mechanism to MPlayer to make it fault-tolerant against transient faults. As one of the requirements of local recovery, the system has to be decomposed into several units that are isolated from each other. Note that the system is already decomposed into a set of modules. Decomposition for local recovery is not necessarily, and in general will not be, aligned one-to-one with the module decomposition of the system. Multiple modules can be wrapped in a RU and recovered together.

In the following sections, we explain how we introduce local recovery to MPlayer, for which we had to decompose the software architecture into several RUs. The communication between multiple units had to be controlled and recovery actions had to be coordinated so that the erroneous RUs can be recovered independently and transparently, while the other RUs are operational.

12.3 The overall process

Figure 12.2 depicts the overall process for introducing local recovery, which consists of the steps Architecture Design, Analysis and Realization. In the Architecture Design step, we expect that a module view [Clements, 2002] of the architecture is provided. Our approach is agnostic to the architecture design method that is used in this step. The module view of the described architecture is provided to the Analysis step as an input. In the Analysis step, the system is analyzed to define the decomposition of the architecture into a set of RUs. In the Realization step, the local recovery is realized with FLORA according to the selected decomposition. In the following sections, we discuss the Analysis and Realization steps in detail.

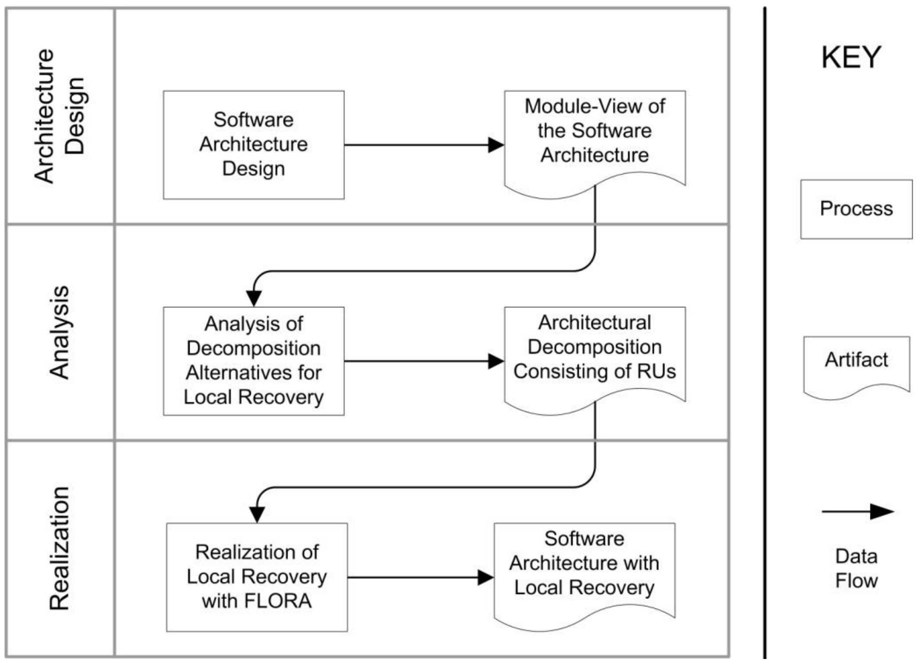

Architecture Design
Software Architecture Design
Module-View of the Software Architecture
Analysis
Analysis of Decomposition Alternatives for Local Recovery
Architectural Decomposition Consisting of RUs
Realization
Realization of Local Recovery with FLORA
Software Architecture with Local Recovery
KEY
Process
Artifact
Data Flow

Figure 12.2 *The overall process.*

12.4 Analysis of Decomposition Alternatives

One of the alternative decompositions for the MPlayer case is to partition the system modules into 3 RUs: 1) RU AUDIO, which provides the functionality of *Libao* 2) RU GUI, which encapsulates the *Gui* functionality and 3) RU MPCORE which comprises the rest of the system. Figure 12.3 depicts the boundaries of these RUs, which are overlayed on the MPlayer software architecture shown in Figure 12.1.

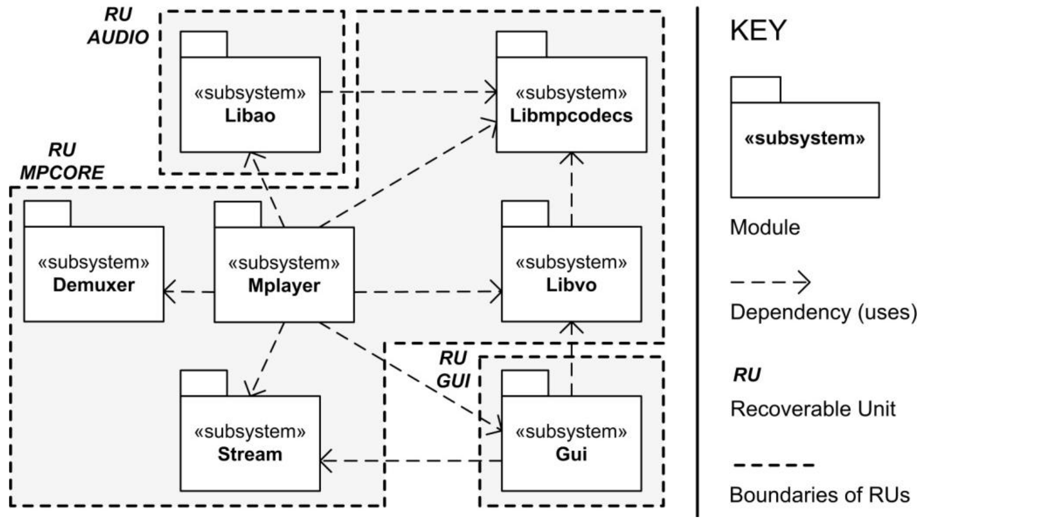
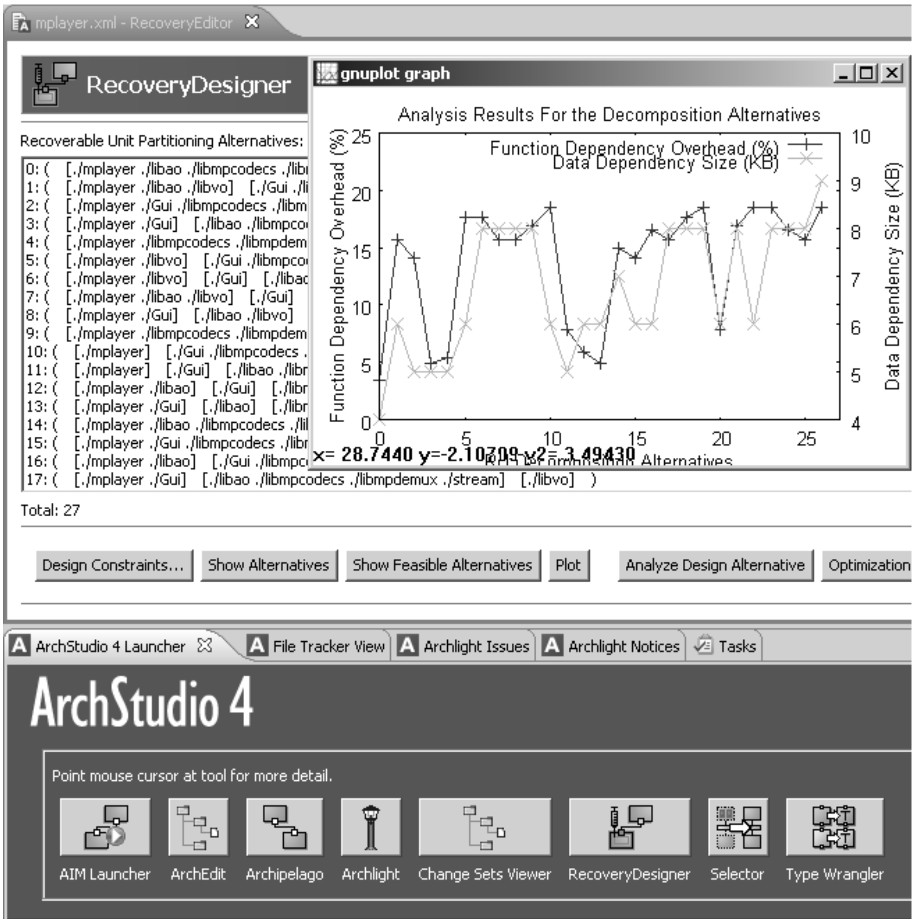


Figure 12.3 *MPlayer software architecture with the boundaries of the recoverable units for an alternative decomposition.*

Figure 12.3 shows only one of the possible decomposition alternatives. There are several ways in which a system can be partitioned into a set of RUs. Each alternative may have both benefits and drawbacks. The number of alternatives can be reduced based on the domain knowledge (e.g. some undependable modules have to be isolated from the critical parts of the system or some modules must not be separated from each other). For the remaining alternatives, there exists an inherent trade-off between the availability and performance criteria. On the one hand increasing availability will require increasing the number of RUs, in which the modules of the system are separately isolated. On the other hand, increasing the number of RUs leads to a performance overhead due to the dependencies between the separated modules in different RUs[1]. Therefore, selecting decomposition alternatives requires the evaluation with respect to these two criteria and making the desired trade-off. We have developed an analysis tool for this purpose. Figure 12.4 depicts a snapshot of this tool, which automatically generates all possible decompositions of system modules into a set of RUs. While doing so, it takes provided constraints into account, specifying which modules must be separated from each other and which modules must be kept together in a RU. The tool estimates the performance overhead and availability for each feasible design alternative. It also implements a set of optimization algorithms to propose the best design alternatives based on its estimations. In the following we briefly explain how performance overhead and availability estimations are performed.

[1] Increasing the number of RUs also leads to an additional development effort, which will be discussed later.



Recoverable Unit Partitioning Alternatives:

```
0: (   [./mplayer ./libao ./libmpcodecs ./libı
1: (   [./mplayer ./libao ./libvo]   [./Gui ./li
2: (   [./mplayer ./Gui ./libmpcodecs ./libm
3: (   [./mplayer ./Gui]   [./libao ./libmpco
4: (   [./mplayer ./libmpcodecs ./libmpdem
5: (   [./mplayer ./libvo]   [./Gui ./libmpco
6: (   [./mplayer ./libvo]   [./Gui]   [./libac
7: (   [./mplayer ./libao ./libvo]   [./Gui]
8: (   [./mplayer ./Gui]   [./libao ./libvo]
9: (   [./mplayer ./libmpcodecs ./libmpdem
10: (   [./mplayer]   [./Gui ./libmpcodecs .
11: (   [./mplayer]   [./Gui]   [./libao ./libr
12: (   [./mplayer ./libao]   [./Gui]   [./libr
13: (   [./mplayer ./Gui]   [./libao]   [./libr
14: (   [./mplayer ./libao ./libmpcodecs ./lil
15: (   [./mplayer ./Gui ./libmpcodecs ./libr
16: (   [./mplayer ./libao]   [./Gui ./libmpc
17: (   [./mplayer ./Gui]   [./libao ./libmpcodecs ./libmpdemux ./stream]   [./libvo]   )
```

Total: 27

```
x= 28.7440 y=-2.10709 y2= 3.49430
```

Figure 12.4 A snapshot of the analysis tool that evaluates software architecture decomposition alternatives for recovery.

12.4.1 Performance overhead analysis

Performance overhead is introduced due to the dependencies between the separated modules in different RUs. We distinguish between two important types of dependencies that cause a performance overhead; i) *function dependency* and ii) *data dependency*.

The function dependency is the number of function calls between modules across different RUs. For transparent recovery these function calls must be redirected, which leads to an additional performance overhead. For this reason, while selecting a decomposition alternative we should consider the number of function calls among modules across different RUs.

The data dependencies are proportional to the size of the shared variables among modules across different RUs. When an existing system is decomposed into RUs, there might be shared state variables leading to data dependencies between RUs. The size of data dependencies complicate the recovery and create performance overhead because the shared data need to be kept synchronized after recovery. This makes the amount of data dependency between RUs an important criterion for selecting RUs.

To be able to measure the function and data dependencies among the modules, we have profiled the system using Gprof [Fenlason, 2000] and Valgrind [Nethercote, 2007] tools, respectively. We have collected information regarding the number and frequency of function calls and size of shared data among the modules. For example, Figure 12.5 depicts a part of the automatically generated module dependency graph for MPlayer, in which the nodes represent source files belonging to various modules (modules are identified by the folder structure and naming convention) and edges represent the function calls among them.

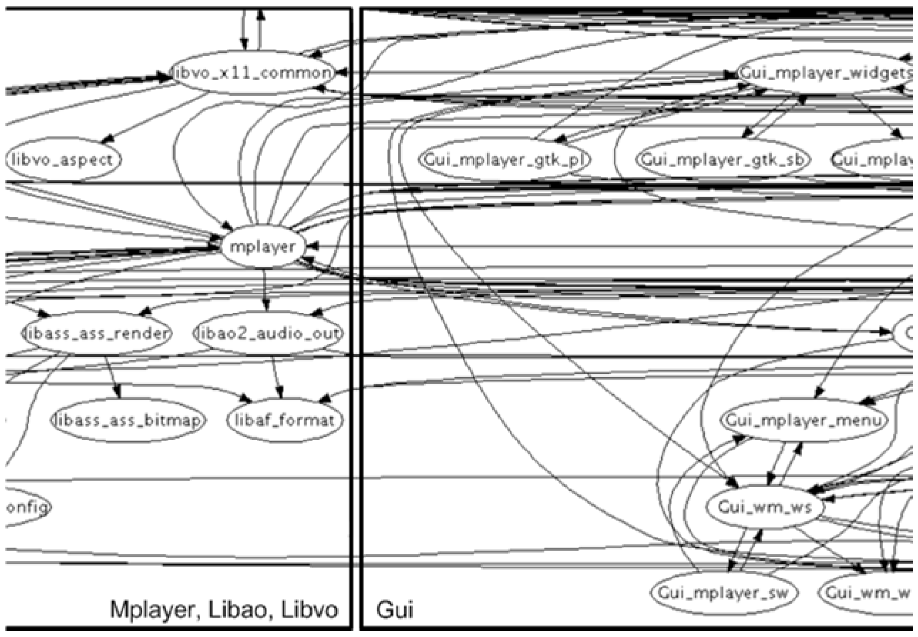


Figure 12.5 *A partial snapshot of the generated module dependency graph together with the boundaries of the Gui module with the modules Mplayer, Libao and Libvo.*

All the collected function call and data access profile data is stored in a database. For each decomposition alternative, the stored data is queried to find out dependencies that pass the boundaries of different RUs. For instance, if the *Gui* module is placed in a different RU than the modules *Mplayer*, *Libao* and *Libvo*, function calls that pass these RU boundaries will be queried as shown in Figure 12.5. The performance overhead is estimated by taking the percentage of such dependencies with respect to the total number of calls. In the case of data dependencies, the estimation is based on the total size of data that is shared among the RUs [Sözer, 2009]. For the example decomposition shown in Figure 12.3, for instance, the analysis tool has calculated the function dependency overhead and the shared data size as 5.4% and 5 KB, respectively.

12.4.2 Availability Analysis

To compare decomposition alternatives with respect to availability, our analysis tool automatically generates analytical models. These models are based on the CTMC (Continuous Time Markov Chains) formalism and they simulate the failure and recovery behavior of the system with the corresponding decompositions. The generated models are provided to a model checker, CADP [Garavel, 2007], which provides availability estimations in turn. To be able to generate the necessary analytical models, we need to specify, in addition to the decomposition, MTTF (Mean Time To Failure) and MTTR (Mean Time To Recover) values for each module of the system. In our case study, we have measured MTTR values based on the time it takes to restart and initialize each module. We have assigned various MTTF values to observe the impact of different scenarios (i.e. what-if analysis). Figure 12.6 shows, for example, a CTMC generated for global recovery. Hereby, all the modules are placed in a single RU.

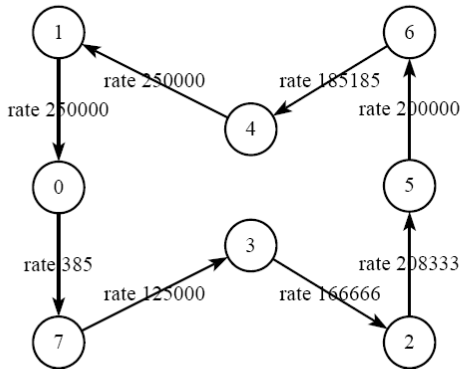


Figure 12.6 *A CTMC generated for global recovery, where all modules of MPlayer are placed in a single RU.*

The initial state of the system is represented by state 0, where the system is available. All the other states represent the recovery stages. Since there is only 1 RU that includes all the 7 modules and since these modules are recovered (initialized) sequentially, there exist 7 states corresponding to the recovery of the system. For example, state 7 represents the system state, where the RU is failed and none of its modules are recovered yet. There is a transition from state 0 to state 7, in which the label of the transition denotes the failure rate (calculated based on the specified MTTF values) of this RU. Upon its failure, all the 7 modules comprised by the RU must be recovered. That is why, there exist 7 transitions from state 7 back to state 0, where the labels of these transitions specify the recovery rates (calculated based on the specified MTTR values) of the modules. CADP model checker can process such a CMTC model and calculate the expected availability of the system in the long run (i.e. steady state). The availability for the example model shown in Figure 12.6 was calculated as 98.59% when all MTTF values for all the modules are specified as 1800 sec. The estimation was 83.60% after we assign the MTTF values 30 sec and 60 sec for *Gui* and *Libao* modules, respectively.

Together with the performance overhead estimations, availability analysis is used for systematic selection of a decomposition alternative. The designer can evaluate the results manually or the tool can run optimization algorithms for large number of decomposition alternatives.

12.5 Realization of local recovery with FLORA

After one of the design alternatives is selected, the software architecture should be structured accordingly. In addition, new supplementary architectural elements and relations should be implemented to enable local recovery. As a result, introducing local recovery to a system leads to additional development and maintenance effort. In this section, we introduce a framework called FLORA to reduce this effort. FLORA supports the decomposition and implementation of software architecture for local recovery. The framework has been implemented in the C language on Linux platform. FLORA assigns RUs to separate operating system (OS) processes, which do not directly communicate to each other. Figure 12.7 depicts the design of the MPlayer after local recovery is introduced using the framework.

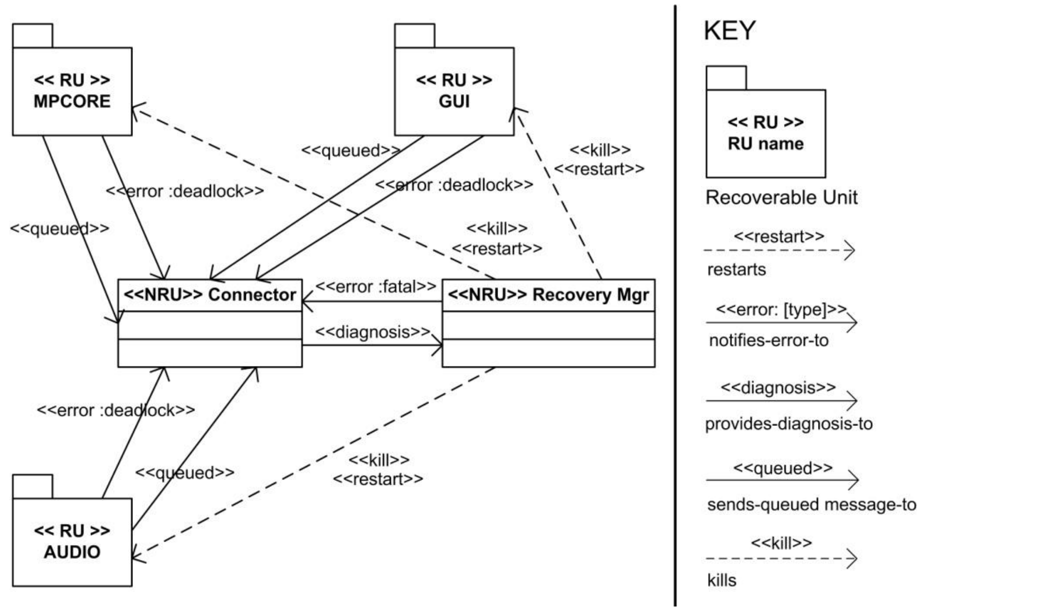


Figure 12.7 *Application of FLORA to MPlayer.*

In Figure 12.7, we can see the three RUs, RU MPCORE, RU GUI and RU AUDIO. In addition, the components *Connector* and *Recovery Manager* have been introduced by the framework. Each RU can detect deadlock errors[2]. Recovery Manager can detect fatal errors[3]. All error notifications are sent to Connector, which comprises the diagnosis facility. Diagnosis information is conveyed to Recovery Manager, which kills a set of RUs and/or restarts a dead RU. Messages that are sent from RUs to Connector are stored (i.e. queued) by RUs in case the destination RU is not available and they are forwarded when the RU becomes operational again.

To apply FLORA, each RU is wrapped using the RU wrapper template as shown in Figure 12.8. The wrapper includes the necessary set of utilities for isolating and controlling an RU (lines 1–3). A set of state variables can be declared to be checkpointed[4] (line 5). If needed, cleanup specific to the RU (e.g. allocated resources) can be specified (lines 8–10) as a preparation for recovery. Post-recovery initialization (lines 12–18) by default includes: i) maintaining the connection with the Connector and the Recovery Manager (line 13), ii) obtaining the checkpointed state variables (line 15) and iii) processing incoming messages from other RUs (line 17). Additional RU-specific initialization actions can also be specified here.

Each RU provides a set of interfaces, which are captured based on the specification in the wrapper (lines 20–24). Each interface defines a set of functions that are marshaled and transferred through Inter-Process Communication (IPC). On reception of these calls, the corresponding functions are called and then the results are returned (lines 26–29). In all the other RUs where this function is declared, function calls are redirected through IPC to the corresponding interface with C MACRO

[2] An RU wrapper detects if an expected response to a message is not received within a configured timeout period.

[3] The Recovery Manager is the parent process of all RUs and receives and handles a signal when a child process is dead.

[4] Checkpoint locations are specified by the designer based on the domain knowledge.

```
1:         #include "util.h"
2:         #include "recunit.h"
3:         #include "regui.h"
4:         ...
5:         #define STATE_VARS … &guiIntfStruct
6:         ...
7:
8:         void cleanUp(){
9:           /* no component specific cleanup */
10:        }
11:
12:        void __ruGui(struct recunit info) {
13:          INIT_RU(SOCK_PATH_RECMGR, SOCK_PATH_CONN)
14:          ...
15:          PRESERVE_STATE
16:          ...
17:          processMsgs();
18:        }
19:
20:        void catchInterfaces(){
21:          BEGIN
22:            CATCH(INTERFACE_GUI, apOnMsgRcvd_gui)
23:          END
24:        }
25:
26:        void onMsgRcvd_gui_guiInit(){
27:          guiInit();
28:          RETURN(INTERFACE_GUI, msg_gui_guiInit)
29:        }
30:        ...
```

Figure 12.8 *RU wrapper code for RU GUI.*

definitions[5]. In Figure 12.9, a code section is shown from one of the modules of RU MPCORE, where all calls to the function `guiInit` are redirected to the function `mpcore_gui_guiInit` (line 1), which activates the corresponding interface (`INTERFACE GUI`) instead of performing the function call (lines 4–6).

```
1:        #define guiInit() mpcore_gui_guiInit()
2:        ...
3:
4:        void mpcore_gui_guiInit() {
5:          CALL(INTERFACE_GUI, msg_gui_guiInit)
6:        }
7:        ...
```

Figure 12.9 *Function redirection through RU interfaces.*

FLORA comprises IPC utilities, message serialization/de-serialization primitives, error detection and diagnosis mechanisms, a RU wrapper template, one central Recovery Manager and one central Connector that communicate with one or more instances of RU.

[5] Function call redirection with C MACRO definitions has been previously applied in KOALA component model [Ommering, 2000]

12.6 Evaluation

To evaluate our approach, we have performed measurements from systems that are decomposed for local recovery. In this section, we briefly explain the implementation issues, how the measurements are performed and the results.

We have used FLORA to introduce local recovery to MPlayer for 3 different decomposition alternatives. i) Global recovery, where all the modules are placed in a single RU ii) Local recovery with two RUs, where the module *Gui* is isolated from the rest of the modules iii) Local recovery with three RUs, where the module *Gui*, *Libao* and the rest of the modules are isolated from each other as shown in Figure 12.3.

To be able to measure the availability achieved with these three implementations, we have modified each module so that they fail with a specified failure rate (assuming an exponential distribution with mean MTTF). After a module is initialized, it creates a thread that is periodically activated every second to inject errors. The Recovery Manager component of FLORA logs the initialization and failure times of RUs to a file during the execution of the system. For each of the implemented alternatives, we have let the system run for 5 hours. Then, we have processed the log files to calculate the times, when the core system module, MPlayer has been down. We have calculated the availability of the system based on the total time that the system has been running (5 hours). For both the analytical models and the error injection, we have assigned 30 sec. as the MTTF of the Gui module, 60 sec. for the MTTF of the Libao module and 1800 sec. as MTTF of the rest of the modules. MTTR values are assigned as measured before from the running system.

Decomposition Alternative	Measured Availability	Estimated Availability
all modules in 1 RU	83.27	83.60
Gui, the rest	92.31	93.25
Gui, Libao, the rest	97.75	98.70
each module in a separate TU	N/A	99.96

Table 12.1 *Comparison between the estimated and measured system availability.*

In Table 12.1, the first column lists the decomposition alternatives. The second column shows the availability measured from the running systems. The third column shows the availability estimated with the analytical models for the corresponding decomposition alternatives. Here, we see that the measured availability and the estimated availability are quite close to each other. In general, the measured availability is less than the estimated availability. This is due to the delays (e.g. error detection, process context switching, IPC) in the actual implementation, which are not reflected to the analytical models. Based on these results, we can conclude that our approach can be used for accurately analyzing, comparing and selecting decomposition alternatives for local recovery.

If all the function calls that pass the boundaries of RUs are defined, FLORA guarantees the correct execution and recovery of these RUs. However, the specification of the RU boundaries with the RU wrapper template requires an additional effort. The main effort is spent due to the definition of the RU wrappers. For the decomposition shown in Figure 12.3, we have measured this effort based on the LOC written for RU wrappers and the actual size of the corresponding RUs. Table 12.2 shows the LOC for each RU (LOC_{RU}), LOC of its wrapper ($LOC_{RU\ wrapper}$) and their ratio (($LOC_{RU\ wrapper}$/LOC_{RU})x100).

As we can see from Table 12.2, we had to write approximately 1K LOC to apply FLORA for the presented case study. The LOC written for wrappers is negligible compared with the corresponding system parts that are wrapped. The size of the wrapper becomes even less significant for bigger system parts. In fact, the wrapper size is independent of the size and internal complexity of the system part that is wrapped. This is because the wrapper captures only the interaction of an RU with the rest of the system.

	LOC_{RU}	**$LOC_{RU\ wrapper}$**	**Ratio**
RU MPCORE	214 K	463	0,22%
RU GUI	20 K	345	1,72%
RU AUDIO	8 K	209	2,61%
Total	242 K	1017	0,42%

Table 12.2 *LOC for the selected RUs (as shown in Figure 12.3), LOC for the corresponding wrappers and their ratio.*

12.7 Related Work

Candea et al. introduced the micro-reboot [Candea, 2004] approach, where local recovery is applied to increase the availability of Java-based Internet systems. Micro-reboot aims at recovering from errors by restarting a minimal subset of components of the system. Progressively larger subsets of components are restarted as long as the recovery is not successful. To employ micro-reboot, a system has to meet a set of architectural requirements (i.e. crash-only design [Candea, 2004]), where components are isolated from each other and their state information is kept in stable repositories. Unfortunately, designs of many existing systems do not have these properties. Such systems have to be decomposed to support isolation and until now the decisions on how to decompose an existing system have been based on qualitative analysis [Candea, 2004-b].

There are several modeling techniques to analyze and improve system availability ([Lai, 2002], [Majzik, 2002]). In general however, these models are specified manually and/or the methodology lacks a comprehensive tool-support, making these models less practical to use.

In [De Florido, 2008], a survey of approaches for application-level fault-tolerance is presented. According to the categorization of this survey, FLORA falls into the category of single-version software fault tolerance libraries (SV libraries). SV libraries are said to be limited in terms of separation of concerns, syntactical adequacy and adaptability [De Florido, 2008]. On the other hand, they provide a good ratio of cost over improvement of the dependability, where the designer can reuse existing, long-tested and sophisticated pieces of software [De Florido, 2008]. An example SV library is libft [Huang, 1995], which collects reusable software components for recovery. However, like other SV libraries [De Florido, 2008] it does not support local recovery.

12.8 Conclusions

In this paper, we have discussed our experiences in introducing local recovery to a system. Local recovery is an effective approach for increasing availability. However, it appears that the required decomposition for local recovery is usually not aligned with the decomposition based on functional concerns. We have proposed a systematic approach for analyzing an existing system to decompose its software architecture to introduce local recovery. The approach provides systematic guidelines to balance the decomposition alternatives with respect to availability and performance. We have presented the framework FLORA that provides reusable abstractions to support the realization of local recovery. We have illustrated FLORA to define three recoverable units (RUs) in the open-source media player called MPlayer. These three RUs were overlaid on the existing structure. The application of the framework, as such, provides a reusable and practical approach to introduce local recovery to software architectures.

Acknowledgements: We thank for the collaboration of Hichem Boudali and Mariëlle Stoelinga from the University of Twente for defining the availability models that are generated for analyzing decomposition alternatives.

12.9 References

[Candea, 2004] G. Candea. *Microreboot: A technique for cheap recovery*. OSDI, USENIX, pp. 31-44, 2004

[Candea, 2004-b] G. Candea. *Improving availability with recursive micro-reboots: A soft-state system case study*. Performance Evaluation, 56(1-4), pp. 213-248, 2004

[Clements, 2002] P. Clements et al. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002

[De Florido, 2008] V. De Florido and C. Blondia. *A survey of linguistic structures for application-level fault tolerance*. ACM Computing Surveys, 40(2), pp. 1-37, 2008

[Fenlason, 2000] J. Fenlason and R. Stallman. *GNU gprof: the GNU profiler*. Free Software Foundation, 2002. http://www.gnu.org/

[Garavel, 2007] H. Garavel et al. *CADP 2006: A toolbox for the construction and analysis of distributed processes*. LNCS 4590, pp. 158-163, 2007

[Huang, 1995] Y. Huang and C. Kintala. *Software fault tolerance in the application layer*. Software Fault Tolerance, M.R. Lyu (Ed.), chapter 10, pp. 231-248, 1995

[Lai, 2002] C.D. Lai. *A model for availability analysis of distributed software/hardware systems*. Information and Software Technology, 44(6), pp. 343-350, 2002

[Majzik, 2002] I. Majzik and G. Huszerl. *Towards dependability modeling of FT-CORBA architectures*. EDCC, pp. 121-139, 2002

[MPlayer, 2007] MPlayer: *Open source media player*. http://www.mplayerhq.hu/ , 2007

[Nethercote, 2007] N. Nethercote and J. Seward. *Valgrind: a framework for heavyweight dynamic binary instrumentation*. SIGPLAN, 42(6), pp. 89-100, 2007

[Ommering, 2000] R. van Ommering, F. van der Linden, J. Kramer, J. Magee. *The Koala component model for consumer electronics software*. IEEE Computer, 33(3), pp. 78-85, 2000

[Parnas, 1972] D. Parnas. *On the criteria to be used in decomposing systems into modules*. Communications of the ACM, 15(12), pp. 1053-1058, 1972

[Sözer, 2009] H. Sözer. *Architecting Fault-Tolerant Software Systems*. PhD thesis, University of Twente, 2009

Chapter 13

Industrial impact, transfer of results, lessons learned

Authors: Roland Mathijssen, Teun Hendriks, Piërre van de Laar

13.1 Introduction

The Trader project is an applied research project that is carried out in an industry-as-laboratory setting [Potts, 1993], with NXP semiconductors as the carrying industrial partner. This setting provides a realistic industrial environment where research ideas and theories can be validated and tested. The results of the various research lines have also resulted in many papers and articles, a number of PhD theses, presentations at scientific workshops and conferences, etc. An overview of the papers and articles is in Appendix A.

In this final chapter of the Trader book we look back at the project to summarize the results and note the lessons learned. Section 13.2 presents a short history of the Trader project. The next Section 13.3 describes the results of the project and the transfers. In Section 13.4 we describe the lessons learned in the Trader project. We specifically look at the concept of industry-as-laboratory and compare this to similar projects. Also the effects of having an overall vision in the project are discussed in this section.

13.2 A short history of the Trader project

13.2.1 First Bsik project

ESI wants to advance industrial innovation and academic excellence in embedded systems engineering (ESE) by creating and applying world-class ESE methods. The developed methodologies must support all aspects of the creation: specification, design, integration, test and validation. To accomplish this, ESI teams up with leading academic and industrial institutes, projects and programs. In long-term projects (often 4 years) ESI and their partners do industry-as-laboratory research to create and validate the ESE methods.

The way of working in the ESI projects is to start a project based on an actual industrial problem, related to one of the topics from the ESI Research Agenda[1]. Together with an industrial partner, called the CIP (Carrying Industrial Partner) and a selection of industrial and academic partners, specifically selected for the industrial problem, the project is started. The partners provide the PhD students and postgraduates to perform in-depth research in an industrial setting, while the CIP takes

[1] Performance, reliability, evolvability and system architecture; see also http://www.esi.nl/institute/research.html

responsibility for assigning a number of their experts on the problem domain. These industrial experts assist both in providing the industrial background knowledge and in facilitating the transfer of the results to the CIP. The project duration of four years acknowledges both the complexity of the research project and employment of PhDs in these projects.

During the project, academic researchers and their industrial colleagues meet regularly to ensure that results are aligned with the industrial question and fit in their industrial context. Also all researchers have regular overall meetings, to keep the various sub-projects lined up. Depending on the actual research topic, some researchers work (part of the time) at the CIP.

The Trader project is the fourth ESI project. It was the first project in a group of six projects that together comprise the ESI Bsik project[2]. The Trader project started in 2004 after an extensive exploratory and preparatory trajectory with Philips Semiconductors and potential academic and industrial partners and ended in 2009.

13.2.2 Project extension

The Trader project started in 2004. Originally the Trader project was projected to run for 4 years, enabling PhD students to do their work in the timeframe of the project. However recruiting PhDs took much longer than expected; the last PhD student started as late as 2006. The late start of a sufficiently resourced project meant that an extension with one year was needed, both to provide the PhD students with sufficient time to do their research in an industry-as-laboratory setting and to achieve the required and expected results.

13.2.3 Changes at the carrying industrial partner

In 2004 the Trader project started with Philips Semiconductors as carrying industrial partner. Being part of Philips Royal Electronics, the focus was mainly on the end-product, i.e. a television system. This TV was produced by Philips Consumer Electronics while Philips Semiconductors provided the chips and most software layers.

In 2006 Philips Semiconductors became an independent company: NXP Semiconductors[3]. The intimate connection between Philips CE and Philips Semiconductors was changed. NXP turned into a silicon and system provider where the link to the actual television customer became more indirect. The main stakeholders were now the OEM companies which produce the TV sets. At the same time, Philips CE's involvement in the Trader project diminished further to solely an information provider and an indirect contact.

Also, Philips Research split its activities; part of Philips Research was incorporated into NXP and became NXP Research. Concurrent with the change from Philips Semiconductors to NXP, the Philips Research contacts with Trader became NXP Research. Shortly after NXP became an independent company, Philips TASS was sold to TSS in April 2007 and became an independent company[4]. Summarizing, in less than one year, the carrying industrial partner, Philips, transformed to NXP Semiconductors, NXP Research and TASS, while Philips CE withdrew as active partner in the project.

[2] http://www.senternovem.nl/BSIK/

[3] http://www.nxp.com/news/content/file_1254.html

[4] http://www.bits-chips.nl/nieuws/tools-toys/bekijk/artikel/tass-in-handen-tss.html

Finally in 2008, at the moment of project extension, mainly due to the economic situation, NXP Home stopped being an active partner in the Trader project. NXP Automotive picked up the role of NXP Home as CIP. This did also imply that the domain shifted from TV to Car infotainment. Fortunately most research lines were sufficiently generic to make this step to another domain possible; however it did cost extra effort.

13.3 Results and transfer

The various research projects all had results which were transferred in some form or other to the CIP. Most chapters have already given a brief description of this transfer. Hence, this chapter only discusses some highlights and lessons learned. To transfer results, ESI uses a model where various phases of research, academic evidence, industrial evidence and operational use follow up one another, see e.g. [Ideals, 2007] page 144 and Figure 13.1. This model often starts with research in an academic setting (phases 0 and 1a). The reason for starting the academic research may well be an industrial problem, but it can also be sheer academic curiosity. In phase 1b an actual industrial problem is taken and the results from 1a are tried on the industrial case. This still happens in a setting where there is minimal interference with everyday industrial business. This is the stage where an industrial proof of concept and demonstrators are developed. Once this phase gives sufficient positive results and confidence, the step is taken to go to an industrial setting in everyday business (phase 1b). The results are applied in the critical path and the researchers gain feedback on applicability in the hectic industrial environment. If this also gives positive results, the next step is to actually transfer the method or tool to industry; the steps to phase 2 and 3. In these phases the researchers gradually step out of the project and the industry becomes the owner. The next sections, including Section 13.4, discuss that making steps towards industrial evidence in a critical setting runs into various challenges.

The Trader project also introduced the concept of Demo-Day. These demo days are further discussed in Section 13.4.3.

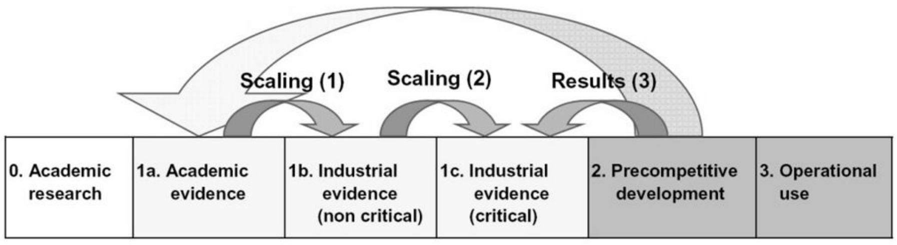

Scaling (1)
Scaling (2)
Results (3)
0. Academic research
1a. Academic evidence
1b. Industrial evidence (non critical)
1c. Industrial evidence (critical)
2. Precompetitive development
3. Operational use

Figure 13.1*: The transfer process (from [Tangram, 2007] and [Ideals, 2007])*

13.3.1 Efficient deployment of coding standards

Applying static software analysis is mandatory in many software development departments and industries. In some cases it is a management decision, while in other cases the OEM customer demands it, trying to ensure higher product quality. The basic assumption is that static code analysis will reduce the error level in produced software. However, solving every warning is a time consuming process, especially when this does not necessarily decrease the number of possible software errors. Also there is no solid proof that every warning relates directly to a potential error.

Modifying a piece of correct code, merely to comply with the rules of a tool, takes time and may even introduce new errors. On the other hand, knowing which rules are most effective to increase the correctness of a program is both valuable for industry and will improve static code analysis in industry.

The results from this research (Chapter 4), indicate relations between certain coding rules and amount of Problem Reports that need to be solved. It actually points to the top-ten rules that help improve software quality. The actual method to find the top-ten of rules is not transferred to NXP (see also Section 13.4.6), however the developer of the tool (QA-c) is interested in the results. Parts of the method may be applied at the original tool vendor. When the actual tool developer and vendor take up this research result, these results are even anchored in a better and broader way.

13.3.2 Stress testing and real-time monitoring

The CPU stressor or cycle eater (Chapter 5) is a result from Trader research that is actually available in the current software tree at the CIP. At integration time, every developer or integrator has the ability to stress the CPU on available clock cycles and memory bandwidth. The resource usage of CPU cycles and bandwidth can also be visualized in real-time. Structurally integration of the use of stress testing in NXP’s test and integration process proved to be more difficult. Even though the method and tool is available to the designers, and the visualization is even used by developers, always applying it and adding it to the standard integration process needs a change in work-flow. The integration process is a somewhat conservative process; as many industrial processes are in order to minimize risks. Errors are sought for, and detected during predefined tests. However a new method to stress the system to make sure that it is sufficiently robust takes time to be accepted as everyday practice.

Further, the concept of stress testing is incorporated in the test and integration course at ESI. Students of this course are provided both with the method to stress test SW and enhanced code stubs for them to easily apply this concept in their company’s setting.

13.3.3 Behavior modeling and model based error detection

The research line on behavior modeling (Chapter 7) was started to enable runtime model based error detection (Chapter 8). To compare actual behavior with expected or modeled behavior in real-time, a light-weight high-level model is needed. During this research the modeling in itself already proved very valuable, both in exploring the system design, and in validating requirements. The model turned written requirements into an executable model that could be exercised, quickly showing contradicting or incomplete requirements. The adoption of such system modeling has high value already during the requirements phase of a project, both for designers to explore the system, and for customers or marketers to discuss the proposed requirements. Next to this, a modeled system has further value during test and validation. Certain tests can be automated; specified system behavior can be compared with actual behavior and automatic test reports can be generated.

Although the value of modeling was seen by NXP Home, and some explorative tests were done, the actual use of modeling was not picked up. NXP Automotive also showed interest, partly because elements of their system were already modeled. Here one sees that the step to apply new methods is smaller once the concepts are already known to an organization. Applying totally new and unknown concepts in an organization proves to be much more difficult.

13.3.4 Spectrum-based fault localization in practice

The concept of Spectrum Based Fault Localization (SFL, see Chapter 10) attracted immediate attention at one of the demo-days (see Section 13.4.3). Many NXP developers showed great interest in this tool. Debugging can be a lot faster with a tool that assists in localizing the most likely places where the root-cause of a SW problem lies. However before NXP wanted to fully implement SFL to support fault localization, more evidence from actual Problem Reports (PRs) was required.

Here this research item ran into a number of challenges typical for an industry such as NXP. The first challenge was that SFL could not immediately be applied on a running software project (phase 1c of Figure 13.1), as risks were considered too high. Solving difficult PRs –where SFL should have most impact– often happens in a hectic environment. It is seen as extra risk to apply new and unknown methods, since it requires resources that are already highly needed to solve the PRs.

Since the technique could not be applied on actual PRs, an archived software version had to be used (phase 1b of Figure 13.1). Restoring the proper software version that contained the errors and reproducing the documented error proved time-consuming and difficult as developers originally assigned to the PR were not available anymore to help out. This also proved to be a lesson for the academic researchers: information that one expects to be present is not always present in real industrial life. In the end, only a handful of PRs could be reproduced for use in SFL evaluation. Some of these PRs gave extra feedback to the researchers on the applicability of SFL on various types of problems. However due to the small number of PRs available to test SFL on the TV software, insufficient evidence was gained to convince NXP that SFL would save time solving PRs. At that moment the choice was made not yet to implement SFL in the SW development process.

When a new SW project was started for the next TV family, a new attempt was made to transfer and use SFL. It again proved difficult to actually get resources to try out SFL in parallel to the project (due to time constraints). In the end, one person could try out SFL on a handful of PRs in the running project. First, to get some experience, SFL was used to analyze a few PRs that were recently solved. Next an open PR was analyzed with SFL. Especially the open PR seemed to give a very promising result. The PR was already open and under investigation for more than 2 weeks. SFL pointed within the hour to the same location as where the research without SFL was at that time. This location proved to be the point where a faulty valuable was written to a register.

A next important step needed to transfer the SFL method was to have the full code instrumented and to have the SFL tools in the tool-chain for applying the SFL method at integration time. A new group of people at another location needed to be convinced of the value of SFL. Due to a near unavailability of NXP resources to support this process at that time, the transfer to the CIP at the time of writing was not yet successful (see also Section 13.4.2).

13.3.5 Open source SFL

As SFL shows promising results, both TU Delft and ESI decided to develop the SFL result further into an open source version that can be applied in virtually any domain. To create this open source version a programmer reworked the TV specific version into a generic version. Also a user manual to apply SFL was written, to facilitate applying the method in other domains. The result can be found on http://fdir.org/zoltar/

This work is somewhat analogous to the work done in industry by engineering, where the output of R&D first goes through the process of making it truly (re-)producible. This method to follow up the work of a research line to make it easier to transfer to other areas is definitely a lesson learned and worthwhile to also apply in the other projects.

13.4 Lessons learned

As already mentioned in the previous sections, the Trader project not only resulted in transfer of research results, but also in a number of other lessons learned. The Trader project was already the fourth large-scale ESI project. Nevertheless, the fact that both a different research challenge was targeted, and that the project was carried out with a different type of industry, caused the project to face a number of new challenges. The next sections will elaborate on these challenges.

13.4.1 Time scale

In the Trader project again one could observe that industrial and academic people live by a different time scale. Similar effects were seen in previous projects where ESI worked together with low- to mid-volume, high-tech industries targeting their products mainly on an industrial market (see the Boderc, Ideals and Tangram projects [Boderc, 2007], [Ideals, 2007], [Tangram, 2007]). However, working together with a high-volume consumer-market industry showed this effect of different time scales even more explicitly. In this hectic consumer market new products are expected roughly twice a year. In relation to a project with academic partners this means that approximately two complete product generations would be developed while the project was ‘just’ investigating the real problem. Between the beginning of the project and the first applicable results, 5 to 6 new product ranges are put into the market. And once research results become available, in the time the project needs to make a sufficient step in research, the industry makes again at least one full product step. This causes the focus of the industrial experts often to differ a lot from the focus of the academic people.

This different pace is also reflected in the average availability to the project of experts from the industry. After one or two years typically, many designers and managers move on to a different project or part of the company. Their replacements bring new contacts to the project. In previous ESI projects, it worked out well to have a small group of highly involved industrial experts. However in an even faster moving industry as the consumer market, keeping this involvement on a proper level is more difficult and will need more attention. In similar projects in the future, even more care must be taken both to select people who are willing to stay with a long-running project and to find means to get continuous management support, even with changes in management (as management also typically moves further after a few years and in this case sometimes even less than a year).

13.4.2 Geographical challenges

Nowadays large companies have their development typically spread out over various locations world-wide. To create one product or component often two or more teams on different geographical locations work together. In the Trader project, most parts of the development related to the Trader project were located in Eindhoven. Some others partners were close by, e.g. in Bruges, a two-hour drive from Eindhoven. However, the effect of a full multi-site project became evident as input and support was needed from other teams at the CIP, such as the software development group in Bangalore, India. That team also needed to implement the SFL tools and to instrument the full SW for our Eindhoven based research team to quickly locate faults during integration time and solve the associated PRs.

Having productive contacts between academic researchers and industry is not self-evident and needs attention and effort. Also other ESI projects showed that it is important for the researchers to work together (on the location of the CIP) for a significant part of their time [Ideals, 2007]. When contacts are suddenly no longer within easy travel distances, when people work in different time zones, and when they have different organizations and working processes, then building up contacts and working closely together to quickly achieve results, suddenly becomes much more laborious.

Working with people at large geographical distances was new to ESI projects. The Trader project approached this geographical challenge by having the local CIP people interfacing with their colleagues abroad. At times, this proved to be a challenge, especially as it requires the creation of shared priorities among different organizations, project priorities and cultures.

13.4.3 Demo-Day

The Trader project was the first ESI project where the concept of a so-called Demo-Day was applied. In previous projects individual results were shown to a small, often selected audience or were presented in the rather static form of a symposium. In the Demo-Day concept, most research lines make a live demonstrator to show this to a large audience at the location of the CIP, inviting a broad audience from management to developers.

Figure 13.2 *Trader Demo-Day at NXP Home (Eindhoven) (Photo: A. Westveer)*

Organizing of such a Demo-Day in house at the CIP has a number of positive effects. First of all, an in house Demo-Day offered the project large exposure to a much broader audience in the company than presentation sessions can reach. Having researchers showing their results with live demonstrations stimulates interaction and lively discussions between potential users and the researchers involved. In previous projects, ESI and the researchers had to search for potential users. Now the potential users simply came to visit the Demo-Day and contacts were made in a natural way. In addition, the Demo-Day proved to be an ideal incentive for the researchers to make a working demonstrator. Without a demonstrator, the work often remained very long in the research phase, and consisted mainly of ideas, papers, and often merely local results for own research. For the industrial contacts, the demonstrators made the research results very tangible, which resulted in both enthusiastic reactions (“I want it now!”) and in-depth discussions, sometimes giving more direction to the researchers or giving opportunities to enhance the industry-as-laboratory method of working.

A Trader Demo-Day was held twice. The first one showed 10 running demonstrators on March 11th, 2008 at NXP Home in Eindhoven (the actual CIP), where an estimated 100 people visited the demos. The second one, a small scale version, showed 5 demos on June 20th, 2008 at NXP CAR in Nijmegen, where the target audience, R&D managers, came to visit the demos. The first Demo-Day in Eindhoven resulted in 10 actual follow-up actions.

Thanks to the success of this Demo-Day, the other ESI projects followed this example. Creating working demonstrators and showing them at the CIP and discussing the results are now one of the standard methods in other ESI projects to transfer results.

13.4.4 Bringing a shared project vision

When the project started, the first months were spent on creating a shared vision; first on how to define the basic problem and next on how to tackle the problem thus defined. For most researchers, though experts in their own research area, the domain of televisions was new. Also the problem description and areas to investigate further initially were still vague. During a number of workshops it became clear that focusing mainly on improving the design phase cannot prevent every possible fault of a complex product. Due to the ever increasing complexity, systems have to be able to deal with errors that will stay undetected during design, integration and test. Such errors can be expected to show up in the field, once the product is sold. Countering such errors implies the need for a system with a form of run-time awareness, which can detect and resolve unexpected system behavior (see Chapter 1). Defining this framework in an early phase of the project helped to give direction to the various sub-projects, most of which are described in this Trader book. Where in some other projects we have seen the risk of the various research topics becoming individual projects, the run-time awareness framework coupled both the research topics and the researchers together into truly one project.

Having an overall vision was of course not sufficient. During the workshops and also afterwards the risk was always present that some academic researchers preferred to work on their favorite research topic only, neglecting the overall goal in which their research was only one piece of the total puzzle. Also having a favorite topic increases the risk that researchers tend to specify the problem in terms of the envisioned solution, thus not seeing the actual problems. An open attitude, created by looking for the overall vision, is needed to see the actual real-world problem that may be different from the expected problem: “Real-world problems are seldom where you expect them to be” [Potts, 1993].

Regular meetings were organized with all the researchers present. In these meetings, researchers were asked to make explicit what input they need and where the output will be used. This helped to keep focus on the overall vision. Also the industrial setting and the creation of demonstrators helped focusing: researchers where often challenged to actually implement their results at NXP. Of course focusing the researchers on this overall vision and industrial context seemed not always in line with their quest for academic results, such as scientific papers. Yet, experimentally validated results are needed for a healthy science. Otherwise, system engineering research will end up in a research crisis like software [Glass, 1994] with a too low ratio of validated results [Tichy, 1995]. However, the next section shows that these two pulling forces can be combined to pull in the same direction.

13.4.5 Industrial versus scientific success

In the field of Embedded Systems Engineering tensions can occur between industrial research and development and academic research. What is considered an industrial success is not always valued scientifically, whereas scientific results are not always easily applicable in industry. The Trader project showed that it is perfectly possible to combine industrial success with scientific success. The research on spectrum-based fault localization is a perfect example that showed that applying, verifying, and testing scientific research in an industrial context can lead to a sound industrial basis and well regarded scientific publications. The SFL research is with sixteen accepted publications one of the highest ranking research lines in Trader.

13.4.6 Tool transfer tensions

Next to the above mentioned challenges, one also sees that many results are in the form of tools that help the development or the architectural process. On the one hand most companies are reluctant to introduce tools that cannot be obtained from well known tool vendors. Often the company does not have resources to support and maintain the tool, so they expect it from a tool vendor. The tools made by the researchers are often specialties and not supported by tool vendors. This makes introduction to the CIP difficult or impossible. One route to follow can be to find a tool vendor that is interested in the new tool. This however proves to be a hard route that often only works when several interested companies urge a tool vendor to pick up the tool.

Further a company is often reluctant to adopt new tools and methods in their current development. Introducing new technology in a running project is seen as an extra risk. A method to transfer new technologies can be to introduce the new technology in a parallel path, next to the running development. Once the new technology has proven itself and the risk of introduction has proven to be low, the company's reluctance can diminish.

13.5 Conclusions

The Trader project has a number of valuable and interesting results. Some of the results were taken up by the CIP. We hoped that more results would have been picked up, however the large changes at the CIP, also induced by the economic situation, made transfer even more of a challenge than envisioned and experienced from previous ESI projects. Nevertheless, both the industrial and academic partners are happy with the results from the Trader project. We hope that this book, bringing all the results together, can also help in creating awareness in industry and academia about product reliability and possible methods to improve it. Especially important is the awareness that there can be a big difference in viewpoint on reliability between users / customers and the developers (see Chapter 2). Finally a number of results from the Trader project are currently being incorporated into a new ESI course on reliability.

Acknowledgements: We gratefully acknowledge the input of many people from NXP over the 5 years of the Trader project.

13.6 References

[Boderc, 2007] M. Heemels, G. Muller (Eds.). *Boderc: Model-based design of high-tech systems; A collaborative research project for multi-disciplinary design analysis of high-tech systems*. Embedded Systems Institute. ISBN 978-90-78679-01-1

[Glass, 1994] R.L. Glass. *The software-research crisis*. IEEE Software. Volume 11, Issue 6, Nov. 1994 pp. 42-47

[Ideals, 2007] R. van Engelen, J. Voeten (Eds.). *Ideals: evolvability of software-intensive high-tech systems; A collaborative research project on maintaining complex embedded systems*. Embedded Systems Institute. ISBN 978-90-78679-03-5

[Potts, 1993] C. Potts. *Software-engineering research revisited*. IEEE Software. Volume 10, Issue 5, Sept. 1993 pp. 19-28

[Tangram, 2007] J. Tretmans (Ed.). *Tangram: Model-based integration and testing of complex high-tech systems; A collaborative research project on multidisciplinary integration and testing of embedded systems*. Embedded Systems Institute. ISBN 978-90-78679-02-8

[Tichy, 1995] W.F. Tichy, P. Lukowicz, L. Prechelt and E.A. Heinz. *Experimental evaluation in computer science: A quantitative study*. Journal of Systems and Software. Volume 28, Issue 1, January 1995, pp. 9-18

Appendix A

Trader Publications

This appendix lists the Trader publications as of July 1st, 2009. An up-to-date list with links to most publications in PDF format can be found on http://www.esi.nl/trader/ (section Publications)

PhD Theses

Hasan Sözer. *Architecting Fault-Tolerant Software Systems*. Doctoral dissertation, University of Twente, the Netherlands, 2009

Ilse de Visser. *Analyzing User Perceived Failure Severity in Consumer Electronics Products – Incorporating the User Perspective into the Development Process*. Doctoral dissertation, Eindhoven University of Technology, the Netherlands, 2008

Articles, book chapters, reports, posters

2009

R. Abreu, P. Zoeteweij, R. Golsteijn, A.J.C. van Gemund. *A practical evaluation of spectrum-based fault localization*. Journal of Systems and Software, 2009, doi:10.1016/j.jss.2009.06.035

R. Abreu, A.J.C. van Gemund. *A Low-Cost Approximate Minimal Hitting Set Algorithm and its Application to Model-Based Diagnosis*. Proceedings of the 8th Symposium on Abstraction, Reformulation and Approximation (SARA’09), Lake Arrowhead, CA, USA, July 2009. AAAI Press

R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. *A Bayesian Approach to Diagnose Multiple Intermittent Faults*. Proceedings of the 20th International Workshop on Principles of Diagnosis (DX’09), pp. 27-33, Stockholm, Sweden, June 2009

R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. *A Model-based Software Reasoning Approach to Software Debugging*. Proceedings of the 22nd International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA-AIE’09), Tainan, Taiwan, June 2009. Studies in Computational Intelligence, vol. 214, pp. 233-239, Springer-Verlag

R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. *A New Bayesian Approach to Multiple Intermittent Fault Diagnosis*. Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI’09), Pasadena, CA, USA, July 2009. AAAI Press

R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. *Localizing Software Faults Simultaneously*. Proceedings of the 9th International Conference on Quality of Software (QSIC’09), Jeju, South Korea, August 2009

R. Abreu, W. Mayer, M. Stumptner, and A.J.C. van Gemund. *Refining Spectrum-based Fault Localization Rankings*. Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC’09) - Software Engineering Track, pp. 409-414, Honolulu, Hawai’i, USA, March 2009

R. Abreu, A.J.C. van Gemund. *Statistics-directed Minimal Hitting Set Algorithm*. Proceedings of the 20th International Workshop on Principles of Diagnosis (DX’09), pp. 51-58, Stockholm, Sweden, June 2009

C. Boogerd, L. Moonen. *A Model-based Software Reasoning Approach to Software Debugging*. Proceedings of the 22nd International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA-AIE’09), Tainan, Taiwan, June 2009. Studies in Computational Intelligence, vol. 214, pp. 233-239, Springer-Verlag

H. Boudali, H. Sözer, M. Stoelinga. *Architectural Availability Analysis of Software Decomposition for Local Recovery*. Proceedings of the 3rd IEEE International Conference on Secure Software Integration and Reliability Improvement, Shanghai, China, July 2009.

T. Janssen, R. Abreu, and A.J.C. van Gemund. *Zoltar: A Spectrum-based Fault Localization Tool*. Proceedings of the 1st International Workshop on Software Integration and Evolution @ Runtime (SINTER’09), Amsterdam, the Netherlands, August 2009. ACM Press.

J. Keijzers, L. Scholten, Y. Lu, E. den Ouden. *Scenario-Based Evaluation of Perception of Picture Quality Failures in LCD Televisions*. CIRP design conference, 30 - 31 march 2009, Cranfield, UK pp. 497-503.

J. Keijzers, E. den Ouden, Y. Lu. *Understanding Consumer Perception of Technological Product Failures: An Attributional Approach*. 27th Conference on Human Factors in Computing Systems ; Poster presentation

J. Keijzers, E. den Ouden, Y. Lu. *Understanding Consumer Perception of Technological Product Failures: An Attributional Approach*. Proceedings of the 27th Conference Extended Abstracts on Human Factors in Computing Systems, pp. 4057-4062, ACM Press (New York), 2009

H. Sözer, B. Tekinerdoğan, M. Akşit. *FLORA: A framework for decomposing software architecture to introduce local recovery*. Software Practice and Experience, Wiley, 39(10). pp. 869-889

2008

R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. *A Dynamic Modeling Approach to Software Multiple-Fault Localization*. Proceedings of the 19th International Workshop on Principles of Diagnosis (DX'08), pp. 7-14, Blue Mountains, NSW, Australia, September 2008

R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. *An Observation-based Model for Fault Localization*. Proceedings of the 6th Workshop on Dynamic Analysis (WODA'08), collocated with the International Symposium on Software Testing and Analysis (ISSTA'08), pp. 64-70

R. Abreu, A. González, P. Zoeteweij, and A.J.C. van Gemund. *Automatic Software Fault Localization using Generic Program Invariants*. Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC'08) - Software Engineering Track, pp. 712-717

R. Abreu, A. González, P. Zoeteweij, and A.J.C. van Gemund. *On the Performance of Fault Screeners in Software Development and Deployment*. Proceedings of the 3rd International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'08), pp. 123-130

M. Agarwal, A.K. Nieuwland. *Real Time Bandwidth Monitoring: IP2032 - A Case Study*. Technical Note NXP-R-TN 2008/00086

E. Brinksma, J. Hooman. *Dependability for High-Tech Systems: an Industry-as-Laboratory Approach*. DATE, pp.1226-1231, 2008 Design, Automation and Test in Europe, 2008. Also appeared as ESI Report Nr. 2008-1

C. Boogerd, L. Moonen. *Assessing the Value of Coding Standards: An Empirical Study*. Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM) pp. 277-286, 2008

C. Boogerd, L. Moonen. *Assessing the Value of Coding Standards: An Empirical Study*. Technical Report TUD-SERG-2008-017, Delft University of Technology, 2008

C. Boogerd, L. Moonen. *On the Use of Data Flow Analysis in Static Profiling*. Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM) 2008

J. Hooman, T. Hendriks. *Model-Based Run-Time Error Detection*. Lecture Notes in Computer Science; Models in Software Engineering; Vol. 5002 pp. 225-236, 2008

J. Keijzers, E. den Ouden, Y. Lu. *The 'Double-Edged Sword' of High-Feature Products: An Explorative Study of the Business Impact*. Proceedings of the 32nd Annual Product Development and Management Association (PDMA) International Research Conference, Orlando, pp. 13-17. USA

J. Keijzers, E. den Ouden, Y. Lu. *Usability benchmark study of commercially available smart phones: cell phone type platform, PDA type platform and PC type platform*. Proceedings of the 10th international conference on Human computer interaction with mobile devices and services; pp. 265-272 ; 2008

W. Mayer, R. Abreu, M. Stumptner, and A.J.C. van Gemund. *Prioritizing Model-Based Debugging Diagnostic Reports*. Proceedings of the 19th International Workshop on Principles of Diagnosis (DX'08), pp. 127-134, Blue Mountains, NSW, Australia, September 2008

H. Sözer, B. Tekinerdoğan. *Introducing Recovery Style for Modeling and Analyzing System Recovery*. 7th Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)

L. Steffens, M. Agarwal, P. van der Wolf. *Real-Time Analysis for Memory Access in Media Processing SoCs - A Practical Approach*. Proceedings of the 2008 Euromicro Conference on Real-Time Systems. pp. 255-265; 2008

B. Tekinerdoğan, H. Sözer, M. Akşit. *Software Architecture Reliability Analysis using Failure Scenarios*. Journal of systems and software, 81 (4). pp. 558-575

P. Zoeteweij, J. Pietersma, R. Abreu, A. Feldman, and A.J.C. van Gemund. *Automated Fault Diagnosis in Embedded Systems*. Proceedings of the 2nd IEEE International Conference on Secure Systems and Reliability Improvement (SSIRI'08)

2007

R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. *On the Accuracy of Spectrum-based Fault Localization*. Proceedings of the Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART'07), pp. 89-98

C. Boogerd. Supporting *Reliable Software Evolution through Program Analysis*. Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR)

J. Hooman, T. Hendriks. *Model-Based Run-Time Error Detection*. Models in Software Engineering : Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers; pp. 225-236

P.J.L.J. van de Laar, R. Golsteijn. *User-Controlled Reflection on Join Points*. Journal of Software, Vol. 2, No. 3, pp. 1-8, September 2007

Z. Ma, D. Scarpazza, F. Catthoor. *Run-time Task Overlapping on Multiprocessor Platforms*. IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia, 2007 (ESTIMedia 2007), pp. 47-52, October 4-5, 2007

Z. Ma, P. Marchal, D.P. Scarpazza, P. Yang, C. Wong, J.I. Gómez, S. Himpe, C. Ykman-Couvreur, F. Catthoor. Systematic *methodology for real-time cost-effective mapping of dynamic concurrent task-based systems on heterogeneous platforms*. Springer, 2007, ISBN 978-1-4020-6328-2

J. Pietersma, R. Abreu, A. Feldman, P. Zoeteweij, A. J.C. van Gemund. *Automated Fault Diagnosis*. Poster session at Nederlands Institute for Research ICT kick-off Event, Utrecht; 2007

H. Sözer, C. Hofmann, B. Tekinerdoğan, M. Akşit. *Detecting Mode Inconsistencies in Component-Based Embedded Software*. DSN Workshop on Architecting Dependable Systems, 27 June 2007, Edinburgh, United Kingdom. pp. 154-160. IEEE Computer Society

H. Sözer, B. Tekinerdoğan, M. Akşit. *Extending Failure Modes and Effects Analysis Approach for Reliability Analysis at the Software Architecture Design Level*. Architecting Dependable Systems IV, Ed. by Rogerio de Lemos et.al., 2007

P. Zoeteweij, J. Pietersma, R. Abreu, A. Feldman, and A.J.C. van Gemund. *Automated Fault Diagnosis in Embedded Software*. Proceedings of the ESI / Bits & Chips Embedded Systems Conference, Eindhoven; 2007

P. Zoeteweij, R. Abreu, R. Golsteijn, A.J.C. van Gemund. *Diagnosis of Embedded Software using Program Spectra*. Proceedings of the 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'07)

P. Zoeteweij, R. Abreu, R. Golsteijn, A.J.C. van Gemund. *Fault Diagnosis of Embedded Software using Program Spectra*. Proceedings of the 3rd European Symposium on Verification and Validation of Software Systems (VVSS 2007), LaQuSo

P. Zoeteweij, J. Pietersma, R. Abreu, A. Feldman, A.J.C. van Gemund. *Software Fault Diagnosis*. Tutorial TESTCOM / FATES / FORTE 2007

2006

R. Abreu, P. Zoeteweij, A.J.C. van Gemund. *An Evaluation of Similarity Coefficients for Software Fault Localization*. Proceedings of the 12th International Symposium on Pacific Rim Dependable Computing (PRDC'06)

R. Abreu, P. Zoeteweij, R. Golsteijn, A.J.C. van Gemund. *Automatic Fault Diagnosis in Embedded Software.* Proceedings of 10th Philips Software Conference (PSC'06), Veldhoven

R. Abreu, P. Zoeteweij, A.J.C. van Gemund. *Program Spectra Analysis in Embedded Software: A Case Study*. Proceedings of the 12th Annual Conference of the Advanced School for Computing and Imaging (ASCI'06), pp. 263-269, Lommel, Belgium, June 2006

C. Boogerd, L. Moonen. *Prioritizing Software Inspection Results using Static Profiling*. Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)

C. Boogerd, L. Moonen. *Ranking Software Inspection Results using Execution Likelihood*. Proceedings of the Philips Software Conference (PSC)

C. Boogerd, L. Moonen. *Ranking Software Inspection Results using Execution Likelihood*. Poster Session at Scientific ICT Research Event Netherlands

J.P.T. Dobbelsteen, R.H.M. Golsteijn, P.J.L.J. van de Laar. *An Infrastructure for Traceability to Increase Insight in Complex Embedded Systems*. Technical Note PR-TN 2006/00506

J. Keijzers, E. den Ouden, A.C. Brombacher. *Evaluating test methods in dealing with customer perceived failures in highly innovative product development*. Proceedings of the IEEE International Conference on Management of Innovation and Technology. (Vol. 2, pp. 576-580)

P.J.L.J. van de Laar. *Combining component-based and aspect-oriented software development in a resource constrained environment*. Technical Note PR-TN 2006/00648

K.S.W. van Langen, M. Opdam, P.J.L.J. van de Laar, and R.H.M. Golsteijn. *Research and design for user-perceived reliability enhancement of teletext and future data carousels*. Philips Research Report PR-TN 2006/00395, 2006

I.M. de Visser, J.A. van den Bogaard. *The risks of applying qualitative reliability prediction methods: a case study*. Proceedings of the Annual Reliability and Maintainability Symposium 2006 (RAMS '06)

I.M. de Visser, Y. Lu, N. Ganesh. *Understanding Failure Severity in New Product Development Processes of Consumer Electronics Products*. Proceedings of the 2006 IEEE International Conference on Management of Innovation and Technology (ICMIT 2006)

P. Zoeteweij, R. Abreu, R. Golsteijn, A.J.C. van Gemund. *Fault Diagnosis of Embedded Software using Program Spectra*. Proceedings of the 12th Nederlandse Testdag, ASML, Veldhoven

2005

R.T.C. Deckers, P.L. Janson; F.H.G. Ogg; P. J. L. J. van de Laar. *Introduction to Software Fault Tolerance ; Concepts and Design Patterns*. Technical Note PR-TN 2005/00451

B. Tekinerdoğan, H. Sözer, M. Akşit. *Software Architecture Reliability Analysis using Failure Scenarios*. WICSA, pp. 203-204, Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA'05), 2005

Appendix B

Glossary and list of abbreviations

A

ADOC	Analog (TV) Digital One Chip; a digital TV processor
ANSI C	The standard published by the American National Standards Institute (ANSI) for the C programming language
AOP	Aspect-oriented programming, a programming paradigm that increases modularity by allowing the separation of cross-cutting concerns, forming a basis for Aspect-oriented software development

B

b or bit	Bit, a binary digit, taking a value of either 0 or 1
B or Byte	Byte, a basic unit of measurement of information storage in computer science, usually consisting of 8 bits
Bsik	Besluit Subsidies Investeringen Kennis-infrastructuur, a scheme from the Dutch government of the proceeds of its natural gas reserves, to subsidize investments in knowledge infrastructure aims, to bring together players from public research and industry and support their joint research efforts with funding of up to 50 percent

C

C	A general-purpose computer programming language originally developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories
Cache	A temporary storage area where frequently accessed data can be stored for rapid access. A cache is frequently used within a CPU to reduce memory access latency
CD	Compact Disc, an optical disc used to store digital data, originally developed for storing digital audio

CDF	Cumulative Distribution Function, describes the probability distribution of a real-valued random variable
CIP	Carrying Industrial Partner, the company that provides the problem and the industrial setting for the Industry as Laboratory project
Codec	Coder – Decoder, a hardware device or computer software used for coding and decoding transformations of data or signal streams
COTS	Commercial of-the-shelf, a term for software or hardware, generally technology or computer products, that are ready-made and available for sale, lease, or license to the general public. They are often used as alternatives to in-house developments
CPU	Central Processing Unit, an electronic circuit that can execute computer programs
Cronbach's alpha	A measure for the internal consistency (reliability) of a psychometric test

D

Deadlock	A situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does
DOORS	A requirements tracking tool from Telelogic / IBM
D-RAM	Dynamic Random Access Memory; data is stored in capacitors which require periodical refreshes to retain the information
DTI	Design Technology Institute, a co-operation between Technische Universiteit Eindhoven (TU/e) and National University of Singapore (NUS)
DVD	Digital Video Disc, an optical disc storage media format

E

Error	The part of the system state that may lead to a failure. E.g. a wrong memory value or a wrong message in a queue
ESI	Embedded Systems Institute

F

FA	Failure Attribution
Failure	An event that occurs when a state change leads to a run that no longer satisfies the external specification
Fault	The adjudged or hypothesized cause of an error which is not part of the system state; e.g. programming mistakes (such as divide by zero) or unexpected input
FUI	Function Importance

G

GNU	A range of applications developed and distributed by the Free Software Foundation. GNU is widely used by UNIX programmers and the acronym GNU means "GNU’s Not UNIX"
GNU GPL	The GNU General Public License (GNU GPL or simply GPL) is a widely used free software license, originally written by Richard Stallman for the GNU project
GUI	Graphical User Interface; a type of user interface which allows people to interact with electronic devices such as computers; hand-held devices such as MP3 Players, Portable Media Players or Gaming devices

H

HDMI	High-Definition Multimedia Interface, a compact audio/video interface for transmitting uncompressed digital data
HDTV	High-definition television, a digital television broadcasting system with higher resolution than traditional television systems
HRT	Hard Real-Time; the completion of an operation after its deadline is useless or may lead to a failure of the complete system

I

IDCT	Inverse Discrete Cosine Transform, a formula or an algorithm often used in signal and image processing, especially for lossy data compression
IMEC	Interuniversity Microelectronics Centre, a micro- and nano-electronics research center located in Leuven, Belgium
IPC	Inter-Process Communication; a set of techniques for the exchange of data among multiple threads in one or more processes
iPod	A brand of portable media players designed and marketed by Apple Inc

J

J-TAG	Joint Test Action Group, the common name used for the IEEE 1149.1 standard entitled Standard Test Access Port and Boundary-Scan Architecture for test access ports used for testing printed circuit boards using boundary scan

K

k	Kilo, one thousand. In software and memory size it may also stand for 1,024 or 2^{10}
Kernel	The central component of a computer operating system, including management of the system's resources
kLOC	Kilo Lines of Code, see LOC

Knapsack Problem	A problem in combinatorial optimization that derives its name from the maximization problem of the best choice of essentials that can fit into one bag to be carried on a trip. Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than a given limit and the total value is as large as possible
Koala	A software component model and architectural description language, developed by Philips Research, optimized to handle complexity, diversity and evolution in a resource constrained domain, such as televisions

L

LCD	Liquid Crystal Display, an electronically-modulated optical device shaped into a thin, flat panel made up of any number of color or monochrome pixels filled with liquid crystals and arrayed in front of a light source (backlight) or reflector. Modern televisions use an LCD to display the pictures instead of the earlier Cathode Ray Tube (CRT)
Linux	An operating system or OS-kernel used by a family of Unix-like operating systems
LL	Low Latency
LLVM	Low Level Virtual Machine; a compiler infrastructure, designed for compile-time, link-time, runtime, and "idle-time" optimization of programs
LOC	Lines of Code, a software metric used to measure the size of a software program by counting the number of lines in the text of the program's source code

M

M	Mega, one million. In software and memory size it may also stand for 1,048,576 or 2^{20}
MATLAB	A numerical computing environment and programming language, maintained by The MathWorks
MBD	Model Based Diagnosis
MIPS	A RISC microprocessor architecture developed by MIPS Technologies
MISRA	Motor Industry Software Reliability Association
MISRA-C	A software development standard for the C programming language developed by MISRA
MMKP	Multi-choice Multi-dimension Knapsack Problem
Mloc	Mega Lines of Code, see LOC
MP3	MPEG-1 Audio Layer 3, a digital audio encoding format using a form of lossy data compression

MPEG (MPG)	Moving Picture Experts Group, formed by the ISO to set standards for audio and video compression and transmission. Also a standard or file format for the generic coding of moving pictures and associated audio information
MPlayer	A free and open source media player
MPSoC	Multi-Processor System-on-a-Chip
MTBF	Mean Time Between Failures; the arithmetic mean (average) time between failures of a system
MTTF	Mean Time To Fail; the average time for a system to fail with the assumption that the failed system is not repaired
MTTR	Mean Time To Recover / Repair; the average time that a device will take to recover / to be repaired from any failure

N

NFF	No-Fault-Found, a term used in the field of failure analysis to describe a situation where an originally reported mode of failure cannot be duplicated by the evaluating technician and therefore the potential defect cannot be fixed
NP	Nondeterministic Polynomial time (complexity), a computational complexity class
NPD	New Product Development, the term used to describe the complete process of bringing a new product or service to market
NTSC	National Television System Committee, the analog television system used in most of the Americas, Japan, South Korea, Taiwan, the Philippines, Burma, and some Pacific island nations and territories
NVM	Non-Volatile Memory; computer memory that can retain the stored information even when not powered
NXP	NXP Semiconductors, a semiconductor company established by Philips in 2006 (formerly a division of Royal Philips Electronics)

O

OEM	Original Equipment Manufacturer, a company that uses a component made by a second company in its own product, or sells the product of the second company under its own brand
OS	Operating System; the interface between hardware and applications; it is responsible for the management and coordination of activities and the sharing of the limited resources of the computer. The operating system acts as a host for applications that are run on the machine
OSD	On Screen Display, an image or text superimposed on a screen picture, commonly used by modern televisions, VCRs, and DVD players to display information such as volume, channel, and time

P

PAL	Phase Alternating Line, a color-encoding system television used in broadcast television systems in large parts of the world
Pareto optimal	A concept originated in economics, named after Vilfredo Pareto. Given a set of alternative allocations of resources, a change from one allocation to another that can make at least one individual better off without making any other individual worse off is called a Pareto improvement. An allocation is Pareto optimal when no further Pareto improvements can be made.
PE	Processing Element
PR	Problem Report, a method to submit problems found and keep track of reported problems of a system and how it is solved or why it is not solved

Q

QA-C	A commercial static code analysis software tool produced by Programming Research for the C language

R

RAM	Random Access Memory, a computer memory that can be read from and written to in arbitrary sequence
RISC	Reduced Instruction Set Computing; a CPU design strategy based on the model that simplified instructions provide for higher performance if this simplicity is utilized to make instructions execute very quickly
RMS	Rate-Monotonic Scheduling, a scheduling algorithm used in real-time operating systems with a static-priority scheduling class
ROM	Read-Only Memory, a class of storage media used in computers and other electronic devices, mainly used to store firmware
RTOS	Real-time Operating System
RU	Recoverable Unit
RV	Runtime Verification

S

SCART	Syndicat des Constructeurs d'Appareils Radiorécepteurs et Téléviseurs, Radio and Television Receiver Manufacturer's Association, a French-originated standard and associated 21-pin connector for connecting audio-visual (AV) equipment together
SCM	Software Configuration Management, the task of tracking and controlling changes in the software
SD	Secure Digital, a non-volatile memory card format for use in portable devices
SECAM	Séquentiel couleur à mémoire, French for "Sequential Color with Memory", an analog color-encoding television system first used in used in broadcast television systems in France

SFL	Spectrum-Based Fault Localization, a diagnosis technique based on statistical analysis of execution profiles
Simulink	A tool for modeling, simulating and analyzing multi-domain dynamic systems, maintained by The MathWorks
Stateflow	An interactive design and simulation tool for event-driven systems, developed by The MathWorks
SUO	System Under Observation

T

TASS	A Dutch (software) service provider active in the area of technical and embedded software
Teletext	A television information retrieval service developed in the early 1970s. It offers a range of text-based information, typically including national, international and sporting news, weather and TV schedules and subtitles (or closed captioning)
TriMedia	A VLIW Media-processor family from NXP Semiconductors
TSU	Time Stamp Unit
TUD	Delft University of Technology
TU/e	Eindhoven University of Technology
TV520	A platform by NXP for LCD television sets
TVoM	TV on Mobile

U

UART	Universal Asynchronous Receiver/Transmitter; a piece of computer hardware that translates data between parallel and serial forms
UPFS	User Perceived Failure Severity
USB	Universal Serial Bus, a serial bus standard to interface devices to a host computer
UT	University of Twente
UML	Unified Modeling Language, a standardized general-purpose modeling language in the field of software engineering

V

VLIW	Very Large Instruction Word, a CPU architecture designed to take advantage of instruction level parallelism
Vproc	Video Processing, a component in the TV Software of NXP

W	
Word	A term for the natural unit of data used by a particular computer design. A word is a fixed-sized group of bits (often 16, 32 or 64) that are handled together by the machine

Appendix C

List of authors

The authors of the Trader book:

Name	**Information**[1]
Rui Abreu	Delft University of Technology r.f.abreu@tudelft.nl
Manvi Agarwal	NXP Research manvi.agarwal@nxp.com
Mehmet Akşit	Twente University m.aksit@ewi.utwente.nl
Cathal Boogerd	Delft University of Technology c.j.boogerd@ewi.tudelft.nl
Francky Catthoor	Interuniversity Microelectronics Centre francky.catthoor@imec.be
Rob Golsteijn	NXP Home rob.golsteijn@mapscape.eu
Teun Hendriks	Embedded Systems Institute teun.hendriks@esi.nl
Jozef Hooman	Embedded Systems Institute Radboud University Nijmegen jozef.hooman@esi.nl
Arjan van Gemund	Delft University of Technology a.j.c.vangemund@tudelft.nl
Jeroen Keijzers	Eindhoven University of Technology j.keijzers@tue.nl
Piërre van de Laar	Embedded Systems Institute pierre.van.de.laar@esi.nl

[1] Affiliation during the Trader project ; current email address (July 2009)

Koen van Langen	Eindhoven University of Technology k.s.w.v.langen @ student.tue.nl
Ilse Luyk	Eindhoven University of Technology i.m.luyk@tue.nl
Zhe Ma	Interuniversity Microelectronics Centre mazhe@imec.be
Somayeh Malakuti	Twente University s.malakuti@ewi.utwente.nl
Roland Mathijssen	Embedded Systems Institute roland.mathijssen@esi.nl
André Nieuwland	NXP Research andre.nieuwland@nxp.com
Matthijs Opdam	Eindhoven University of Technology m.opdam@student.tue.nl
Hasan Sözer	Twente University sozerh@ewi.utwente.nl
Bedir Tekinerdoğan	Twente University bedir@cs.bilkent.edu.tr
Peter Zoeteweij	Delft University of Technology Peter.Zoeteweij@gmail.com

For more information: office@esi.nl

Appendix D

List of consortium partners

Delft University of Technology	Delft, the Netherlands
Embedded Systems Institute	Eindhoven, the Netherlands
Eindhoven University of Technology	Eindhoven, the Netherlands
Interuniversity Microelectronics Centre	Leuven, Belgium
Leiden University	Leiden, the Netherlands
NXP Automotive	Nijmegen, the Netherlands
NXP Home	Eindhoven, the Netherlands
NXP Research	Eindhoven, the Netherlands
Philips Consumer Electronics	Bruges, Belgium
TASS	Eindhoven, the Netherlands
University of Twente	Enschede, the Netherlands

Embedded Systems
INSTITUTE

Trader: Reliability of high-volume consumer products

A collaborative research project on the reliability of complex embedded systems

Summary

The Trader Project is an industrial-academic research project, led by the Embedded Systems Institute in Eindhoven, The Netherlands. For a period of five years, researchers and engineers from NXP Semiconductors have worked closely together with researchers from the Delft University of Technology, Eindhoven University of Technology, University of Twente, Leiden University, IMEC, TASS and the Embedded Systems Institute. The research objective was to maintain or even improve the reliability in the face of ever increasing consumer electronics complexity.

NXP Semiconductors (formerly Philips Semiconductors) acted as the Carrying Industrial Partner. The HOME group of NXP Semiconductors, a world leading supplier of technology for digital television, provided the 'Industry-as-laboratory' research case. This enabled the research team to develop and validate the proposed reliability enhancing techniques and methods on leading edge technology.

This book summarizes the findings of the Trader project. Special attention is given to models and methods that predict, analyze and evaluate system reliability as perceived by end users. The ultimate goal is to create systems that have some form of high-level self-awareness and are able to detect and resolve unexpected system behavior. The methods and techniques have been applied to the domain of digital television and have shown promising potential to cope with the continuous increase in consumer electronics complexity.

Embedded Systems Institute

The Embedded Systems Institute is a public-private partnership founded in 2002 and supervised by a number of Dutch universities and companies: Delft University of Technology, Eindhoven University of Technology, University of Twente, NXP Semiconductors, ASML, Océ and Philips. Its mission is to advance industrial innovation and academic excellence in embedded systems engineering. A broad research program is executed, mostly based on cases from industrial practice. The research is carried out in partnership with academia, industrial partners and other research institutes. ESI also maintains an extensive knowledge dissemination program, including a wide variety of courses for system architecting and engineering.

Visit us at **www.esi.nl**