

ABSTRACT

JETLEY, RAOUL P. A Conformance Review Strategy for Regulating Safety-Critical Software. (Under the direction of Professor S. Purushothaman Iyer).

Safety is an important concern for software used in life-critical systems such as air transport, nuclear power and medical devices. The critical nature of these systems necessitates that the software used therein be reliable and free of errors. It becomes imperative, therefore, to have a stringent review process in place to ascertain the correctness of the software and to ensure that it meets all requirements and standards. Regulatory agencies encourage the use of formal methods-based techniques in the development of safety critical software. However, most manufacturers are reluctant to use these techniques, citing them as too complex and time consuming. As a result, (potentially life-threatening) errors are often not discovered until the software is already on the market. When such an error is eventually discovered, it becomes essential to trace the failure to its exact source in the implementation and to assure that the error correction restores the overall safety and effectiveness of the device.

In this dissertation, we present how efficient premarket and postmarket conformance reviews of designs and implementations can be carried out using formal methods-based techniques, to enable the process of reviewing software in safety-critical devices. To facilitate premarket conformance reviews, we introduce the notion of *usage models*, standardized formal models that serve as design templates. We present an approach to conformance checking of safety-critical software through formal verification and automated test case sequences derived from these usage models.

To provide for efficient postmarket reviews, we establish a methodology based on integrating program slicing with model abstraction to trace software failures to their root cause. We formalize this methodology by presenting an iterative algorithm for *abstraction-driven slicing*, and realize this algorithm through the implementation of the CAdS — a forensic analysis tool for C programs.

We provide case studies involving typical medical device software to illustrate the various concepts involved. We use results from these studies to establish the

effectiveness of our proposed approach and compare it to conventional static analysis techniques such as program slicing.

We demonstrate how our approach to premarket review is used to uncover safety errors in software implementations and show how abstraction-driven slicing can be used to improve the efficiency of postmarket reviews by up to 30%.

A Conformance Review Strategy for Regulating Safety-Critical Software

by

Raoul P. Jetley

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh

June, 2006

Approved By:

Dr. Laurie Williams

Dr. Matthias Stallmann

Dr. S. Purushothaman Iyer
Chair of Advisory Committee

Dr. Alexander Dean

To my parents.

Biography

Raoul Jetley was born on October 28, 1976, in New Delhi, India and spent most of his formative years in the quiet little town of Jamnagar. He received a Bachelors degree in Computer Science from Bombay University in 1998. Raoul enrolled at North Carolina State University in Fall 2000, and shortly thereafter began working under the direction of Dr. Purush Iyer. He received a Master's degree in Computer Science in December 2002.

Raoul has enjoyed a brief stint working in the IT industry, during which he served in the capacity of a programmer analyst at Shreyas Business Solutions Pvt. Ltd. and as a technical specialist at SETLabs, Infosys. His research interests include formal methods, automata theory, and artificial intelligence. His non-research interests include cricket and British heavy metal.

Acknowledgements

It is a myth that a dissertation is the soul-wrenching creation solely of its author's time, toil and tenacity. Many people conspired to drag this author kicking and screaming towards his goal. I thank these people for conspiring to do so.

First and foremost, I would like to thank my advisor and guide, Dr. Purush Iyer for his constant encouragement, support and patience, without which this work would not have been possible. Dr. Iyer has always had extraordinary faith and confidence in my abilities, even at times when I had little faith in myself. I cannot thank him enough for that.

I am grateful to my committee members for their valuable comments and suggestions. I would especially like to thank Dr. Laurie Williams and Dr. Tao Xie, whose suggestions have helped put my work in perspective and contributed greatly towards the shaping of this dissertation.

A number of people and organizations have extended their support to this research. I am grateful to all of them. I would like to thank Paul Jones and the CDRH for their guidance and insights into regulatory review processes. I would also like to thank Dr. Judy Prewitt and Rick Sutton at Duke Medical, Mike Parsons at CMS Inc., Dr. Fred Pearce at WRAIR and Eric Linner at Baxter for contributing data and code for the various code studies. Special thanks to Dr. Gene Stark for providing the applet for the CARA model.

Last but not the least, I would like to thank my family, friends and colleagues, who have always stood by me through thick and thin.

Contents

List of Figures	viii
List of Tables	x
1 Introduction	1
1.1 A Formal Methods Development Approach	2
1.2 The Premarket Review Process	4
1.2.1 Usage Models	4
1.2.2 Incorporating Usage Models in Premarket Review	5
1.3 An Approach to Postmarket Review	7
1.3.1 Combining Slicing with Model Abstraction	9
1.3.2 Potential Problem with Proposed Methodology	10
1.3.3 Automating Postmarket Review	11
1.4 Contributions	11
1.4.1 Case Studies	12
1.4.2 Assessing Case Studies	13
1.5 Thesis Organization	14
2 Background and Related Work	16
2.1 Model Based Software Development	16
2.1.1 Automata-based Modeling	17
2.1.2 Product Line Architecture	19
2.1.3 Related Approaches to Product-line Engineering	20
2.2 Verification and Validation	22
2.2.1 Temporal Logic	22
2.2.2 Linear Temporal Logic (LTL)	23
2.2.3 Model Checking	23
2.2.4 Model-based Testing	24
2.3 Program Slicing	27
2.3.1 Static Slicing	27

2.3.2	Dynamic Slicing	29
2.3.3	Conditioned Slicing	29
2.3.4	Amorphous Slicing	30
2.4	Model Abstraction	32
2.4.1	Model Abstraction Techniques	33
2.5	Post-mortem Forensic Analysis	36
2.5.1	Comparing Post-mortem Analysis Techniques	36
2.5.2	Corrective Maintenance vs. Post-mortem Analysis	38
2.6	Formal Methods Based Tools	39
3	The Premarket Review Process	42
3.1	Overview of the Premarket Review Process	43
3.2	Defining a Usage Model	46
3.2.1	Layered Architecture	47
3.2.2	Observer-based Architecture	48
3.3	Verifying Usage Models	49
3.4	Usage Scenario Generation and Testing	51
3.5	Case Study: The Generic infusion Pump	52
3.5.1	Overview of the GIP Model	52
3.5.2	The GIP Usage Model in Matlab	55
3.5.3	Verifying the GIP Usage Model	57
3.5.4	Generating Usage Scenarios for the GIP Model	60
3.6	Case Study: The CARA System	62
3.6.1	The CARA Usage Model	63
3.6.2	Verifying the CARA Usage Model	66
3.6.3	Generating Usage Scenarios for the CARA Model	67
4	Postmarket Forensic Analysis	69
4.1	Facilitating Program Comprehension	70
4.2	The Abstraction-driven Slicing Approach	72
4.2.1	Methodology	72
4.2.2	Computing a Slice using Codesurfer	74
4.2.3	Using CWolf to Abstract C Programs	75
4.3	CAdS — Automating Post-mortem Analysis	77
4.3.1	CAdS System Overview	78
4.3.2	Model Extraction	79
4.3.3	Graph Visualization	82
4.3.4	Slicing the CWI File in CAdS	83
4.3.5	Abstraction Map Generation	85
4.4	Example: Tracing Failure Report to Software	86

5	Putting it Together: Software Simulation Checking	91
5.1	Assessing Impact of Change in Software	92
5.1.1	Using Simulation Checking to Compare Models	93
5.1.2	Using CWB-NC for Semantic Checking	96
5.2	Simulation Checking for the GIP System	97
6	Results and Observations	100
6.1	Results from the Generic Infusion Pump Study	101
6.1.1	Verification of the GIP Usage Model	101
6.1.2	Testing a GIP-based Implementation	102
6.1.3	Forensic Analysis of the PCA Prototype	102
6.1.4	Assessing Change using Preorder Simulation	105
6.1.5	Effort Expended	106
6.2	Results from the Premarket Review for CARA	107
6.2.1	Discrepancies	108
6.2.2	Missing Specifications	109
6.2.3	Testing the CARA Applet	110
6.2.4	Effort Expended for Reviewing CARA	110
6.3	Results from the Forensic Analysis of XiO	111
6.3.1	Evaluating the Approach	111
6.3.2	Effort Analysis	113
6.4	Summary	114
7	Conclusion and Future Work	117
7.1	Improving Premarket Review	118
7.2	Enhancing Abstraction-driven Slicing	119
	Bibliography	121

List of Figures

1.1	Usage models and the premarket review process	6
1.2	Using slicing and abstraction for postmarket review	10
2.1	The product line life cycle	20
2.2	A model-based testing workflow	25
2.3	Sample C code for a taxation program and slice w.r.t. line 32	28
2.4	Dynamic slice for input $\{\text{income} = 7200, \text{age} = 35\}$	30
2.5	Conditioned slice for the condition $\text{income} \geq 65000$	31
2.6	The model abstraction process	33
3.1	A layered architecture for Usage models	47
3.2	Example of an observer-based architecture for usage models	49
3.3	Overview of the usage model for infusion pump software	54
3.4	The Simulink interface to the infusion pump usage model	55
3.5	Verifying the usage model in Reactis	59
3.6	Sample LTL formulae used in Uppaal to verify the GIP usage model	60
3.7	Part of the usage scenario for the PCA prototype implementation	62
3.8	Overview of the CARA usage model	64
3.9	Sample LTL formulae used to verify the CARA usage model	67
3.10	Screenshot of the applet used to simulate the CARA system	68
4.1	Computing a backward slice using a program dependence graph (PDG)	75
4.2	Architecture of CWolf	76
4.3	A simple C program to recognize a^*b^* and its minimized LTS	77
4.4	Abstraction and label maps for the program of Figure 4.3 (A)	78
4.5	The CAdS System Architecture	79
4.6	The effect of data abstraction on an LTS	81
4.7	The DTD for the Graph File	82
4.8	A Partially Expanded Graph in CAdS	83
4.9	Portion of an abstract model generated for the XiO software.	88

4.10	Result of a forward slice on <code>plan_has_photon_nonseg_port_or_mlc</code>	89
4.11	An example of error tracing using the CAdS tool	90
5.1	Detecting regression errors in revised code	94
5.2	Checking consistency of updated usage models	95
5.3	Using CWB-NC to assess changes in modified software	97
5.4	Comparing an updated model to the original using Simulation checking	98
6.1	Detecting an error during testing for the GIP system	102
6.2	Number of iterations for forensic analysis of the PCA prototype	103
6.3	Successive slice sizes during forensic analysis of the PCA prototype .	104
6.4	Distribution of blocks (states) per each level for CAdS	105
6.5	Effort distribution for (A): Premarket review and (B): Forensic analysis for the GIP study	106
6.6	Number of iterations for forensic analysis of the XiO system	113
6.7	Comparison of slice sizes for the XiO forensic analysis	114

List of Tables

2.1	Comparison of Post-mortem analysis techniques	38
4.1	Abstractions provided by CWMB	80
6.1	Effort required for the CARA study	111
6.2	Effort required for forensic analysis of XiO	114

Chapter 1

Introduction

Given a diverse collection of stakeholders, there is considerable interest in the requirements engineering community in supporting the tasks of eliciting, analyzing and negotiating requirements. Typically, the main stakeholders are developers, users and customers of computer systems. However, in the case of safety-critical computer systems there is another important stakeholder — a safety regulatory authority. In many areas such as air transport, nuclear power and medical devices, there are special state regulatory bodies whose basic tasks are independent control of safety, certification and licensing of activity connected with safety. The primary aim of these bodies is to assure that software released to market does not endanger (or compromise the safety of) end users for these systems. In the case of computer control systems, there are two main tasks, or reviews performed by the regulatory body:

1. Establishing regulatory requirements for systems and assessing software during certification or licensing (regulatory assessment). The purpose of this assessment is to be convinced that the system and software answer established regulatory requirements.
2. Performing post-mortem forensic analysis to trace system failures to their source in the software. This is usually performed on software that has already been

released to the market.

The first of these reviews is performed before the software is released to market, to check for conformance to (safety) standards. The second review is required when a defect or anomaly is detected in software already on the market. Borrowing terminology from the medical device industry [Koh96, Glo03], we refer to these as **premarket conformance review** and **postmarket forensic review** respectively.

These premarket and postmarket reviews are complex, labor intensive processes, and have a high emphasis on software safety. With this in view, formal methods have often been recommended as a means for carrying out these tasks [Rus93, HB99].

The application of formal methods during certification has been considered to a much lesser degree than for software development, and is not widely put into practice at present. Several theories and tools have been proposed for this purpose. However, these have been deemed to be too impractical or not comprehensive enough to be of much use. As a result, formal methods are still to be used extensively in a regulatory environment. Nevertheless, the application of formal methods in this area has potential to be useful and successful. In this thesis, we present one scheme for a leveraging existing formal methods-based techniques to provide a viable approach to review of safety-critical software. The approach presented is specific to the problems faced by most regulatory bodies, but would benefit equally developers, end-users and regulatory analysts.

1.1 A Formal Methods Development Approach

Formal methods provide a methodological framework for system specification, production and verification. These methods provide the means of development and assessment of a system or parts of a system, based on principles of logic, set theory, abstract algebra or graph theory, and which allow for mathematical manipulation and proof of properties.

The process of applying formal methods to a chosen application typically involves the following phases:

1. requirements elicitation/formal specification,
2. modeling,
3. verification (model checking),
4. implementation,
5. testing, and
6. maintenance and post-mortem analysis.

Formal specification is defined as the characterization of a planned or existing system expressed in a formal language. The term *modeling* refers to the mathematical representation of a system underlying the systems specification. The distinction between the two phases is somewhat artificial and should not be taken too literally, since there is a considerable amount of overlap between them, with the former generally leading into the latter.

Formal verification, usually achieved through model checking [CES86, Mer01], involves validation of the specifications, proving key properties and invariants, and establishing the consistency of axioms. *Implementation* involves converting (either manually or automatically) the verified model to executable code, similar to code derivation in the model-driven approach to software development [Ken02, RFW⁺04].

Testing and *maintenance* activities are similar to those in conventional paradigms (e.g., [Boe88, Boo86]) and are mostly independent of the formal approach. Lastly, *post-mortem or forensic analysis* refers to the process of tracing software failure to source code, and is used by maintenance engineers and regulatory reviewers for establishing the cause of the failure.

In the ideal case, software manufacturers would use formal methods during the early phases of the System Development Life Cycle (SDLC) to reduce the possibility of errors creeping in the software. However, most manufacturers do not employ formal methods, contending that they involve expending huge amounts of effort and time, are too complicated and cumbersome to use (especially by the average analyst), and are difficult to learn [BM92].

To ensure safety of the software, therefore, formal methods could be employed during the latter stages of the SDLC, viz., during the testing and maintenance phases. Doing so would lower the emphasis on design-time formal analysis and allow for integration of testing and maintenance with the formal approach as well. One such scheme, as presented here, involves the use of formal methods for the premarket and postmarket regulatory review processes. Generally speaking, the premarket review would employ formal techniques for the testing phase of the SDLC, while postmarket review would be addressed during corrective maintenance and post-mortem analysis.

Such a scheme would require the involvement of manufacturers, testers, maintenance engineers, as well as regulatory agencies, and would thus encourage the use and possible standardization of formal methods based practices.

1.2 The Premarket Review Process

Typical regulatory (premarket) review procedures are process-oriented and based on generalized guidelines such as normative documents of the regulatory authority, national and international standards (IEC, ISO etc.), and recommendations of international organizations (IAEA etc.). These reviews, although not entirely without their benefits, do not assure strict conformance to safety and are unable to ensure that all safety requirements are satisfactorily met by the software. In order to perform a more stringent and complete conformance check, software implementations themselves need to be evaluated against safety requirements.

As a means to provide for such an evaluation, and to incorporate a formalized testing methodology for premarket review, we introduce the concept of a formal *usage model*.

1.2.1 Usage Models

A *usage model* characterizes the use of software in the intended environment and incorporates a common set of properties and safety requirements that can be defined

and analyzed rigorously using formal methods based techniques. The concept of a usage model was introduced independently by John Musa [MM98] and Poore et al. [WPT95, PT99], who define it as *a formalism that enables the application of many statistical principles to software testing and forms the basis for efficient testing in support of decision making*. The software implementations sharing this common set of properties are said to belong to the same *software family*. When used in the premarket review process, the usage model acts as a standardized design model for the software family that can be used by manufacturers as a reference for their individual implementations.

Importantly, the usage model can also be used to derive a suite of test cases or *usage scenarios* against which specific implementations can be validated. This suite includes a comprehensive set of usage scenarios that must always be satisfied (corresponding to safety properties), optional scenarios that correspond to features that are desirable but not mandatory (best practices), and scenarios that must never hold true (error conditions). Using this test suite for validation would ensure that (at least) minimum product level safety guidelines would be satisfactorily met.

This approach to premarket review would ensure that the minimum safety standards are satisfied, and at the same time, save effort and minimize errors associated with the model building process while contributing to a reduction in regulatory recall and corrective action costs.

1.2.2 Incorporating Usage Models in Premarket Review

Figure 1.1 gives a schematic of how a usage model is utilized to enhance premarket regulatory review process. The scheme as depicted in the figure involves deriving a test suite from a formal usage model, essentially a one-time effort, and using this test suite to assess individual software implementations.

The process of deriving the test suite consists of three separate activities or phases. The first phase deals primarily with establishing requirements for the usage model and delineating the system specifications. The second phase comprises of formally verifying the defined model. Lastly, in the third phase, usage scenarios are derived

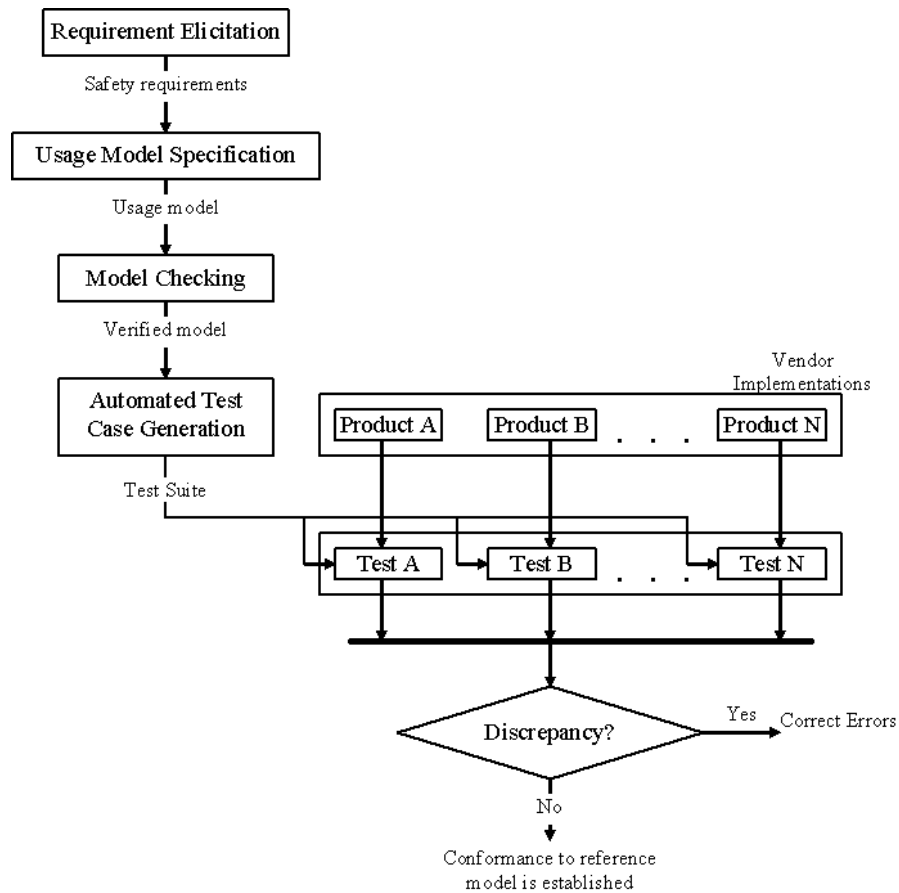


Figure 1.1: Usage models and the premarket review process

from the verified model.

To have the model serve as a benchmark for all software implementations in a software family, it is imperative to ensure that the model is sound and correctly incorporates the major functionalities of the device. To ensure this, the requirements are collected in consultation with a set of experts (regulatory and manufacturers). With the help of these experts, common characteristics of the device are identified, and used to define the specifications for the usage model.

While it may not be possible to capture all functionalities and behavior for each and every device in the generic usage model, the model must consist of a set of minimum safety properties that all implementations of the device must possess. The

implementations are, in effect, evaluated against these safety properties during testing. A failed test scenario indicates a violation of a safety condition, that may lead to a potential hazardous malfunction.

The usage model can be defined using formal methods-based specification techniques and is (usually) represented as a state machine or finite state automaton. The advantage of using a formal model is that it can be rigorously verified using model checking and exhaustive state explorations to ensure its correctness and adherence to the safety requirements.

Once the model is verified, it is used to automatically derive test sequences, or usage scenarios. Each sequence generated, in essence, is an execution path in the usage model and corresponds to a set of user inputs or events for the system. All such distinct paths (test sequences) and their corresponding results are recorded and maintained as a test suite for products in the software family. As a manufacturer develops an application/product, it is checked against the test suite for conformance. Any discrepancy between the expected results (as obtained through the usage model) and observed outputs indicates a failure of compliance for the product. If the results of the tests are in agreement, however, then the product is deemed compliant with the established safety criteria.

1.3 An Approach to Postmarket Review

Ideally, with extensive premarket review, the need for forensic analysis should be minimal. However, there could be situations not covered by the initial usage model that could have lead to a catastrophic failure. Further, it is likely that software in the field might have errors introduced in it due to bug fixes or adaptive maintenance, and thus differs from the original software that was reviewed. A major failure under these circumstances could then necessitate a postmarket review, to determine the cause of the failure and to affix blame for the same.

A significant part of the postmarket regulatory review process involves post-mortem forensic analysis to determine the cause of a malfunction and trace it to

its source in the software. Typically, during post-mortem analysis, the forensic investigator (or analyst) investigates the cause of software failure by tracing incidents reported in failure reports and error logs to their origin in source code. To do this, the analyst requires (at least) a basic understanding of the program structure and system design. This understanding is usually achieved by manually scanning through the source code and the system documentation.

Manual scanning of code, however, is rather inefficient and time consuming; especially for large, complex applications with several possible interleavings across several threads. The process is further exacerbated by insufficient documentation and poor design. As a result, the analyst often has to spend a considerable amount of time and effort trying to understand the code by manually scanning through it.

To ease this burden for the analyst, the forensic analysis process could be aided by the use of static analysis tools and techniques. The most popular of these methods, used extensively by debuggers and maintenance engineers, is program slicing [Wei81] — a technique to facilitate understanding and debugging of programs by focusing on selected aspects of semantics. Slicing aids analysis by deleting those parts of the program which can be determined to have no effect upon the semantics of interest, and focuses attention on the segment of code which may contain a fault.

The slicing-based approach is popular owing to its simplicity and ease of use. However, it is not always the most efficient, especially when used for large, highly cohesive programs. The slices obtained for these programs are usually too large, and require several refinements and iterations before the error is successfully traced to its source. Moreover, it is not usually possible to execute or dynamically debug the code during post-market review, rendering infeasible the use of runtime analysis techniques such as dynamic slicing and execution backtracking [ADS93]. Consequently, the analyst is constrained by the limitations of static slicing and ends up spending considerable effort during the post-mortem analysis process.

1.3.1 Combining Slicing with Model Abstraction

One way to overcome this problem and improve the efficiency of forensic analysis is to combine static slicing with model abstraction [CC77, CC79]. Using abstraction in conjunction with slicing provides the analyst with an abstract semantic representation of the program that improves his understanding of the program and helps define better slicing criteria in order to derive more concise slices. The slices in turn contribute to deriving more refined abstract models for the program. Using the two in tandem reduces the number of iterations and slices required during analysis and thus leads to a reduction in effort required for post-mortem analysis.

Figure 1.2 depicts a scheme for how program slicing can be combined with model abstraction to aid forensic analysis.

Postmarket review, as shown in this figure, is an iterative process, with successive slices used to refine the abstraction, and successive (refined) models producing smaller, more compact slices. The process starts by using the reported failure to form an initial slicing criterion. This criterion is used to obtain a slice of the program that helps the analyst narrow down the code responsible for the failure. An abstract model is then derived from this slice and used to form the slicing criterion to further refine the slice and narrow down the cause of failure. The process continues till the slices (and models) are refined enough to successfully trace the cause of the failure.

The critical part in this scheme is the extraction of an abstract model from the source code. The abstract model is obtained by mapping statements in the concrete program domain to states in the abstract model domain. Two popular strategies for extracting abstract models from source code are data abstraction and predicate abstraction. The basic idea of data abstraction [CC79] is to evaluate data variables in the program over a smaller abstract domain, while still preserving the existential behaviors of the program execution. Predicate abstraction [SH97] on the other hand, encodes concrete states using the truth evaluation of a set of predicates present therein. Thus, any concrete states sharing the same assignment to all of the predicates are abstracted into the same representation in the abstract state.

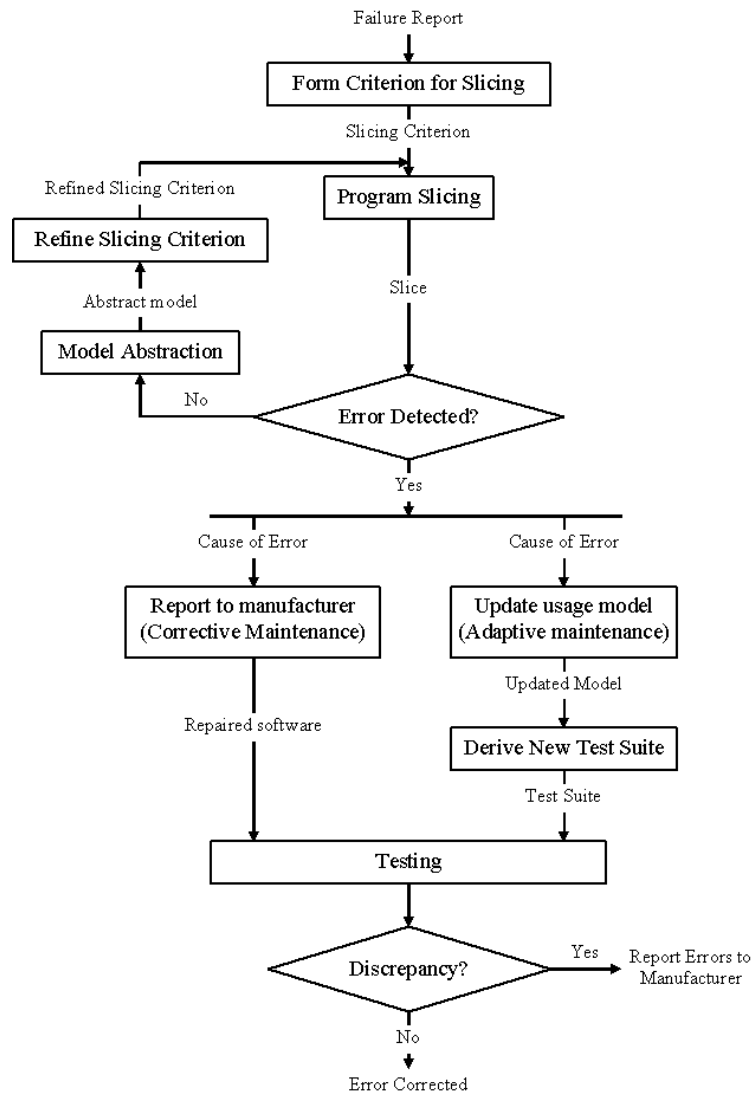


Figure 1.2: Using slicing and abstraction for postmarket review

1.3.2 Potential Problem with Proposed Methodology

A major problem with abstraction is that the extracted model may not accurately depict the behavior of the software. The abstraction may be too coarse or too fine; resulting in a model that either produces spurious errors, or is not detailed enough to detect all the errors in the software. A precise abstraction should be able to determine the slightest change in the source code and produce a model that can be distinguished

from the model generated before modifying the code.

We assess the precision of the abstraction in our approach by comparing models extracted from original code to those extracted after an error is fixed. The Concurrency Workbench of the New Century (CWB-NC) [RS96] is used to detect whether the two models are semantically equivalent. If CWB-NC is unable to distinguish between the two models, it would mean that the abstraction used is too coarse. This comparison between models also helps assess the impact of the error correction and ensures that the changes made during error correction do not have any sort of negative impact on the software.

1.3.3 Automating Postmarket Review

To improve the efficiency of the postmarket forensic analysis process, we automate the abstraction-driven slicing approach shown in Figure 1.2. This automation is achieved through the implementation of the **C language Abstraction driven Slicer (CAdS)** tool, that combines slicing and abstraction for the C language to aid forensic analysis. We explain the architecture for the CAdS tool in Section 4.3.

1.4 Contributions

This thesis describes a framework for using static analysis techniques to aid regulatory analysis. An important contribution of the work is that it combines existing static analysis approaches, such as model checking, abstract interpretation, and program slicing to define procedures for regulatory conformance reviews. The premarket and postmarket review procedures presented here prescribe an efficient methodology for assuring software safety in regulatory-controlled safety-critical systems. Specifically, the following are the contributions of this thesis:

1. We introduce the concept of formal usage models as a means to standardize safety requirements and validate software implementations.

2. We present an approach to premarket conformance review for safety-critical software. The approach is based on formal verification of usage models, and using these verified models to generate automated test suites for validating specific software implementations.
3. We establish the concept of abstraction-driven slicing, an integrated approach to postmarket forensic analysis using a combination of static slicing and model abstraction. The integrated approach is an improvement over traditional approaches to program comprehension, and provides a better understanding of the program structure and semantics. This leads to more efficient debugging and forensic analysis.
4. We present an iterative algorithm for abstraction-driven slicing and demonstrate the effectiveness of the approach with the help of real-life case studies.
5. We develop the architecture for CAdS, an automated tool for forensic analysis. CAdS is based on the abstraction-driven slicing paradigm and provides algorithms for interprocedural static slicing, interactive graph visualization and abstraction criterion generation.

1.4.1 Case Studies

To explain and illustrate the conformance review methodologies, we evaluate the viability and efficiency of our approach against three separate safety-critical systems. All three of these systems deal with the analysis of medical device software. We chose medical devices as our example since they embody typical features exhibited by most high-confidence systems. Medical devices are highly safety-critical devices based on embedded system architecture, with software used to control the underlying hardware components and constantly monitor the patient for vital signs. The end-user for these devices is not an expert, but a medic (or in some cases, the patient), with limited knowledge the system. This makes it all the more important for the software to be as reliable as possible.

To implement our review process, we build a standardized design model for a Generic Infusion Pump (GIP). We use the model to verify the system requirements, and the corresponding implementation to perform forensic analysis as a means to trace reported software failures to the source code.

Apart from the generic infusion pump, we also consider other, more specialized implementations to evaluate the proposed methodology. The Computer-Aided Resuscitation Algorithm (CARA), designed by the Walter Reed Army Institute of Research (WRAIR), is used for model checking system specifications and help exhibit the utility of formal verification for a large-scale medical device.

The XiO radiation therapy planning software, developed by Computerized Medical Systems (CMS) Inc., is used as a case study for postmarket forensic analysis. Both the GIP system and the XiO software are also used to evaluate the CAdS tool.

Our research was carried out in collaboration with, and with support from the US FDA. The specifications and source code for the case studies were provided by the Center for Devices and Radiological Health (CDRH), US FDA.

1.4.2 Assessing Case Studies

The major objective of the premarket review is to assure that implementations in a software family conform to a minimum common set of safety criteria. To achieve this, we define formalized usage models for the GIP and CARA case studies. The models are verified through formal verification techniques and used to derive automated test suites. These test suites are then used to validate specific software implementations. The number of errors detected during these processes of verification and validation serves as a measure of the effectiveness for the premarket review approach.

To evaluate the postmarket review process, we consider the ability of the process to trace failures to their source and the effort required to trace these errors. The effort is measured in terms of the number of iterations needed to establish the cause of failure (generally speaking, a higher number of iterations results in greater effort). The approach is compared against static program slicing — a popular forensic analysis technique to assess its efficacy and savings in effort provided.

1.5 Thesis Organization

This thesis is organized into seven chapters as follows:

Chapter One: *Introduction*, which you are reading now, contains the motivation and goals of the research embodied in this thesis.

Chapter Two: *Background* covers the theoretical aspects of topics addressed in later chapters and cites related work pertinent to this dissertation. In particular, it discusses model checking, program comprehension, product line architectures, program slicing and model abstraction.

Chapter Three: *The Premarket Review Process* presents the approach to premarket review based on formal usage models. The three stages involved in the review process are detailed. These include defining usage models, using model checking to verify the models, and deriving usage scenarios to test individual applications. The GIP and CARA systems are used as examples to illustrate the three stages in detail.

Chapter Four: *Postmarket Review Strategies* gives an outline of the processes involved during postmarket review, in particular for post-mortem forensic analysis of software. The algorithm for abstraction-driven slicing is introduced as a means to enhance program comprehension, and a framework to automate this process is presented. The XiO and GIP case studies are used to illustrate the effectiveness of the approach.

Chapter Five: *Putting it Together: Software Simulation Checking* binds the premarket and postmarket review processes together using the notion of simulation checking to bridge the gap. Simulation checking is used to ascertain that the abstract model extracted from the code conforms to the safety properties specified in the usage model.

Chapter Six: *Results* lists the results obtained for the case studies and examples introduced in previous chapters. For premarket review, results are presented for model checking and testing based on usage models. The post-mortem analysis process is evaluated by comparing it against static slicing, a popular tool for debugging and program comprehension.

Finally, *Chapter Seven: Conclusion* summarizes this thesis and provides a road map for future research.

Chapter 2

Background and Related Work

In this chapter, we introduce the concepts required in latter parts of the thesis and elaborate on related work in the respective areas. We start off by introducing model-based software development in Section 2.1. In Section 2.2, we address means to verify these models and using them to test software implementations. We then explain various approaches to program slicing in Section 2.3, and describe different methods to extract abstract models from source code in Section 2.4. In Section 2.5 we establish the role of program comprehension during post-mortem forensic analysis and compare existing techniques to our proposed abstraction-driven slicing paradigm. Lastly, in Section 2.6 we list the various formal methods based tools used to facilitate the concepts introduced.

2.1 Model Based Software Development

A formal model defines a mathematical representation suitable for formalizing the application domain and for calculating and predicting the behavior of the application in that context. Formal models are more precise than informal descriptions written in natural language or in quasi-formal notations, such as pseudo code, diagrammatic

techniques, and many CASE notations. One aspect of precision is the need to examine and make explicit all underlying assumptions; hence, mathematical models also tend to force a more thorough analysis. This analysis is carried out using established methods of mathematical reasoning, e.g., model checking.

2.1.1 Automata-based Modeling

Formal models can be classified in terms of two broad classes: functional and automata-based. In the functional model of computation every problem is viewed as an evaluation of a function. The solution to a given problem is specified by a complete and unambiguous (often recursive) functional description. Automata-based models, on the other hand, typically consist of an abstract representation of system state and a set of operations that manipulate the state to effect a transition from the current to the next state. Automata-based models are usually employed for formal modeling, owing mainly to their expressiveness and formalized means for verification. In this thesis too, we develop and use automata-based usage models to aid premarket review processes. With this in view, we give the following definitions.

2.1.1.1 Finite State Automata

An automaton (plural automata) is a finite-state transition system consisting of a set of states and a set of state-to-state transitions that occur on input symbols chosen from a given alphabet. Formally, a deterministic finite state automaton is defined as a 5-tuple $M = (S; \Sigma; \delta; s_0; F)$, where S is a finite set of states, Σ is a finite event alphabet, s_0 is the initial state, $F \subseteq S$ is the set of final states, and $\delta \subseteq S \times S$ is the transition relation for M , such that $\forall s \in S (\exists s' \in S (s, s') \in \delta)$.

Finite state automata may be deterministic, meaning that there is a unique transition from a given state on a given input, or non-deterministic, meaning that there are zero, one, or more such transitions. Non-deterministic finite automata, usually used in model-checking, are abbreviated as *Kripke structures*, given by the triple $M=(S, \Sigma, \delta)$.

2.1.1.2 ω -Automata

To accommodate eventualities, it is necessary to use a class of automata that accepts infinite words (sequences), a facility provided by ω -automata. Like a conventional automaton, an ω -automaton consists of a set of states, an input alphabet, a transition relation, and a distinguished initial state. The difference between the two classes of automata occurs in the definition of acceptance. Acceptance for a conventional automaton is defined in terms of a final state. Since the notion of final state is not useful for a class of machines that accepts infinite words, acceptance must be defined in some other way.

A (non-deterministic) ω -automaton is thus defined as a 5-tuple $(S; \Sigma; \delta; s_0; \mathcal{F})$, where S , Σ , and s_0 are as defined above, $\mathcal{F}$ is an acceptance condition, and $\delta: S \times \Sigma \rightarrow \mathcal{P}(S)$ is a transition function. The automaton is deterministic if for every state $s \in S$ and every $a \in \Sigma$, $|\delta(s, a)| \leq 1$.

A path in an ω -automaton, M , is an infinite sequence of states $s_0 s_1 s_2 \dots \in S$ that begins in s_0 and has the following property: $\forall i \geq 1, \exists a_i \in \Sigma: \delta(s_i, a_i) = s_{i+1}$. A path $s_0 s_1 s_2 \dots \in S^\omega$ in M is a run of an infinite word $a_1 a_2 \dots \in \Sigma^\omega$ if $\forall i \geq 1: \delta(s_i, a_i) = s_{i+1}$. The infinitary set of a sequence $s_0 s_1 s_2 \dots \in S^\omega$, written as $\inf(s_0 s_1 s_2 \dots)$, is the set of all states that appear infinitely many times in the sequence [CBK90].

2.1.1.3 Büchi Automata

A Büchi Automaton M is an ω -automaton where the acceptance condition, $\mathcal{F}$, is defined as follows. $\mathcal{F} \subseteq S$ is a set of states (as in the case of a $*$ -automata) and a path p is accepted by M if $\inf(p) \cap \mathcal{F} \neq \emptyset$ [CBK90]. An infinite word is accepted by a Büchi automaton if it has an accepting run in the automaton. Büchi automata are generally used in model checking as an automata-theoretic version of a formula in linear temporal logic (LTL) [Var96, Wol02].

2.1.1.4 Timed Automata

Timed automata are a generalization of ω -automata and are used to model real-time systems over time. Like ω -automata, timed automata generate (accept) infinite sequences of states. However, timed automata must also satisfy timing requirements and produce (accept) timed state sequences. Timed automata may be given various semantic interpretations, including point-based strictly-monotonic real-time (the original interpretation), interval-based variants, interleaving, fictitious clock, and/or synchronicity [AH92].

2.1.2 Product Line Architecture

To have a formal model serve as a shared design (as in a usage model) for software applications, it needs to capture generic properties common across all implementations. This feature of usage models is based upon the concept of software product-line architecture [WL99a, WL99b, CNN01].

The software product line approach aims at producing a set of products that are developed on a basis of a common software product family architecture and a set of shared assets [Bos00]. The shared assets and the software product family architecture are developed for reuse in a development process and reused to build products in a prescribed way in a deployment process. The creators of the product line use a systematic process that exploits the commonality across the family, while allowing variation among the members.

The goal of product line engineering is to support the systematic development of a set of similar software systems by understanding and controlling their common and distinguishing characteristics. It is thus an approach for software reuse driven by the concepts from the real-world domain of the software products, which are used to tackle the main reuse challenges. These challenges are depicted in Figure 2.1 in relation to the overall life cycle of a product line. The concepts, entities, and relationships in the application domain are analyzed and used to build a product line infrastructure, which includes reference architecture for the systems in the domain.

Concrete applications can then be constructed largely by instantiating and reusing this product line infrastructure.

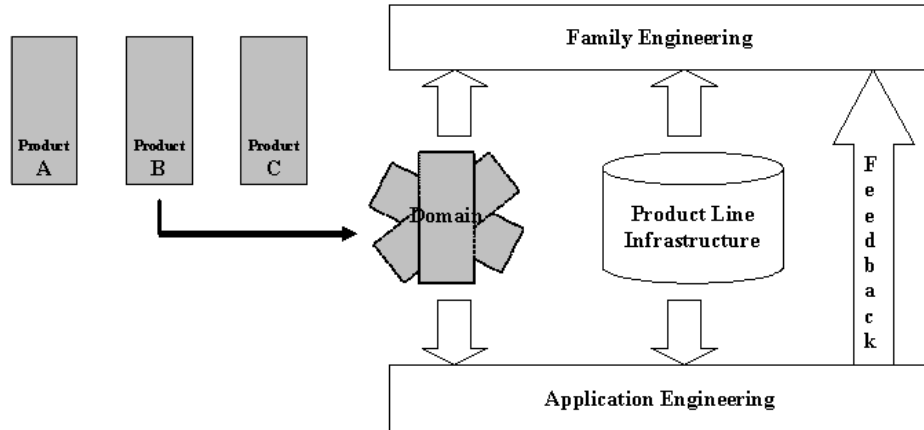


Figure 2.1: The product line life cycle

2.1.3 Related Approaches to Product-line Engineering

There have been several process models dividing the product-line engineering process into domain engineering and application engineering. These methods have first concentrated on domain engineering, and have been afterwards extended with application engineering.

FODA (Feature-Oriented Domain Analysis) [KCH⁺90] considers the features of similar applications in a domain. Features are defined as capabilities of the applications considered from the end-users point of view, and cover both common and variable aspects among related systems. FODA includes different analyses such as requirements analysis and feature analysis, and produces a domain model covering the differences between related applications. FODA is currently a part of the model-based approach of domain engineering [dom00]. FODA has been further extended into FORM (Feature-Oriented Reuse Method) [KKLL99, KLD02] to include also software design and implementation phases. FORM covers both analysis of domain features and using these features to develop reusable domain artifacts.

ODM (Organization Domain Modeling) [Sim95, SCK⁺96] mainly concentrates on the domain engineering of legacy systems. However, it can be applied to the requirements for new systems. ODM is tailorable and configurable, and it can be integrated with other software engineering technologies. It combines different artifacts such as requirements, design, code, and processes from several legacy systems into reusable common assets. ODM is supported by DAGAR (Domain Architecture-based Generation for Ada Reuse) [KS96].

RSEB (Reuse-Driven Software Engineering Business) [JGJ97] is a systematic, model-driven reuse method. It composes sets of related applications from sets of reusable components. RSEB uses UML (Unified Modeling Language) [RJB99] to specify application systems, reusable component systems, and layered architectures. Variabilities between systems are expressed with variation points and attached variants. FeatuRSEB (Featured RSEB) [GFdA98] connects features (from FODA) with RSEB. FODA and RSEB, in fact, have much in common. Both approaches are model-driven methods providing several models corresponding the different view points of the domain, and are thus compatible with each other.

FAST [CHW98] is a development process for producing software in a family-oriented way that applies software product-line architecture principles into software engineering process. The FAST process separates product-line engineering process into two main parts. One step concentrates on providing the core assets including the environment for implementing each product. The other step utilizes the environment in the production of different software products belonging to the family.

PuLSE (Product Line Software Engineering) [SJKM05] divides product-line life cycle into three parts. In the initialization phase, PuLSE is customized to fit the particular application. In the second phase, product-line infrastructure is constructed. This step includes scoping, modeling, and architecting the product-line. In the third phase, the product-line infrastructure is used to create individual products. This concerns instantiating the product-line model and architecture. Each of these phases is associated with product-line infrastructure evolution. Further, each phase considers changing requirements and changing concepts within the domain.

2.2 Verification and Validation

Once a formal model is defined, it needs to be verified against the system requirements before it is used as the basis for implementation. A popular approach to verification is through model checking, which uses temporal logics to assure that desired system properties are satisfied. The software implementation itself can be validated using model-based testing, i.e., by using test cases derived from (verified) models to test the implementation. In this section, we give a brief introduction to the concepts involved for each of these approaches.

2.2.1 Temporal Logic

Temporal logic (also known as tense logic) [Pnu77, Bur84] augments the standard operators of propositional first-order logic with tense operators that are used to formalize time-dependent conditions. The simplest temporal logic adds two operators: the (weak) future operator, F , and the (weak) past operator, P . The formula Fq is true in the present if q is true at some time in the future and, similarly, the formula Pq is true in the present if q is true at some time in the past. These operators can be combined to assert quite complex statements about time-dependent phenomena. For example, $q \Rightarrow F P q$ can be interpreted as “if q holds in the present, then at some time in the future q will have held in the past”. The duals of these operators, $\neg F \neg$, usually abbreviated G and $\neg P \neg$, usually abbreviated H , yield the corresponding strong future and past operators. $Gq \equiv \neg F \neg q$ means that q is true at every moment in the future, and $Hq \equiv \neg P \neg q$ means that q is true at every moment in the past.

The purpose of temporal logics is to specify properties of dynamic systems. These properties can be classified as either

- Desired properties. Often liveness properties like “In every infinite run action ‘a’ occurs infinitely often”, or
- Unwanted properties, i.e., error conditions. For example, properties like “Process A and process B are in the critical section at the same time”

and used to check the system model for conformance to liveness and safety criteria.

Temporal logic can be classified as Combinational Tree Logic (CTL) and Linear Temporal Logic (LTL). While both these logics are comparable with respect to their expressiveness, LTL is preferred in most model checkers owing to its support for on-the-fly verification [GPVW96, Hol96].

2.2.2 Linear Temporal Logic (LTL)

Linear time corresponds to the usual notion of time as a linearly ordered set, measured either with the real or the natural numbers. An LTL formula ϕ can be defined to have the following syntax:

$$\phi ::= \text{true} \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \langle a \rangle \phi \mid \phi_1 \mathcal{U} \phi_2 \mid \phi_1 w\mathcal{U} \phi_2$$

For a transition system with states S let $\text{runs}(S)$ be the set of all runs (starting at any state). The denotation $\llbracket \phi \rrbracket$ of the formula ϕ is then defined inductively as follows:

$$\begin{aligned} \llbracket \text{true} \rrbracket &:= \text{runs}(S) \\ \llbracket \neg\phi \rrbracket &:= \text{runs}(S) - \llbracket \phi \rrbracket \\ \llbracket \phi_1 \wedge \phi_2 \rrbracket &:= \llbracket \phi_1 \rrbracket \cap \llbracket \phi_2 \rrbracket \\ \llbracket \langle a \rangle \phi \rrbracket &:= \{ \pi \in \text{runs}(S) \mid \text{firstact}(\pi) = a \vee \pi^1 \in \llbracket \phi \rrbracket \} \\ \llbracket \phi_1 \mathcal{U} \phi_2 \rrbracket &:= \{ \pi \in \text{runs}(S) \mid \exists i \pi^i \in \llbracket \phi_2 \rrbracket \wedge \forall j < i \pi^j \in \llbracket \phi_1 \rrbracket \} \\ \llbracket \phi_1 w\mathcal{U} \phi_2 \rrbracket &:= \{ \pi \in \text{runs}(S) \mid (\exists i \pi^i \in \llbracket \phi_2 \rrbracket \wedge \forall j < i \pi^j \in \llbracket \phi_1 \rrbracket) \vee \forall i \pi^i \in \llbracket \phi_1 \rrbracket \} \end{aligned}$$

Intuitively, $\llbracket \langle a \rangle \phi \rrbracket$ means that the next action enabled in one of the runs is a . $\llbracket \phi_1 \mathcal{U} \phi_2 \rrbracket$, read as ϕ_1 ‘Until’ ϕ_2 means that the path has a suffix that satisfies ϕ_2 and all prefixes satisfy ϕ_1 . $\llbracket \phi_1 w\mathcal{U} \phi_2 \rrbracket$, or ϕ_1 ‘weak-Until’ ϕ_2 represents the property as long as ϕ_1 is false, ϕ_2 must be true.

2.2.3 Model Checking

Model checking is a method to algorithmically verify formal systems. This is achieved by verifying whether the system model satisfies a formal specification. The

system model is usually defined as an automaton (or network of automata). The specification is written terms of temporal logic formulae (LTL).

Temporal logics play an important role in formal verification. They are used to specify properties of processes. Such properties are for example “The process is deadlock-free. (A deadlocked state is not reachable)” or “In every infinite execution of a process action b must occur infinitely often”.

Formally, the problem can be stated as follows: given a desired property, expressed as a temporal logic formula ϕ , and an automata-based model M with current state s , model checking evaluates whether $M, s \models \phi$.

There have been several case studies of model checking for safety-critical software. Sreemani and Atlee [SA96] used SMV to analyze the A-7E aircraft software requirements. Bharadwaj and Heitmeyer [BH99] continued this line of work, using SMV and Spin for other SCR requirements. From an informal specification of a hydroelectric power plant, Pugliese and Tronci [PT96] developed a process-algebra specification, which was then verified with an in-house BDD-based model checker. Crow and Di Vito [CV98] verified invariants of the requirements for a software subsystem on the Space Shuttle of NASA with the explicit model checker $Mur'\phi$ [Dil96]. Using Spin, Havelund et al. [HLP01] found errors in a space craft controller, while Schneider et al. [SECH98] validated a model of a fault-tolerant system specified as a hierarchical state machine.

An important difference between this thesis and existing work on model checking is that while previous work has been concentrated on exhaustive verification of particular (full-fledged) systems, we use model checking to ensure correctness of usage models, which are used as a framework for larger, more varied system implementations.

2.2.4 Model-based Testing

Model-based testing [FvBK⁺91, DJK⁺99] refers to software testing where test cases are derived in whole or in part from a model that describes some (usually functional) aspects of the system under test.

The test cases derived from the formal model are functional tests on the same level of abstraction as the model. These test cases are collectively known as the abstract test suite. The abstract test suite cannot be directly executed against the system under test because it is on the wrong level of abstraction. Therefore an executable test suite is derived from the abstract test suite that can communicate with the system under test. This is done by mapping the abstract test cases to concrete test cases suitable for execution. An advantage of this mapping is that the same set of abstract test cases can be used to generate multiple (concrete) executable test suites for different implementations. A representation of the model-based testing workflow is shown in figure 2.2.

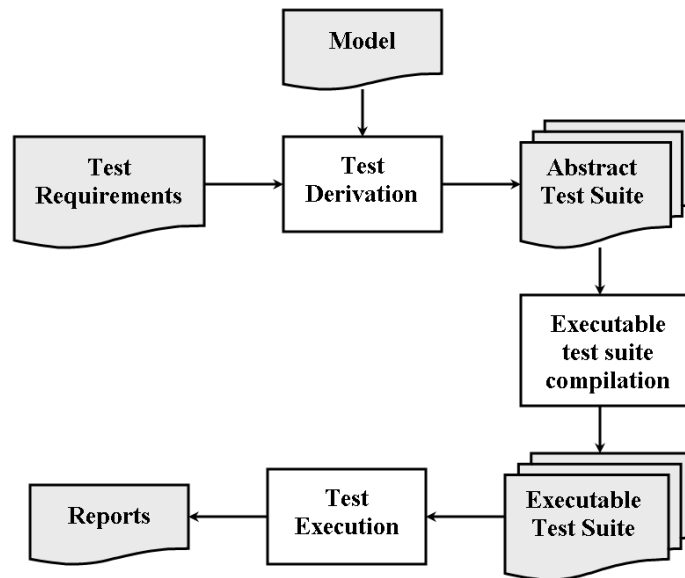


Figure 2.2: A model-based testing workflow

Because test suites are derived from models and not from source code, model-based testing is usually seen as one form of black-box testing. In some aspects, this is not completely accurate. Model-based testing can be combined with source-code level test coverage measurement, and functional models can be based on existing source code in the first place.

There have been several efforts involving the use of formal methods to generate test

cases for safety-critical systems. One approach to automated test generation is based on the ability of model checkers to generate counterexamples to invalid assertions: roughly speaking, to generate a test case that will exercise a behavior characterized by a predicate p , the property ‘always not p ’ is model checked and the counterexample to this property provides the required test case (if there is no counterexample, then the property is true and the proposed test case is infeasible).

Callahan, Schneider and Easterbrook in [CSE96] present an approach to verify the execution traces created by monitoring statements during white-box testing using a model-checker as a semantic tableau. Multiple execution traces are organized into equivalence partitions based on requirement specifications written in LTL, and counter-examples generated by the model-checker are used to produce new test cases for unpopulated equivalence partitions.

Beyer et al. [BCH⁺04] similarly present a methodology to extend the BLAST model checker [HJMS03] to determine program locations where a certain predicate p is true, and automatically generate a sequence of vectors (or test cases) that reflect the truth value of p for the concerned program locations.

Specification-based Regression Testing [XDR04b, XDR04a] targets on ways to use specifications to track these changes and improve regression testing. The approach uses model checking to reason between two versions of software specifications and generate safe and minimal regression tests for the updated software specifications.

AsmL, a popular language for model-based testing has been developed at the Foundations of Software Engineering (FSE) group at Microsoft Research [GGSV02]. AsmL builds on the concept of abstract state machine [Gur95] and provides a modern specification environment that is object-oriented and component based. Typically, the models used are FSMs, which are derived from an abstract state machine (ASM) specification. This allows for integration of ASM specs with existing tools for test case generation from FSMs.

The approach suggested in this paper does not make use of model-checking to generate test cases per se, but requires exhaustive verification as a pre-requisite to the test case generation process, making the process independent of a specific formal analysis tool or platform.

2.3 Program Slicing

While test cases derived from formal models help during premarket testing, debugging and maintenance activities rely upon static analysis techniques. The most popular of these methods is program slicing [Wei81] — a technique to facilitate understanding and debugging of programs by focusing on selected aspects of semantics. Program slicing uses a user-defined criterion to extract statements from a program. Any statement that does not satisfy this slicing criterion is removed from the original program. The resultant reduced code fragment constitutes the slice for the program for the given criterion.

2.3.1 Static Slicing

In his initial work on slicing, Weiser defines the slicing criterion as a program point (statement) and set of variables. This form of slicing has since come to be known as static slicing, because the slicing criterion contains no information about how the program is to be executed.

A static slice for a program p is constructed with respect to a tuple, (p, x, V) , where x is a point in the program (typically a node of the Control Flow Graph for p), and V is a set of variables belonging to p . A statement may be removed from p to form a static slice, s of p , iff it cannot affect the value of any variable in V when the next statement to be executed is at point x .

As an example, consider the C program given in Figure 2.3 (the example is taken from [FDHH04] and lists part of the code for the 1999-2000 UK taxation system). The slice for this program with respect to the criterion $(p, 32, \{\text{income}\})$ is shown as highlighted code in the figure. As can be seen from the slice, all statements (more specifically, program points) that do not impact the value of the variable `income` at line 32 are excluded from the slice. This type of slice is referred to as a ‘backward slice’, since it includes all program points that may influence whether control reaches x , and all points that may influence the values of the variables used at x when control gets there. Alternatively, a ‘forward slice’ from program point x includes all points

affected by the computation or conditional test at x.

```

1: #include <stdio.h>
2: int personal;
3: char code;
4: main () {
5:     int tax, income, age, pc10;
6:     int married, blind;
7:     scanf ("%d, %d", &age, &income);
8:     scanf ("%d, %d", &blind, &married);
9:
10:    pc10 = 1500;
11:    if (income<=personal) tax=0;
12:    else {
13:        income = income-personal;
14:        if (income<=pc10) tax = income/10;
15:        else {
16:            tax = pc10/10;
17:            income = income-pc10;
18:            if (income<=28000)
19:                tax = tax+(income*23)/100;
20:            else {
21:                tax = tax+(28000*23)/100;
22:                income = income-28000;
23:                tax = tax+(income*40)/100;
24:            }
25:        }
26:    }
27:    if (!blind && !married && age<65)
28:        code = 'L';
29:    else if (!blind && age<65 && married)
30:        code = 'H';
31:    else code = 'T';
32:    printf("The income is: %d\n",income);
33:    printf("The tax is: %d\n",tax);
34: }

```

Figure 2.3: Sample C code for a taxation program and slice w.r.t. line 32

While a backward slice includes program points all the way back to the beginning of execution, and a forward slice includes points all the way forward to the end of execution, a ‘chop’ introduced in [JR94] is used to show the influence of one set of program points (the chop-sources) on another (the chop-targets). A coarser (but more efficient) chop, termed a ‘fast chop’, can be computed as the intersection of a backward slice and a forward slice.

2.3.2 Dynamic Slicing

Static slicing is a useful abstraction technique for understanding program behavior. However, when dealing with large-scale systems, it can often result in rather large and unmanageable slices. This is particularly true for well-constructed programs, which are typically highly cohesive, resulting in programs where the computation of the value of each variable is highly dependent upon the values of many other variables.

As an alternative, Korel and Laski suggest the dynamic approach to program slicing [KL88]. A dynamic slice is the part of the program that affects the computation of a variable (or set of variables) of interest during program execution on a specific input. The criterion for dynamic slicing is obtained simply by augmenting the static slice criterion with the input to the program. The slice is computed by evaluating the program for the given input and limiting the slice to only the statements that would be executed for this input. In this respect it differs from static slicing, which would include all possible executions for the specified criterion, not just for a particular input.

A dynamic slice for a program p can be defined with respect to a tuple (p, I, x, V) , where p , x and V are as defined for the static slice, and I is the input to p for which the slice is to be determined. All statements of p that are not visited during the execution on input I and those that do not affect the value of variables in V when the next statement to be executed is at point x are excluded from the slice s , of p .

Figure 2.4 shows the dynamic slice for the program given in Figure 2.3, for the criterion $(p, I_0, 33, \{\text{tax}\})$ where I_0 is the input $\{\text{income} = 7200, \text{age} = 35\}$ for the given program.

2.3.3 Conditioned Slicing

Dynamic slicing, by virtue of being input-specific, is much better suited as a debugging tool. However, it cannot be readily used for applications like program comprehension and visualization, where the program execution remains unspecified. For these applications, a more static approach is required. On the other extreme,

```

1:  #include <stdio.h>
2:  int personal=5980;
4:  main () {
5:      int tax, income, pc10;

10:     pc10 = 1500;
13:     income = income - personal;
14:     tax = income/10;
33:     printf ("The tax to be paid is : %d\n", tax);
34: }

```

Figure 2.4: Dynamic slice for input {income = 7200, age = 35}

using pure static slicing may not be very helpful in this situation either, since it can result in inordinately large slices. Conditioned slicing, introduced by Canfora et al. [CCL98] bridges this gap between static and dynamic slicing. The conditioned approach evaluates the slice with respect to a given condition along with the slicing criterion used for static slicing. The resultant slice is (usually) considerably smaller than a static slice, but not as localized as a dynamic slice.

A Conditioned slice is constructed with respect to a tuple, (p, x, V, π) , where x is a point in the program, V is a set of variables, and π is some condition on the value of variables defined in p . A statement may be removed from a program p to form a slice s of p , iff it cannot affect the value of any variable in V when the next statement to be executed is at point x and the initial state is such that the condition π is satisfied (holds true) at point x of program p .

Figure 2.5 shows the conditioned slice for the program of Figure 1 for the criterion $(p, 33, \text{tax}, \text{income} \geq 65000)$.

2.3.4 Amorphous Slicing

The slicing techniques discussed thus far have all been syntax-preserving slices, i.e. they merely remove non-relevant statements from the program without altering its (syntactic) structure. Amorphous slicing, introduced by Harman and Danicic [HD97], on the other hand, uses semantic preserving transformations to simplify the slice.

```

1:  #include <stdio.h>
2:  int personal=5980;
4:  main () {
5:      int tax, income, pc10;

10:     pc10 = 1500;
13:     income = income - personal;
16:     tax = pc10/10;
17:     income = income-pc10;
21:     tax = tax+(28000*23)/100;
22:     income = income-28000;
23:     tax = tax+(income*40)/100;
33:     printf ("The tax to be paid is : %d\n", tax);
34: }

```

Figure 2.5: Conditioned slice for the condition $\text{income} \geq 65000$

This syntactic freedom allows amorphous slicing to perform greater simplification with the result that amorphous slices are often smaller than their syntax-preserving counterparts.

The key to amorphous slicing is the transformation that is used to simplify the program. An amorphous slice can be constructed using any program transformation which simplifies the program and which preserves the effect of the original program with respect to the slicing criterion.

Amorphous slicing can be defined with respect to the transformation used to simplify the program. Let α be a transformation relation on a program that produces p^α as the simplified program when applied to a program p . Thus, an amorphous slice can be defined w.r.t. a tuple (p^α, x', V') , where x' is a point in the simplified program p^α , corresponding to a program point x in p , and V' is a set of variables in p^α corresponding to the variable set V in p . A statement may be removed from p^α to form an amorphous slice s , iff it cannot affect the value of any variable in V' when the next statement to be executed is at point x' .

The approach to abstraction-driven slicing presented in this thesis is similar in many ways to amorphous slicing. The abstract interpretation acts as a transformation relation to generate an abstract model, which is then used to yield further slices. Just like amorphous slicing, the abstraction-driven slicing approach allows for pro-

gram transformation as well as statement deletion. The difference though is that in abstraction, the program is not changed syntactically; instead an abstract model is constructed and the program statements are mapped onto the states in the model (each state in the model corresponds to one statement in the program). This allows for a better understanding of the program, while at the same time providing the simplicity of static, syntax-preserving slicing.

2.4 Model Abstraction

Model checking is based on the abstract-check-refine paradigm: build an abstract model of the software system, then check the desired property based on the system specifications, and if the check fails, refine the model and start over. The most critical (also the most time-consuming) task in this approach is defining a model that sufficiently and correctly captures the features of the system under consideration. Since it is developed by the analyst, it is quite possible for the model itself to be erroneous and incomplete. Such errors, if present, could be very hard to detect and could easily undermine the effort spent in analysis. To avoid (or in any case, reduce) such errors, and to expedite the model building process, several methods have been proposed that automatically extract (or abstract) the model from the code. Figure 2.6 shows the general scheme for model abstraction. The abstraction process uses a set of predefined abstract interpretations (AIs) to convert a given program to an abstract model based on the (abstraction) criteria defined. Abstract interpretation is a framework for executing a program, using values from an abstract domain in place of the original concrete values. A formal definition of abstract interpretation can be found in [CC77]. The model generated by the abstraction process, usually represented as a labeled transition system (LTS), can be used to verify a temporal property ϕ and ascertain the validity of the program, or generate an error trace as a counter-example.

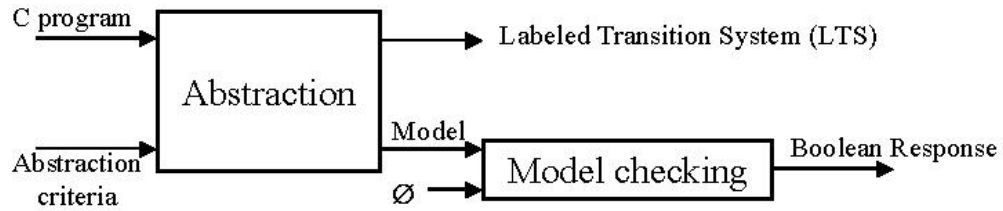


Figure 2.6: The model abstraction process

2.4.1 Model Abstraction Techniques

There are a number of techniques and tools used for abstracting models from source code. These techniques differ mainly in the connotation assigned to the derived model. Based on the nature of the model abstracted, these techniques can be classified into several categories, as described below.

2.4.1.1 Data Abstraction

Data abstraction uses an abstract interpretation to map constructs in the concrete world (whose semantics are defined by the C language definition) to an abstract world, defined as a lattice of symbols. These worlds are connected by two functions. The abstraction function, α , maps from a set of concrete elements to an abstract element. The concretization function, γ , maps from an abstract element to a set of concrete elements. Furthermore, each domain has the same set of operators, so that we can perform exactly the same computations in either the concrete or abstract domain.

C Wolf [DuV02, DI02] is an example of a data abstraction tool, developed at NC State University. It uses abstract interpretations to automatically generate data-abstracted finite models of C programs. The abstractions are defined for the variables in the original program and map to a smaller set of abstract values in the corresponding model. The output of the model generator is either a graph with labeled states and edges for viewing by the user, or a labeled transition system (LTS) in a format readable by the Concurrency Workbench of the New Century (CWB-NC). We discuss data abstraction using C Wolf in more detail in the next section.

Bandera [CDH⁺00] is another tool that makes use of user-defined data abstractions to generate models from concurrent Java programs. Bandera has a generic scheme for users to express abstractions, and then apply a particular abstraction to a specific program. Bandera also employs slicing techniques to remove parts of the program that are irrelevant to establishing properties of interest (the properties themselves are expressed in linear temporal logic). Bandera can generate models for several reasoning systems, including Spin [Hol91] and SMV [McM92, CMCHG96].

2.4.1.2 Predicate Abstraction

The predicate abstraction approach collectively abstracts all program variables by replacing them with a set of boolean variables. The SLAM project [BR02] automatically generates and refines sets of predicates based upon a property (expressed as an automaton) that the user wishes to check. Given a set of chosen predicates (or their negations) that definitely hold at that point. Those program points where such a characterization is not possible lead to non-deterministic choice points. The result is a non-deterministic schema over boolean variables, one for each predicate of interest, which can be represented as a binary decision diagram (BDD). The SLAM project has several other tools to analyze/model-check these boolean programs. SLAM has been used extensively to identify bugs in device drivers for Windows NT [BR01].

2.4.1.3 Metacompilation

In this approach, the compiler for a programming language is extended with a user-customizable language, or table, which can be used to specify the model to be generated in the input language of a model checker. Two tools that belong in this category are FeaVer [Hol00, HS01] and xgcc [LCED01]. In both these cases, it is up to the user to generate the models that are believed to be an appropriate abstraction of a program. FeaVer generates Promela models that can be verified using Spin. xgcc is used together with the Mur ϕ model checker [DDHY92, Dil96]. Mur ϕ is based on a Pascal-like imperative language, similar to that of Spin.

2.4.1.4 Specification-less approach

In this approach, specific properties of concurrent programs, such as deadlocks, are targeted. To show these specific properties, slicing is typically used. This approach has been used in Java PathFinder [VHBP00] and JCAT/YAV [Ios00]. The JavaPathFinder (JPF) is a model checker for Java programs designed for memory efficiency, and to support backwards execution. The only properties that JPF can check are deadlocks and invariant conditions.

YAV and its predecessor JCAT are both Java model checking tools developed at Politecnico di Torino by Iosif, Demartini, and Sisto. YAV accepts a Java program as input and generates as output a program in an extension of the Promela language. This extended Promela code is processed by dSpin, an enhancement of Spin with support for the modeling of dynamic memory operations, pointers-to-functions, and a function call-stack. YAV relies on the state-space caching algorithm of Spin to perform incomplete model checking of the resulting Promela program, which is still useful for detecting many bugs.

2.4.1.3 Runtime monitoring

This approach does not use abstraction at all. It is instead based in instrumenting a program, for a specific property of interest, with additional code. The instrumented code, when run, monitors the program execution such that if an erroneous state is reached, then appropriate action, such as stopping gracefully or logging data, can be carried out. This approach has been used in Verisoft [God97], a state-space exploration tool for C programs.

2.5 Post-mortem Forensic Analysis

2.5.1 Comparing Post-mortem Analysis Techniques

Traditionally, static slicing techniques have been used as an aid to program comprehension during post-mortem forensic analysis. However, the slices obtained through static slicing can often be too large and may not give a clear understanding of the code when dealing with large-scale, complex systems. Several alternative techniques have been used, as a result, to aid forensic analysis.

Programmers often make use of static syntax checkers like Lint [Joh78] and Polyspace [Deu03] to analyze and debug their programs. These tools though are, by their nature, limited to checking for syntactic anomalies and are not much help to forensic analysts with no prior knowledge of the software. Type checking tools such as CQual [FFA99, ZEJ02] and Anno Domini [EHM⁺99] extend the notion of syntax checking by performing qualifier inference to check for conformance to system properties. These methods are however, limited to checking for type equivalence only.

Dynamic slicing is a useful technique for debugging, as it limits the scope of the slice to a specific execution trace. However it does not lend itself readily to post-mortem analysis due to the infeasibility of execution backtracking. Conditioned slicing solves this problem by evaluating the slice with respect to a given condition along with the criterion used for static slicing. The resultant slice is (usually) considerably smaller than a static slice, but not as localized as a dynamic slice. Conditioned slicing, thus, does help an analyst by reducing the size of successive slices, but does not contribute greatly to the forming of accurate hypotheses during the error detection process. This owes to the fact that conditioned slicing, like static slicing is a syntax-preserving technique, and contributes to program comprehension in more or less the same way.

Another popular technique for post-mortem analysis is systematic testing, as used in Verisoft [God97]. Systematic testing explores the state space of a system, searching for coordination problems (deadlocks, etc.) and assertion violations in a software system by generating, controlling, and observing the possible executions and inter-

actions of all its components. However, it requires a thorough understanding of the system design to be used as an effective forensic tool.

Model abstraction extracts an abstract model of the program based on data and/or predicates, providing the analyst with a bottom-up view of the program. Strategic programming [LVV04] uses generic data-processing actions to traverse through program structures and provide a similar bottom-up understanding of the code. Abstraction though provides an added advantage of producing verifiable models that can be checked using model-checking for conformance to safety requirements.

Our approach of abstraction-driven slicing is similar in many ways to amorphous slicing. The abstract interpretation acts as a transformation relation to generate an abstract model, which is then used to yield further slices. Just like amorphous slicing, the abstraction-driven slicing approach allows for program transformation as well as statement deletion. The difference though is that in abstraction, the program is not changed syntactically; instead an abstract model is constructed and the program statements are mapped onto the states in the model (each state in the model corresponds to one statement in the program). This allows for a better understanding of the program, while at the same time providing the simplicity of static, syntax-preserving slicing.

Table 2.1 lists the various techniques for post-mortem analysis and compares them against the abstraction-driven slicing approach. The following criteria are used to assess the tools:

1. Compilable code required: Does the code under review need to be compilable?
2. Executable code required: Does the code need to be executable?
3. False positives: The number of errors reported that do not actually correspond to the software malfunction.
4. Completeness: Can all errors in the code be successfully detected?
5. Iterations needed to debug: The number of iterations that are required to successfully trace an error.

Table 2.1: Comparison of Post-mortem analysis techniques

	Compilable code required	Executable code required	False positives	Completeness	Iterations needed to debug	Size of slice	Preserves syntax	Level of program understanding needed
Static Slicing	Yes	No	Many	No	Many	Large	Yes	Low
Syntax Checkers (Lint, Polyspace)	Yes	No	Many	No	NA	NA	Yes	Low
Dynamic Slicing	Yes	Yes	Few	No	Few	Small	Yes	Low
Model Checking (Abstraction)	Yes	No	Many	Yes	Few	NA	No	High
Systematic Testing (Verisofi)	Yes	Yes	Few	Yes	Few	NA	Yes	High
Type Checking (CQual, Anno Domini)	Yes	No	Many	Yes	Many	NA	Yes	Low
Strategic Programming	Yes	No	Many	No	Few	NA	Yes	High
Amorphous Slicing	Yes	No	Many	No	Many	Small	No	Low
Abstraction-based Slicing	Yes	No	Many	Yes	Few	Small	No	Low

6. Size of slice: The average size of a slice generated.

7. Preserves syntax: Is the original syntax of the program preserved?

8. Level of program understanding required: The level of (a priori) understanding of the code required to apply this technique.

As can be seen from the table, the abstraction-driven slicing approach is well-suited for post-mortem analysis, as it requires comparatively fewer iterations, generates smaller slices, and does not necessitate a high level of program understanding.

2.5.2 Corrective Maintenance vs. Post-mortem Analysis

Error detection during forensic analysis shares much in common with corrective maintenance. Maintenance engineers, like forensic analysts, often make use of post-mortem analysis to track the cause of reported bugs to their exact location in the source code. PSE (Post-mortem Symbolic Execution) [MSA⁺04a] is a static analysis technique developed at Microsoft Research that addresses tracing of software failures during maintenance by producing execution traces that show how a program is driven to a given failure. In particular, PSE handles a particular class of software defects, viz., typestate errors. PSE performs a backwards value flow analysis starting from the point of failure, transferring the blame for the error along the execution trace until it reaches a point where the value was created, indicated by a failing execution trace, or reaches a contradiction, in which case the particular trace can be ruled out.

Another methodology for assisting source code analysis by improving the analyst's understanding of the software is provided by ARTISAn [JEM05]. ARTISAn is a static analysis framework for Java similar to CAdS, in that it aids program analysis by enhancing the analyst's understanding of the software. The system comprehension is provided via a combination of low-level purpose views and high-level usage views. The purpose view is derived by classifying and labeling program constructs, while the usage view is obtained through a def-use analysis to identify regions of elements based on invocation and inheritance relationships. In tandem, these views provide the user with a better understanding of the system, and an opportunity to faster locate the parts that are of particular interest. The principle remains the same in CAdS, only we replace the concept of views with the top-down and bottom-up comprehension paradigms respectively.

2.6 Formal Methods Based Tools

A number of different software tools were used to assist in the premarket reviews and forensic analysis for the various case studies. A brief introduction to these tools is given in this section.

Matlab [Mol80] is a commercial software package developed by Mathworks for numeric computation, technical graphics and visualization, and an intuitive programming language for applications in engineering and science. Matlab provides a family of add-on application-specific solutions called toolboxes that extend the MATLAB environment to solve particular classes of problems. **Simulink** is one such toolbox that provides an interactive system for the nonlinear simulation of dynamic systems. Simulink is a graphical program that allows systems to be modeled as real-time or discrete systems. It is often used in conjunction with **Stateflow**, another toolbox used to visually model and simulate complex reactive systems based on finite state machine theory. Stateflow is an interactive design and simulation tool for event-driven systems that provides language elements to describe complex logic in state transition and flow diagram notations.

Reactis [rea03] is an embedded-software design automation (ESDA) tool suite developed by Reactive Systems. It is based on the Simulink/Stateflow modeling paradigm, and can be used to test, simulate and validate Matlab models.

Reactis consists of three main components: a Tester, a Simulator, and a Validator. The Reactis Tester automatically generates test suites from Simulink/Stateflow models. The test suites provide comprehensive coverage of different test-quality criteria, and while at the same time minimizing redundancy in tests. The Reactis Simulator enables users to visualize model execution in a manner similar to those of traditional debuggers. The Reactis Validator performs automated searches of models for violations of user-specified requirements, returning a test case as a counter-example in case a violation is detected.

Uppaal [BLL⁺95] is a tool suite for modeling and verification of real-time systems, based on constraint-solving, and on-the-fly techniques, developed jointly by the Design and Analysis of Real-Time Systems group at Uppsala University, Sweden and Basic Research in Computer Science at Aalborg University, Denmark. Uppaal provides a graphical interface to define system descriptions as networks of timed automata and uses Linear Temporal Logic (LTL) formulae to verify the specified model.

Spin is a generic verification system that supports the design and verification of asynchronous process systems [Hol91, Hol97]. Spin verification models are focused on proving the correctness of process interactions, and they attempt to abstract as much as possible from internal sequential computations. Process interactions can be specified in Spin with rendezvous primitives, with asynchronous message passing through buffered channels, through access to shared variables, or with any combination of these.

Spin accepts design specifications written in the verification language Promela (a Process Meta Language), and it accepts correctness claims specified in the syntax of standard Linear Temporal Logic. The description of a concurrent system in Promela consists of one or more user-defined process templates, or **proctype** definitions, and at least one process instantiation.

Codesurfer is a code analysis and inspection tool developed by GrammaTech Inc[ARTZ03] that provides extensive support for program slicing as a means to nav-

igate through code. The tool uses a System Dependence Graph (SDG) to internally represent the program being analyzed, with each node in the SDG representing a fragment of code, or a program point. Slicing is implemented in Codesurfer as queries over a particular SDG. These queries can be used to evaluate forward slices, backward slices, and chops (a chop shows the influence of one set of program points on another). The query module itself can be configured to operate in several different modes. Based on the mode selected, queries can be derived with respect to program points, variables or functions.

CWolf [DuV02, DI02] is a tool developed at NC State University for automatically generating data-abstracted finite models of C programs. The front end for CWolf consists of a compiler, which reduces a C program to an intermediate representation, known as the CWolf Intermediate (CWI) format. The back-end is a model generator component, which is built around an abstract virtual machine used to execute abstracted C programs. The output of the model generator is either a graph with labeled states and edges for viewing by the user or in a format readable by the Concurrency Workbench of the New Century (CWB-NC). Abstraction of C programs using CWolf is explained in detail in Chapter 4.

The **Concurrency Workbench of the New Century (CWB-NC)** is a verification tool that provides the user with a facility to verify automata-based models using model checking and semantic equivalence techniques. The model can be defined in a number of system-design notations, including CCS, CSP and LOTOS, and verified against μ -calculus or GCTL* properties[CS96]. Similarly, different equivalences can be used to assess the behavior of the model, including preorder checking and bisimulation equivalences [Mil89].

Chapter 3

The Premarket Review Process

Most regulatory agencies enforce at least some level of premarket conformance to certify whether the submitted software is safe or not. However, most of this review is focused on the development process and does not have the means to actually verify the software itself.

While it is impossible for a third party agency to exhaustively verify the code, the usage model based review strategy could serve as a viable alternative to assessing the safety of the software. The review process, introduced in [Section 1.2](#), involves the following major activities:

1. defining a usage model for a software family,
2. verifying the usage model to ensure it meets requirements, and
3. deriving automated test case sequences and using these to validate individual software implementations. This validation can be performed by the regulatory agency, or by the manufacturers themselves, who would need to provide proof of conformance during premarket certification.

The review process, detailed in this chapter, is best explained with the help of examples. We use our experiences with medical device software as a means to illustrate

the process. In particular, we use two separate systems — the Generic infusion pump (GIP) and the computer-aided resuscitation algorithm (CARA) as running examples throughout this chapter.

The **Generic infusion pump or GIP** is a generalized version of an infusion pump, a safety-critical medical device, being developed at the CDRH/OSEL [JJ06]. **CARA** is a closed-loop software system developed at the Walter Reed Army Institute of Research (WRAIR) that is used to drive a high output infusion pump used for fluid resuscitation of patients suffering from conditions that lead to hypotension due to loss of blood.

Individual activities of the premarket conformance review process are discussed in the different sections of chapter. Section 3.1 gives a general overview of the premarket review process from the regulators’ perspective. Section 3.2 explains usage models and their architectures. Verification strategies for usage models are discussed in Section 3.3. Section 3.4 explains how usage scenarios can be extracted with the help of these models. Lastly, the two case studies, GIP and CARA, are discussed in Sections 3.5 and 3.6 respectively. Results of evaluating the two studies against extracted usage scenarios are listed in Chapter 6.

3.1 Overview of the Premarket Review Process

The scheme for premarket review is given by Algorithm 1. The algorithm takes as input requirements specifications for a group of related software, or software family, along with a set of minimum common safety properties.

A *safety property*, typically expressed as a functional requirement, is used to ensure that the software does not result in a hazardous situation. Specifically, it is used to ensure that the usage model does not lead to an illegal (error) state. For example, a safety property for an infusion pump software could be defined as ‘*If the patient’s blood pressure falls below 90mm Hg within 10 seconds of stopping infusion, a level 2 alarm must be triggered by the pump*’. To enforce that this property holds for the pump software, it is expressed in a formal notation and verified against the usage

Algorithm 1 Premarket Conformance Review

Input: A software family S characterized by system requirements R , a set of safety properties P for S , a set of software implementations I based on R .

Output: boolean conform, indicating whether $i \in I$ conforms to safety properties given by P .

Define a usage model M based on the requirements R .

Verify usage model M using formal verification techniques (i.e., model checking and/or targetted simulation) to check whether $M \models P$.

if $M \not\models P$ **then**

 Revise usage model M . Go to Step 1.

end if

Derive set of usage scenarios T_s from usage model M .

for all i in I **do**

 Validate i against derived test suite T_s (i.e., check whether $i \models T_s$).

if $i \models T_s$ **then**

 conform $\leftarrow$ true.

else if $i \not\models T_s$ **then**

 conform $\leftarrow$ false.

end if

end for

model for the software.

The usage model is defined in terms of a finite state machine (or Kripke structure). Using this formal representation allows the model to be verified against the safety properties, using exhaustive state exploration techniques. The model itself comprises of a set of states corresponding to the various configurations the software can be in, and transitions for each of the states, corresponding to the events possible for a particular configuration. An event may be either a human input (such as a user selection), or an internal system event (such as an interrupt or exception).

Each event can further be associated with a probability value, indicating the probability of occurrence for that event with regard to the current state. The probability of occurrence for each event can range from a minimum value of 0 (indicating an impossible scenario) to a maximum of 1 (indicating that the event is always possible). For each state, an event is chosen non-deterministically (with a higher probability event having a better chance of being selected). Upon execution of the chosen event, the model transitions to a new state, determined by a transition relation δ defined as

$$\delta : Q \times E \times [0, 1] \times Q$$

where, Q is the set of all states in the model, E is the set of events for the system, and $[0, 1]$ is the range of reals between 0 and 1 defining the probability of occurrence for each event at a given state.

Once the usage model is defined, it is exhaustively verified against the set of safety properties. A number of verification techniques can be used to this end, e.g., symbolic execution [BW94], theorem proving [ORS92]. In this thesis, we propose two (popular) approaches to formal verification — targeted simulation [Het88] and model checking [Mer01].

Targeted simulation is a verification technique that allows users to instrument models with checks for assessing model behavior and targets for guiding the generation of test data. The simulation can be used to automate requirements-driven testing and to verify model behavior. Model checking is a technique for verifying finite state concurrent systems through an exhaustive search of the finite state space of the system to determine if a safety property is true or not. If the system fails to satisfy a desired property, the model checker produces a counterexample that demonstrates a wrong behaviour. This faulty trace provides insight to understanding the real reason for the failure as well as important clues for fixing the problem.

If during verification, a safety property is not satisfied, it corresponds to an error in the usage model. The counterexample, if produced, is used to establish the cause of the error. The error may be a problem in the usage model itself, or an anomaly in the system requirements. In either case, the model is revised and subsequently re-verified. This process continues till all safety properties are sufficiently satisfied by the usage model.

Next, the verified model is used to derive a set of usage scenarios against which specific software implementations are validated. The usage scenarios are derived by navigating through the execution paths of the usage model. For each state (or

transition) encountered during the traversal, a sequence of test statements is added to the usage scenario. The set of these usage scenarios corresponds to the test suite for the implementation.

Once test suites are generated, they are used to validate software implementations based on the particular usage model. The validation is performed in a black-box fashion, allowing regulators to test software without detailed knowledge of the code. The errors detected during validation are noted and reported to manufacturers. Alternately, the manufacturers can be provided with the test suites for relevant products and mandated to validate their software against these suites before submitting it for review.

3.2 Defining a Usage Model

The usage model is at the heart of the premarket review process. Effectiveness of the conformance review depends to a large extent, on the accuracy of the model and its ability to correctly capture critical features of the software design. From a safety point of view, there are two criteria that the usage model must meet — on the one hand, it should be general enough to encapsulate the major features of the software across implementations. On the other hand, the model should be specific enough to educe characteristics of individual implementations. The right balance between generalization and specificity for a usage model depends on particular software and level of safety required for it. A number of different architecture patterns can be used for defining the structure of a usage model [BMR⁺96]. These architecture patterns can be used as templates for defining formal usage models and allow for a structured design to enable efficient usage scenario derivation. For the examples we deal with in this thesis, we define and use schemes for layered and observer-based architectures. We briefly explain the two here.

3.2.1 Layered Architecture

The layered architecture provides a decomposition of features such that most interactions occur only between neighboring layers. The layers are arranged according to their level of generality, with application program interfaces (APIs) provided to facilitate communication between layers. The most generic (or generalized) part of the model is at the lowest layer. Each successive layer sacrifices generality while expanding on specific features of the software. Figure 3.1 shows the schema for the layered usage model architecture.

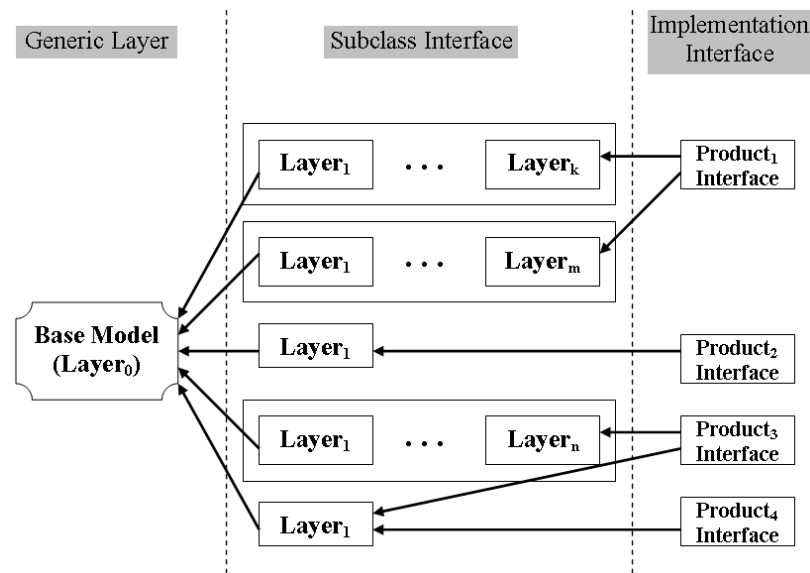


Figure 3.1: A layered architecture for Usage models

The lowest (leftmost) layer in the figure consists of the generic layer and corresponds to safety features common to all products in the software family. Subclass interfaces extend the generic layer to incorporate characteristics of a particular type, or subclass, of systems. As an example, the generic layer could define safety properties for software used in all cardiac defibrillators, while the subclass interface layer would extend it to describe features for internal defibrillators, automated external defibrillators (AEDs), implantable cardiac defibrillators etc. The subclass interface itself could consist of further sublayers if required (e.g., monophasic or biphasic AEDs).

The outermost layer, implementation interface, deals with individual products or implementations. This layer provides a mapping between the abstract functionality defined in the model and its manifestation in the actual product.

The layered architecture is the most common architecture for usage models, as it allows for subclassing of safety features and provides enhanced maintainability and extensibility across different layers of the model.

We use this architecture to model the GIP system, with the base layer describing features of the generic pump, and outer layers delineating aspects of specific implementations.

3.2.2 Observer-based Architecture

The essence of the Observer-based model architecture is that one or more objects (called observers or listeners) are used to observe an event on an observed object (itself an active component in the model). The relationship between the observed object and the observers is usually one-to-one, but in some cases an observer may also listen to events across several objects.

When an event is raised, each observer takes a ‘snapshot’ of the system at that instant and correspondingly executes a set of actions to respond to the event (e.g., generating a test case sequence). Typically, the observed objects are independent components (or subsystems) loosely-coupled with each other. Figure 3.2 shows the general scheme for such an observer-based model architecture. The components in the model are independent processes (or automata) that communicate with each other through synchronous or asynchronous messages. The observers are used to capture events on these components.

We use the observer-based architecture to define the usage model for the CARA system. The model is defined as a network of timed automata, with each automaton corresponding to a subsystem in the model.

Irrespective of the architecture used for modeling, all usage models make use of formal methods based techniques to capture software specifications and safety conditions. The major advantage of using a formal model is that it can be rigorously

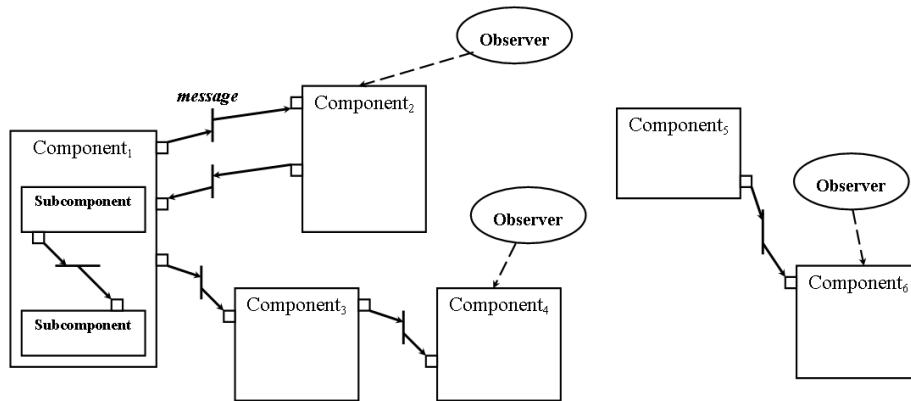


Figure 3.2: Example of an observer-based architecture for usage models

verified using model checking and exhaustive state explorations to ensure its correctness and adherence to the safety requirements. The basic characteristics of the model, whether it be the core of the layered architecture or individual sub-components in the subsystem architecture, are thus defined in terms of formal models.

3.3 Verifying Usage Models

The usage model, defined formally, can consist of thousands of different states, with several thousand transitions, and a number of associated events and constraints influencing its behavior. To use this model as a basis for premarket review, it is imperative to ascertain that the model satisfies all system requirements.

Verifying the model against requirements essentially corresponds to checking for two types of properties:

1. Liveness properties, used to ensure that the model does satisfy the desired properties. In terms of LTL, a liveness property is represented as $F\phi := \text{true} \mathcal{U} \phi$, and means that the formula ϕ eventually holds on a path.
2. Safety properties, used to check if the model can result in an unwanted or error condition. In terms of LTL, a safety property is represented as $G\phi := \phi \mathcal{W} \mathcal{U}$

false, and means that the formula ϕ always holds on a certain path in the automaton.

One way to check for these properties is through a simulation of the model, and constantly monitoring the state space. The simulations performed could be **random** or **guided**. A random simulation simply traverses the automaton non-deterministically. A guided simulation on the other hand, uses state-based constraints and coverage heuristics [TFC⁺01] to determine the states and transitions to explore. The simulation terminates when all states (and transitions) are visited, all liveness properties are satisfied, or simply when a user-defined maximum search limit is reached.

Guided (or targeted) simulation uses a ‘test and check’ approach for verifying models. Users first instrument their models with checks that monitor whether model behavior is consistent with the safety properties. The models are then simulated to create and run test data to see if any checks can be violated. These tests can then be used to diagnose the cause of any abnormal behavior detected. Checks can be as simple as monitoring range constraints on individual signals or as complicated as determining whether or not a model satisfies a system-level requirement. Checks are usually expressed in terms of assertions, conditions that must never be violated, or simulation targets that must be reachable for the corresponding property to be satisfied.

A more rigorous way to verify (formal) usage models is via model-checking. The model checking procedure used to verify usage models is based on an interpretation using a finite-state symbolic semantics of networks. The logic is interpreted with respect to symbolic states of the form (l, D) , where D is a constraint system and l a control vector [Pet99]. Thus, a symbolic state (l, D) represents all the states l that satisfy the constraint D . Based on this notion of symbolic state, the heart of the model-checking procedure is an abstract reachability algorithm, which reduces the reachability problem to that of solving simple constraint systems. The algorithm is used to check whether an automaton may reach a state satisfying a given state formula ϕ [IV85].

Generally, the implementation the reachability algorithm is extended so that a diagnostic trace is automatically generated as a side-effect in case reachability is established. If the symbolic state-space is examined in a breadth-first manner (corresponding to organizing the waiting set as a queue), the trace is guaranteed to be the shortest possible.

3.4 Usage Scenario Generation and Testing

Test case sequences, or usage scenarios, are derived from the formal model by performing a walk through or guided simulation through the state space of the model. For layered architecture, the test suite is generated with the help of the application interface, while for observer-based architecture, the suite is derived using observers in the model. A more general approach is to provide an application programming interface (API) for the model that can be used and extended by model components to generate implementation-specific test suites. The components are used to provide a mapping between the abstract state space of the model and the concrete structures in the code. Every time a target state (or transition) is encountered, the component generates stimuli that the corresponding event would produce in the actual product.

The various stimuli generated during the traversal are recorded to produce a (unique) usage scenario. An exhaustive collection of all such scenarios for an implementation is referred to as a *test suite*.

The criteria for recording the stimuli are defined with respect to the syntax of the usage model. Each model, defined in terms of an automaton, contains states and transitions. Each transition in turn may have a label that includes all or some of the following: an event, a condition, a condition action, and a transition action. A transitions condition action is executed whenever it is deemed ready to fire. The transition's action is executed only when it is included in such a firing transition.

The criteria for generating test case sequences can thus be based on the following coverages:

1. State coverage. This criterion is met when each state in the model has been

entered at least once.

2. Condition action coverage. A condition action is deemed to be covered if it has been executed at least once, or, if the segment has no condition action, if its condition has evaluated to `true` at least once. This criterion is satisfied when all target conditions have been covered.
3. Transition action coverage. This criterion is met when all target transition actions have been executed at least once. If a transition has no associated action, it is ignored by this criterion.

The test cases generated via usage models help the regulator validate individual implementations using a black-box testing approach. This is ideally suited to the premarket review process, as the regulator is not concerned about details of the implementation during this process.

3.5 Case Study: The Generic infusion Pump

To illustrate the ideas presented in the previous sections, we have developed a usage model for infusion pump software. An infusion pump is an example of a safety-critical medical device that uses software to control the underlying hardware and events. Our case study is based on requirement specifications for a generic infusion pump (GIP) model, being developed at the CDRH/OSEL. We use the infusion pump case study as a reference to explain our approach to premarket review.

3.5.1 Overview of the GIP Model

The specifications for the GIP are defined with respect to a set of possible situations that the pump could be in and a corresponding set of input events. A situation in the GIP represents the configuration of the pump and its peripherals at a given instant of time. For example, the situation `On/Cold/Alarming` indicates that the

pump is switched on, is currently dormant (cold), and has the alarm turned on due to some malfunction. Events are used to model inputs provided by a human user or the system environment. For example, a human input could be pressing a button or administering a dose, while environment events could include timeouts or interrupts indicating a malfunction.

Infusion pumps can be categorized into five major types or categories viz., large volume, patient controlled analgesic (PCA), elastomeric, implanted and ambulatory. All these pumps are similar to each other, in that they all are used to administer ‘fluids’ into a patients circulatory system. However, there exist significant differences across the different categories in terms of specific features such as the mode of infusion, volume injected, level of patient interaction, etc.

Similarly, the software used to control these pumps also share certain common functionalities, but differ when it comes to pump-specific features. The infusion pump usage model exploits this relation between the various pump categories and represents it using a *three-tiered layered architecture*. This representation includes a **core model** at the generic (innermost) layer that captures the features common across all implementations, a front-end for the GIP interface (subclass interface) and extensions to the core, called **wrappers**, which act as the implementation interface and model the specific aspects of the different pump implementations.

An overview of the usage model for infusion pump software is shown in Figure 3.3. The model consists of, apart from the core, front-end and wrappers, a user interface, a pump module, and a simulation of the patient undergoing treatment.

The core of the usage model essentially captures the various situations and events in the GIP and represents these as a Kripke structure, $M_{GIP} = (S, \Sigma, \delta)$, where S , the set of states for M_{GIP} represents the situations for the infusion pump; Σ corresponds to the input events for the pump; and δ depicts how the generic pump would react to user events. Wrappers are needed to model specific characteristics of the various pump categories. Models for the different categories (or pump types) in this scheme are characterized by the types of inputs and events they allow. Each category model therefore corresponds to a set of wrappers that provide a mapping from real world values to abstract boolean inputs for the core. Abstracting the events in this manner

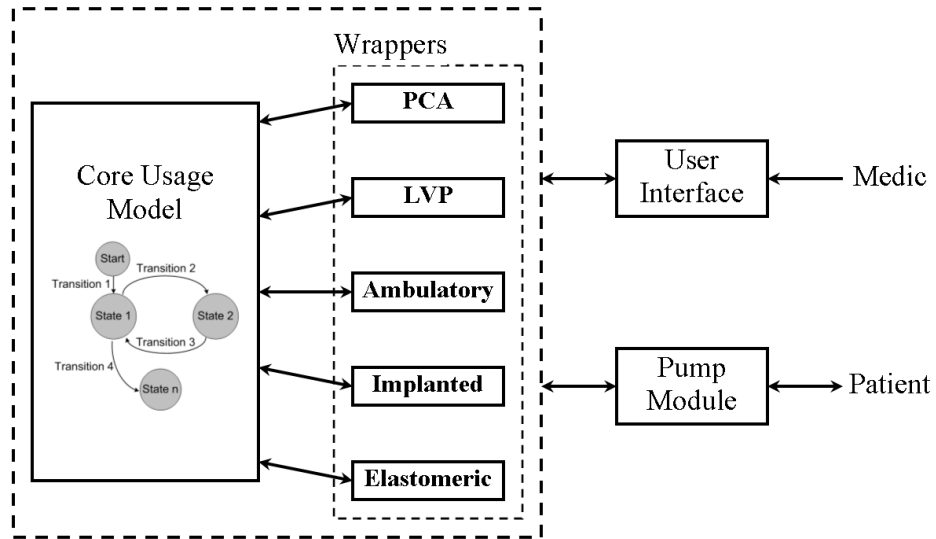


Figure 3.3: Overview of the usage model for infusion pump software

not only facilitates modeling pump categories, but also helps reduce the state space (and thus the complexity) of the core model. Moreover, abstracting event handling details away from the core improves the adaptability of the (overall) usage model and makes it much easier to incorporate changes therein.

The user interface (UI) for the usage model is simple, but comprehensive. The UI provides all possible buttons (both hard and soft) and screens available to the medic, but allows for these to be disabled if necessary. The screens corresponding to the Point-of-Care Unit (PCU) for example, are enabled for large volume pumps, but disabled when generating test sequences for ambulatory or implantable infusion pumps. The pump module represents the physical pump itself, along with its peripherals such as the syringe, boluses, and vials. The module is used to simulate the working of the physical pump and provides system generated events such as interrupts and timeouts. The patient module is passive in most cases, except for the PCA pump model, wherein it can cause an event by requesting for an analgesic dose. Ideally, this module would be more comprehensive and would simulate the patients reactions to the medication being administered (e.g., by monitoring its heart rate, blood pressure etc.). Defining such a simulation model however, is a non-trivial task and an area of

active research on its own.

3.5.2 The GIP Usage Model in Matlab

The usage model is defined using the Stateflow toolbox of Mathworks Simulink, which is a part of the Matlab tool suite. Matlab was chosen as a modeling tool owing to its support for state transition notation and numerical computations.

Figure 3.4 shows the front-end interface for the model designed in Simulink. The pump is modeled as a closed loop system with the input events being sent to the model at fixed step discrete time intervals of 0.5 seconds.

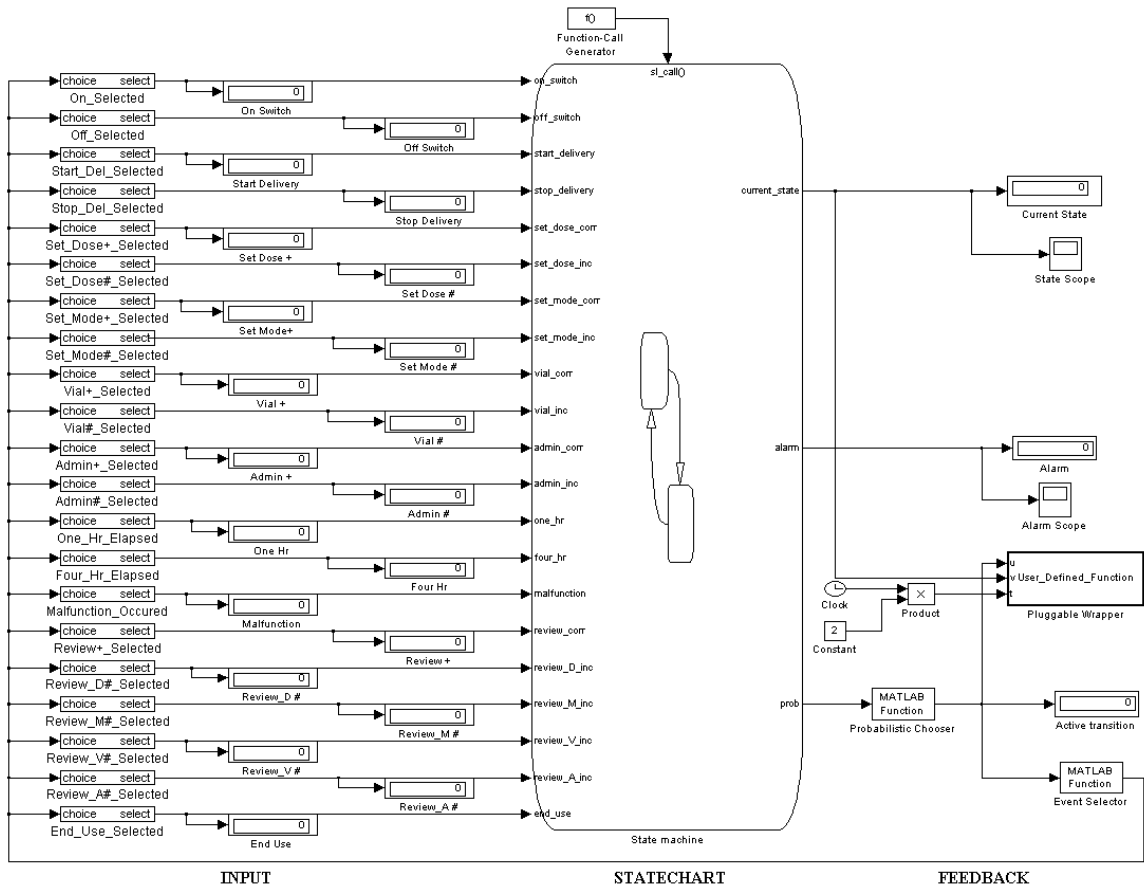


Figure 3.4: The Simulink interface to the infusion pump usage model

The model can be viewed as comprised of three subsystems, labeled **INPUT**, **STATECHART** and **FEEDBACK** respectively. The **INPUT** subsystem is used to model inputs and other events for the infusion pump software to the **State machine** block. The events are modeled simply as integer values between 1 and 21, representing one of the 21 possible inputs for the infusion pump.

The **STATECHART** subsystem has 21 inputs corresponding to each of the input events possible, and three outputs — **current state**, an integer indicating the current configuration the state machine is in; **alarm**, a boolean flag set when the system is in an alarming state; and **prob**, an array listing the probability for all possible transitions for the current state. The function call generator **f()** is used to trigger a new transition in the state machine every 0.5 seconds. The **State machine** block consists of 292 states and approximately 3,500 transitions. A customized parsing tool is used to automatically extract the model from the specifications provided in order to improve the efficiency and minimize errors during the model building process. The Stateflow API is used to code the states and transition behavior.

The **FEEDBACK** subsystem is used primarily to display the status of **State machine** and to select a next valid transition as input to complete the feedback loop in the model. The block **Probabilistic Chooser** uses a Matlab function to non-deterministically select a transition using the values in the array **prob** (a higher probability value translates into an increased chance of that transition being selected). The chosen transition index is then used to select an appropriate event and re-entered to the **INPUT** subsystem at the next 0.5 second discrete interval, thus continuing the loop over the next cycle.

The block **Pluggable Wrapper**, as the name suggests, is used to define a mapping between the abstract states in the Stateflow state machine to corresponding functionalities in the actual software implementations. The block, implemented in Matlab/C, is also used to generate test cases for the implementations through guided simulations. For the purpose of our case study, we have defined a mapping to a prototype PCA software, developed at the CDRH. Mappings could easily be defined for other, similar implementations as well and wrappers implemented for these devices accordingly.

3.5.3 Verifying the GIP Usage Model

To verify the correctness and completeness for the infusion pump model, we have to ensure the correct operation of the pump for all possible combinations of situations and events defined by the specifications. To achieve this, a number of global properties are defined that when completely satisfied would ensure conformance for every situation and event in the infusion pump model. This set of global properties consists of the following:

1. All possible situations must be reachable.
2. All valid events must be executable (follows trivially from 1).
3. There must be no deadlocks in the system.
4. There must be no livelocks of length $|Q|$ in the system, where Q is the set of all situations.
5. The combined probability for all events for a given situation must always be 1.
6. If the probability for an event is zero for a given situation, then that event must never be executed.
7. If the probability for an event is non-zero for a given situation, then it must be possible to execute that event.
8. The event ‘malfunction’ must always trigger the alarm (i.e., it must always result in a situation labeled ‘Alarming’).

To assess these properties against the usage model, we use the Reactis [rea03] and Uppaal [BLL⁺95] tool suites. The two tool suites are used to verify different aspects of the model, and to ensure completeness of the verification process.

In particular, Reactis allows for guided simulations through the state space of the Matlab model to assure coverage for all situations and events. Uppaal on the other hand, provides more conventional model checking by evaluating the usage model against linear temporal logic (LTL) formulae. Reactis is chosen as a verification tool

owing to its compatibility with Matlab, while Uppaal is used due to its support for LTL and easy-to-use specification language. Similar tools such as Spin [Hol97] and SMV [McM92, CMCHG96] could be used in its stead as well.

3.5.3.1 Verification using Reactis

Reactis is an embedded-software design automation (ESDA) tool suite developed by Reactive Systems. It is based on the Simulink/Stateflow modeling paradigm, and uses program flow analysis to automatically perform exhaustive trace verification for a Matlab model. This verification is usually performed against a set of assertions and user-defined targets. Any undesirable property that is satisfied during model execution results in an assertion violation. On the other hand, a user-defined target must be reachable to ensure that a desired property is satisfied.

Reactis does not support a closed-loop model; thus the feedback loop in the usage model is replaced by an equivalent open-loop input module. The inputs are chosen at random with an equal probability distribution, and configured randomly to generate an integer with a value between 1 and 21. Also, the Matlab block `ProbabilisticChooser` is replaced by a state machine emulating a human user.

Figure 3.5 shows a snapshot during verification in Reactis. The (top) left panel in the figure shows an assertion block with inputs defined as the current state and probability values for each event. The input probabilities are compared against the transitions active at the current state. Any discrepancy between the two (e.g., a valid transition with a null probability value) is detected by the `Relational Operator` block and an assertion violation signaled.

The bottom panel in the figure shows the trace for a particular state (state 61). The valid transitions active at the state are listed in row 2 (each valid transition is represented by a 1 in the 21 element array). The corresponding probability values for the state are listed in row 3, and rounded off to the highest integer in row 4. The `Relational Operator` block compares the values in row 2 and row 4. The output of this block is listed in row 5. All `false` entries in the list represent an inconsistency between transitions and event probabilities, and therefore correspond to an error in

the system specifications.

The model defined in Uppaal is in essence similar to the **State machine** block in the Matlab model and captures the behavior of the usage model using states and transitions in a similar manner. Uppaal does not support real numbers however, therefore a simplification is made while assigning the probabilities for transitions. Each valid transition (i.e., a transition with a probability value greater than 0) is assumed equally likely. This assumption does not affect the reachability of the states, since any transition with a non-zero probability value must occur eventually.

The Uppaal model is ported directly from the Stateflow block and consists of the same number of states and transitions. The coverage for the model is verified using LTL formulae that are evaluated against the state machine defined in the model. Figure 3.6 lists some typical LTL formulae that are verified against the infusion pump usage model. Following conventions used in Uppaal, $\mathbf{E}\psi$ indicates that an LTL formula ψ holds for some run of the automaton. $\mathbf{A}\psi$ means that the formula ψ is **true** for all runs.

EF (pump.State245) State 245 in the infusion pump model can be reached. EF (pump.Trans1925 == true) Transition 1925 must eventually be executed. EG ($\forall s \text{ State}(s) \wedge (\text{pump.s} \Rightarrow \text{pump.State1})$) It is possible to visit State1 in all possible paths. AG ($\neg \text{deadlock}$) The system is deadlock free. AF (pump.State1) At least one path has S1 true all the time.
--

Figure 3.6: Sample LTL formulae used in Uppaal to verify the GIP usage model

Anomalies detected in the model are resolved to ensure correctness of the model before it is released publicly (to the manufacturers), or used to derive test suites. In the case of the infusion pump model, most of the anomalies detected using Reactis and Uppaal pertain to missing transitions and incorrect probability assignments. These discrepancies are resolved by adjusting the behavior of the state machine, and the model re-verified to ensure conformance to specifications.

3.5.4 Generating Usage Scenarios for the GIP Model

The GIP usage model, once exhaustively verified and assured to be correct, is used to generate usage scenarios for particular implementations of the system. In our study, we generate usage scenarios for a prototype implementation of a PCA pump,

developed at the CDRH. The software for the prototype is implemented in C, and consists of around 20,000 lines of code.

The set of usage scenarios is derived with the help of pluggable wrappers. The wrappers are plugged in the GIP model in place of the **Pluggable Wrapper** block explained in Section 3.2. For the GIP model, a wrapper could be used for large volume, PCA, implantable, elastomeric, or ambulatory pumps. In our study, we use a wrapper to generate a test suite for a prototype PCA implementation. The usage scenarios generated thus are used to validate this prototype, with each scenario in the test suite corresponding to a unique path or trace through the GIP model.

The traces for the test suite are obtained by simulating the GIP model in Reactis. Reactis provides both random and targeted simulations for providing these traces. The targeted simulation mode addresses coverage for both states and transitions in the simulated model. For each trace generated by Reactis, the PCA wrapper produces a sequence corresponding to the usage scenario for the specific (PCA) implementation.

The PCA wrapper translates each transition in the trace to an input event for the prototype. A usage scenario thus produces a sequence of input events and the corresponding (expected) response for the PCA implementation.

Figure 3.7 shows part of the usage scenario generated for the PCA prototype. The scenario is produced in a syntax that is suited to comparison with the output from the PCA test harness. A **Prompt** in the sequence indicates the message that is prompted on the pump's user interface. **Input** indicates an input event for the pump (along with the corresponding code for the event). **System Interrupt** similarly indicates an internal event generated by the pump. **Response** lists the response that is expected for the given input. A sequence of these inputs and response tuples forms the usage scenario for the implementation.

To test the prototype PCA implementation, a test harness is incorporated within the PCA software to simulate the input events given in the usage scenario. The responses (response codes) for the events are recorded and finally compared against the expected responses from the generated scenarios using a customized parser. The development of the test harness and testing for the prototype PCA implementation were carried out in-house at the CDRH.

```

Prompt: "Please set the dose for the patient" -> Input:I005:The flow rate was set to 30
mL/hr. The VTBI was set to 300 mL -> Response:R001:No response.

Input:I002:The ON button was pressed -> Response:R102:The pump was switched ON.

Prompt: "Please select the mode of operation" -> Input:I008:The mode was set as
ANESTHESIA -> Response:R122:The mode set is NOT in accordance with the prescription.

Prompt: "Please set the dose for the patient" -> Input:I005:The flow rate was set to 30
mL/hr. The VTBI was set to 300 mL -> Response:R001:No response.

Prompt: "Please set the dose for the patient" -> Input:I006:The flow rate was set to
30.3 mL/hr. The VTBI was set to 300 mL -> Response:R109:The pump started infusion at a
rate NOT in accordance with the prescription.

System Interrupt:I113: Timeout occurred after one hour of inactivity. Pump going into
power safe mode -> Response:R203:Pump in power safe mode.

Prompt: "Please set the dose for the patient" -> Input:I005:The flow rate was set to 30
mL/hr. The VTBI was set to 300 mL -> Response:R001:No response.

Input:I018:Reviewed pump configuration and changed the mode. The mode was set as
ANESTHESIA -> Response:R122:The mode set is NOT in accordance with the prescription.

```

Figure 3.7: Part of the usage scenario for the PCA prototype implementation

3.6 Case Study: The CARA System

CARA is intended to support primary intravenous fluid resuscitation therapy to rapidly restore intravascular volume and blood pressure in patients with clinical shock, hypotension, and hypoperfusion states as a result of hemorrhagic blood loss, occult hemorrhage, neurogenic shock, and septic shock. The intended use of the system is for combat casualties, but in time it could be used for civilians as well, in cases like accidents or trauma, or for patients suffering from heart diseases and diabetes. Being a highly safety-critical system, to the extent that the patients life depends on it, it becomes imperative that CARA has zero tolerance for errors.

The CARA system works on three platforms: Life Support for Trauma and Transport (LSTAT), DataPak, and Warfighter Physiological Status Monitor (WPSM), collectively known as Hosts. The control software uses the patients blood pressure to guide a proportional closed-loop control algorithm, which controls a high-output infusion pump. The normal use is for a physiological monitoring device (PMD) like the LSTAT or DataPak to be attached to the patient by a caregiver, identified as a Medic, Nurse, or Doctor. Once the PMD is attached, vital signs are taken and regis-

tered to CARA. CARA responds according to the condition of the patient, which is determined by the patients blood pressure.

3.6.1 The CARA Usage Model

The usage model for CARA is developed using the Observer architecture, with individual sub-components loosely coupled with each other, as in the subsystem architecture. Each sub-component in turn is modeled as a timed automaton sharing a common global clock. The independent sub-components communicate with each other using asynchronous messages.

The approach taken for analyzing the requirements was to build models with the hope that inconsistencies in the requirements would become apparent. To that end, we initially built a model in Uppaal [BLL⁺95]. Based on that experience we built a very similar, but more detailed, model in Spin [JCI04]. The two models, while similar in nature in that both are networks of automata, differed in how certain timing aspects were captured. While Uppaal allows the use of multiple real-time clocks and the ability to express constraints on those clocks, Spin does not have the ability to express these constraints in a straightforward manner; necessitating the clocks to be modeled explicitly.

3.6.1.1 CARA components in Uppaal and Spin

An overview of the components used in the CARA model is shown in Figure 3.8. Based on the system requirements, the Uppaal and Spin models consist of the following components:

- **The CARA Control Algorithm:** The CARA process is designed as an elaborate automaton with various locations representing possible states that the system can exist in at any given time. Transitions between states are modeled based on the values of global variables corresponding to the mode, the blood pressure source, cuff readings, etc. and the current clock readings. *Channels*

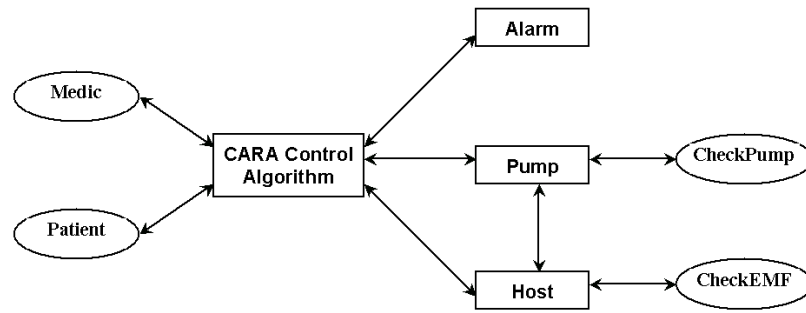


Figure 3.8: Overview of the CARA usage model

are used to issue alarms in case an error is detected and also to synchronize with other processes. The channels are constantly checked for any indication of errors or missed readings that might cause an alarm. Since multiple synchronizations are not allowed in Uppaal [LPY97], committed locations are used to simulate actions that are performed simultaneously. The Spin model is similar except that the clocks were simulated.

- **The Host:** The Host module is used to monitor the patient and provide feedback to the CARA algorithm. The CARA system operates in one of three modes: a Wait mode, during which all the subsystems are available but the pump is not connected to the Host; a Manual mode, during which the pump is connected to the Host, but CARA is dormant; and an Autocontrol mode, during which the system operates entirely according to the CARA algorithm.

In the Manual mode, the Host collects blood pressure readings and other vital signs from the patient and continually displays them on the screen for the medic. The medic though has complete control over the pump and decides the flow rate and volume to be infused by the pump. In the Autocontrol mode, the Host continues collecting readings from the patient, and in addition also feeds a control voltage back to the CARA control algorithm. The Host is also used to display and log all its readings while in this mode.

- **The Pump:** The Pump process is used to generate readings (cuff and beat-to-beat) based on the mode and blood pressure source values. The process

is modeled to lie dormant till it receives a message to generate a cuff reading (indicating that the system is in Autocontrol mode and that the cuff pressures are inflated). Upon receiving this message, the process generates a blood pressure reading non-deterministically. It then continues in a loop, checking for the blood pressure source and producing cuff pressure or beat-to-beat readings correspondingly. At any instant, if the system reverts to Manual mode, the process breaks out of the loop and waits for the next request for a (cuff) reading.

- **The Alarm:** The alarm process simulates the possible alarm conditions that might arise during the operation of the CARA system. A distinction is made between level 1 and level 2 alarms, and each alarm condition is associated with a certain priority. The lowest priority is used to indicate that none of the other conditions held true, and corresponds to a ‘No alarm’ state. The process also incorporates the silence and reset buttons with a silence time corresponding to each of the alarm conditions. On receiving a silence signal, the alarm is disabled for the duration given by the silence time. On receiving a reset signal, all alarms are flushed and the alarm is set to the lowest priority.

While the components described above are common to both the Uppaal and Spin models, the context in which these components work, viz. the Observers are modeled differently in the two models.

3.6.1.2 Observers in the Uppaal Model

The observers in the Uppaal model include the processes **Medic**, **Patient**, **CheckPump**, and **CheckEMF**. The **Medic** process mimics the role of a medic controlling the functioning of the CARA system. This consists of sending messages to start and terminate Autocontrol mode (based on the current mode and the status of the pump and the Host) and to decide whether or not to override a non-corroborated blood pressure source.

The **Patient** process simply makes a transition on each reading, updating the flow rate and the average blood pressure in the process. The **CheckPump** and **CheckEMF**

processes (which in the real system monitor the volume of fluid infused and the back-EMF, respectively) are modeled as observers. Both processes maintain a set of variables to indicate the status of the system. These variables, corresponding to the back-EMF, impedance, connections, airlock, and occlusion, are non-deterministically assigned truth values in the course of the various transitions. The variables, in turn, trigger the various error signals sent to the Alarm Unit.

3.6.1.3 Observers in the Spin Model

Some of the components, such as `CheckPump` and `CheckEMF`, are modeled at a greater level of detail in the Spin model. In particular, the process of taking averages of blood pressure readings and the process of adjusting the volume of fluid flow based on the current pressure is modeled based on linear control.

3.6.2 Verifying the CARA Usage Model

The verification for the CARA model is performed with the help of the Uppaal and Spin model checkers. Both these model checking tools search the state space of the system to test for invariant and reachability properties.

Uppaal uses a query analyzer, a graphical user interface that takes LTL formulae as input and uses the model checker to verify them against the model defined. Specifications of the CARA system (as given by the requirements document) are translated into LTL formulae and verified for correctness using the model checker.

Spin provides two ways to verify the system definition. Both involve an exhaustive state space search mechanism. The first involves checking for assertion violations included in the process definitions themselves. This, however, is quite expensive for the CARA system and often results in memory overflows. To avoid this, assertion violations are checked using abstractions of the model, or with only partially defined system definitions (i.e., without some of the processes included in the system definition).

Figure 3.9 gives a few sample LTL formulae against which the CARA model is verified. The same formulae are used for verification using both Uppaal and Spin.

AG (mode == AC) $\Rightarrow$ (connected == true)
The pump is always connected (to the Host) in autocontrol mode.
(connected == false) U (mode == manual mode == AC)
The pump is disconnected until CARA is in manual or autocontrol mode.
AG (mode == AC) $\Rightarrow$ (connected == true)
The pump is always connected in the autocontrol mode.
(alarmPriority == 11) $\Rightarrow$ (EF $\neg$ (mode == AC))
Alarm 11 (no BP reading for 3 minutes) cannot occur autocontrol mode.
AG (flowRate == 6) $\Rightarrow$ (backEMF > 1.0 V)
If the flow rate is 6 liters per hour, the back EMF must be greater than 1V.

Figure 3.9: Sample LTL formulae used to verify the CARA usage model

3.6.3 Generating Usage Scenarios for the CARA Model

Testing for CARA is done against a prototype of the system developed at the State University of New York (SUNY), Stony Brook [Sta04]. The SUNY simulation is based on the same requirement specifications as our model, and implemented as a Java applet that mimics the operation of the real-life CARA system. A screenshot of the applet is shown in Figure 3.10.

The CARA applet supports only random simulations of the system; i.e., it does not allow for targeted simulations. Therefore, an alternate semi-automated strategy is used to validate the system against usage scenarios. To accomplish this, the CARA applet is first modified to record all the different execution traces and system states. A sample trace recorded during random simulation of the CARA applet is shown in the bottom frame in Figure 3.10. Since the CARA applet involves only a few states per process, full state coverage is ascertained from the traces with relative ease.

Once the execution traces from the applet are available, they are used to manually perform guided simulations in Uppaal and Spin. The responses from

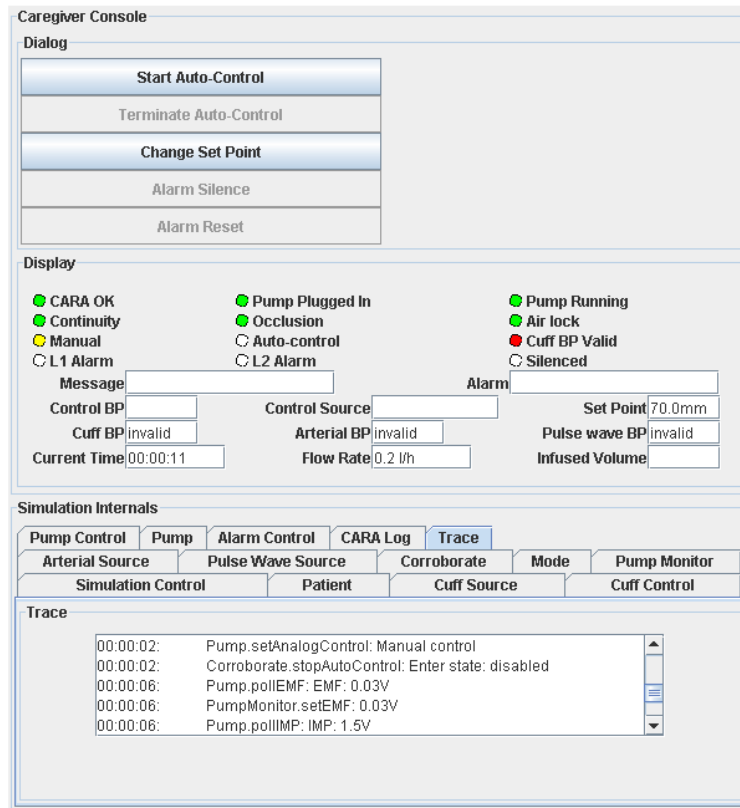


Figure 3.10: Screenshot of the applet used to simulate the CARA system

the model simulations are finally compared against the execution traces to validate correctness for the applet. Any discrepancy between the two indicates an error in the implementation of the CARA applet.

Results from the verification and testing for the two case studies are given in detail in Section 6.

Chapter 4

Postmarket Forensic Analysis

Forensic analysis — or the process of tracing software failure to source code — is one of the most challenging activities in the life cycle of any software system [MSA⁺04b]. It can be especially intricate when dealing with large, real-time systems. Typically, such systems consist of several different concurrent processes and event-based operations, making it even more difficult to analyze and debug the system.

Some studies have suggested that up to 50% of all effort spent in software development is during the testing and maintenance phases [ARTZ03]. A major chunk of this maintenance effort is spent in simply trying to understand the code and locating the cause of a failure. The scenario is further complicated in the case of post-mortem analysis, where documentation may be sparse and the original developers, usually third party vendors, may no longer be available. Additional effort needs to be spent in such cases to understand the code before it can be debugged/analyzed.

Traditionally, program slicing methods have been used to perform post-mortem analyses [SK00]. There have been several studies showing how program slicing can be used to assist debugging [GL91, Tip95]. However, by its very nature, slicing provides only a limited, scaled-down view of the program. To completely understand the system, replete with process dependencies and control flows, a wider, more macro-

scopic view of the entire system needs to be provided. Moreover, when carrying out post-mortem analysis, regulators do not always have access to compilable code or implementation environments (e.g., for radiation therapy machines or spacecraft control software). This makes the use of dynamic analysis techniques, such as dynamic slicing and backtracking, almost impossible for these software. As a result, only static slicing (or a variation thereof) can be used for post-mortem analysis. The problem with static slicing, however, is that the resultant slices are generally too large to be of much use to the analyst. This problem is even more pronounced when dealing with large programs.

We use the abstraction-driven slicing method to address this problem. By combining abstraction with slicing, we aim to improve the analyst’s understanding of the software, and eventually lead to a more efficient approach to forensic analysis. In this chapter, we describe the integrated approach to abstraction-driven slicing, and explain how it can be leveraged to aid post-mortem forensic analysis.

Specifically, in Section 4.1 we give an overview of the program comprehension process, and how it can be achieved through a combination of slicing and abstraction. Section 4.2.1 explains the methodology for the abstraction-driven slicing approach. Section 4.3 details how this approach can be automated through the implementation of the CAdS forensic analysis assistant. Experiences with the approach are examined in Section 4.4 with the help of an example to illustrate the use of abstraction-driven slicing for tracing failure reports to errors in software.

4.1 Facilitating Program Comprehension

To understand how a better comprehension of a program can improve its analysis, we first need to understand how programmers and analysts comprehend code. The process of understanding the code, or “program comprehension” is defined by deLucia et al. [dLFM96] as the process of abstracting higher level descriptions of the system — which employ typical application domain concepts and terms — from lower level descriptions, like program structure and data flow. Typically the comprehension of

an existing software system consumes from 50% to 90% of all maintenance effort [dLFM96]. This effort can be greatly reduced if the analyst could have an efficient means to automatically abstract the high level descriptions from the source code, and not have to manually scan the entire code in order to do so.

Several cognition models have been proposed to explain how program comprehension works. Two of the most popular among these are the top-down and bottom-up models [Bro83, Pen87]. The top-down model of program comprehension is typically invoked when the code under consideration is familiar and a description of the conceptual components of the application domain and of the way they interact is available. The analyst understands code by exploiting this knowledge to formulate hypotheses about the meaning of the program segments being analyzed. The hypothesis are confirmed by scanning the code for beacons — pieces of code implementing typical program constructs, and iteratively refined, producing new sub-goals to be verified by scanning code again. The process halts when each component of the application domain has been identified in the code.

Program slicing is a popular tool for aiding top-down comprehension. Like all top-down approaches, it is best used when dealing with familiar code, and serves as an effective code scanning technique that allows the analyst to verify and refine hypotheses.

The bottom-up model of program comprehension is invoked when the code under consideration is completely new to the analyst. The first mental representation of the code the analyst builds is a control-flow abstraction called the program model. The program model is created by chunking of microstructures into macrostructures and via cross-referencing. Starting from the program model, a further model can be abstracted which maps the control-flow knowledge about code to the real world domain knowledge. The process continues recursively by formulating new models to aggregate these plans into higher order plans.

Abstraction, by virtue of automatically extracting models from source code is a good facilitator for bottom-up comprehension. A structured visual representation of the code acts as the program model for the analyst and helps form a quick and clear mental representation of the program being analyzed.

A more holistic, integrated model is suggested by von Mayrhauser and Vans [vMV95], which proposes that code comprehension involves both top-down and bottom-up activities. The model is based on the premise that the comprehension process does not proceed exclusively in either the top-down or the bottom-up direction, but instead switches continuously between the two approaches. Our approach to post-mortem error detection, viz., the abstraction-driven slicing process, is based upon such an integrated model of program comprehension. Top-down comprehension in the process is realized by traditional static slicing, while bottom-up comprehension is provided by the program models generated via abstraction.

4.2 The Abstraction-driven Slicing Approach

4.2.1 Methodology

The methodology for post-mortem forensic analysis through a combination of slicing and abstraction is given in Algorithm 2. The approach combines the top-down cognition model (realized by static slicing) with automated bottom-up model abstraction, enabling the analyst to comprehend the code better and in effect analyze it more efficiently.

The key to this approach lies in its iterative nature. The process starts off by producing a large, top-level model, M for the (default) abstraction criterion. The analyst uses M to get an initial understanding of the program and defines a slicing criterion to obtain a slice S corresponding to M . Over each successive iteration, S keeps getting more refined, and smaller in size. Similarly, the program model, M , is highly abstract to begin with, but successive models keep getting more specific and detailed during each iteration. The successive iterations keep returning more refined slices and specific models till either the problem is successfully resolved, or the slice is shown not to contain the error under consideration.

An important feature of the algorithm is that both the slice S and the model M are configurable. This provides the analyst with the flexibility to redefine the slicing

Algorithm 2 Abstraction-driven Slicing

Input: A program P , a slicing criterion $C = (x, V)$, where x is a program point and V is a set of variables.

while Problem not successfully detected **do**

Obtain a slice S for the criterion C .

Choose an abstraction map AM and a label map LM for the slice S , based on heuristics given by Algorithm 3.

Obtain an abstract model M for the slice S using AM and LM .

if Problem narrowed to a specific part of model M **then**

Choose a refined slicing criterion $C = (x, V)$ and continue.

else

Pick one of the following

Choose a new abstraction map AM and label map LM to generate a new model M .

Change slicing criterion $C = (x, V)$ and generate a new slice S .

end if

end while

(or abstraction) criteria at any point during the process, as required for the analysis. However, with configurability there is a wide possible set of slicing criteria, and abstraction and label maps that an analyst could use. Based on our experiences with the approach, we have synthesized a set of heuristics that can be used for obtaining an abstraction criterion. These heuristics are provided in Algorithm 3. The heuristics essentially suggest that that abstraction and label map should be generated based on the slicing criterion. In particular, they suggest that all global variables that are present in a slice, and conditionals that affect the variables in a slicing criterion are to be watched. We formalize this concept in the Abstraction map generator module of the CAdS tool. Details of the module’s implementation are given in Chapter 4.3.

One problem, theoretically speaking, with providing this sort of configurability to the analyst is that a successor slice could be larger than its predecessor during some iteration (i.e. the set of states in M' be a superset of the states in M). In a real-world scenario, however, this would not be a problem, as eventually the slice would be refined to the appropriate size over successive iterations.

To evaluate the approach, we use the Codesurfer and CWolf tools in combination to facilitate static slicing and model abstraction respectively. We refer to this

Algorithm 3 Heuristics for defining Abstraction and Label Maps

Input: A program P , a slice S .

Output: Abstraction Map AM and Label Map LM for constructing a model M .

Add all functions that are included in the slice to the AM .

Add each variable defined in the slicing criterion $C = (x, V)$ with a default partition value (free) to the AM .

For all functions defined in the AM add a call statement with a default label set to the name of the function, to LM .

for all Variable v in the slicing criterion $C = (x, V)$ **do**

For each conditional expression involving v add a watch for the expression in the LM . This would need to be revised in case the expression are evaluated against other variables (or non-literals).

if v is a global variable **then**

if v is on the left hand side of an assignment **then**

add $write(v)$ to the LM .

end if

if v is used on the right hand side of an assignment statement **then**

Add $read(v)$ to the LM .

end if

end if

end for

technique involving the use of two independent tools to provide forensic analysis as **semi-automated abstraction-driven slicing**. A more complete automated approach is provided by the CAdS forensic analysis assistant, explained in Section 4.3.

4.2.2 Computing a Slice using Codesurfer

Codesurfer, introduced in Section 2.6, uses a modified version of the algorithm introduced by Ottenstein and Ottenstein [OO84] to compute static slices for C programs. The algorithm is based on the notion of a program dependency graphs (PDG) [FOW87]. A PDG for a program is computed by combining control dependence and flow dependence edges for the control dependency graph (CFG) of the program. A control dependence edge exists between nodes p and q of a CFG if a condition defined at p holds **true** at q . Similarly, a flow dependence edge is established from p to q if the value of a variable assigned at p may be used at q .

The PDG-based algorithm given by Ottenstein and Ottenstein applied to intra-procedural slicing. To incorporate inter-procedural slicing, Horwitz et al. extend the concept of a PDG to a system dependence graph (SDG). The algorithm for inter-procedural slicing is detailed in [HRB88].

Figure 4.1 shows a fragment of C code and the corresponding PDG. Solid lines in the graph correspond to the edges in the CFG for the program. Dashed lines represent flow dependence edges. To compute a backward slice with respect to the last statement in the code (`printf(i)`), edges from the corresponding PDG block are traced to all of its predecessors. Each of these predecessors are then traced to their own predecessor blocks, shown as bold in the figure. The algorithm continues recursively till no new blocks are encountered. The set of all visited blocks gives the required slice (highlighted in code).

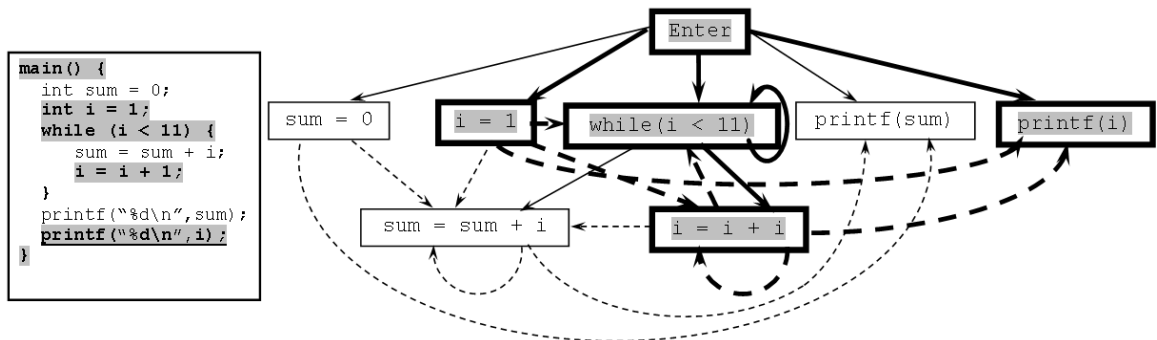


Figure 4.1: Computing a backward slice using a program dependence graph (PDG)

4.2.3 Using CWolf to Abstract C Programs

We explain the process of extracting abstract models from software with the help of the CWolf abstraction tool. CWolf is a tool developed at NC State University for automatically generating data-abstracted finite models of C programs. The front end for CWolf consists of the CWolf C Compiler (CWCC), which reduces a C program to an intermediate representation, known as the CWolf Intermediate (CWI) format. The back-end consists of a CWolf Model Builder (CWMB), a model generator com-

ponent which is built around an abstract virtual machine used to execute abstracted C programs.

The model generator operates on three input files. The first is the CWI file containing a compiled C program. The second is an abstraction map (AM) file, which describes abstractions for each variable in the input program. The third is a label map (LM) file, which is a text file describing the labels for the transitions of the generated model. The output of the model generator is either a graph with labeled states and edges for viewing by the user, or a labeled transition system (LTS) in a format readable by the Concurrency Workbench of the New Century (CWB-NC). A simplified system architecture reflecting this view of CWolf is shown in Figure 4.2.

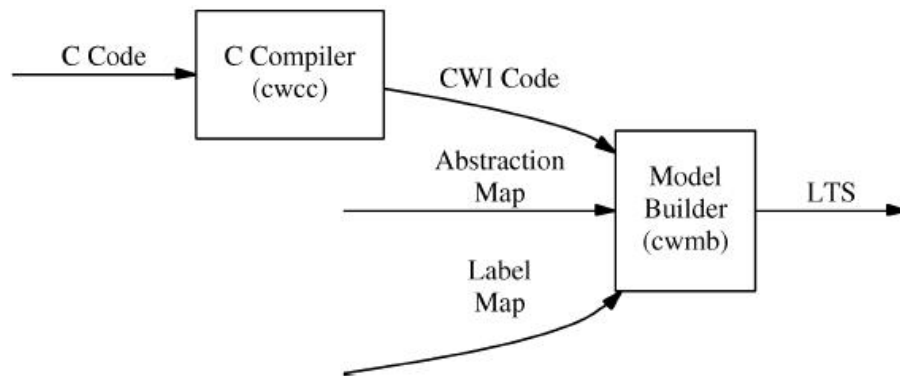


Figure 4.2: Architecture of CWolf

An example of the use of CWolf is given in Figure 4.3. In part (A) is a C-program which is being abstracted and Part (B) is its abstraction. For CWolf to develop this diagram two directives have to be provided: (a) an abstraction map which says what variables have to be abstracted and how, and (b) what the important events are and how they have to be labeled in the abstraction. An example abstraction map and label map are given in Figure 4.4. As can be observed from this figure, the variables `ch` and `state` are the important variables of the program. In particular, the abstraction map says that what matters is whether `ch` is the character `'a'` (ASCII code 97), `'b'`, `'c'`, `EOF`, `'\n'` (ASCII code 10) or `'\r'`. The label map, in a similar fashion, says that when these characters are seen an event labeled `a`, `b`, `eof`, `X` (denoting other

characters) is to be generated. The result is, of course, Figure 4.3 (B).

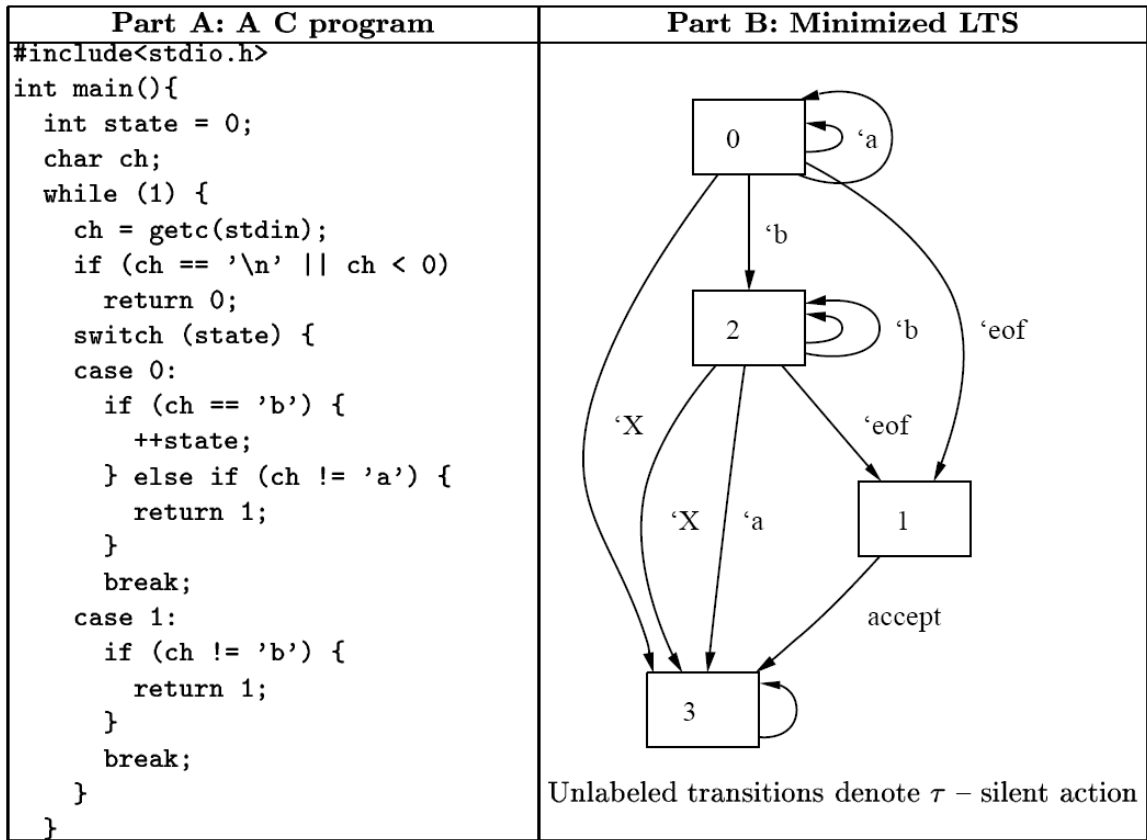


Figure 4.3: A simple C program to recognize a^*b^* and its minimized LTS

4.3 CAdS — Automating Post-mortem Analysis

The abstraction-driven approach provided considerable improvements over the use of static slicing for post-mortem analysis. However, a significant amount of effort is still expended in defining abstraction criteria and mapping abstract models to slices derived from the code. To reduce this overhead, we have automated these two activities in CAdS — the **C** language **A**bstraction-driven **S**licer.

The framework for CAdS is based upon the architecture of CWolf, which we extend to include functionalities for slicing and automated abstraction criteria derivation.

fsm.am	fsm.lm
<pre> file "fsm.c" { fun main () : top { var state : part(0,1,2); var ch : part(0, 10, 11, 97, 98, 99); } </pre>	<pre> exit == 0 => accept; watch (main:ch == 97) => 'a; watch (main:ch == 98) => 'b; watch (main:ch < 0) => 'eof; watch (main:ch == 10) => 'eof; watch (main:ch >= 0) => 'X; </pre>

Figure 4.4: Abstraction and label maps for the program of Figure 4.3 (A)

Clearly, any lessons learnt from our work can also be transfered to other software model-checkers, for instance [CDH⁺00, HJMS03].

4.3.1 CAdS System Overview

An overview of the CAdS system architecture is shown in Figure 4.5. The system consists of three major components — a model generator, a CWI file slicer, and an abstraction map generator. A user interface is provided to the analyst to view the generated models and specify criteria for slicing.

The model generator takes as input the CWI file produced by the CWCC, and an abstraction criterion (a default criterion is used when none is specified). The model builder abstracts the program according to the criterion specified and generates a corresponding LTS. The graph builder component then aggregates the blocks in the LTS to produce a hierarchical graph representation of the model, output as an XML file.

The user interface parses this XML file and displays it as an interactive, navigable graph, providing the analyst with a bottom-up understanding of the program. Each block in the graph is associated with a corresponding line in the original program/slice, and allows the analyst to specify a slicing criterion by selecting a block (or set of blocks) in the graph. Once specified, the CWolf slicer uses this criterion to slice the CWI file.

The abstraction map generator uses the updated (sliced) CWI file and the user-

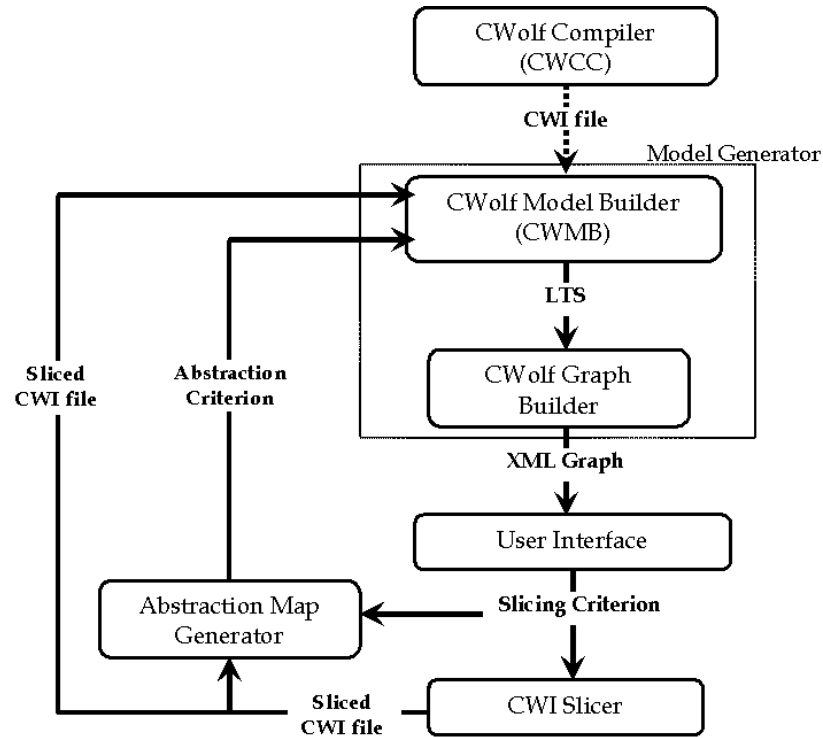


Figure 4.5: The CAdS System Architecture

defined slicing criterion to build a new abstraction criterion for the model generator. A new model is generated with respect to this criterion and the entire process repeated over the next iteration.

4.3.2 Model Extraction

Extraction of a model from a CWI file is done over two steps — first the CWMB extracts an abstract model (LTS) from the CWI file, and then the graph builder component converts this LTS to a structured graph representation.

4.3.2.1 The CWolf Model Builder (CWMB)

The CWMB uses non-relational data abstraction [CC79] to approximate values of a set of considered data variables, ignoring all other information in any state during

program execution. In particular, it focuses exclusively on the abstraction of integer variables [DuV02]. The data values of interest are specified in the abstraction map file, and constitute the abstraction criteria for CWMB. To assist the analyst, CWMB provides a set of predefined parameterized abstractions. The syntactic terms for the available abstractions are **top**, **mod**, **part**, **minmax**, and **free**. A summary of these abstractions is given in Table 4.1.

Table 4.1: Abstractions provided by CWMB

Syntax	Description
top	All values are abstracted to a single value $\top$.
minmax	Values are abstracted to an upper and lower bound $\langle l, h \rangle$.
mod (k)	Values are abstracted to a set of remainders modulo k .
part ($a_1, a_2, \dots, a_k$)	Values are abstracted to a partition of the set of integers. The partition is $[-\infty \dots a_1 - 1], [a_1 \dots a_2 - 1], \dots [a_{k-1} \dots a_k - 1], [a_k \dots \infty]$.
int	The value is either a precise integer or $\top$.
τ array	The value is an array of values of abstracted type τ , where $\tau \in \{\text{int}, \text{bool}, \text{pointer}, \text{part}(a_1, a_2, \dots, a_k)\}$.
free	The abstraction is chosen dynamically (assuming the abstraction of the latest assigned value)
bool	Values are true , false , maybe

To understand how an abstraction determines the extracted model, consider the example program shown in Figure 4.6 (A). In the example, the variable **ch** is read from input, and at the state (state 1) right after the execution of **getc** statement, no conclusion can be made on **ch**'s value. Conventional model checking will enumerate all integer values. However, only the values 0, 97, 98 for **ch** affect the state switch in the program, while all other values are not important. Thus, **ch** can be abstracted using a partition abstraction as $\{\langle -\infty, -1 \rangle, \langle 0, 96 \rangle, \langle 97, 97 \rangle, \langle 98, 98 \rangle, \langle 99, +\infty \rangle\}$. The resulting LTS is shown in Figure 4.6 (B).

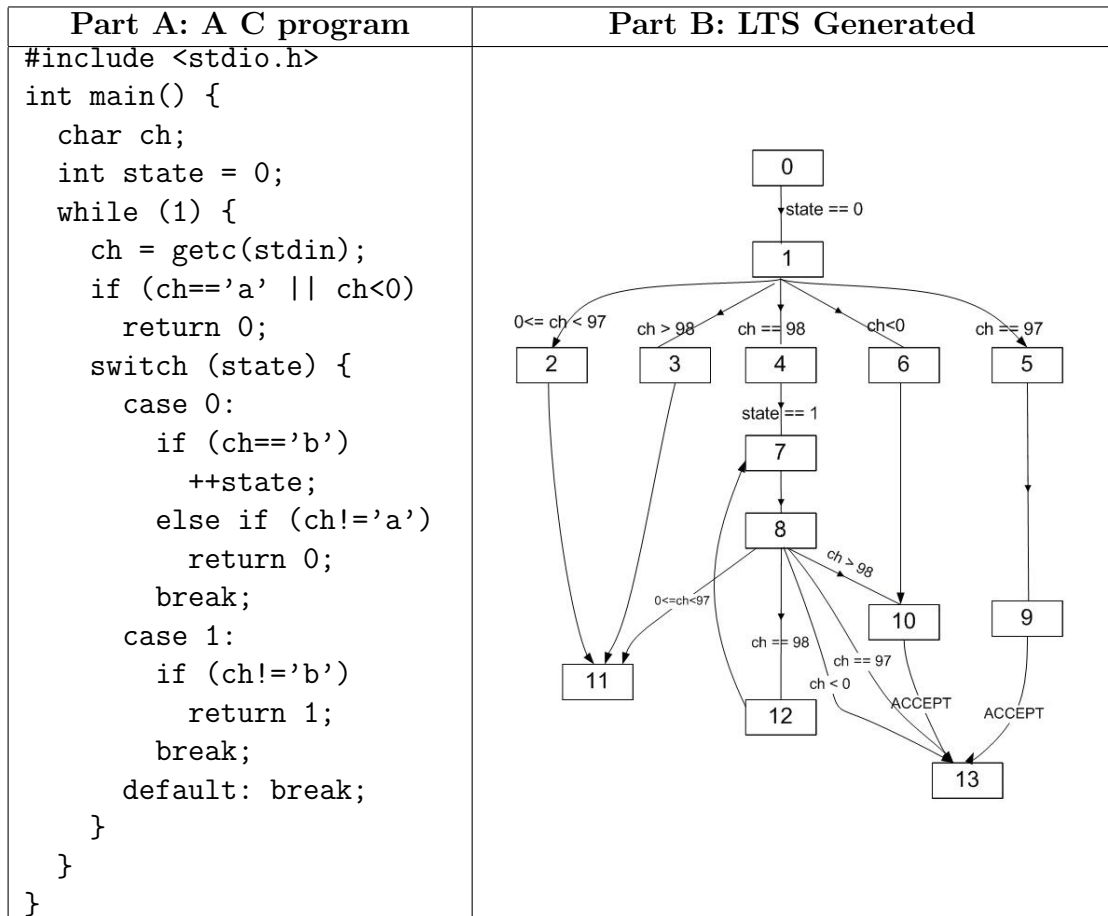


Figure 4.6: The effect of data abstraction on an LTS

4.3.2.2 The Graph Builder

The graph builder implements a depth first search algorithm to identify the dependencies and hierarchical relationships between blocks of the LTS generated by CWMB. Blocks are classified according to their functionality — those used to invoke user-defined functions are labeled as **FunctionCalls**, while all others are labeled simply as **Nodes**. The depth first search algorithm traverses each branch in the LTS, adding each **Node** encountered to the current (parent) **FunctionCall** block. If a new **FunctionCall** is encountered along the branch, the parent block gets pushed onto a stack and the internal structure for the new **FunctionCall** is determined. Once done, the new block is popped from the stack and added to its parent, thus allowing

```

<!ELEMENT Graph      (FunctionCall+, Edge+)>
<!ELEMENT FunctionCall (Id, FName, Level, Component+)>
<!ELEMENT Component  (Id, Level, (FunctionCall | Node)+)>
<!ELEMENT Node       (Id, Level, (LineNumber, Statement)+)>
<!ELEMENT Id         (#PCDATA)>
<!ELEMENT FName      (#PCDATA)>
<!ELEMENT Level      (#PCDATA)>
<!ELEMENT LineNumber (#PCDATA)>
<!ELEMENT Statement  (#PCDATA)>
<!ELEMENT Edge       (Source, Destination, Label?)>
<!ELEMENT Source     (#PCDATA)>
<!ELEMENT Destination (#PCDATA)>
<!ELEMENT Label      (#PCDATA)>

```

Figure 4.7: The DTD for the Graph File

for arbitrary nesting of functions.

A set of blocks within each **FunctionCall** that form a strongly connected component is aggregated and labeled as a **Component**. A **Component** thus represents a recursive (or mutually recursive) control flow in the slice. A **Component** can consist of **Nodes** as well as **FunctionCalls**. Control flow in the graph is represented by **Edges** between the **FunctionCalls** (and corresponding edges between the descendant **Components** and **Nodes**).

The graph structure is stored in an XML file that can be read and parsed by the user interface. The Document Type Definition (DTD) for the XML graph file is given in Figure 4.7. The DTD defines the layout of the graph file and specifies the format for storing the block hierarchy and edges between blocks in the graph.

4.3.3 Graph Visualization

The CAdS user interface is used to display the XML graph generated and facilitate a bottom-up understanding of the slice. The interface makes use of the Java JGraph API [BH02] to visualize control dependencies and hierarchies in the graph. The initial representation depicts a top-level dependency graph between the **FunctionCalls**

in the slice. Each `FunctionCall` can be expanded to view its internal structure, consisting of `Components`, `Nodes` and nested `FunctionCalls`. The blocks in view are arranged using a Tree layout algorithm [Bra96]. Each block in the graph corresponds to a set of statements in the slice. These statements can be viewed by double-clicking on the block. The analyst can select one or more of these statements and add them to the slicing criterion, which is then passed to the slicer as the criterion for slicing the CWI file. See Figure 4.8 for an illustration.

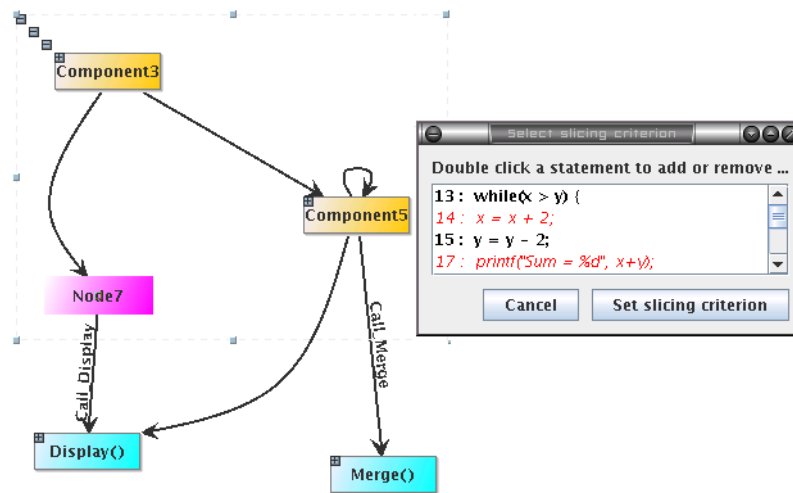


Figure 4.8: A Partially Expanded Graph in CAdS

4.3.4 Slicing the CWI File in CAdS

The CWI file essentially represents a network of basic block elements in the program (a basic block is a linear sequence of program statements with one entry point and possibly multiple exits). Slicing this file, therefore, is semantically the same as slicing the C program itself, with the added advantage that the slice generated need not be compiled again. This makes it possible to slice the CWI file recursively, i.e., define slices on already sliced CWI files — an important aspect of the post-mortem analysis process.

The CWI slicer uses a modified version of the Horwitz-Reps-Binkley algorithm [HRB88] to evaluate slices. A System Dependency Graph (SDG) is used to define

the data-flow and control-flow dependencies between the basic blocks. The control dependencies between basic blocks are computed following the approach given in [FOW87]. Control dependencies between statements are easily derived from this, since all statements in one basic block depend on the same predecessor, given the fact that there is only one entrance for each block.

To determine the data-dependency subgraph, the CWI slicer maintains a summary for each function in the program. A function summary consists of information about global variables the function defines and uses, together with formal parameters whose content might be modified during execution of the function. Function summaries are derived using a working list algorithm that determines the intra-procedural data-dependency calculations for each function. During this calculation, whenever a statement calling some function is encountered, the current summary of the callee function is fetched and used to update the def-use information for this statement. Upon completion of the data-dependency calculation, the function is popped from the working list, and a new summary is obtained for it. If this new function summary is not consistent with the old one, all functions that call this function are appended to the end of the working list. The process continues till the working list is empty. To ensure the correctness of the intra-procedural data-flow analysis, a **start** block is added to the beginning of each function graph, defining all global variables and formal parameters. Similarly, a **stop** block using all global variables and return registers is appended at the end of the function graph.

Once the SDG is constructed, it is combined with the function call graph. This includes adding a **call** edge from all call sites to the **start** blocks of callee functions, and adding **return** edges from the immediate successors of call sites to the **stop** blocks of callee functions, provided the callee functions modify some global variables or formal parameters. Computing the slice then corresponds to traversing backwards through the nodes in the dependency graph. The slice is evaluated in two phases, as in the Horwitz-Reps-Binkley algorithm. Given a slicing criteria (x, V) , phase one marks all vertices in the graph that are backwards reachable from x through **call** edges and data- or control-dependency edges. Phase two corresponds to marking all vertices that are backwards reachable from x through **return** edges and data-

or control-dependency edges. Finally, unmarked statements are removed from each basic block, and the CWI file is updated by deleting unreachable blocks.

4.3.5 Abstraction Map Generation

The size of state space and level of detail provided by the LTS depends on the abstraction criterion specified. When using manual abstraction and slicing (semi-automated abstraction-based slicing), this definition of the criterion is specified by users, resulting in an overhead in effort if the generated slice is large. The CAdS tool solves this by automatically generating abstraction criteria by collecting relevant data values for ‘interesting’ variables in the slice and associating these variables with partition abstractions customized by data values collected. This task is achieved through the abstraction map generator component in the tool.

A variable is considered interesting if it is impacted by the slicing criterion or it is located in conditionals in the slice. More formally, the set of interesting variables, $V(AM_i)$ in the i^{th} iteration can be defined as

$$V(AM_i) = V(AM_{i-1}) \cup V(SC_i) \cup V_{conditionals}(Slice_i)$$

where,

$V(AM_{i-1})$ is the set of variables used in the abstraction map for the previous iteration (the abstraction map for the initial iteration is empty), $V(SC_i)$ is the set of variables in the slicing criterion for the current slice, and $V_{conditionals}(Slice_i)$ is the set of variables in conditionals for the current slice. Conditionals include the predicates used in if-then-else statements, while loops, and switch constructs. Conditional variables are included as the evaluation of these variables directly influences the programs behavior.

Although it might be possible that some variable $x \in V(AM_{i-1})$ from a previous iteration does not exist in current slice, its inclusion in $V(AM_i)$ would not affect the model construction. Instead, its existence in the current abstraction map would make the collection of relevant data values terminate in shorter time and with more precise results.

For example, for the following switch statement

```
switch (x) {
  case 10:
    do_something(); break;
  case 20:
    do_something_else(); break;
}
if (y > x+1) ...
```

the relevant values for x here would be computed as 10 and 20, and the partition would be defined as $\{(-\infty, 9], [10, 10], [11, 19], [20, 20], [21, +\infty)\}$.

Another set of partition values would be obtained from conditionals in current slice that are in the format of Presburger arithmetic formulae. In the above example, if we already have 10 and 20 as relevant values for x , then this information can be passed through and used to derivate key values for y that exist in the conditional $y > x + 1$. After a simple computation, 11 and 21 would be obtained as the partition values, and the new partition for y would be $\{(-\infty, 10], [11, 11], [12, 20], [21, 21], [22, +\infty)\}$.

With these partition abstractions for y , we can obtain more precise result on the truth-value of the conditional $y > x + 1$ during the model construction. The calculation of relevant values is accomplished by symbolically executing the current slice. The process terminates when the execution converges, i.e. no new data value is added for any variable in the interest variable set. For those variables not in the set, abstractions are either assigned by the user or by default set as **free**.

4.4 Example: Tracing Failure Report to Software

The abstraction-driven slicing approach is used to analyze two medical device systems: a prototype implementation of the GIP system, and the XiO teletherapy planning software made available to us by CMS Inc. Both systems are analyzed using the semi-automated approach (using CWolf and the Codesurfer slicing tool), and with the help of the CAdS forensic analysis assistant. Results of the analyses are recorded in Chapter 6.

The GIP system was introduced in Chapter 3. For forensic analysis, we use a prototype of a PCA pump software implementation based on the GIP specifications. Post-mortem analysis is used to trace errors found through validation against usage scenarios during premarket analysis.

The second study involved post-mortem analysis of the XiO radiation therapy planning software. XiO is a radiation therapy planning system used to develop treatment plans for cancer patients, and provides a graphical environment with a combination of drop down file menus and icons to provide access to planning functions.

The post-mortem analysis of the XiO software is carried out on the basis of failure reports received from the field. The failure reports, provided by the manufacturer, have been collected during everyday real-life use of the software. In this section, we illustrate the use of abstraction-driven slicing by using the approach to analyze one such failure report for the XiO system

For the purpose of evaluating our approach, we limit ourselves to the software interface for the teletherapy module, and the errors detected therein. The module considered is approximately 40K lines of C code. The observed fault is reported in the form of a failure report, indicating what went wrong during the operation of the radiation therapy machine. The form and nature of the report is typical to those submitted to regulators and maintenance engineers during forensic analysis.

The text of the XiO failure report is reproduced below:

If a plan has a photon beam and an electron beam, both with ports, and the electron beam is active, selecting Port -> Convert aperture does nothing. It should prompt for the beam on which to do the conversion, i.e., the photon beam.

To trace the fault to the source of the error in the software, the following sequence of steps is carried out:

1. Using a (high-level) control flow graph for the source and prior knowledge of the code (through inspection), it is deduced that the functions corresponding to the Port menu are invoked from the function `DoPortMnu()`.

2. The `DoPortMnu()` function essentially consists of a giant switch statement, corresponding to each menu item in the Port interface. An abstraction with respect to the function is taken to get this understanding. A portion of the abstracted model is presented in Figure 4.9.

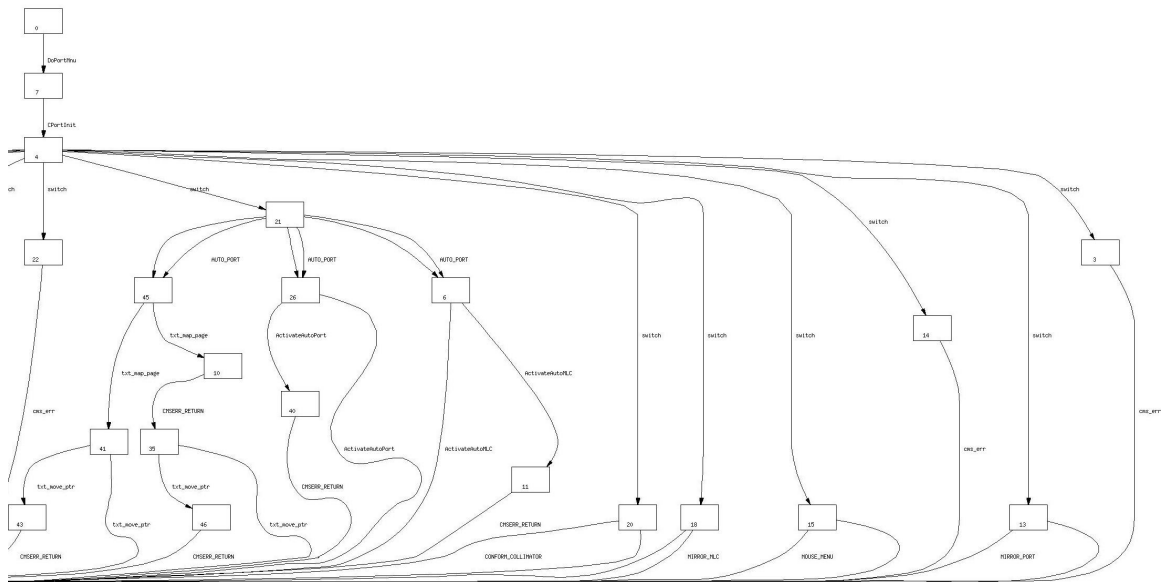


Figure 4.9: Portion of an abstract model generated for the XiO software.

3. A successor slice on case `ePMN_CONVERT_M` shows that the `void` function `GetBeamForPortUse()` is invoked. A forward slice for this function is taken, but is found to be too large. So, another high-level abstract model is generated, this time with respect to the function `GetBeamForPortUse()`.
4. Using the new abstract model, it is discovered that the `void` function `InitTelePortPulldowns()` actively deals with the beam (more specifically, the modality of the beam). A chop between the two functions confirms this.
5. A chop between the two variables `ePHOTON` and `cur_beam` shows that the boolean variable `plan_has_photon_nonseg_port_or_mlc` is set if the current beam is a photon beam. This variable is then used to enable (or disable) the `Convert aperture` menu item. Since this variable is checked for every beam on the plan, it would be `true` if the plan has a combination of electron and photon beams.

6. A forward slice on `plan_has_photon_nonseg_port_or_mlc` shows that the `Convert` aperture menu item would be enabled, but not active (in display mode only) if a photon beam is present (either by itself or with an electron beam). Consequently, when the electron beam is active, the system is fooled into assuming that the photon beam is active as well, and does not prompt for selecting a new beam. Also, since the current beam is electron, it does not perform the required conversion. Figure 4.10 shows part of the code relevant to the slice. Shaded portions indicate statements included in the slice. Statements in the slice relevant to the variables `plan_has_photon_nonseg_port_or_mlc` and `ePHOTON` are underlined.

```

void InitTelePortPulldowns (void) {
  Boolean_t    plan_has_beams; /* True if the plan has beams. */
  int          group;         /* The port menu group */
  beamll_t     *bm;           /* active beam data */
  ...
  for (bm = g_cport_ex_info.pd->plan->cur_sub_p->subp.tel->beam_list;
       bm != (beamll_t *)NULL ; bm = bm->next )
  {
    /*_ Check for a customized port. _*/
    if( bm->beam.trtmtaids.port_flag == TRUE &&
        bm->beam.trtmtaids.cust_port != (port_t *)NULL)
    {
      only_has_seg_mlc = FALSE;
      /* Need to check port_list to see if port shape is defined*/
      if( bm->beam.trtmtaids.cust_port->port_list!=(portll_t *)NULL)
      {
        plan_has_port = TRUE;
        plan_has_nonseg_port_or_mlc = TRUE;
        if( bm->beam.modality == ePHOTON )
          plan has photon nonseg port or mlc = TRUE;
      }
    }
  }
  ...
}

```

Figure 4.10: Result of a forward slice on `plan_has_photon_nonseg_port_or_mlc`

The error is traced in a similar fashion with the help of the CAdS tool. In this case, the LTS generated by CWolf are replaced by the hierarchical graphs produced by CAdS. A snapshot of the graph generated is shown in Figure 4.11. The graph shows a `Node` being invoked as a result of the call to the `DoPortMnu()` function, and

the C code corresponding to the **Node**. Further, line 94 of the code is used as part of the slicing criterion for the next iteration in the analysis. This criterion is used to slice the CWI file, and to obtain the abstraction criterion (AM and LM files) to generate the LTS (and graph) corresponding to the sliced CWI file. The hierarchical nature of the graph allows users to navigate through only relevant parts (FunctionCalls and Nodes) of the model. Other **FunctionCalls** and **Nodes** not pertinent to the analysis are left unexplored.

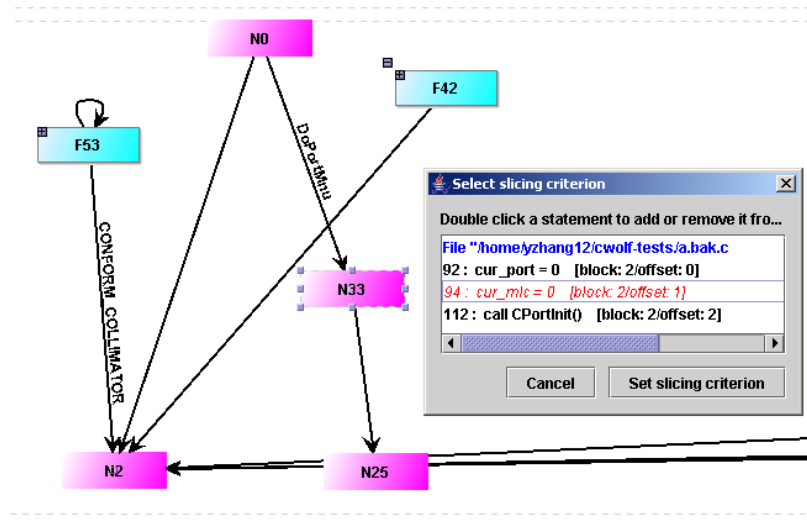


Figure 4.11: An example of error tracing using the CAdS tool

The slicing in CAdS is limited to computing static backward slices only, since CAdS does not currently support forward slices or chops. This limits the scope of the analysis to an extent. However, the graph constructed by CAdS itself aids comprehension by virtue of displaying concrete code associated with each abstract block.

Chapter 5

Putting it Together: Software Simulation Checking

Forensic analysis through the abstraction-driven slicing approach provides a regulator with the ability to trace software-related failures to their source. Once the cause of the error is ascertained, the device may be recalled, or punitive charges levied against the manufacturer. More importantly, the error can be fixed appropriately, and preventive actions taken to avoid such malfunctions in the future.

However, potential problems could arise from fixing of the error itself. All too often a patch in the software that addresses a certain error could result in other bugs elsewhere in the code. Such errors, often referred to as regression bugs, are a major concern for safety-critical systems. A noteworthy statistic in this respect is that of the 242 software-related recalls conducted by the US FDA between 1992 and 1998, 192 (or 79%) were due to defects that were introduced when changes were made to the software after its initial production and distribution [fda02].

Regression testing techniques [GHK⁺01] are typically used to detect such bugs. In our case too, most of the newly introduced errors can be detected with the help of exhaustive testing, and by evaluating the updated code against usage scenarios for

the software. However, test cases and usage scenarios may not always be exhaustive enough to uncover these errors, and the bug as a result would remain undetected in the software till the next failure event. Moreover, there may not exist detailed usage models (and correspondingly usage scenarios) for certain software categories, making it all the more imperative to have a means for assuring the integrity of the updated software. For systems that do have usage models defined, it is similarly necessary to ensure that updating these models does not affect the intended semantics of the system.

To provide these assurances, we make use of the notion of semantic preorder checking. In particular, we investigate simulation checking to assess whether updated software implementations preserve the semantics of the original code and usage models respectively.

In Section 5.1 of this chapter we introduce the concept of simulation preorder and give a scheme for detecting regression errors by checking for preservation of behavior in updated software and usage models. In Section 5.2 we illustrate the scheme using the GIP system as an example.

5.1 Assessing Impact of Change in Software

A major role of semantic definitions is to identify systems that are behaviorally equivalent. One particular notion that has gained wide acceptance is the notion of bisimulation [Mil89], which characterizes when two systems are equivalent, i.e., when one can be replaced by another in any context without changing the overall system. However, when evaluating changes made to a system, we compare ‘similar’ (as opposed to equivalent) systems. In order to do this, we use the slightly weaker, but more practical concept of simulation checking [vG90]. A process P ‘simulates’ a process Q provided P can do everything that Q can do, and possibly more.

Formally, a model M_1 is said to be simulated by model M_2 if all possible execution traces in M_1 can be generated using M_2 as well. For two Kripke structures $M_1 = (S_1, \Sigma_1, \delta_1)$ and $M_2 = (S_2, \Sigma_2, \delta_2)$, a relation $\rho \subseteq S_1 \times S_2$ is a simulation if whenever

$$(p, q) \in \rho, \text{ then for each } \alpha \text{ in } \Sigma_1:$$

$$\forall p' \text{ with } p \xrightarrow{\alpha} p', \exists q' \text{ with } q \xrightarrow{\alpha} q' \text{ and } (p', q') \in \rho$$

where,

S_1 is the state space for M_1 ,

S_2 is the state space for M_2 ,

Σ_1 denotes the set of actions (event alphabet) for M_1 ,

Σ_2 denotes the set of actions (event alphabet) for M_2 ,

$\delta_1 \subseteq S_1 \times S_1$ is the transition relation for M_1 ,

$\delta_2 \subseteq S_2 \times S_2$ is the transition relation for M_2 ,

$p, p' \in S_1$, and $q, q' \in S_2$.

A **simulation preorder** is the largest simulation relation, denoted by $\sqsubseteq$. A **simulation equivalence** $\simeq$ is defined as $\sqsubseteq \cap \sqsupseteq$.

5.1.1 Using Simulation Checking to Compare Models

To check that an updated system S' preserves the semantics of the original system S , we basically need to test whether S' can simulate S . Thus, to check whether an updated program p' preserves the semantics of an original program p , we compare abstract models M_p and $M_{p'}$ for programs p and p' respectively. If $M_p \not\sqsubseteq M_{p'}$, it follows that the updated program p' does not preserve all properties of the original program p . If on the other hand, $M_{p'}$ does preserve the semantics of M_p , i.e., $M_p \sqsubseteq M_{p'}$, then p' may or may not preserve p . In this case, the updated program p' would need to be re-validated against test cases (or usage scenarios) to check for compliance.

Figure 5.1 shows a general scheme for detecting revised code for regression errors. The process, as shown in the figure, starts by abstracting a set of models, M_p for a program p using abstraction criteria C . Once the program p is modified to fix a reported bug B , the same criteria C is used to abstract a corresponding set of revised models $M_{p'}$ for the revised program p' . The criteria C would ideally consist of all variables and data structures directly affected by the bug fix.

A simulation preorder is then used to check whether $M_{p'}$ preserves M_p through

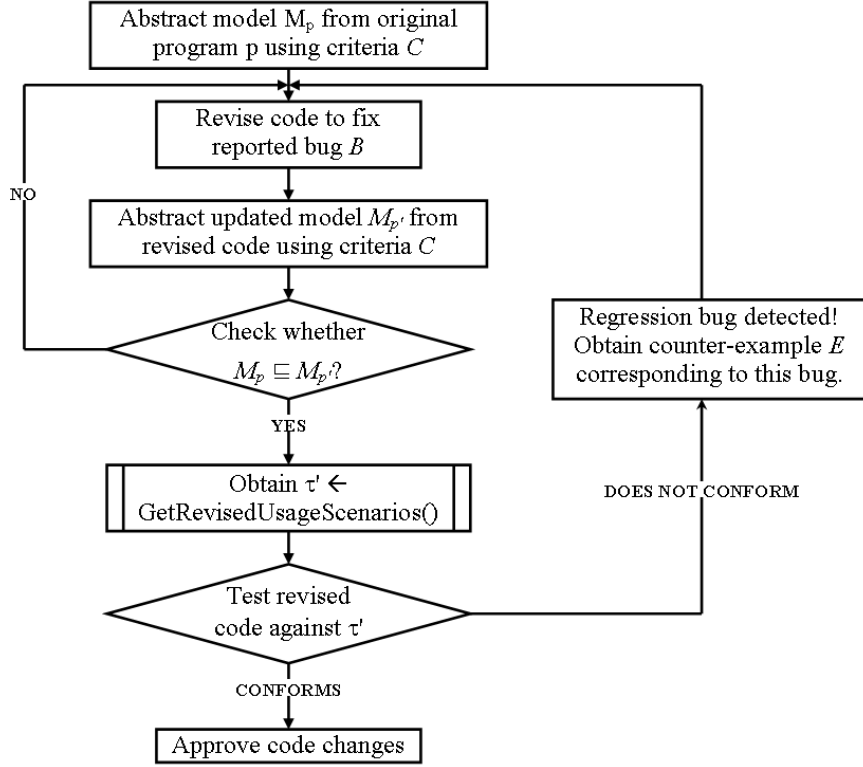


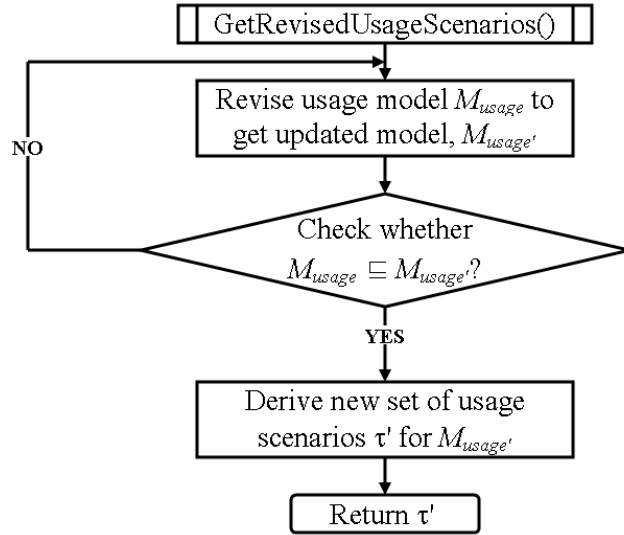
Figure 5.1: Detecting regression errors in revised code

a simulation relation (i.e., if $M_p \subseteq M_{p'}$). If $M_p \not\subseteq M_{p'}$, it indicates the presence of a regression bug that needs to be fixed. A trace or counter-example E is obtained indicating an execution sequence in M_p that is not present in $M_{p'}$. This counter-example can then be used to detect exactly how the two models differ, thereby helping to trace the regression bug in program p' . Once all such bugs are removed, and $M_p \subseteq M_{p'}$ is successfully established, p' is further tested by evaluating it against all usage scenarios, τ derived from the usage model.

If the original program p was tested against τ prior to its release, the detection of bug B would indicate that the existing scenarios τ do not address all hazardous conditions and new scenarios need to be generated to address the reported error. Since the scenarios are derived from usage models, the model M_{usage} itself may need to be updated to incorporate the new safety conditions before new usage scenarios are derived from it. Figure 5.2 depicts the procedure for obtaining a set of revised

usage scenarios τ' from the updated model $M_{usage'}$.

To ensure that the updated usage model is consistent with the original model, it is compared against the original using a simulation preorder. As shown in Figure 5.2, the usage model M_{usage} is updated to reflect the change in the software's behavior. Next, a preorder relation is used to check behavioral equivalence of the updated model against the original (i.e., to check if $M_{usage} \sqsubseteq M_{usage'}$). If $M_{usage'}$ is found not to simulate M_{usage} , it indicates that the updated model does not satisfy properties met by M_{usage} and needs to be revised again to correct this anomaly. If on the other hand $M_{usage'}$ does simulate M_{usage} , it is used to derive a new set of usage scenarios, τ' using which the revised code can be validated.



In case the usage model (and correspondingly the set of usage scenarios) for the software is not available, the code is tested solely against the manufacturer's test cases. If after this testing, no further errors are detected in the code, the change is approved and the modified code released accordingly.

5.1.2 Using CWB-NC for Semantic Checking

To compare behavior of different models, abstracted or usage, we use the Concurrency Workbench of the New Century (CWB-NC). The CWB-NC is an automated behavioral relation based verification tool that supports a number of modeling languages including CCS, CSP and LOTOS. The models once defined (or translated) in any of these languages can then be checked for semantic equivalence or preorder using predefined commands in the workbench.

Simulation preorder of models is checked in CWB-NC by the `le` command. The syntax for the `le` command is given as

```
le [-S semantic-type] model1 model2
```

`le` computes whether *model₁* and *model₂* are related by the behavioral preorder indicated by `semantic-type`. The currently available semantic-type arguments are `may` (may-preorder) and `must` (must-preorder).

When two models are not `may` equivalent (or not related by the `may` preorder), the CWB-NC returns an action trace which one may perform but the other may not. Correspondingly, when two models are not `must` equivalent (or not related by the `must` preorder), the CWB-NC returns a trace sequence that one must pass and the other may fail. The returned traces are simply space delimited sequence of actions.

The following example shows the use of traces as diagnostic information:

```
cwb-nc> eq -S may 'a.b.nil + a.c.nil' 'a.b.nil'
FALSE...
a.b.nil + a.c.nil has trace:
  a c
a.b.nil does not.
```

As shown by the example, the two CCS definitions `a.b.nil + a.c.nil` and `a.b.nil` are not `may` equivalent since the trace `a c` is satisfied by the former, but not the latter.

5.2 Simulation Checking for the GIP System

The scheme for detecting regression errors through simulation preorder checking, given in the previous section is used to assess the effect of changes in the prototype PCA implementation. The process is integrated as part of the forensic analysis and testing of the prototype.

The PCA prototype, based on the GIP specifications, is implemented in the C language, and implements the various situations and events defined in the GIP model. Supplemental code relating to specific pump functionality is added to facilitate source compilation and to ensure a more complete implementation. In all, the code comprises of 23 files, consisting of approximately 20,000 lines of code. The prototype is tested against the set of usage scenarios derived from the GIP mode, as explained in Chapter 3. In all, forty-one errors have been detected during this process.

The bugs in the code are subsequently fixed and models abstracted from revised code are checked against models abstracted from the original code to evaluate whether the semantics for the original are preserved by the new model. Figure 5.3 shows this being checked in CWB-NC.

```

cwb-nc> load test05.ccs
Execution time (user,system,gc,real):(0.931,0.000,0.160,0.931)
cwb-nc>
cwb-nc>
cwb-nc> le -S may Impl Impl2
Building automaton...
.....1000.....2000.....3000.....4000.....5000
.....6000...
States: 6393
Transitions: 8619
Done building automaton.
Building automaton...
.....1000.....2000.....3000.....4000.....5000
.....6000...
States: 5805
Transitions: 8039
Done building automaton.
Transforming automaton...
Done transforming automaton.
TRUE
Execution time (user,system,gc,real):(1.062,0.000,0.230,1.062)
cwb-nc>

```

Figure 5.3: Using CWB-NC to assess changes in modified software

As shown in Figure 5.3, The CCS process `Impl` describes the abstract model for the

original program (M_p), while process `Impl2` is the model abstracted from the program after fixing the bug ($M_{p'}$). The same abstraction criterion is used to generate both `Impl1` and `Impl2`. The two processes are checked using the `may` preorder. As shown in the figure, `Impl2` successfully simulates `Impl1`. This does not prove that the revised code is correct however; to ensure that, the code must be tested against the new usage scenarios.

To obtain the updated set of usage scenarios however, the GIP usage model itself needs to be changed. Once modified, it is checked against the original model to ensure that the semantics of the original are preserved (or in this case, that all safety conditions met by the original are satisfied). Figure 5.4 shows this check being performed in CWB-NC.

```
cwb-nc> load gip.ccs
Execution time (user,system,gc,real):(0.541,0.000,0.060,0.541)
cwb-nc>
cwb-nc> le -S may Spec1 Spec2
Building automaton...
States: 292
Transitions: 3369
Done building automaton.
Building automaton...
States: 292
Transitions: 3370
Done building automaton.
Transforming automaton...
Done transforming automaton.
FALSE...
Spec1 has trace:
ON MALFUNCTION OFF VIALcorr
Spec2 does not.
Execution time (user,system,gc,real):(17.896,0.000,5.308,17.896)
cwb-nc>
cwb-nc>
cwb-nc> le -S may Spec1 Spec3
Building automaton...
States: 292
Transitions: 3369
Done building automaton.
Building automaton...
States: 292
Transitions: 3370
Done building automaton.
Transforming automaton...
Done transforming automaton.
TRUE
Execution time (user,system,gc,real):(19.137,0.000,5.009,19.137)
```

Figure 5.4: Comparing an updated model to the original using Simulation checking

The specification shown in Figure 5.4 consists of three processes. **Spec1** represents the original usage model for the GIP. **Spec2** is an incorrectly modified version of **Spec1**. The process **Spec3** represents a correctly revised model for GIP. A **may** preorder between **Spec1** and **Spec2** indicates that **Spec2** does not simulate **Spec1**, and produces a trace that can be satisfied by **Spec1**, but not **Spec2**. A similar **may** preorder between **Spec1** and **Spec3** shows that all paths in **Spec1** can be traced by **Spec3**, and thus it can be successfully simulated.

Once the updated model is ascertained to be consistent with the original, it is used to derive the set of new usage scenarios. The PCA prototype is finally re-validated using these new usage scenarios to ensure compliance with the updated GIP model.

Chapter 6

Results and Observations

In this chapter, we list the results obtained from the three studies used through this thesis. We use the results from the GIP, CARA, and XiO studies respectively to illustrate the viability and usefulness of the premarket and postmarket review strategies introduced.

The GIP example is one study among the three that spans the entire spectrum of the concepts introduced in this thesis; i.e., it deals with defining usage models, verification of these models, validation against usage scenarios, and post-mortem analysis to fix errors detected. The CARA system is used only during the premarket conformance review, while the XiO software is used for postmarket forensic analysis.

We thus list the results for both premarket and postmarket analyses of GIP in Section 6.1. In Section 6.2 we discuss results of formal verification and testing for the CARA system. In Section 6.3 we list the results from the forensic analysis of the XiO radiation therapy planning system. An assessment of the observed results is discussed in Section 6.4.

6.1 Results from the Generic Infusion Pump Study

6.1.1 Verification of the GIP Usage Model

The GIP usage model, as explained in Chapter 3, was verified using the Reactis and Uppaal tools. Safety properties were specified for the system and verified against the model using LTL formulae, assertions and user-defined targets.

Five errors were uncovered during this verification. The errors dealt with either anomalies in the system specifications or errors in the model itself. Importantly, we were able to ensure that this list of errors was comprehensive, and that the model was correct for all other cases (with respect to the safety conditions considered). The errors detected were classified in the following three categories, with category 1 errors being the most severe and category 3 errors the least severe:

1. **Null probability.** One error was detected due to a null probability assigned to a valid transition. This meant that this event would never be executed, despite there being an active transition between the respective source and destination states.
2. **Missing transitions.** There was one error due to a missing transition for a defined probability value. This could lead to a potential deadlock whenever the event with the associated probability would be executed.
3. **Missing self-transition probabilities.** There were three errors due to missing self-transition probabilities. These errors were more subtle, since they did not result in invalid transitions or unreachable states. They did, however, translate to exaggerated probabilities for other transitions originating at the given state, indicating an anomaly in the requirements. Such anomalies could lead to incorrect system behavior in the case of internally generated events such as timeouts at 1 hour and 4 hour intervals.

6.1.2 Testing a GIP-based Implementation

The specifications for the GIP system were used to implement a prototype software for a PCA infusion pump. The prototype, developed at CDRH, was based on the requirement specifications provided for the GIP and extended to incorporate specific features for a typical PCA pump. A test suite (or set of usage scenarios) for this prototype was derived from the usage model for the GIP. The test suite consisted of approximately 39,000 usage scenarios that were then used to validate the PCA prototype. A total of **forty-one** errors were detected using the test suite for this implementation.

The validation was performed in a black box manner, with expected responses for usage scenarios compared against the corresponding observed responses for the prototype. The outputs from each were compared to detect if there were any discrepancies. Figure 6.1 shows portion of a typical scenario for an error detected. The expected and observed responses differ in the column highlighted, indicating that when the dose is changed incorrectly (at $t = 59$), the alarm is not triggered as it is supposed to.

Sequence (t)	...	54	55	56	57	58	59	60	61	...
Expected		I006:R108	I011:R001	I005:R001	I002:R102	I008:R122	I006:R109	I113:R203	I002:R102	
Observed		I006:R108	I011:R001	I005:R001	I002:R102	I008:R122	I006:R115	I113:R115	I002:R102	

Figure 6.1: Detecting an error during testing for the GIP system

6.1.3 Forensic Analysis of the PCA Prototype

Thirty-one of the forty-one errors detected during testing of the prototype PCA implementation were traced to source code using the abstraction-driven slicing approach. To assess the efficiency and impact of the approach, the forensic analysis was carried out in three stages: in the first stage, errors were traced using only static slicing; in the second stage, semi-automated abstraction-driven slicing was used to trace the errors; finally the process was repeated using the CAdS forensic analysis

tool.

The three approaches were compared with respect to the number of iterations required to trace a failure to its source in the code. For the abstraction-driven approaches, this translates to the number of iterations required through Algorithm 2. For static slicing, the abstraction step of the algorithm is skipped.

As shown in Figure 6.2, the number of iterations required to trace an error to source code were significantly higher for static slicing when compared with those required during abstraction-driven slicing (both in the semi-automated mode and when using CAdS). The average number of iterations needed when using only static slicing was 4.39 per error. When using semi-automated abstraction and slicing, it reduced to 3.97. When using CAdS, this number dropped further to 3.77.

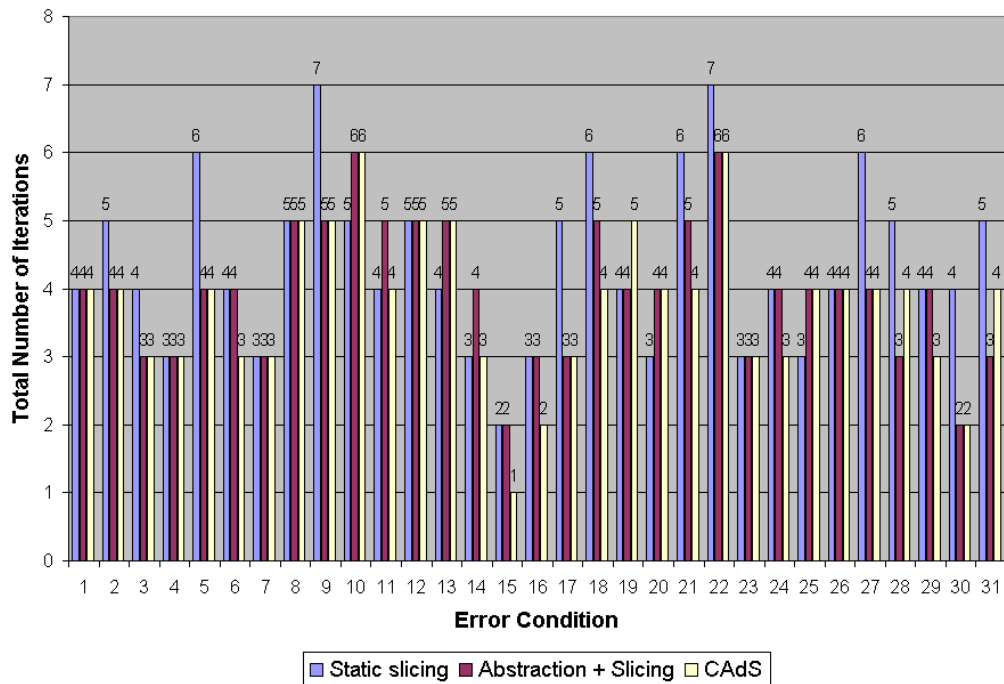


Figure 6.2: Number of iterations for forensic analysis of the PCA prototype

The reduction in the iterations for abstraction-driven slicing was complemented by a corresponding reduction in the size of the slice over each iteration. Figure 6.3 shows the comparison between the three approaches with respect to the average size

of the slices considered. As can be seen from the figure, the average size of successive slices kept reducing in each of the approaches, as the slicing criteria were refined. Level 1 in the figure indicates a top-level slice, while successive levels correspond to iteratively refined slices. As noted in the figure, the slices used in the abstraction-driven approaches were consistently smaller over successive slices.

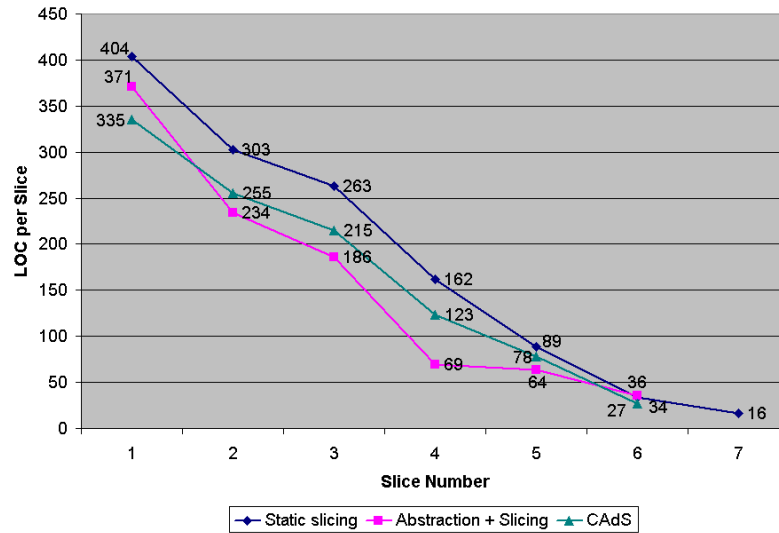


Figure 6.3: Successive slice sizes during forensic analysis of the PCA prototype

The size of slices was generally slightly larger when using CAdS as opposed to the semi-automated abstraction-driven slicing approach. This can be attributed to the fact that slicing in CAdS was limited to backward static slicing only (forward slices and chops are not yet supported by CAdS). However, CAdS used a hierarchical structure to display the models generated, making its graphs easier to navigate and understand, as opposed to LTSs generated by CWolf. Structuring blocks in levels allowed for navigation through only those blocks that were pertinent to the problem while ignoring blocks in other branches (or differing levels). Figure 6.4 illustrates the distribution of CAdS blocks over various levels for models corresponding to the PCA prototype. As shown in the figure, the bulk of the blocks were at levels 4 and higher, indicating that the states in the models were well distributed across different branches (and subbranches) in the graph.

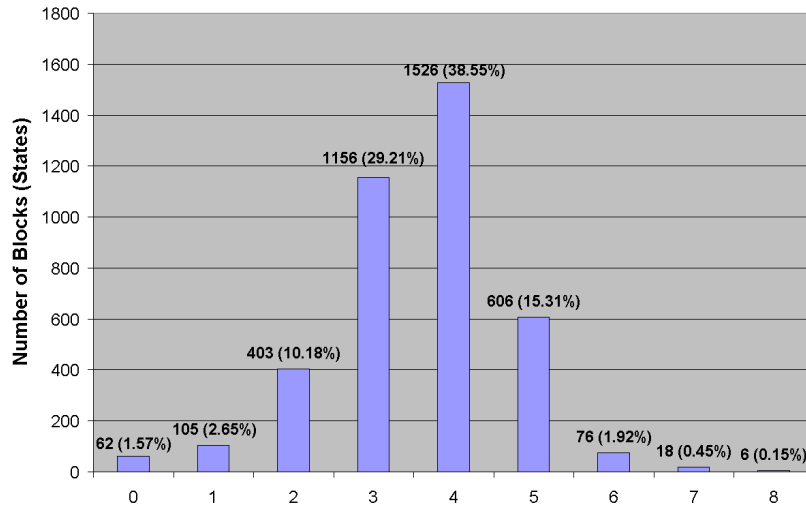


Figure 6.4: Distribution of blocks (states) per each level for CAdS

6.1.4 Assessing Change using Preorder Simulation

Once the programs for the prototype PCA implementation were modified to fix the errors detected during forensic analysis, abstract models for these programs were checked against models abstracted from the original code (using the same criteria for both abstractions). The new models were checked using a `may` preorder to ascertain whether they preserved properties of the original models. Only when the new models were proven to be similar to the original, were the programs re-validated against (updated) usage criteria for the GIP model.

To generate a new set of usage scenarios for re-validating the PCA prototype, the GIP usage model was revised, with the errors detected during verification and testing fixed accordingly. The revised model was then compared against the original usage model to ensure that all safety properties were still satisfied by the new model. The revised model was verified using Uppaal and Reactis to ensure this compliance, before being used to derive the updated set of usage scenarios for the PCA prototype.

Simulation preorders were also used to evaluate whether the granularity of the abstractions used in forensic analysis. The usefulness of an abstraction can be measured by its ability to produce distinguishable abstract models for subtle variations

in the source (i.e., in the concrete domain). We evaluated the abstraction mappings used during forensic analysis of the PCA implementation by comparing the models used to detect bugs in the source code with similar models generated from rectified code. The same abstraction mappings were used to generate both sets of models. `must` preorders were used to perform the simulation checking.

The corrections made to the software to mitigate the errors either introduced new events for existing structures, or modified existing properties for these structure. Using the notion of preorder checking provided by CWBNC, we were able to distinguish the corrected models from the original models in 24 of the 31, or 77%, of the corrections made. The changes that could not be traced to the new models involved redundant transitions and states that were never executed or which lay outside the scope of the abstract interpretation defined (e.g., modification of existing floating point values).

6.1.5 Effort Expended

The effort expended during the analysis of the GIP system is shown in Figure 6.5. The total effort expended during the premarket review phase, including effort spent on deriving usage scenarios and testing of the PCA prototype implementation was 410 person hours. The distribution for this effort is depicted in Figure 6.5(A).

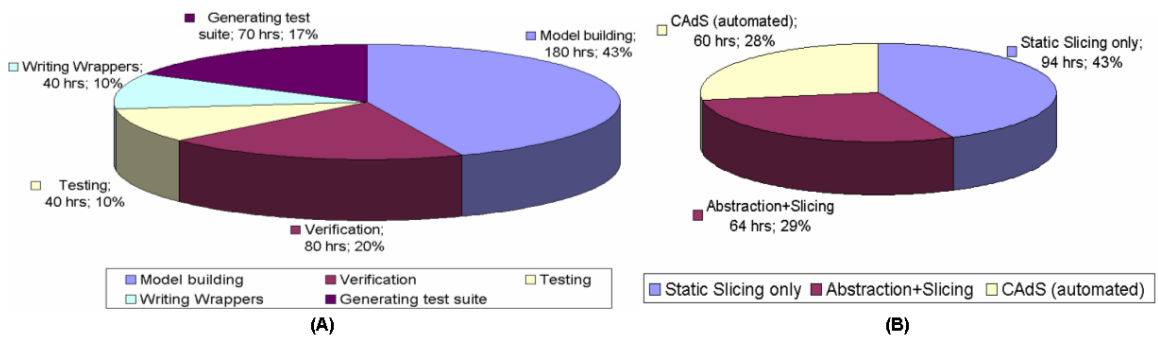


Figure 6.5: Effort distribution for (A): Premarket review and (B): Forensic analysis for the GIP study

As can be seen from the figure, the major effort was expended in defining the system specifications and developing the model itself. A further 19% of the effort was spent during verification of the model, contributing to 63% of the effort being spent in one-time activities. Only 37% of the effort (about 150 person hrs) was spent on implementation-specific activities of developing wrappers, extracting test suites, and the actual testing itself (including writing the test harnesses). Although this effort constitutes a little overhead for both regulators and the manufacturers, the extra effort is fully justified in view of the additional safety assurance provided.

The effort spent during forensic analysis of the PCA prototype was 218 person hours, shown in Figure 6.5(B). This included the effort required for analyses using static slicing, semi-automated abstraction-driven slicing, and the CAdS forensic analysis assistant. The savings in effort over pure static slicing was about 32% when using the semi-automated approach, and 36% when using CAdS.

6.2 Results from the Premarket Review for CARA

During the modeling and verification of the CARA system using Spin and Up-paal, there a number of errors were detected in the system specifications. The errors detected were settled by WRAIR on report, either by clarifying the ambiguity, or making suitable changes to the requirement documents. The errors were broadly classified into two categories:

1. Discrepancies found in the requirements. These correspond to conflicting requirements found within the system specifications provided.
2. Missing requirements or specifications that were ambiguous or provided insufficient information.

Apart from these errors, six minor inconsistencies detected in the specifications across the various documents provided. These inconsistencies were clarified by WRAIR upon request, usually during the modeling phase itself.

6.2.1 Discrepancies

There were **three** discrepancies detected in the system requirements. These are as given below:

1. The first discrepancy detected was related to cuff pressure readings. According to one specification, when cuff pressures are used, CARA should immediately initiate five readings at an interval of 5 minutes. However, another requirement gave a table for cuff pressure intervals ranging from 1 minutes to 10 minutes. This was in contradiction to the earlier specification. On reporting this error, a change was made to the requirements stating that the frequency table should dictate the number of cuff readings done at all times. References to the five specifically required initial cuff readings were removed.
2. The second discrepancy also involved cuff pressure readings. According to one specification, cuff pressure readings could be at an interval of as much as 10 minutes. However, another requirement stated that CARA must return to manual mode if no cuff pressure reading is detected in 3 minutes. When this issue was raised, we were told that the second requirement pertained only to cases where a primary blood pressure source (apart from cuff pressures) was available. When cuff pressures were the only source, then CARA was to revert to manual mode if two successive readings are missed.
3. The third discrepancy involved the set point value in the requirements document. All the values specified in the requirements, including a few lookup tables, assumed that the set point was always maintained at its default value of 70mm Hg. However, if the medic modified the set point, these values would need to be changed as well. This discrepancy was confirmed and rectified on report.

6.2.2 Missing Specifications

There were **four** instances during modeling where the specifications were found to be incomplete. These omissions were detected while modeling the CARA specifications and capturing safety properties for the model. A list of these missing specifications is given below:

1. Insufficient information was provided by the requirements document regarding the use of uncorroborated beat-to-beat sources. According to the requirements, each uncorroborated blood pressure reading was to be corroborated with the cuff pressure. However, it did not specify the action to be taken when the cuff pressures were not available. Clarifications were provided by WRAIR, and it was suggested that an arterial line could be used if the cuff pressures were not available.
2. The requirements document contained an alarm table specifying various alarm conditions, along with the priorities and silence times for each. However, the alarm table was found to be incomplete, as a number of alarm conditions and their respective priorities and silence times were not listed therein. A comprehensive alarm table was provided thereafter.
3. A similar error was noted in another requirement. The requirement stated that if the beat-to-beat signal were lost for more than 3 minutes, CARA would revert to using secondary blood pressure sources using appropriate quality control procedures. It was not clear what steps would be taken if no secondary sources were available. Also, the reference to the “appropriate quality procedures” was ambiguous. On report, it was suggested that CARA would revert to Manual mode in such cases.
4. One of the most important features of the CARA model was the feedback loop that controlled the flow rate of the fluids to be infused. However, the relation between the flow rate and back-EMF and the relation between back-EMF and the control voltage were not clearly delineated in any of the specifications. When

we reported this, we were told that this is a field that is still under study, and a certain amount of abstraction could be used to model these dependencies. Later on, a table was provided that gave a mapping between the control voltage and the flow rate. The relation between average blood pressure readings and the control voltage was assumed to be linear.

6.2.3 Testing the CARA Applet

As explained in Chapter 3, the CARA applet developed at SUNY, Stony Brook was tested against the usage sequences in the Uppaal and Spin formal models. Traces were obtained from the CARA applet, simulating the various events and inputs possible for the system. The traces were then used to simulate the exact same events in the usage model. Responses from the two (CARA applet and models) were noted and compared to detect errors, if any.

No new errors were detected during the testing of the CARA applet. The applet had been exhaustively verified at SUNY before its release. However, complete state and transition coverage for the traces was ensured, assuring that the CARA applet met all safety conditions defined.

6.2.4 Effort Expended for Reviewing CARA

A major part of the effort expended during the premarket review of CARA was spent in modeling and verification of the CARA usage model. The Uppaal and Spin tools were employed for this purpose.

The effort required for modeling the CARA system in Uppaal and Spin was also found to be comparable. Whereas the learning curve for Uppaal was less steep, it was easier to debug the model in Spin, owing to its superior simulation environment. Again, since the system was modeled in Uppaal before Spin, the basic structure of the design for Spin was already available. This resulted in a lesser amount of time required for modeling the system therein. Table 6.1 gives an account of the effort spent for review of the CARA system.

Table 6.1: Effort required for the CARA study

Activity	Time Spent
Review of the requirement specifications	60 person hours
Modeling CARA in Uppaal	120 person hours
Modeling CARA in Spin	200 person hours
Verification in Uppaal	80 person hours
Verification in Spin	60 person hours
Total Effort	520 person hours

From Table 6.1, the total effort required for the modeling and verification of the CARA system was calculated to be 520 person hours. This might seem to be a considerable amount on the face of it; however, the effort noted above was calculated taking into consideration the time taken to learn the two tools and getting familiarized with the system domain. Once comfortable with the nuances of the tool, the effort for such modeling and verification would be greatly reduced.

6.3 Results from the Forensic Analysis of XiO

The third study dealt with forensic analysis of the XiO radiation therapy planning software, a commercial software package developed by CMS Inc. Failures encountered during real-world operation of the software were recorded and traced to their sources in the code using forensic analysis techniques.

6.3.1 Evaluating the Approach

To assess the impact of forensic analysis for the XiO radiation therapy planning software, the analysis was carried in three phases, similar to the GIP study. In the first phase, we used only static program slicing to detect errors in the software. In the second phase, model abstraction was combined with slicing to perform the analysis using the semi-automated abstraction-driven slicing approach. In the third phase, the analysis was performed with the help of the CAdS forensic analysis assistant.

In all, **twenty-six** failure reports were analyzed using the two approaches, of which **thirteen** were traced successfully to their source.

The 50% success rate for the post-mortem analysis can to an extent be attributed to the fact that only partial code was available for analysis. The analysis was based on the `port` module of the XiO system, consisting of 40,000 lines of code. However, this module was only part of a larger system, with several other modules. Code for these modules was not available during forensic analysis. These modules could lead to (or impact) failures in the `port` module, but could not be traced without analyzing the source code for the same.

Forensic analysis for XiO was performed using an iterative approach. When using slicing alone, the analysis started off by considering a large static slice for a selected set of variables. This slice was evaluated and refined against a revised slicing criterion to get a new, smaller slice. The process produced smaller slices in each successive iteration, until the problem was detected, or the slice was found not to contain the error; in which case, a new slicing criterion was defined and examined.

When using abstraction-driven slicing (semi-automated and CAdS), a high-level model was generated for the entire program to get an overview of the program functionality. Using this knowledge, a slicing criterion was selected to obtain the initial slice. Subsequent iterations kept generating more specific models and refined slices in accordance with Algorithm 2 described in Chapter 4.

To compare the two approaches, we considered the following parameters against which the efficiency of the analysis was evaluated:

- the number of iterations required to trace each fault to source code,
- the average size of each slice per iteration, and
- the total effort expended to resolve each error.

Figure 6.6 shows the total number of iterations required to trace a fault in the XiO system to its source in the software. The iterations are shown for each of the failure conditions resolved with respect to the three analysis approaches. The average number of iterations required when using static slicing alone was 10.31 per error. For

the two abstraction-driven approaches, the number of iterations were lesser at 9.08 and 8.92 per error for the semi-automated approach and the CAdS tool respectively.

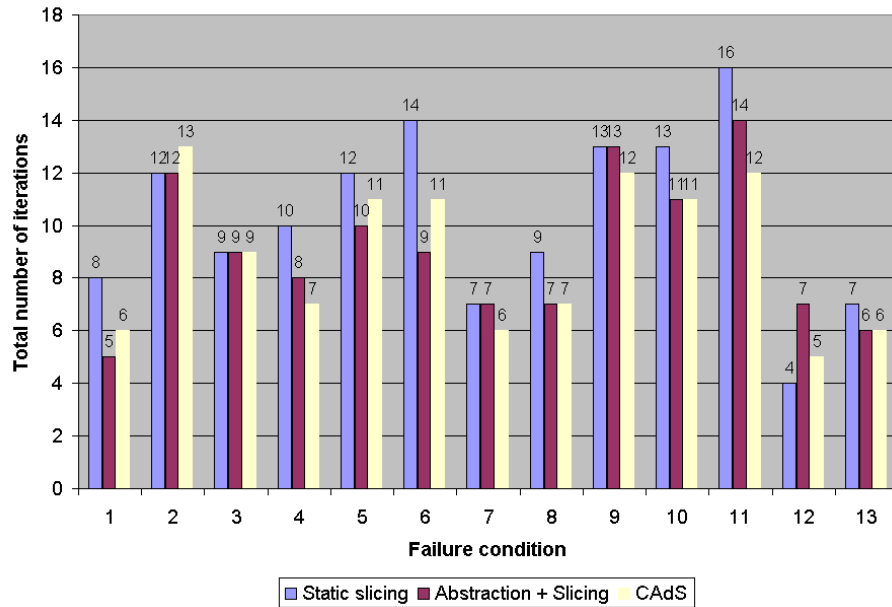


Figure 6.6: Number of iterations for forensic analysis of the XiO system

Figure 6.7 compares the average size of a slice in the three approaches. As in the GIP study, the slices obtained through static slicing were found to be consistently larger than those for the abstraction-driven approaches. The size of slices obtained using the semi-automated approach were slightly smaller than those for CAdS, particularly during the early stages of the analysis.

6.3.2 Effort Analysis

The effort required for each of the three approaches is compared in Table 6.2. Both semi-automated abstraction-driven slicing and CAdS were found to be more efficient than pure static slicing. When using Codesurfer and CWolf for abstraction-driven slicing, the effort required was 21.4% lesser than that for static slicing. Similarly, when using CAdS, the effort required was 19.4% lesser than that for static slicing. The slight

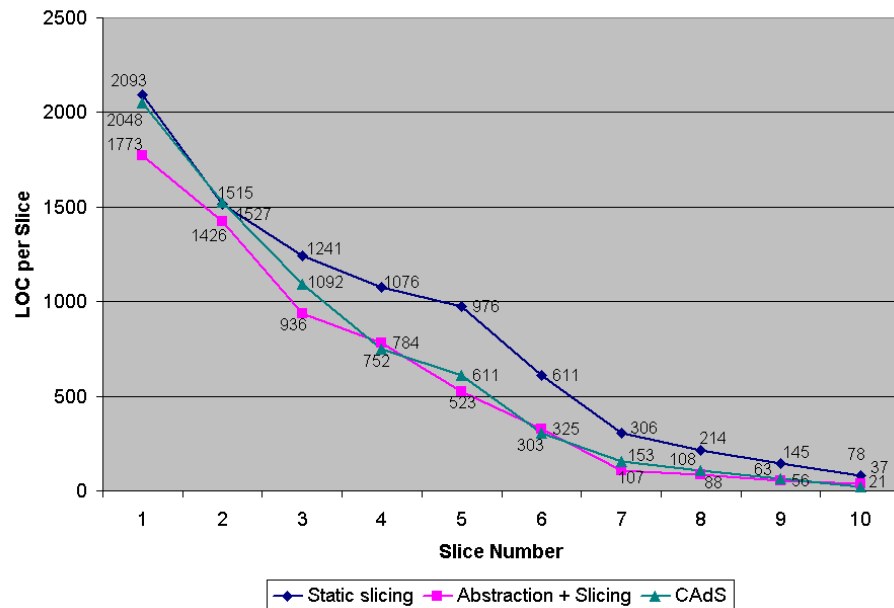


Figure 6.7: Comparison of slice sizes for the XiO forensic analysis

discrepancy indicating higher effort for CAdS as compared to the semi-automated approach can be attributed to the limitations of the CAdS slicer (as opposed to various slicing options provided by Codesurfer).

Table 6.2: Effort required for forensic analysis of XiO

	Static Slicing	Abstraction-driven slicing (semi-automated)	CAdS
Total Effort Required	221 person-hr	174 person-hr	178 person-hr
Effort per Error	8.50 person-hr	6.69 person-hr	6.84 person-hr

6.4 Summary

Results for the GIP and CARA studies illustrated the advantages of using formal methods based approaches to premarket review. A number of errors were detected in the system specifications for both studies that would otherwise have been uncovered

much later in the SDLC. These comprised of design errors that were overlooked by conventional review strategies like Cleanroom engineering. Moreover, formally verifying usage models (and test suites/usage scenarios derived from them) up front ensures that the usage models conform to safety properties.

As evinced by validation of the PCA prototype, usage scenarios provided a fairly exhaustive test suite to evaluate implementations against safety standards. Usage model based premarket review could thus be a valuable tool for regulatory agencies during certification and conformance reviews.

A less tangible advantage of (formal) usage models was that while modeling the system itself, a much better understanding of the requirement specifications was obtained, making it much easier to design and implement the system in the later stages of the development cycle. A number of issues were raised while modeling the GIP and CARA systems, with many ambiguous requirements being clarified, and the model revised accordingly. The extra effort expended at this stage thus translates to substantial savings during implementation. This would be an important incentive for manufacturers for using formal methods in their design (or for incorporating usage models in therein).

With regard to forensic analysis, the GIP and XiO studies amply demonstrated the efficacy of the abstraction-driven slicing approach. The method was shown to be consistently better than pure static slicing at tracing failures to source code. The bottom-up comprehension provided by abstraction complemented by the top-down understanding from slicing, resulting in fewer iterations during post-mortem analysis of the software.

Automating the process using CAdS further helped reduce the effort in medium-sized programs (as in the PCA implementation). CAdS primarily addressed reduction in effort for specifying abstraction criteria and the manual mapping of slices on to abstract models. In the GIP study, this comprised a substantial percentage of the analysis effort, and was significantly reduced through CAdS. However, for larger programs (as in the XiO study), majority of the effort was spent in navigating through source code and abstract models. The effort spent in formulating abstraction criteria was much lesser in comparison. As a result, the advantages offered by CAdS were

somewhat marginalized. With an improved slicing algorithm and better visualization algorithms, the efficiency of the CAdS tool could be further increased, resulting in greater savings in effort during forensic analysis.

Chapter 7

Conclusion and Future Work

In this thesis, we have given a strategy for premarket and postmarket reviews for safety conformance of safety-critical software. We have demonstrated that a formalized approach, based on construction of models from designs and implementations, could lead to a faster review of medical device applications and provided experimental evidence to show the viability of our approach. Experimental evidence suggests that the technical approach of using formal methods is effective for regulatory reviews. Preliminary studies with medical device software have illustrated the utility and efficacy of the approach from the regulator's (in this case, the CDRH's) perspective. CDRH is currently in the process of evaluating the premarket and postmarket review strategies with a view to include these as part of their guidelines to software verification and validation.

However, a general acceptance of the methodology by both regulators and software manufacturers would depend upon a number of political considerations, and many comprehensive studies that show the advantage of using formal-methods based tools. Thus, we look upon this study as the first step in the process of developing an efficient and universally acceptable methodology for design and review of safety-critical software. Some of the issues that need to be further explored to make this possible include developing comprehensive usage models for various categories of safety-critical

software and providing a scalable implementation of automated abstraction-driven slicing.

7.1 Improving Premarket Review

Usage models for safety-critical software can vary from having a few states to large-scale systems with tens of thousands states. Modeling these large systems is a non-trivial task and requires significant effort, mainly in laying out the architecture and structure of the various components in the system. In this thesis, we have defined the layered and observer-based architecture patterns to facilitate the modeling process (for the GIP and CARA studies respectively). This, however, is by no means an exhaustive set. Several new architecture patterns would need to be identified to accommodate the entire spectrum of safety-critical systems. Some architectures like those for object-oriented software and distributed systems could be adapted from existing standards like DoDAF and TADG [BMR⁺96]. Other, more specific architectures would only be identified while designing usage models for typical safety-critical software systems. Finally, an architecture suite would be defined, consisting of templates for usage models, algorithms for deriving usage scenarios from the models, and a set of formal methods tools best suited for modeling and verification of the specific architecture.

An important component of the usage model is the interface used to provide mappings between the generalized formal model and specific software implementations. The interfaces used in the GIP study were not relatively simple, owing to the fact that the implementation (i.e., the PCA prototype) was based on the same specifications as the model. However, for larger, more complex systems, the interfaces would be much more complicated. A framework would thus need to be developed for defining comprehensive interfaces for individual implementations. Each interface would be used to derive exhaustive usage scenarios or test suites for that implementation.

These interface mappings would require a thorough understanding of specific software implementations and, would thus need to be defined by individual manufacturers

(or by regulators in accordance with manufacturers). To facilitate this, Application Programming Interfaces (APIs) could be specified for the usage model using which interfaces from various vendors could harness and manipulate the model. The APIs would be standardized and included as part of the usage model architecture as well.

7.2 Enhancing Abstraction-driven Slicing

The integrated approach to post-mortem analysis presented by the abstraction-driven slicing paradigm offers significant advantages as opposed to plain static slicing. The most obvious advantage is the reduction in effort as a direct consequence of automating the bottom-up model abstraction process. Also, using partition abstractions provides a good heuristic to assess the slice refinement criteria. This results in fewer, more refined slices, and consequently, savings in effort.

Obtaining an abstracted model also provides the analyst the facility to use this model for conventional model checking. This abstracted model could be used to verify temporal constraints and assertions, thus providing a more exhaustive and complete analysis than is possible through plain program slicing. This facility is of great help especially when performing static analysis for real-time and distributed systems.

The methodology does have its limitations as well. One problem with the approach, that is a legacy of static slicing, is that it can only be used to detect ‘errors of commission’, but is not very good with ‘errors of omission’ (although it has been argued that slicing can be used to aid the detection of such errors [HD97]). This problem could be addressed to some extent by combining model checking with this approach to verify the abstracted model. However, studies need to be carried out to determine how useful model checking on partial models would be and how seamlessly such an integration could be achieved.

Another limitation of the CAdS tool is that it uses a rather rudimentary implementation of a backward static slicer. To fully exploit the utility of the integrated approach though, a more sophisticated slicer needs to be provided. Specifically, the analyst should have the ability to compute forward slices and chops over the CWI file

as well. Alternatively, the CAdS tool could be integrated with a commercial slicing engine, such as Codesurfer.

CAdS uses a simple user interface with a **Tree layout** algorithm to display abstracted graphs. The **Tree layout** algorithm manages the hierarchical structure of the graph quite well, but is not so efficient when dealing with nested blocks and collapsed states. As a result, extra effort is sometimes needed to navigate through the graph. The efficiency of the CAdS display can be improved by using more sophisticated layout algorithms, or by interfacing the front-end of the tool with a commercial graph visualization package, such as JGraphPad.

The current implementation of CWMB, and as a result CAdS, deals with data abstraction for integer values only. This limits the scope of abstraction to simple, integer data types. Extending the scope of the abstraction to include array elements and integer fields in user-defined data types would provide more precise abstraction refinements and more relevant details in the generated models. This in turn would lead to fewer iterations in the post-mortem analysis, and lead to a further reduction in effort.

As software becomes more complex, taking over more and more competent human diagnostic and intervention activities, the need for more rigorous conformance reviews becomes paramount. To perform premarket reviews, regulators need a methodology to adequately ensure conformance to safety standards. Similarly, failures encountered once a device is on the market must be effectively traced to their source to ensure appropriate remedial measures. The processes presented in this thesis provide a formal methods based approach to address these concerns and improve regulatory review processes. While the focus of the two research efforts presented are different, they share common theories, and are complimentary. Applying these shared techniques can strengthen the analysis process and the resulting rigor of the regulatory processes discussed.

Bibliography

- [ADS93] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software — Practice and Experience*, 23(6):589–616, 1993.
- [AH92] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: A survey. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 74–106, London, UK, 1992. Springer-Verlag.
- [ARTZ03] Paul Anderson, Thomas Reps, Tim Teitelbaum, and Mark Zarins. Tool support for fine-grained software inspection. *IEEE Software*, 20(4):42–50, 2003.
- [BCH⁺04] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 326–335, 2004.
- [BH99] Ramesh Bharadwaj and Constance L. Heitmeyer. Model checking complete requirements specifications using abstraction. *Autom. Softw. Eng.*, 6(1):37–68, 1999.
- [BH02] Jay Bagga and Adrian Heinz. JGraph — A Java based system for drawing graphs and running graph algorithms. *j-LECT-NOTES-COMP-SCI*, 2265, 2002.
- [BLL⁺95] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real-Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer-Verlag, Oct 1995.
- [BM92] L. M. Barroca and John A. McDermid. Formal methods: Use and relevance for the development of safety-critical systems. *Comput. J.*, 35(6):579–599, 1992.

- [BMR⁺96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.
- [Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.
- [Boo86] Grady Booch. Object-oriented development. *IEEE Trans. Software Eng.*, 12(2):211–221, 1986.
- [Bos00] Jan Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.
- [BR01] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.
- [BR02] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3. ACM Press, 2002.
- [Bra96] Franz-Josef Brandenburg, editor. *Graph Drawing, Symposium on Graph Drawing, GD '95, Passau, Germany, September 20-22, 1995, Proceedings*, volume 1027 of *Lecture Notes in Computer Science*. Springer, 1996.
- [Bro83] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.
- [Bur84] J. P. Burgess. Basic tense logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic: Volume II: Extensions of Classical Logic*, pages 89–133. Reidel, Dordrecht, 1984.
- [BW94] Bernard Boigelot and Pierre Wolper. Symbolic verification with periodic sets. In *CAV*, pages 55–67, 1994.
- [CBK90] E. M. Clarke, I. A. Browne, and R. P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω -automata. In *CAAP '90: Proceedings of the fifteenth colloquium on CAAP'90*, pages 103–116, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

- [CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
- [CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.
- [CCL98] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology special issue on Program Slicing*, 40(11–12):595–607, December 1998.
- [CDH⁺00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.
- [CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.
- [CHW98] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Softw.*, 15(6):37–45, 1998.
- [CMCHG96] Edmund M. Clarke, Kenneth L. McMillan, Sérgio Vale Aguiar Campos, and Vassili Hartonas-Garmhausen. Symbolic model checking. In *CAV*, pages 419–427, 1996.
- [CNN01] Paul Clements, Linda Northrop, and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, August 2001.
- [CS96] Rance Cleaveland and Steve Sims. The NCSU concurrency workbench. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 394–397, New Brunswick, NJ, USA, 1996. Springer Verlag.
- [CSE96] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model-checking. Technical Report NASA-IVV-96-022, NASA Independent Verification and Validation Facility, Fairmont, WV, August 1996.
- [CV98] Judith Crow and Ben L. Di Vito. Formalizing space shuttle software requirements: Four case studies. *ACM Trans. Softw. Eng. Methodologies*, 7(3):296–332, 1998.

- [DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *ICCD '92: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 522–525. IEEE Computer Society, 1992.
- [Deu03] Alain Deutsch. Static verification of dynamic properties, November 2003.
- [DI02] Daniel C. DuVarney and S. Purushothaman Iyer. C wolf - a toolset for extracting models from c programs. In *FORTE '02: Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*, pages 260–275. Springer-Verlag, 2002.
- [Dil96] D. L. Dill. The mur ϕ verification system. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 390–393, New Brunswick, NJ, USA, 1996. Springer Verlag.
- [DJK⁺99] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the International Conference on Software Engineering (ICSE)*, May 1999.
- [dLFM96] Andrea de Lucia, Anna Rita Fasolino, and Malcolm Munro. Understanding function behaviors through program slicing. In *WPC '96: Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)*, page 9, Washington, DC, USA, 1996. IEEE Computer Society.
- [dom00] Domain engineering: A model-based approach, January 2000.
- [DuV02] Daniel C. DuVarney. *Abstraction-Based Generation of Finite State Models from C Programs*. PhD thesis, North Carolina State University, Raleigh, North Carolina, USA, 2002.
- [EHM⁺99] Peter Eidorff, Fritz Henglein, Christian Mossin, Henning Niss, Morten Heine B. Sørensen, and Mads Tofte. AnnoDomini in practice: A type-theoretic approach to the year 2000 problem. In Jean-Yves Girard, editor, *Proc. Symposium on Typed Lambda Calculus and Applications (TLCA)*, volume 1581 of *LNCS*, pages 6–13, L'Aquila, Italy, April 1999. Springer-Verlag.

- [fda02] General principles of software validation; Final guidance for industry and FDA staff. Center for Devices and Radiological Health, US Food and Drug Administration, January 2002.
- [FDHH04] Chris Fox, Sebastian Danicic, Mark Harman, and Robert M. Hierons. Consit: a fully automated conditioned program slicer. *Softw., Pract. Exper.*, 34(1):15–46, 2004.
- [FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 192–203, New York, NY, USA, 1999. ACM Press.
- [FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.
- [FvBK⁺91] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Trans. Softw. Eng.*, 17(6):591–603, 1991.
- [GFdA98] M. L. Griss, J. Favaro, and M. d' Alessandro. Integrating feature modeling with the rseb. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 76, Washington, DC, USA, 1998. IEEE Computer Society.
- [GGSV02] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating finite state machines from abstract state machines. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 112–122, New York, NY, USA, 2002. ACM Press.
- [GHK⁺01] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, 2001.
- [GL91] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.
- [Glo03] Implementing a post market surveillance programme, September 2003.
- [God97] Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186. ACM Press, 1997.

- [GPVW96] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, 1996. Chapman & Hall, Ltd.
- [Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
- [HB99] Michael G. Hinchey and J. P. Bowen. *High-Integrity System Specification and Design*, editor S. A. Schuman. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
- [HD97] Mark Harman and Sebastian Danicic. Amorphous program slicing. In *WPC '97: Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, page 70. IEEE Computer Society, 1997.
- [Het88] Bill Hetzel. *The complete guide to software testing (2nd ed.)*. QED Information Sciences, Inc., Wellesley, MA, USA, 1988.
- [HJMS03] T.A. Henzinger, R. Jhala, R. Majumdar, and G. SUTRE. Software verification with BLAST. In *Proc. 10th Int. SPIN Workshop (SPIN'2003), Portland, OR, USA, May 2003*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer, 2003. Tool paper.
- [HLP01] Klaus Havelund, Mike Lowry, and John Penix. Formal analysis of a space-craft controller using spin. *IEEE Trans. Softw. Eng.*, 27(8):749–765, 2001.
- [Hol91] Gerard J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., 1991.
- [Hol96] Gerard Holzmann. On-the-fly model checking. *ACM Comput. Surv.*, 28(4es):120, 1996.
- [Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
- [Hol00] Gerard J. Holzmann. Logic verification of ansi-c code with spin. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 131–147. Springer-Verlag, 2000.
- [HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN*

- 1988 conference on Programming Language design and Implementation*, pages 35–46, New York, NY, USA, 1988. ACM Press.
- [HS01] Gerard J. Holzmann and Margaret H. Smith. Software model checking: extracting verification models from source code. *Softw. Test., Verif. Reliab.*, 11(2):65–79, 2001.
 - [Ios00] Radu Iosif. Formal verification applied to java concurrent software. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 707–709. ACM Press, 2000.
 - [IV85] Ravishankar K. Iyer and Paola Velardi. Hardware-related software errors: Measurement and analysis. *IEEE Transactions in Software Engineering*, 11(2):223–231, 1985.
 - [JCI04] Raoul Praful Jetley, Cohan Carlos, and S. Purushothaman Iyer. A case study on applying formal methods to medical devices: computer-aided resuscitation algorithm. *Software Tools for Technology Transfer*, 5(4):320–330, 2004.
 - [JEM05] Vladimir Jakobac, Alexander Egyed, and Nenad Medvidovic. Improving system understanding via interactive, tailorable, source code analysis. In *FASE*, pages 253–268, 2005.
 - [JGJ97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.
 - [JIJ06] Raoul Praful Jetley, S. Purushothaman Iyer, and Paul L. Jones. A formal methods approach to medical device review. *IEEE Computer*, 39(4):61–67, April 2006.
 - [Joh78] S. Johnson. Lint, a c program checker, 1978.
 - [JR94] Daniel Jackson and Eugene J. Rollins. A new model of program dependences for reverse engineering. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 2–10, New York, NY, USA, 1994. ACM Press.
 - [KCH⁺90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University, November 1990.
 - [Ken02] S Kent. Model Driven Engineering. In *Proceedings of IFM 2002*, LNCS 2335, pages 286–298. Springer-Verlag, 2002.

- [KKLL99] Kyo Chul Kang, Sajoong Kim, Jaejoon Lee, and Kwanwoo Lee. Feature-oriented engineering of pbx software for adaptability and reuseability. *Softw., Pract. Exper.*, 29(10):875–896, 1999.
- [KL88] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.
- [KLD02] Kyo C. Kang, Jaejoon Lee, and Patrick Donohoe. Feature-oriented project line engineering. *IEEE Softw.*, 19(4):58–65, 2002.
- [Koh96] Arthur C. Kohler. A 510(k) primer: FDA’s premarket notification process. *Medical Device and Diagnostic Industry Magazine*, pages 127–141, January 1996.
- [KS96] Carol Diane Klingler and James Solderitsch. Dagar: a process for domain architecture definition and asset implementation. In *TRI-Ada ’96: Proceedings of the conference on TRI-Ada ’96*, pages 231–245, New York, NY, USA, 1996. ACM Press.
- [LCED01] David J. Lie, Andy Chou, Dawson Engler, and David Dill. A simple method for extracting models from protocol code. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 192–203. IEEE Computer Society and ACM SIGARCH, 2001.
- [LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.
- [LVV04] R. Lammel, E. Visser, and J. Visser. The essence of strategic programming. draft; available at <http://www.cwi.nl/ralf/>, 2004.
- [McM92] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.
- [Mer01] Stephan Merz. Model checking: A tutorial overview. In F. Cassez et al., editor, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer-Verlag, Heidelberg, 2001.
- [Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
- [MM98] John Musa and John D. Muda. *Software Reliability Engineered Testing*. McGraw-Hill, Inc., New York, NY, USA, 1998.

- [Mol80] Cleve B. Moler. MATLAB user's guide. Technical report, University of New Mexico, Department of Computer Science, November 1980.
- [MSA⁺04a] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. Pse: explaining program failures via postmortem static analysis. *SIGSOFT Softw. Eng. Notes*, 29(6):63–72, 2004.
- [MSA⁺04b] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. Pse: explaining program failures via postmortem static analysis. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 63–72, New York, NY, USA, 2004. ACM Press.
- [OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184. ACM Press, 1984.
- [ORS92] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.
- [Pen87] N. Pennington. Comprehension strategies in programming. In G. M. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 100–113. Norwood, NJ: Ablex Publishing Company, 1987.
- [Pet99] Paul Pettersson. *Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice*. PhD thesis, Uppsala: Acta Universitatis Upsaliensis, 1999.
- [Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.
- [PT96] Rosario Pugliese and Enrico Tronci. Automatic verification of a hydro-electric power plant. In *FME*, pages 425–444, 1996.
- [PT99] J. H. Poore and Carmen J. Trammell. Application of statistical science to testing and evaluating software intensive systems. In *SESD '99: Proceedings of the Science and Engineering for Software Development: A Recognition of Harlan D. Mills' Legacy*, page 40, Washington, DC, USA, 1999.

- [rea03] Model-based testing and validation of control software with reactis, November 2003.
- [RFW⁺04] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie. *Model Driven Architecture with Executable UML*. Cambridge University Press, March 2004.
- [RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.
- [RS96] R. Cleaveland and S. Sims. The NCSU concurrency workbench. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 394–397, New Brunswick, NJ, USA, 1996. Springer Verlag.
- [Rus93] John Rushby. Formal methods and the certification of critical systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Also issued under the title *Formal Methods and Digital Systems Validation for Airborne Systems* as NASA Contractor Report 4551, December 1993.
- [SA96] Tirumale Sreemani and Joanne M. Atlee. Feasibility of model checking software requirements. In *COMPASS'96, Proceedings of the 11th Annual Conference on Computer Assurance*, pages 77–88, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.
- [SCK⁺96] Mark Simos, Dick Creps, Carol Klingler, Larry Levine, and Dean Allemang. Organization domain modeling (ODM) guidebook, version 2.0. Technical Report STARS-VC-A025/001/00, Lockheed Martin Tactical Defence Systems, June 1996.
- [SECH98] Francis Schneider, Steve Easterbrook, John R. Callahan, and Gerard J. Holzmann. Validating requirements for fault tolerant systems using model checking. In *ICRE '98: Proceedings of the 3rd International Conference on Requirements Engineering*, pages 4–13, Washington, DC, USA, 1998. IEEE Computer Society.
- [SH97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.
- [Sim95] Mark A. Simos. Organization domain modeling (odm): formalizing the core domain modeling life cycle. In *SSR '95: Proceedings of the*

- 1995 Symposium on Software reusability*, pages 196–205, New York, NY, USA, 1995. ACM Press.
- [SJKM05] Klaus Schmid, Isabel John, Ronny Kolb, and Gerald Meier. Introducing the pulse approach to an embedded system population at testo ag. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 544–552, New York, NY, USA, 2005. ACM Press.
 - [SK00] Raymond Smith and Bogdan Korel. Slicing event traces of large software systems. In *Automated and Algorithmic Debugging*, 2000.
 - [Sta04] E. W. Stark. Formal specification of CARA in Java. *Software Tools for Technology Transfer*, 5(4):331–350, 2004.
 - [TFC⁺01] Serdar Tasiran, Farzan Fallah, David G. Chinnery, Scott J. Weber, and Kurt Keutzer. A functional validation technique: Biased-random simulation guided by observability-based coverage. In *International Conference on Computer Design*, pages 82–88, September 2001.
 - [Tip95] Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
 - [Var96] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency : structure versus automata*, pages 238–266, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.
 - [vG90] R. J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In *CONCUR '90: Proceedings on Theories of concurrency: unification and extension*, pages 278–297, New York, NY, USA, 1990. Springer-Verlag New York, Inc.
 - [VHBP00] W. Visser, K. Havelund, G. Brat, and S. Park. Java pathfinder - second generation of a java model checker, July 2000.
 - [vMV95] Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.
 - [Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, 1981.
 - [WL99a] David M. Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison Wesley, 1999.

- [WL99b] David M. Weiss and Chi Tau Robert Lai. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [Wol02] Pierre Wolper. Constructing automata from temporal logic formulas: a tutorial. *Lectures on formal methods and performance analysis: first EEf/Euro summer school on trends in computer science*, pages 261–277, 2002.
- [WPT95] Gwendolyn H. Walton, J. H. Poore, and Carmen J. Trammell. Statistical testing of software based on a usage model. *Software—Practice and Experience*, 25(1):97–108, 1995.
- [XDR04a] Lihua Xu, Marcio S. Dias, and Debra J. Richardson. Generating regression tests via model checking. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMP-SAC), Hong Kong*, pages 336–341, September 2004.
- [XDR04b] Lihua Xu, Marcio S. Dias, and Debra J. Richardson. Regression testing via model checking. In *Proceedings of the IASTED International Conference on Software Engineering (SE2004), Innsbruck*, pages 336–341, February 2004.
- [ZEJ02] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using equal for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, pages 33–48, Berkeley, CA, USA, 2002. USENIX Association.