ABSTRACT

ZHANG, YI. Improving Software Comprehension In Regulating Safety-Critical Systems. (Under the direction of Professor S. Purushothaman Iyer.)

One primary concern regarding software employed in safety-critical systems is that it should not endanger end users of these systems. Regulatory agencies have been given the task of ensuring that software, when contained in safety-critical systems, conforms to a set of stringent standards. Moreover, these agencies are required to comprehend a piece of software, in the event of its failure, to identify causes of the failure. Software comprehension is, thus, needed to assess software safety and assign blame for a reported software failure.

A number of challenges in regulatory practices prevent regulators from achieving a fast software comprehension: 1) There is lack of a good communication channel between regulators and manufacturers, forcing regulators to acquire knowledge of applicant systems from poorly-organized documentation; 2) Traditional program comprehension techniques, such as static slicing, demand that regulators have intimate knowledge of the software. Moreover, when the code is large in size, programming slicing usually yields a slice that is too large to be understood; 3) Given a reported failure, current post-mortem analysis techniques cannot assist regulators to choose wisely on the range of code that contains the software defect causing the reported failure. Therefore, regulators have to perform costly comprehension on the entire system to trace the failure.

In this thesis, we present how the accuracy and efficiency of software comprehension can be improved by answering the above-mentioned challenges. First of all, we present a structured, graphical model, called Goal Graph, to improve communication between regulators and vendors. Our case study on applying Goal Graph to the class of generic large-volume infusion pumps demonstrates that the model is flexible and expressive for realistic regulation activities.

Secondly, we present a method integrating abstraction and slicing to ameliorate the problem of understanding large slices. The method allows iterative invocations of abstraction and slicing, allows automatic generation of abstraction criteria, allows continuous slicing, and allows a better visualization of abstract executions. Our case study illustrates that effort is reduced in post-mortem analysis when slicing and abstraction are combined.
Lastly, we propose a framework to utilize error reports to help regulators localize a failure to a specific portion of code. In this framework, we present solutions to both formalizing error reports and identifying execution sequences that satisfy the error report. Moreover, we devise an algorithm to derive slicing criteria automatically, allowing regulators to use slicing without any knowledge of the system being analyzed. We finally present a case study involving a medical device that illustrates how these automatic techniques can aid program comprehension.
Improving Software Comprehension In Regulating Safety-Critical Systems

by

Yi Zhang

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, North Carolina

2008

APPROVED BY:

Dr. Laurie Williams
Dr. Tao Xie

Dr. S. Purushothaman Iyer
Chair of Advisory Committee

Dr. Matthias Stallmann
DEDICATION

To my parents,
and
Yang Zhang, the true love of my life.
BIOGRAPHY

Yi Zhang was born on July 4, 1978, in Fuzhou, a small eastern city in China. In this quiet and peaceful town he spent most of his formative years, until he left for college in 1995. After five years of undergraduate study, he received his B.E. degree in Computer Science from University of Science and Technology of China in 2000. At the same year, he enrolled at Institute of Software, Chinese Academy of Sciences, from which he received his M.E. degree in Computer Science in 2003. Also in 2003, Yi enrolled at North Carolina State University, and shortly thereafter began working under the direction of Dr. Purush Iyer.

Yi focuses his research interests on formal methods, static program analysis, and compiler construction and optimization. His non-research interests include soccer, badminton, historical novels and Chinese classic music.
ACKNOWLEDGMENTS

The writing of this dissertation has been one of the most significant academic challenges I have ever had to face. Without the help, guidance and support from many people, this work would not have been finished. It is to them that I owe my deepest gratitude. First and foremost, I would like to acknowledge the guidance, patience and understanding of my advisor, Dr. Purush Iyer. His wisdom and knowledge shared with me provided an indispensable groundwork to the accomplishment of this dissertation. More importantly, his constant confidence in my abilities persistently encouraged me and helped me maintain a positive attitude throughout my PhD study at NC State.

I am grateful to my doctoral committee members for their valuable discussions, insightful input and accessibility. I would especially like to thank Dr. Tao Xie, for his suggestions and comments providing me a novel perspective to re-evaluate my work, and for the ample information he supplied on static analysis techniques and tools.

I would like to sincerely thank Paul Jones, Brian Fitzgerald and the CDRH, for giving me the opportunity to get involved in realistic regulation practices. I am also grateful to Dan DuVarney for his excellent implementation of the CWolf tool, which acted as an important platform in my work. A special thanks to Raoul Jetlety for his cooperation with me to realize the CAdS tool, as well as his knowledge and insights on regulation and program comprehension.

Finally, and mostly importantly, I would like to thank my girlfriend Yang. Her support, encouragement and patience were undeniable the bedrock upon which the past five years of my life have been built. Her devotion to me in all sorts of loneliness and separation reassures me of her love and fidelity, and grants me the power to steadfastly pursue my dream. I thank my parents, Meibao and Jiarui, for their confidence in me and allowing me to be as ambitious as I wanted. It was under their watchful eyes that I gained so much drive and courage to tackle challenges head on.
# TABLE OF CONTENTS

## LIST OF TABLES

- vii

## LIST OF FIGURES

- viii

## 1 Introduction

- A Requirement Model for Pre-Market Review ............................................... 3
- Comprehending Code in Post-Market Review ..................................................... 7
  - Abstraction for Program Comprehension ....................................................... 7
  - Error Report Driven Failure Analysis ............................................................ 8
- Contributions ........................................................................................................ 10
- Thesis Organization ............................................................................................... 12

## 2 Background and Related Work

- Assurance and Validation ..................................................................................... 14
  - Assurance Evaluation and Assurance Case ....................................................... 15
  - Validation and Verification ............................................................................... 16
- Program Slicing ..................................................................................................... 19
  - Static Slicing ..................................................................................................... 20
  - Dynamic Slicing ................................................................................................. 20
  - Conditioned Slicing ......................................................................................... 21
  - Amorphous Slicing ......................................................................................... 22
- Model Abstraction .................................................................................................. 22
  - Metacompilation ............................................................................................... 23
  - Abstraction ....................................................................................................... 23
- Failure Analysis Techniques .................................................................................. 31
  - Failure Investigation and Reporting Schemes ................................................... 31
  - Post-mortem Analysis Techniques .................................................................... 32

## 3 Goal Graph -

- A Conceptual Requirement Model ....................................................................... 35
  - Hierarchy of The Model .................................................................................... 36
  - Device ............................................................................................................... 37
  - Components ...................................................................................................... 39
  - Requirements .................................................................................................... 40
  - Pre-Conditions .................................................................................................. 43
  - Constraints ........................................................................................................ 45
  - Connecting Goal Graphs and Assurance Cases ................................................ 47
  - Discussions ....................................................................................................... 48
  - Related Work ..................................................................................................... 50
  - A Case Study on Large Volume Infusion Pumps ................................................ 52
3.5.1 Top Component Hierarchy ..................................  53
3.5.2 Delivery Mechanism ......................................  54
3.5.3 Correctness of Infusion Process .........................  56
3.5.4 Reliability of Infusion Execution .........................  58

4 Abstraction for Program Comprehension .........................  62
  4.1 Abstraction-Driven Slicing ..................................  63
  4.2 Automatic Generation of Abstraction Criteria ..............  67
  4.3 Slicing over CWI Representations ..........................  72
  4.4 Abstracting Models ........................................  75
  4.5 Case Studies ..............................................  78

5 Error Report Driven Software Failure Analysis ..................  82
  5.1 Maluse Case ..............................................  83
  5.2 Logical Explanation of Maluse Cases .......................  87
  5.3 Framework Description ....................................  89
    5.3.1 Abstract Model Construction ..........................  92
    5.3.2 Slicing Criterion Derivation ..........................  96
  5.4 Decomposition of Maluse Cases ............................  97
  5.5 Case Study .............................................. 100
    5.5.1 Program Instrumentation ............................... 100
    5.5.2 A Running Example .................................... 101
    5.5.3 Experimental Results ................................. 103

6 Conclusions and Future Work .................................... 106
  6.1 Summarization ............................................ 106
    6.1.1 A Conceptual Requirement Model ...................... 106
    6.1.2 Improving Software Comprehension in Post-market Analysis .... 107
  6.2 Enhancing The Goal Graph Model ............................ 108
  6.3 Improving The Error Report Driven Failure Analysis ....... 109
  6.4 Connecting Pre- and Post-Market Review with Goal Graphs .... 111

Bibliography ...................................................... 113
LIST OF TABLES

Table 2.1 Well-Known Non-Relational Abstractions ................................................. 27
Table 3.1 Table for Device *Generic LVP* ................................................................. 39
Table 3.2 Table for Component *Fluid Delivery* ......................................................... 42
Table 3.3 Table for Requirement *Safe Infusion* ......................................................... 42
Table 3.4 Table for Constraint *Safe KVO Rate* ......................................................... 45
Table 4.1 Information Types .......................................................................................... 70
Table 4.2 Ranks and Interesting Value Lists for Expressions ......................................... 70
Table 4.3 DTD for the Graph File .................................................................................. 76
Table 4.4 Effort Required for Analyzing the XiO System ............................................... 79
Table 5.1 An Example Error Report .............................................................................. 84
Table 5.2 The Maluse Case Grammar ............................................................................ 85
Table 5.3 Maluse Case for the Example Error Report .................................................... 86
Table 5.4 The Semantics of CTL .................................................................................... 87
Table 5.5 Abstraction Types Supported by CWolf ......................................................... 92
Table 5.6 Base Abstraction Types for Variables in Predicates ......................................... 93
Table 5.7 The Model Generation Process ...................................................................... 95
Table 5.8 Derivation of Slicing Criterion ....................................................................... 97
Table 5.9 First Maluse Case for the Example ................................................................. 101
Table 5.10 Second Maluse Case for the Example ......................................................... 102
Table 5.11 Experiment Results for the XiO Analysis ................................................. 103
LIST OF FIGURES

Figure 1.1 Structured Argument .................................................. 5
Figure 1.2 Information Flows between Three Parties ......................... 6

Figure 2.1 Model Based Pre-Market Review Process ......................... 19
Figure 2.2 CWolf Architecture ..................................................... 29
Figure 2.3 Failure Reporting Process ............................................. 32

Figure 3.1 A Hierarchy of Infusion Pump Types ............................... 38
Figure 3.2 Goal Graph Structure .................................................. 41
Figure 3.3 From Goal Graph to Assurance Case ............................... 47
Figure 3.4 Top Level Components of LVP Software ......................... 52
Figure 3.5 Goal Sub-Graph for Fluid Delivery ................................. 55
Figure 3.6 Goal Sub-Graph for Safe Process ................................. 60
Figure 3.7 Goal Sub-Graph for Safe Infusion ................................. 61

Figure 4.1 The CAdS System Architecture ..................................... 66
Figure 4.2 An Example of Partition Abstraction ............................... 69
Figure 4.3 An Example of Abstraction Map Generation ..................... 71
Figure 4.4 The Coarse and Refined Model Generated ....................... 72
Figure 4.5 A Partially Expanded Graph in CAdS ............................. 77
Figure 4.6 The Number of Iterations for Analyzing the XiO System ....... 78
Figure 4.7 The Number of Iterations for Analyzing the PCA Prototype .. 80
Figure 4.8 Comparison of Slice Sizes for the XiO Analysis ................. 81
Figure 4.9  Comparison of Slice Sizes for the PCA Analysis .......................... 81

Figure 5.1  Error Tracing Process ................................................................. 91

Figure 5.2  Hierarchy of the Detected Error Trace ............................................. 102

Figure 5.3  Structure of the Resultant Slice ..................................................... 103

Figure 5.4  Comparison of Iteration Numbers between CAdS and the Framework ... 104

Figure 6.1  Connecting Pre- and Post-Market Review ...................................... 111
Chapter 1

Introduction

Over the past few decades, software has played an increasingly important role in the development and deployment of safety-critical systems. Taking the medical care area as example, we can see that computer-controlled medical devices with powerful functionalities have been developed. As safety-critical systems continue to grow both in size and functionality, so does the complexity of the internal architecture of the software used and the complexity of the human-computer interaction. Consequently, it is becoming more difficult to remove subtle defects from software before its release. According to an analysis that U.S. Food and Drug Administration (FDA) performed on 3140 medical device recalls between 1992 and 1998, 242 of them (7.7%) were attributable to unnoticed software defects [1]. Moreover, a complicated software system also makes it challenging, in the event of its failure, to investigate the cause of failure and, thus, to take necessary corrective actions.

For safety-critical systems, such as nuclear power plant, air transport systems, and medical devices, there are independent regulatory authorities whose task is to control, certify and license safe systems. One of their primary aims, thus, is to ensure that software running on these systems functions as intended, and at the same time it does not endanger end users of these systems. In order to address this aim, a regulatory authority usually performs two types of reviews over software:

1. Establish a set of regulatory standards (or requirements) for software contained in safety-critical systems, and evaluate software according to the regulations.

2. Investigate the cause of a reported software failure, usually by tracing the failure back to its source in the software; and by adopting fair and objective disciplinary actions,
either enforce corrective changes to the problematic software or issue a system recall.

The intention of the first review, which is referred to as Pre-Market Review in [2], is to prevent software that is not in conformance with functionality and safety requirements from being released to the market. The second review, on the other hand, is required to remove a reported software defect or anomaly that threatens public safety. We borrow the terminology used in [3] and refer to this review as Post-Market (Forensic) Review in this thesis. Both pre- and post-market reviews are complex and laborious processes, and rely heavily on software comprehension.

Software comprehension, as a process of gaining an understanding of how a software system functions, provides the premise for regulators to reason about whether or not a system submitted to the review has faithfully establish regulatory requirements both in its design and implementation. In post-market review, on the other hand, software comprehension allows regulators to check if the reported failure does result from a particular run while the software system is functioning. Moreover, the understanding acquired by software comprehension might assist regulators to reveal what to blame as the real failure cause: logic mistakes behind system design, code flaws, or the mismatch between design and implementation. What is revealed by the analysis could then be used by regulators as objective evidence to support their subsequential corrective actions.

However, software comprehension is not an easy task for regulators, both in the pre- and post-market review processes. In pre-market review, the primary obstacle to regulators getting an insightful comprehension lies in the fact that they usually have different interest than manufacturers of the same system. To be more specific, regulators usually obtain their understanding on a system in a requirement- or risk-oriented process, attempting to check the system can fulfill every individual (safety) requirement or mitigate each previously-identified risk. However, manufacturers emphasize more on how a system is implemented to function as needed by end users. It is difficult for regulators and manufacturers to unambiguously share their interest with each other, due to the lack of efficient communication means. As a result, submissions from manufacturers usually reflect interest from only one side, and force regulators to understand manufacturers’ intention before acquiring an understanding that helps with regulation.

In post-market review, regulators proceed to comprehend a problematic system by focusing more on its source code. However, comprehending source code would not be easy
unless regulators have answers to the following questions:

1. From where should the comprehension start and to what extent should the comprehension continue? Since only a part of the implementation is problematic to cause the reported failure, it is not economic to trace the failure in the entire system. But how can regulators choose a particular piece of code and be certain that it contains the defect to be discovered?

2. Given the piece of code that has the defect to be uncovered, how can regulators efficiently obtain a understanding of the code if it is large?

In order to improve software comprehension in regulation activities, research objectives of the thesis can be phased as follows:

1. Devise an efficient and flexible communication means for all stakeholders relevant to pre-market review, such that they can elicit, analyze and negotiate interest (or in another words, requirements) with each other.

2. Devise an efficient approach for regulators to acquire a high level recognition model for a piece of code.

3. Devise an approach for regulators to localize, with confidence, the reported failure down to a specific portion of code.

1.1 A Requirement Model for Pre-Market Review

As there is no standard procedure to check the validity and reliability of software products, regulatory agencies, such as US FDA, carries out its Pre-market Review by reviewing the design of software in a medical device. To facilitate the process, manufacturers are required to submit their software development life-cycle artifacts. Typically, these artifacts include:

- requirement specifications
- design and implementation specifications
- documents on quality assurance plans and results
- risk analysis and management documents
As manufacturers do not follow a common format, regulators have to review numerous documents, vastly different in each submission, to verify that an application provides sufficient safety and quality assurance. The great variety of documents used makes the Pre-market Review process time-consuming. Furthermore, manufacturers might be required to submit supplemental documentation to resolve confusions that regulators encounter in the original submission, which lengthens the review process. Unfortunately, any delay in the Pre-market Review process could lead manufacturers to postpone release of their product; thus, causing them significant economic loss.

Much research effort has been expended on expediting the Pre-market Review process, but a viable solution has eluded researchers to date. In European safety-critical industries, there is a trend of recommending that Assurance Case [4] to be used a means to facilitate communication between regulators and manufacturers. Much like a legal case, an assurance case is essentially an easily reviewable format that structures evidence to support claims regarding a system’s conformance to assurance-related requirements, where the linkage between evidence and claims is explicitly annotated with arguments. As illustrated in Figure 1.1, the explicit structure that an assurance case possesses assists manufacturers to convince regulators in less effort: manufacturers first submit their systems to review by claiming that sufficient assurance-related properties have been fulfilled in these systems; regulators then trace along the explicit claim-argument-evidence chains, and check whether claims from manufacturers can be sufficiently justified by corresponding arguments and evidences.

It should be noted that a number of documentation techniques similar to assurance cases have been proposed and applied in other areas. One instance of these techniques is Safety Case [5] that, suggested by a broad community, has been applied to standardize the evaluation procedures on information secure systems. Compared to assurance cases, safety cases have similar structures, while cast most attentions onto safety properties.

Generally, construction of an assurance case is a two step process:

1. Firstly, claims are broken down to sub-claims until they are simple enough to be supported by objective evidence, and,

2. Secondly, arguments are added to connect lowest-level claims to evidence that justify the claims.
Considering the above creation procedure, it is easy to see that the notion of assurance case, while reasonable, could suffer from overuse of implementation details in reality [6]. To be more specific, implementation details enter in the creation of assurance cases in two ways:

- For complicated systems, it is possible for a high-level claim to be decomposed in different but equivalent ways. Manufacturers tend to choose a decomposition that allows them to match low level implementation details against the lower-level claims more conveniently.

- The decomposition of claims is stopped whenever evidences are believed to be sufficient enough to justify bottom-level sub-claims. However this belief, typically, comes from a comprehensive understanding of the implementation.

Laden with too many implementation details, an assurance case makes it necessary for regulators to obtain a comprehensive knowledge of a system if they are wish to check the correctness and consistency of the case. Consequently, the assurance case approach does not alleviate regulators’ workload substantially. The problem on hand is, then, to devise a method for stating requirements of a system such that:

a it is free of implementation details,

b it allows for a gradual comprehension of a system without a steep learning curve,
c. it allows manufacturers to connect requirements with corresponding claims, as well as necessary evidences and arguments,

d. it allows for easy documentation of a review process, and finally

e. it allows for easy exchange of information between a collection of stakeholders, such as manufacturers, physicians and reviewers.

To answer the questions stated above, we designed a structured, graphical model, called Goal Graph, to encapsulate regulators’ insight over a common set of products, as well as the set of safety requirements that they think are important. This model can be used by regulators to communicate assurance concerns about a family of systems with other parties involved. With an easily browsable model, manufacturers can deduce assurance claims with no confusion, and hence construct a convincing application to support their claims.

With the aid of a graphical model, experts or professionals in the field can also comprehend and validate the knowledge that the regulatory agency possesses on a common set of products. Furthermore, the hierarchical nature of the model allows experts to add/refine what is built by regulators. The revised graph, blessed by experts, can be communicated back to regulators, helping them meet regulatory requirements. Figure 1.2 demonstrates how information, regarding a medical device under consideration, flows between medical professionals, regulators, and manufacturers in the presence of Goal Graph.

Figure 1.2: Information Flows between Three Parties
1.2 Comprehending Code in Post-Market Review

Post-mortem forensic analysis determines the cause of a malfunction and traces it to its source in the software. Typically, during post-mortem analysis, the analyst (or reviewer) investigates the cause of software failure by tracing problems 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 interleaving across several threads. The process is further exacerbated by insufficient documentation and poor design. To ease this burden for the analyst, the post-mortem analysis process could be aided by the use of static analysis tools and techniques. Among these methods, program slicing has been historically considered as efficient and easy-to-use. Program slicing [7] facilitates 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 widely adopted in program comprehension [8] 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 [9]. Consequently, the analyst is constrained by the limitations of static slicing and ends up spending considerable effort during the post-mortem analysis process.

1.2.1 Abstraction for Program Comprehension

One way to overcome this problem is to provide the analyst a more recognizable model of the slice, i.e. with such a model the analyst can achieve his comprehension more efficiently than studying the slice itself or any of its flow-chart representation, e.g. system-
dependency graph. Abstraction [10, 11] is one ideal technique serving to provide the analyst a more recognizable model. An abstract model that can be automatically extracted from a system shows all possible executions of the system at a certain level of approximation. Combined with abstraction, slicing can work more efficiently during the post-mortem analysis, because the analyst can improve his understanding of the slice and define better slicing criteria in order to derive more concise slices upon traversing the abstract model.

The extent to which abstraction can help improve the analyst’s comprehension of the program depends on the level of approximation. If the abstract model is too coarse, the knowledge that the analyst can exploit about the program would be undermined by spurious execution paths brought into the model. However deciding whether the abstraction is ‘appropriate’ is both application- and human-dependent, and it is not possible to design an automatic algorithm that can be universally used in any application domain to find the best abstraction. Abstraction Refinement [12] is a semi-automatic approach that, when it terminates, can discover an abstraction approaching the optimal one. Starting from an initial coarse abstraction, Abstraction Refinement iteratively refines abstractions until sufficient details are discovered.

We show how to integrate abstraction into the post-mortem analysis process by combining it with slicing. On one hand, abstraction provides the analyst with an abstract semantic representation of the program that improves his understanding of the program and helps define better slicing criteria. On the other hand, abstraction is refined continuously by deriving more precise abstraction criteria from successive slices.

1.2.2 Error Report Driven Failure Analysis

One complaint about slicing is that it requires knowledge of an exact location in a program as slicing criterion. In common practice, however, the analyst (or reviewer) often comes from a third-party, which means he has no intimate knowledge of the system being reviewed to relate the failure reported to a clear location in the implementation. This situation gets worse when the analyst has access to partial code and/or sparse documentation.

Error reports from the field, while not providing clear clues for slicing criterion, do possess information depicting the scenario in which the system failures they describe occur. Our observation is that error reports describe briefly the system context under which the failure is detected, a short action history that users take leading to the failure, and the
symptom from which users identify the occurrence of the failure. We hypothesize that the process of discovering system failures can be expedited if information (in error reports) can be made use of.

The problem, however, is that error reports are written in natural language while software tools work on the basis of mechanical representations. Building a natural language processing system to convert natural language error reports to a mechanical form is a challenging, and important, problem. But, such an effort would be wasteful if comprehension driven by error reports is not tenable. Therefore, we ask the question – if error reports were to be available in a form suitable for mechanical manipulations (a) what should be their shapes?, and (b) how can this representation be used to identify a slicing criterion which can be used to obtain a slice containing (potentially) the cause(s) of a reported error?

We already know that the abstract model, by virtue of exploring all possible execution paths of the system, is suitable for finding the system’s executions satisfying certain pre-defined properties. Therefore, if we looked at a particular error report as a special property that specifies a temporal order between events triggering the occurrence of the reported failure, it is possible that the abstract model will also be suitable for isolating those system execution sequences that satisfy the error report, which in another words, captures the usage scenario specified by the error report. Providing that it is possible to recover the failure scenario, the task of identifying the accurate cause of the failure will become a relatively simple task.

Given a property, an abstract model has the advantage over concrete models that it allows relatively short execution sequences to be isolated as an evidence to the property. The flip side of the coin, however, is that the existence of abstraction might lead to spurious execution sequences being isolated. Since our primary concern is to ease the reviewer’s comprehension, either on a piece of code or on a particular execution, it is preferable trace a failure in an abstractly constructed model provided this model does not cause unacceptably high false positive rate.

Our decision to construct abstract error traces is also supported by the fact that error reports can be incomplete (i.e., may not contain all the interactions the user had with the system). Consequently, there is no way we can build exact execution sequences or slices. Thus, an algorithm is needed to abstract programs so that all information from the error report is preserved in the model.

To address the above questions, we introduce a uniform representation, called
Maluse Case, to encapsulate useful information in error reports, and propose a framework to utilize Maluse Cases to facilitate the failure tracing process. In this framework, abstraction is applied to extract an abstract model from source code. During the model construction process, abstract execution paths obtained so far are checked whether they can simulate the short action history formalized in the Maluse Case, and at the same time the specified error trigger can be satisfied. Abstract execution paths satisfying the check disclose the portion of code possibly related to the error. Moreover states at the tail of such paths imply explicit sites where error symptom can occur, and hence indicate appropriate slicing criterions. Slicing can then be performed over such paths based on the selected criterion, providing analysts smaller pieces of code to study and to ascertain the cause of an error. If a slice obtained at last step does not contain, as believed by analysts, contain defects that would cause the purported failure, it can then be used to refine the abstraction; a process that can be iteratively carried out until the source of an error is discovered.

1.3 Contributions

This thesis provides approaches for regulators to improve their comprehension of software artifacts, at both design and implementation levels, in regulatory activities. For pre-market review, we aim at providing an efficient communication medium between regulators and manufacturers such that intentions, plans, and mechanisms that manufacturers use to construct products can be unambiguously communicated to regulators, allowing regulators a quick comprehension of applicant’s product. For post-market review, we improve the reviewer’s comprehension on the purported failure by providing him easily viewable models of code that is considered suspicious, where the suspicion regarding an error can come either from a review’s hypothesis or from error reports. Specifically, primary contributions of this thesis can be summarized as follows:

1. We present the design of a structured requirement model, called Goal Graph, to formally organize safety requirements elicited by regulators. The model is suitable for use as a communication vehicle between parties that have different interests, given the fact that our model possesses the following characteristics:

   - The model has a graphical representation, allowing users to easily traverse it.
• The model can be readily customized for regulating different types of devices, leaving its internal structure untouched.

• The information contained in each node of the model is semi-structured and suitable for database manipulations.

We investigated the efficacy and flexibility of the model by applying it onto generic large-volume infusion pumps.

2. We propose an approach that integrates abstraction into the abstraction-driven slicing framework, serving to provide the reviewer a bottom-up recognition model of the program. This recognition model helps the reviewer understand the program and derive better slicing criteria. More specifically, the approach proposed provide the following solutions to primary difficulties arising during the integration:

• We propose an algorithm to automatically generate refined abstraction criteria based on a slice and a slicing criterion. This enables abstract models to be constructed to obtain successively precise models; hence it improves the reviewer’s comprehension of the program being analyzed.

• An implementation of a slicer directly over an intermediate representation of C programs, which enables successive slicing over the same program.

• We present an algorithm to extract further-abstract models from abstract models (LTSs, particularly). This kind of further-abstract models aids the reviewer to decompose source code based on conceptual functionalities and dependencies.

3. We propose a framework that utilize error reports as guidance for failure tracing analysis. For a purported failure, the framework isolates system executions that could possibly contain the failure scenario, as prescribed by the error report, and hence provides the reviewer clear clues to identify the cause of a failure. In particular, the framework is built up around the following ideas:

• A definition of Maluse Cases allowing us to encode error reports in a mechanical form, and an investigation of their properties. In particular, we show how Maluse Cases can be looked upon as temporal logic formulae, thus clarifying their semantics.
• An abstraction-guided algorithm to construct an abstract error trace from an error report/Maluse Case.
• A technique to identify slicing criterion from an abstract error trace.

We conduct a significant case study (software in a medical device) to illustrate that our framework does lead to easier identification of errors.

1.4 Thesis Organization

This thesis is organized into six chapters as follows:

Chapter 1: Introduction presents the motivation and goals of the research embodied in this thesis.

Chapter 2: Background and Related Work briefly introduces the theoretic notions used in later chapters. In particular, this chapter addresses the concept of assurance and techniques for assurance evaluation. It also covers primary theories and techniques in the field of abstraction, program slicing, and post-mortem analysis. Moreover, research work pertinent to this thesis are also cited and compared with our work.

Chapter 3: Goal Graph - A Conceptual Requirement Model explains our work on designing and assessing a structured conceptual requirement model – Goal Graph – to formalize safety requirement elicited by regulatory agencies. The chapter also includes an algorithm of deriving assurance cases from a particular goal graph in a straightforward manner. A case study on creating a complete goal graph for generic large-volume infusion pumps is used to establish the efficacy and flexibility of the model in realistic regulation practices.

Chapter 4: Abstraction for Program Comprehension: explains the role that abstraction plays in the abstraction-driven slicing framework, and details our solution on how to integrate abstraction into our slicing framework seamless. The solution covers three primary issues: automatic generation of abstraction criteria, directly slicing over immediate representations of C programs, and visualizing abstract models. Two case studies – XiO and GIP – are used to illustrate the effectiveness of the integrated framework.
Chapter 5: Error Report Driven Software Failure Analysis proposes a framework of encapsulating useful information from error reports into a unified representation and utilizing this representation to guide error detection process. A decomposition approach is also presented in this chapter for extending the framework onto tracing failures in modular-based systems. A Case study on the XiO system examines the effectiveness of this framework, and finally

Chapter 6: Conclusions and Future Work 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 later parts of the thesis and elaborate on related work in the respective areas. First of all, in Section 2.1 we explore the concept of assurance, assurance evaluation standards, and validation techniques. Secondly, in Section 2.2 we outline various categories of program slicing techniques. Model abstraction methods, especially techniques that fall into the abstraction paradigm, are discussed in Section 2.3. Finally, in Section 2.4, we describe the basic process of failure reporting and investigation, and discuss techniques that have been used in post-mortem analysis.

2.1 Assurance and Validation

The term Assurance has been used in research literature and practices in various ways. According to [4], for example, assurance is a term related to reducing the level of uncertainty in estimation, predication, information, inference, or the achievement of a goal. In [13], on the other hand, assurance is normalized as

\[
\frac{R_{\text{max}} - R_{\text{min}}}{C_{\text{max}}}
\]

where \( C_{\text{max}} \) represents the largest possible value of consequence that a risk can be encountered by a system, and \( R_{\text{max}} \) and \( R_{\text{min}} \) quantify the maximum and minimum estimation of risk that be imposed on the system respectively.

When combined with different types of properties, assurance provides users a measurement of several aspects of a system and helps users to rationally decide whether to put the system into use in designated environment. For example, software security assurance
assesses the degree to which a software system can be trusted to operate at a level of security, such that any loss of data, resources and services caused by potential harm is limited to a pre-defined level.

When a software artifact is employed inside or with devices, the assurance concerning the software usually considers the following properties:

- Correctness deals with whether the implementation is a necessary and sufficient representation of the specification.
- Effectiveness relates to the suitability of the selected functionalities in fulfilling intended uses with ability to counter identified risks.
- Usability refers to the ease with which a system can be configured and its functionalities used without compromising system assurance.
- Workmanship refers to product or system quality relative to the state of the art, including maintainability, expendability, and durability.

2.1.1 Assurance Evaluation and Assurance Case

A number of standards have been proposed and recognized to normalize the process of evaluating assurance about products or systems, each of which targets a particular application domain and focuses on a specific category of properties. The Common Criteria (CC) [14], for instance, standardizes the process of evaluating the security assurance of software products. A CC evaluation results in an Evaluation Assurance Level (EAL) assigned to a software product indicating the level of assurance requirements the product satisfies.

Basically, these assurance evaluation standards provide high level categorization of products and systems, provided objective evidences can be collected to justify its assurance. In general, evidences considered as objective and acceptable by these standards can be categorized into the following types:

- System Evidence, such as test results and configuration parameters; these come from examining a product or system directly.
- Process Evidence, such as defined process metrics and performance data; these come from examining whether the development, evaluation, and operation processes are reasonable and have been followed.
• People Evidence, such as experience data and training data; these come from examining the individuals and organizations.

• Environment Evidence, such as backup mechanisms and tool capabilities, which confirm that developmental and operational environments are acceptable.

For regulating safety-critical systems, it is important to show that a system exhibits an acceptable level of assurance on complex properties. In such circumstances, it imposes a burden on manufacturers to organize evidences in a satisfactory way such that regulators can be convinced of the assurance claims of their products. Considering this situation, assurance cases [4] has been proposed and widely adopted in European safety-critical industries.

An assurance case, as shown in Figure 1.1, is essentially an easily reviewable format that structures evidence to support claims regarding a system’s conformance to safety-related requirements, where the linkage between evidence and claims is explicitly annotated with arguments. Several notation schemes, such as Goal Structuring Notation [15], have been proposed to facilitate the creation and documentation of assurance cases.

The process of developing an assurance case is simplified, somewhat, through the introduction of assurance case patterns [16]. Assurance case patterns maintain the structure, but not the specific details, of an argument and therefore can be instantiated in multiple situations as appropriate. In this way, patterns offer the benefits of reuse and repeatability of process, as well as providing some notion of coverage or completeness of the evidence.

2.1.2 Validation and Verification

Generally, validation processes are conducted to check that the product design satisfies or fits the intended usage. However, in applications where highly risky circumstances are often encountered, validation tasks should also justify that a product can protect high value assets against significant risks. Therefore, the necessity of a rigorous development environment with stringent validation measurements might outweigh the consideration of minimizing economic costs. In the EAL stack, for instance, evaluation levels higher than EAL4 require that the development environment used by manufacturers should incorporate (semi-)formal design and validation techniques, such that products can be designed and constructed to fulfill high assurance requirements. With this in view, formal methods, such as model checking, have often been recommended to be introduced into industrial validation practices.
Model Checking

Model checking is based on constructing a finite model $M$ of the system to be verified. It also depends upon interesting properties to be encoded as some formula $\Phi$ in modal/temporal logics. Finally, the model checking procedure systematically checks that all executions in the model $M$ satisfy the the formula $\Phi$.

Although a number of structures can be used as system models, state transition system (and, in particular, Kripke structure) is the most widely used one [17]. A state transition system is a tuple $(\Sigma, R)$, where $\Sigma$ is the state space, and $R \subseteq \Sigma \times \Sigma$ stands for transitions between states. A Kripke structure is a state transition system $M$ associated with a labelling function $I : \Sigma \rightarrow AP$ that assigns to states propositions, from $AP$, that they satisfy. In the rest of this report, Kripke structure will be used as the system model in order to discuss software model checking in a uniform way.

In the context of model checking, desired properties are defined as expected behavioral patterns of systems. A number of temporal logics have been proposed as appropriate for expressing system properties, e.g. Linear Tree Logic (LTL), Computation Tree Logic (CTL) or $\mu$-calculus. Different temporal logics posses different levels of expressivity, but all of them focus on two aspects of system behaviors: 1. safety (or liveness) represents that correct (resp. expected) behaviors occur globally (resp. eventually) during system executions; 2. universal (or existential) properties assert expected properties hold for all (resp. some) possible system executions.

The remarkable advantage of model checking lies in fact that the verification process becomes totally automatic. Moreover when a system does not satisfy a desired property, model checking tools can produce counterexamples to aid in debugging the system. Such features have made model checking successful in verifying hardware systems and communication protocols where systems have finite state space.

However, the effectiveness of model checking is restricted to finite state designs and cannot handle software verification. It is usually difficult or even impossible to construct an explicit model for a software system, because of its pervasive use of data variables over infinite domains, complex data structures and dynamic objects. Moreover, when a new (parallel) component is added to the software, the complicated interaction between it and legacy components could lead to an exponential increase in the size of state space.

To cope with the state explosion problem, several state-space reduction techniques
have been proposed. Symbolic model checking \cite{18} encodes the state transitions as logic formulae which can be efficiently stored in compact formats such as Binary Decision Diagram (BDD) \cite{19}; On-the-fly model checking \cite{20} only explores the portion of state space that is necessary for verifying the property; Partial order reduction \cite{21} eliminates redundant interleaving of actions from different subsystems that will not affect the verification; Moreover, as components in a system may have same behavior patterns, an equivalence relation can be defined because of this symmetry. Thus symmetry model checking \cite{22} computes such equivalence relations, and verifies the induced quotient system. Although these model reduction techniques improve the efficiency of model checking, they cannot ultimately conquer the problem of undecidability.

**Formal Method Based Pre-market Review**

Jetley et al. in \cite{23} presented a (semi-)automatic solution to Pre-market Review tasks, in order to improve strictness and efficiency of validating a family of software implementations that share common intended uses and assurance requirements. In this framework, as depicted in Figure 2.1, the concept of usage model \cite{24} is firstly borrowed to formalize and standardize a (minimum) set of principles, either functionality- or assurance-oriented, that the family of implementations are expected to follow.

The usage model thus specified is then passed through a static model checker for formal verification on its correctness and completeness. After the verification and necessary refinement, the usage model could be utilized by model based testing techniques (such as \cite{25}) to derive a common set of test cases, or in another word usage scenarios, against which all implementations in the family can be validated.

A problem restricting the feasibility of the above framework is the difficulty of ensuring that the usage model is sound and that it correctly incorporates major functional and assurance requirements for a common range of implementations. While consultation with experts in the field serves as a primary way to collect such requirements, any misunderstanding and confusion in the consultation will affect the quality of the final usage model specified. Therefore, our Goal Graph model, because it is structured and formalized, can be integrated into the above framework as a communication medium, helping reduce uncertainties and ambiguities during the consultation process.
2.2 Program Slicing

In post-mortem analysis, the reviewer of software systems, either the regulator or the maintenance engineer, needs to investigate the cause of software failure by mapping incidents reported by end users to their origin in source code. To accomplish his job, the reviewer is required to obtain a basic understanding of the program structure and system design. Typically this understanding is achieved by manually scanning through the source code and system documentation. However, manual scanning is inefficient and time-consuming, especially when the system being analyzed is large, complex and consists of interleaving threads. Insufficient documents and poor design exacerbate this process. To ease the burden for the reviewer, the post-mortem analysis process can be aided by the use of static tools and techniques. The most widely adopted of these methods is program slicing [7] – 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. There have been several variations to the basic scheme of program slicing proposed over years. Based on the subset of program statements considered, program slicing techniques can be divided into three categories: static, dynamic, and conditioned slicing. A good survey on program slicing can be found in [8].

2.2.1 Static Slicing

In its purest form, static slicing defines the slicing criterion as a program point $x$ and set $V$ of variables. A static slice for a program $p$ can then be calculated as all statements in $p$ that has impact on the evaluation of any variable in $V$. 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 $x$. Having backward and forward slicing as tools to collect program points along the execution from or to a particular point, program chopping presented in [26] constructs 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.

2.2.2 Dynamic Slicing

Although being useful for understanding program behavior, static slicing can often result in rather large and unmanageable slices when dealing with large-scale systems. 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. In comparison with static slicing to consider program statements executed on any possible input, dynamic slicing proposed by Korel and Laski [27] narrows the calculation of a slice to 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. In this extension, the criterion in static slicing is simply augmented with the input
to the program. Formally, a dynamic slicing criterion is defined as a tuple \((I, x, V)\), where 
\(x\) and \(V\) are as defined for the static slicing, and \(I\) is the input to the program for which 
the slice is to be determined. The dynamic slice for a program \(p\) is computed by removing 
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 
\(x\). By focusing on a specific execution, dynamic slicing is more suitable for program debug-
ging. For post-mortem analysis, its feasibility is limited since the program execution in such 
applications remains unspecified. In such circumstance, the only information available to 
the reviewer is the error report, source code and the system documentation. In general the 
source code available might be partial because the target system might deploy off-the-shelf 
software or software venders hesitate to disclose their overall implementations, which make 
the code inexecutable. Moreover restoring a specific execution require providing the target 
program a precise input, and the ambiguous characteristic of most error reports makes this 
difficult or impossible.

### 2.2.3 Conditioned Slicing

Conditioned slicing, introduced by Canfora et al. [28] bridges this gap between 
static and dynamic slicing by considering these statements in the program that will be 
executed given a subset of input provided to the program. Generally conditioned slicing 
can obtain considerably smaller slices than static slicing, but not as localized as a dynamic 
slice.

Several ways have been proposed to characterized the subset of input. The most 
commonly used one is to specify a condition that should be satisfied when any execution 
taking any member of the subset of inputs reach at a particular program point. More 
formally, the conditioned approach evaluates the slice with respect to a given condition along 
with the slicing criterion used for static slicing. A Conditioned slice is then constructed with 
respect to a tuple, \((p, x, V, \pi)\), where \(x\) and \(V\) are as defined for static slicing, and \(\pi\) is 
some condition on the value of variables defined in the target program. A statement may 
be removed from a program \(p\) to form the slice for \(p\), if it cannot affect the value of any 
variable in \(V\) during some execution of \(p\) that can reach \(x\) with \(\pi\) being satisfied at \(x\). 
Hence a straightforward approach to implement conditioned slicing is a two-step process: 
first perform data-flow analysis over \(p\) to identify and eliminate the part of \(p\) that will not
lead any execution of it to a state where the execution reaches $x$ and at the same time $\pi$ will hold true, and then static slicing is performed over the resultant part. Although multiple optimizations have been proposed, most techniques of conditioned slicing follow the above process.

2.2.4 Amorphous Slicing

Slicing techniques discussed thus far are also referred as Deletion Based slicing in that they preserve the syntax structure of the original program. The calculation of slices proceeds by deleting these statements considered as irrelevant based on particular criteria. Amorphous slicing [29], on the other hand, uses semantic preserving transformations to simplify the slice.

Without sticking to the syntax of the original program, amorphous slicing is capable of performing greater simplification than its syntax-preserving counterparts, which results in often smaller slices. The effectiveness of amorphous slicing depends on the transformation it deploys to simplify the program. Any transformation works for amorphous slicing if it can simplify the program and at the same time preserves the effect of the original program with respect to the slicing criterion. Denoted the transformation applied as $\alpha$, an amorphous slice can be defined w.r.t. a tuple $(p^\alpha, x', V')$, where $p^\alpha$ is result of applying $\alpha$ over the original program $p$, $x'$ is a point in $p^\alpha$, corresponding to a program point $x$ in $p$, and $V'$ is a set of variables in $p^\alpha$ corresponding to the variable set $V$ in $p$. A statement may be removed from $p^\alpha$ to form an amorphous slice $s$, if and only if it cannot affect the value of any variable in $V'$ when the next statement to be executed is at point $x'$.

2.3 Model Abstraction

Compared to informal description techniques, such as pseudo code, diagrammatic charts and most notation techniques, a formal model presents a strict mathematical representation of the system under analysis. This representation soundly simulates and predicts the system behavior, and hence helps the reviewer understand and reason about the system.
2.3.1 Metacompilation

While verifying large-scale implementations, either by testing groups or by independent reviewers, it is desirable that formal representations could be extracted automatically from source code. Compared to techniques that develops models, of programs, from scratch, Metacompilation makes use of equivalent facilities in existing model checking tools. To be more specific, the model extraction process in the Metacompilation paradigm is usually carried out in two phrases:

- Implementations written in a programming language are translated by a compiler frontend into model descriptions that verification tools would accept. It is up to users to define and customize a protocol prescribing the translation, in order to ensure an appropriate abstraction of the original program could be acquired.

- Model descriptions generated are then input into model checkers, such as SPIN [30], guiding these tools to build up models automatically.

Two tools that belongs in this category are FeaVer [31, 32] and xgcc [33]. FeaVer pre-compiles a C program into a Promela description, which can be verified by the model checker SPIN. xgcc works similarly as FeaVer, except that its output is targeted at the Mur∅ model checker [34].

2.3.2 Abstraction

In practice, however, chances are that users have to try a set of abstractions for a specific system before an appropriate one can be reached. In such situations, it is of importance that users can reason explicitly on relative precisions among these abstractions, in order to balance the size of the model constructed and the information to preserve. Abstract Interpretation (AI for short) [10, 11] is the most commonly accepted framework to reason about abstractions.

Abstract Interpretation

The most commonly used category of formal models is automata-based, where the behavior of the target system is simulated by a set of transitions over system states. Automata-based models are suitable for further reasoning and verification, because of their expressiveness and formalized representation.
However, it is usually difficult or even impossible to construct an explicit automata-based model for a software system, because of its pervasive use of data variables over infinite domains, complex data structures and dynamic objects. Moreover when a new component is added into the software, the complicated interaction between it and legacy components could lead to an exponential increase in the size of state space of the model.

Abstract Interpretation formalizes the concept of abstraction in a lattice-like framework, and is the only state-space reduction technique that can always extract finite models from large-scale systems by providing a unified approach for designing abstract semantics that soundly approximates programs’ concrete semantics. Also A.I. helps users organize a set of abstract semantics of a program, each one different from the other, into a lattice. This lattice facilitates comparison between potential abstractions, and the choice of a abstract semantics with appropriate level of precision.

The execution of programs can be interpreted as manipulations over a set of interesting objects, which is called as the concrete domain. A.I. is usually formalized as a theory connecting the concrete domain and the abstract one approximating it. Typically, this connection is captured as a Galois connection. Let the concrete domain \((P, \subseteq, \cup, \cap)\) and the abstract domain \((A, \preceq, \wedge, \vee)\) be complete lattices; the correspondence between concrete and abstract domain is given by a Galois connection, i.e. an adjunction of an abstraction function \(\alpha : P \rightarrow A\) and a concretization function \(\gamma : A \rightarrow P\), such that: 1) \(\alpha\) and \(\gamma\) are monotonic, and 2) \(\forall c \in P \forall a \in A \alpha(c) \preceq a \Leftrightarrow c \subseteq \gamma(a)\). If moreover \(\forall a \in A \alpha \circ \gamma(a) = a\), this Galois connection is called a Galois insertion.

In order to devise an abstract semantics applicable for analysis of systems, the core step is to provide safe abstract descriptions for all operators over the concrete domain \(P\), thus all computations over the concrete domain can be safely mimicked by corresponding computations over the abstract domain. For a concrete function \(f : P \rightarrow P\), its abstract description \(f_\alpha : A \rightarrow A\) is safe w.r.t. \((\alpha, \gamma)\) if it satisfies \(\alpha \circ f \circ \gamma(a) \preceq f_\alpha(a)\) for any \(a \in A\). This definition of safety for an unary function over the concrete domains can be easily lifted to arbitrary functions \(g : P^n \rightarrow P^m\) by extending it point-wise.

Abstract Model Construction

Considering that the set of system behaviors is essentially a transition relation \(R\) over system’s state space \(M\), populating abstract transitions in the abstract model can be
implemented as calculating a safe (either under- or over-) approximation $R_A$ for $R$.

In [35], a technique known as existential abstraction was introduced to over-approximate the concrete transition relation by defining $R_A$ as $\forall a_1, a_2 \in M_A \ R_A(a_1, a_2) \Leftrightarrow \exists s_1, s_2 \in M \ (s_1 = h^{-1}(a_1) \land s_2 = h^{-1}(a_2) \land R(s_1, s_2))$, where $M_A$ is the state space of the abstract model and $h$ is a homomorphism from $M$ to $M_A$.  

On the other hand, research in [36, 37, 38] present approaches to under-estimate the system behavior. An abstract transition set $R^C_A$ under-approximates $R$ if $R^C_A(a, b) \Leftrightarrow \forall a' \in \gamma(a) \exists b' \in \gamma(b) \ R(a', b')$. Research in [39, 36] demonstrates that properties that can be characterized in existential-quantifier-free temporal logics are preserved from the abstract model to its concrete counterpart if the set of abstract transition over-approximates the set of concrete ones, and properties in universal-quantifier-free temporal logics are preserved in the other direction if the set of abstract transitions under-approximates its concrete counterpart.

**Abstraction Refinement**

An ideal abstract model for a given program should be as compact as possible, and at the same time, should keep necessary information of the original program. However, whether an abstraction of a given program is at the appropriate level depends on what properties will be verified against it, and in general finding the best abstraction for an infinite system is uncomputable [40]. On the other hand, given a property there always exists an optimal and finite abstraction for a given program [41]. An instance of such algorithms is Abstraction Refinement that starts from an initial coarse abstraction, and continuously refines abstractions by iterations until sufficient details are discovered. The original idea of Abstraction Refinement comes from Localization Reduction in [42], and has been studied in [12, 40, 43, 44, 45].

Basically Abstraction Refinement is a process of iterating the following two steps:

1. identifying a set of information used to instruct refinement, and
2. performing a refinement on current abstraction based on the information obtained at step 1.

---

1 A homomorphism between two set is a transformation of one set into another that preserves in the second set the operations between the members of the first set.
According to the information identified at step 1, various abstraction refinement approaches can be classified into two categories: 1) the information comes from a model check run, i.e. spurious counterexamples or 2) directly from the system description and property specification.

The former paradigm, known as Counterexample Guided Abstraction Refinement, was firstly proposed in [12]) and takes effect only when counterexamples detected have only finite length. In this paradigm, every abstract state in the counterexample is checked whether its concrete counterpart can be specified by the strongest post-condition of its ancestor [12], or by the weakest pre-condition of its descendant [46]. Any state failing in the check implies the spuriousness of the counterexample, and directs the way how current abstraction could be refined.

In contrast to using error traces, [43] proposed a program transformation technique to rewrite the syntax of a program using a set of predicates. The revised program results in the same control structure as the original one, with all data variables extracted away. The abstraction refinement is accomplished in this approach by continuously extending the set of predicates with new elements during the program rewriting process. Having all refinements done syntactically, this approach can be applied to infinite systems, and the resulting program is still open for other symbolic reduction techniques. But its efficiency is restricted by its requirement to check each predicates with every program action, even though most of which is unnecessary.

Abstract Domains

Much research has been done in designing specific abstractions for large-scale software systems, especially for those implemented in modern programming languages. Among the approaches proposed data abstraction and predicate abstraction have been shown as effective for the analysis of contemporary software systems.

1. Data Abstraction

The basic idea of data abstraction is to evaluate data variables in the program over a much smaller abstract domain, while still preserving the existential behaviors of the program. Data abstractions can be divided into non-relational and relational data abstraction depending on the information it can track about data variables.
Non-relational data abstraction approximates values of a set of considered data variables and ignores all other information in any state during the program execution. Table 2.1 briefly summarizes well-known non-relational data abstractions, and their abstraction and concretization functions.

2. Geometric abstractions

Geometric abstractions [47, 48, 49, 50, 51], also known as relational data abstractions, are more precise than non-relational ones in that relationships between values of interesting data variables are preserved by these abstractions. As the trace semantics of a program can be abstracted by a collecting semantics of a set of points in $\mathbb{R}^n$ space, assuming relationships between $n$ data variables in the program are of interest, geometric abstractions further approximate these points using geometric shapes in $\mathbb{R}^n$ space that contain them. Existing geometric abstractions track only linear relationships between values of program variables, which can be characterized by a set of linear (in)equalities over these variables. For example, the relationships between $n$ variables that convex polyhedra abstraction [47] can preserve are a set $C$ of linear constrains in the format of $c_1 x_1 + \ldots + c_n x_n \leq c$. Convex polyhedra abstraction approximates all the points in $\mathbb{R}^n$ that satisfies these constrains as the minimal convex hull in $\mathbb{R}^n$ containing these points. In general, an abstract domain of higher expressivity comes with a higher complexity overhead for operations over them. Therefore, selection of appropriate geometric abstractions for formal verification of programs depends on users’ decisions on the tradeoff between the precision of information they want to obtain and the tractability of abstract domains.

<table>
<thead>
<tr>
<th>Abstractions</th>
<th>$\alpha(S)$</th>
<th>$\gamma(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top</td>
<td>$\perp$ if $S = \emptyset$</td>
<td>$\emptyset$ if $x = \perp$</td>
</tr>
<tr>
<td></td>
<td>$\top$ Otherwise</td>
<td>$\top$ Otherwise</td>
</tr>
<tr>
<td>Sign</td>
<td>$+$ if $\forall x \in S \ x &gt; 0$</td>
<td>$Z^+$ if $x = +$</td>
</tr>
<tr>
<td></td>
<td>$-$ if $\forall x \in S \ x &lt; 0$</td>
<td>$Z^-$ if $x = -$</td>
</tr>
<tr>
<td></td>
<td>$\perp$ if $S = \emptyset$</td>
<td>$\emptyset$ Otherwise</td>
</tr>
<tr>
<td></td>
<td>$\top$ Otherwise</td>
<td>$\top$ Otherwise</td>
</tr>
<tr>
<td>Interval</td>
<td>$[\bigcap S, \bigcup S]$ if $S \neq \emptyset$</td>
<td>$\emptyset$ if $x = \perp$</td>
</tr>
<tr>
<td></td>
<td>$\perp$ Otherwise</td>
<td>${ z \mid l \leq z \leq u }$ if $x = \perp$</td>
</tr>
<tr>
<td>$\text{Mod}_k$</td>
<td>${ x \mod k \mid x \in S }$</td>
<td>${ p + i \ast k \mid i \in \mathbb{Z} \text{ and } p \in x }$</td>
</tr>
<tr>
<td>Partition</td>
<td>${&lt; a, b &gt; \in T \mid \exists x \in S \ a \leq x \leq b }$</td>
<td>$\bigcup_{&lt; a, b &gt; \in x} { z \mid a \leq z \leq b }$</td>
</tr>
</tbody>
</table>
3. **Predicate abstraction** Predicate abstraction is a primary instance of abstraction techniques systematically applied in static analysis and model checking. The basic idea of predicate abstraction is to encode concrete states using a set of predicates; thus concrete states sharing the same assignment to all of the predicates are abstracted into the same representative in the abstract model. Predicate abstraction was first designed by Graf and Saidi [52], in which finite or infinite state systems described as guarded programs were abstracted into abstract graph, and any $CTL^*$ properties without existential quantifiers would be checked against these systems. The work presented in [53] and [40] extended the application of predicate abstraction to C programs. In these works, predicate abstraction was used as the core technique for constructing abstract models from realistic programs, and interesting program invariants like safe list manipulations can be checked against the original programs. Case studies in these projects show that software model checking based on predicate abstraction can scale up to large programs.

**Tools to Extract Abstract Models**

The design of tools to build abstract models from programs needs to answer the following questions: (a) what range of abstractions should the tool support? (b) how to ensure the soundness of abstract domains and its operations? and (c) how to combine different abstractions? Also given that a model generated could still be of a significantly large state space, a successful tool should implement methods to store abstract models compactly.

1. **CWolf**

CWolf [54], developed at NCSU, is a toolkit designed for extracting finite labeled transition systems from C programs. An input of C programs is passed to a C parser embedded in CWolf, and the resulting intermediate program is executed symbolically to generate the finite synchronization skeleton of the original program. The skeleton obtained is open for further reduction by computing its bisimilar quotient system, or can be passed to Concurrency Workbench of New Century (CWB-NC) [55] for model checking purposes. The obligation of abstracting integer variables in the source program is left to users by requiring them to provide an abstraction map file binding particular abstractions with variables of interest. Moreover, users should
present a *label map* file enumerating interesting events (e.g. function calls, accesses to variable etc.) which will be marked on transitions in the final transition system. The architecture of CWolf is shown in Figure 2.2.

The set of data abstractions that CWolf supports is predefined in the system, which makes the proof of the soundness of abstractions and operations over them transparent to users. Also embedded in the system is a coercion scheme between different abstraction types. However, to increase the tool’s flexibility, most of abstractions are parameterized, and these parameters are customized by users in the *abstraction map* file. In detail, CWolf supports *unary, modulo, interval, partition* and *free* abstractions for integer variables in the source code. Here, the specific abstraction *free* is not an actual abstraction, but used as an indication that the abstraction type of the variable should be chosen dynamically. For array variables, CWolf assumes all its elements share the same abstraction. The CWolf system should be able to determine the size of this array, otherwise, the whole array will be abstracted as $\top$, which extracts away all information about the array.

CWolf focuses on sequential C programs, thus the use of threads is not allowed. For concurrent programs, models can be extracted by slicing them by hand into sequential components, and then combing them using the parallel composition feature of CWBN-C, which require a significant amount of users’ work. Also abstractions for non-trivial data structures like records, unions are not considered.

2. Bandera
Bandera [56] was a toolkit developed by Dwyer et al. to automatically extract finite state models from Java source code. The model generated can then be input to several model checkers like SPIN [30] and Java PathFinder [57] by encoding the model in the description language, rewriting the desired property in the specification language that the model checker accepts. Also Bandera interprets counterexamples returned by mapping them back to the source code. Compared to other Java verification tools like Java PathFinder Bandera can construct more compact but safe model for java programs, because of the property-guided abstraction technique it uses. Another contribution of Bandera is the visualization and navigation support it provides to instruct users on selecting appropriate abstraction for interesting variables, which reduces the effort from users. The abstraction types underlying Bandera are interval, modulo, and set abstractions on integer variables [58].

Combination of Slicing and Abstraction

Abstraction and program slicing can both benefit from the incorporation of one into another. Slicing can help the abstract model construction algorithms focus on parts of source code that are of importance. The yardstick to decide whether an element of source code is important depends on the application domain.

Bandera [56] is the most notable instance of slicing being used to improve the efficiency of the abstraction process. In Bandera, the abstract model is constructed in a property-oriented way, i.e., the abstract model extracted will later on be checked against some linear temporal logic property. Therefore, the property which the abstract model will be checked against provides a clear clue to judge whether an element of source code is important, and provides a guideline to derive the slicing criterion.

Bandera works on Java programs, and the slicing scheme in it considers dependencies not only due to data- and control-flow but also due to synchronization of threads. Moreover, while deriving slicing criterion from the property to be checked, Bandera doesn’t consider how to make the property reflect the hypothesis of possible error locations.

In the other direction, performing slicing over an abstract model rather than over conventional dependency graphs of the system will provide more precise, and smaller, slices. Abstract slicing [59] treats slicing as a model checking task over the model constructed using predicate abstraction. In this approach the property a statement has control or data
influence on the given location is characterized as a formula in the branching time temporal logic CTL. Therefore finding all statements satisfying this property is converted to a model checking problem to find all states in the abstract model obeying the CTL formula.

The work presented in Abstract slicing does not consider how to generate a sufficient set of predicates from which the abstract model is constructed. Neither does it consider how to decide accurate hypothesis of error locations as slicing criterions. Our abstraction-driven slicing, in contrast, keeps refining abstractions used and ascertaining hypothesis of slicing criterion via an iterative process. And in our maluse-case-guided error detection approach, coming up with a hypothesis of slicing criterion is reduced to computing conditions stated in an error report. The refinement of abstraction in this approach is also suggested by slicing.

2.4 Failure Analysis Techniques

2.4.1 Failure Investigation and Reporting Schemes

Failure reports yield information about the hazards that threaten safety-critical applications. They, therefore, provide important means of validating risk assessments and safety cases. As a result, a growing number of international standards require that failure-reporting systems to be integrated into safety assurance management schemes. For example, IEC 61508 is widely used as a standard for the development of safety-critical applications that incorporate computer systems. This standard includes recommendations for manufacturers to implement procedures which ensure that hazardous failures are analyzed and the probability of a repeat occurrence is minimized.

The amount of resources that are devoted to the investigation and analysis of failures is, typically, determined by an initial assessment of risks associated with a recurrence of an adverse event. It is, however, possible to identify a number of stages that are common to most failure investigations. For instance, Figure 2.3 provides an overview of the reporting process recommended in the EUROCONTROL guidelines for reporting occurrence in European Air Traffic Management [60].

After a failure has been reported, it is important to gather data about what happened during the incident. This data gathered can aid analysts or investigators to reconstruct the failure scenario. The reconstruction stage usually focuses on the latent, or
Figure 2.3: Failure Reporting Process

long-term, failures that created the conditions in which an incident occurred, which is important for identifying the successful barriers that might protect against similar incidents in the future.

Failure investigation does not end once the causes of an event have been identified. Recommendations must be identified and implemented, if, in the future, the safety of the system is to be protected. It is particularly important to monitor the success or failure of any changes that are made in the aftermath of failure; there is a danger that recommendations may fail to address the underlying causes of adverse events. Finally, it is important to ensure that insights gained from a failure investigation are disseminated both within and between organizations, as a way to protect other stakeholders against similar failures or to elicit information about similar failures that may not have previously been reported.

2.4.2 Post-mortem Analysis Techniques

As an important means to aid reviewers acquire a comprehensive understanding of a program, static slicing techniques have been historically used in post-mortem forensic
analysis to establish the real cause of a reported failure. 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. Moreover, employing static slicing in post-mortem analysis involves forming accurate hypotheses of slicing criteria during the error detection process. Without deep understanding of the system, iterative sessions of slicing process are often required before the actual error cause is identified. The number of iteration depends on the accuracy of hypotheses.

Conditioned slicing does not contribute to the formation of accurate hypotheses either. However, it can reduce the size of slices obtained, compared to static slicing, by focusing on a specific portion of code. This is similar to our approach in Chapter 5 where we make use of the Prerequisite component of Maluse Cases to constrain the initialization of execution traces.

Both slicing-like techniques and our error reported guided approach can benefit from techniques ([61],[62]) that connect the syntax of failure reports and the target software. With a clear connection between terms used in failure reports and entities in an implementation available, it is easier for reviewers to invoke slicing procedures with more accurate hypotheses of slicing criterions. For our approach, on the other hand, such a clear connection can aid eliciting accurate Error Conditions in Maluse Cases, making the error detection process terminate in a less number of iterations.

Besides slicing-based techniques, several techniques and tools have been presented to aid the reviewer to improve his understanding of source code. ARTISAn [63] is a static analysis framework for Java. ARTISAn enhances analysts’ comprehension of programs by providing them a combination of low-level purpose views and high-level usage views, where the purpose view is derived by classifying and labelling program constructs and the usage view is obtained through a def-use analysis to identify regions of elements based on invocation and inheritance relationships. Strategic programming [64] uses generic data-processing actions to traverse through program structures and provide a similar bottom-up understanding of the code.

Error detection during post-mortem analysis shares much in common with corrective maintenance. PSE (Post-mortem Symbolic Execution) [65] 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., type state 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.
Chapter 3

Goal Graph -
A Conceptual Requirement Model

In traditional engineering systems the stake-holders (manufacturers, end-users) drive the development of requirements for a product. However, lately, in the arena of computer networking the International Engineering Task Force (IETF) – a group representing several manufacturers – is in charge of developing new designs/requirements that everyone can follow. This cooperation among potential rivals is needed to make computer networks work. In the arena of medical devices, however, there are no requirements for components, or devices, to work with each other; consequently, there is no need for vendors to cooperate on product lines. The burden of making sense of a number of devices, all of which purportedly have the same functionality, now falls on regulatory agencies.

While it is true that manufacturers have to convince US FDA, by presenting evidences, that its devices work as intended, it is difficult for the regulatory agency to understand and believe these claims of the manufacturers. The primary reason for this situation comes from the fact that manufacturers and regulators have different interests. From the regulation point of view, the requirements document of a software or hardware should reflect the stated needs of the customer or those of statutes [1]. Usually, such a requirement is represented as a property of the physical world that the system should work in. Requirements from the manufacturer’s point of view, however, normally takes the shape of specifications of the system itself, its intended use, and characteristics of the behavior of the software at its interface.
When composing their claim-argument-evidence chains, manufacturers usually overuse implementation details, which makes it difficult for regulators to understand or to feel confident about a device under review. It should be noted that the overuse of details, by manufacturers, could either be due to the fact that the system was decomposed and requirements elaborated that way or it could be that the manufacturer does not have enough time to look at their system from a different perspective.

Therefore, a possible approach to expedite the Pre-Market Review process would be for the regulators to provide guidance to manufacturers on what the appropriate level of manufacturing details, in an application, are. Our Goal Graph model can be looked as an instance of this approach, with the following constraints on them:

- The model shall act as a better knowledge model than what is current practise, and it should allow regulators to encapsulate their understanding and assurance concerns about a common set of medical devices.

- When communicated to manufacturers, such a model could serve as a check list of assurance-related requirements – a list that could be used by manufacturers not only to design and implement their products, but also to compose assurance cases for their products by providing evidence and arguments to the set of requirements.

Hence, the direct benefit of using such a model is that it would relieve regulators from reviewing information/documents unrelated to their task of checking assurance-related requirements, and the Pre-market Review process can then be accelerated.

In this chapter, we will describe a structured, graphical model, called Goal Graph to fulfill the above objectives. The rest of this chapter is organized as follows: the basic structure of Goal Graph is explained in section 3.1; a straightforward algorithm is then presented in section 3.2 to build assurance cases from a Goal Graph; section 3.3 contains a discussion of issues related to developing Goal Graph models; section 3.4 compares our work with others, and lastly, we report our experience of applying the Goal Graph model to generic large volume infusion pumps in section 3.5.

### 3.1 Hierarchy of The Model

A Goal Graph is a graphical model capturing functionality and assurance requirements of a particular system. It can be used by regulators to publish their assurance con-
cerns regarding all systems in a certain realm. System vendors can also use *Goal Graphs* as a starting point to constitute and communicate their assurance claims, as well as relevant objective evidences.

Technically, a *Goal Graph*, as shown in Figure 3.2, is a four-tiered structure, where nodes in each tier are associated with information of a particular kind:

- The top level of a *Goal Graph* establishes the application domain and the use environment, by enumerating the set of rules that characterize all medical devices in a certain category.

- The second level consists of a tree-based description of functionalities for systems in the category, which in turn inspires the derivation of assurance requirements.

- The third level, called *Requirement Hierarchy* level, summarizes assurance requirements derived from functionalities described in the second level. Starting from these requirements, system vendors can construct their claim-argument-evidence chains, enabling regulators to concentrate on higher-level claims.

- The bottom level enumerates a set of quantitative constraints over the system’s performance, which make it possible to connect pre- and post-market analysis.

### 3.1.1 Device

An important fact about software assurance is that no software system, either stand-alone or contained in a complicated device, can be assured for its reliability and safety in all respects and under all conditions. This fact places a ground rule for regulators and manufacturers in introducing assurance-related requirements: to be useful, a requirement must articulate accurately and precisely the properties the system is expected to exhibit, and the assumption about the system’s environment upon which the system’s intended use can be performed. In another word, before reviewing a particular applicant device, the regulatory agency should have a strict means such that the device can be unambiguously categorized and associated with the right set of assurance standards that the device should be checked against with.

There obviously exists multiple ways to categorize a broad set of devices, each of which corresponds to different interest from different stakeholders. Particularly for medical devices, there are three basic types of rules to categorize a device: the environment it can be
Figure 3.1: A Hierarchy of Infusion Pump Types

deployed in, functionalities it can achieve, and its intended use. For example, Diagram 3.1 depicts the hierarchy of the various infusion pump classes in the market. The root of the tree represents the broad category of all infusion pump devices (IP in the figure). Infusion pumps are first classified as external and implanted (in vivo), based on its functioning environments. External pumps can be classified into two main classes - large- or small-volume pumps, each of which differentiates from the other in its functionality, i.e. the volume of fluid it can deliver in one session. Large-volume infusion pumps are further classified based on their intended use, as enteral, therapeutic, and analgesic pumps. Moreover, small-volume pumps can also be divided from the intended use perspective into patient-controlled or non-patient-controlled sub-types.

The top level Device of a Goal Graph is introduced to record how devices in a broad domain are divided from the regulatory agency’s perspective. Every node in the level corresponds to a rigid class of devices realizing same functionalities, working under similar environments, and supposedly being applied to analogous applications.

When the set of assurance requirements have been elucidated explicitly for a device, the regulatory agency also need to decide the level of acceptance to evaluate how the device
Table 3.1: Table for Device *Generic LVP*

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can be used for home therapy?</td>
<td>Yes</td>
</tr>
<tr>
<td>Provides provide treatments?</td>
<td>No</td>
</tr>
<tr>
<td>Is drug or biologic involved?</td>
<td>Yes</td>
</tr>
<tr>
<td>Consists of multiple types of pumps?</td>
<td>No</td>
</tr>
<tr>
<td>Provides diagnostic information?</td>
<td>No</td>
</tr>
<tr>
<td>Provides gravity infusion?</td>
<td>No</td>
</tr>
<tr>
<td>Is a blood establishment device?</td>
<td>No</td>
</tr>
<tr>
<td>Key Words</td>
<td>Generic, Large Volume, LVP</td>
</tr>
</tbody>
</table>

fulfills these requirements. Basically, this decision is made depended on the device’s level of concern, i.e., an estimate of severity of injury that it could permit or inflict on a patient or operators as a result of device failures, flaws, or simply by virtue of employing it for its intended use [66].

At current stage, our model does not directly document the level of acceptance, but chooses to reflect this issue implicitly. To be more specific, our model associates each *Device* node with a table of properties about the corresponding class of devices. These properties, working together, could affirmatively determinate the level of concern for these devices. For example, Table 3.2 lists eight properties for generic large volume infusion pumps, each of which being true will cause the regulatory agency believes that corresponding devices are of major level of concern. As a result, manufacturers are required to perform rigorous verifications on these devices to prove risks have been confidently removed.

### 3.1.2 Components

One possible starting point for assurance analysis would be taking a particular device into consideration. But it is not appropriate for us, given that we are interested in all systems of a particular category. Considering any particular device will likely get us entangled in infinitely many ways of designing and implementing a same functionality. Therefore, we choose to rely on on requirement documentation of existing successful devices, which enables us to capture a minimum set of functionalities that a system in the category should accomplish in order to pass the pre-market review. These functionalities characterize intrinsic properties of systems in the category, and can be used as a reasonable resource for derivation of assurance requirements. Furthermore, the connection between user requirements and functionalities discovered in this stage is also useful for the construction of
traceability chains within the entire system.

This paper focuses on providing a structured view of system functionalities, and leaves it as an open problem on how to extract and evaluate a set of essential functionalities. Functionalities captured can be modelled as a directed graph in the Component Hierarchy level of a Goal Graph, where each node of the graph represents a single functionality; moreover, a table presenting any additional information regarding the functionality that could be useful for comprehension can be associated with this graph node. The format of tables residing in nodes is shown in Table 3.2, in which the key field Functionality records a textual description of tasks that the corresponding functionality should accomplish. Finally, edges within the graph explicitly illustrate dependencies between functionalities.

For a particular functionality, its quantitative and status attributes can be identified and stored in the Attribute List field of its corresponding table. The quantitative attribute of a functionality, for example Volume To Be Infused (VTBI) of the infusion mechanism, serves as a parameter of the system performance and provide a clear clue to derive quantitative requirements. A status attribute, on the other hand, can only be evaluated to a set of specific values, and imposes a process-related requirement. A typical example of status attributes is Delivery Mode in Table 3.2.

In complicated systems functionalities are likely to interact with each other. For example, the delivery of fluid to patients requires that the user interface (either audio or visual) be used to ensure safe delivery of fluid to the patient. Partitioning functionality, of course, requires experience from the domain. In Goal Graphs, interactions between functionalities is simply recorded in Component Referred fields of tables associated with those functionalities involved. Such style of cross-referencing allows a regulator to browse and cross-check correctness of the functionality graph.

3.1.3 Requirements

Williams et al. [67] define claims in assurance cases as statements that a subject has a particular attribute or property. Considering device functionalities as a special category of subjects, requirements in our model can be treated as a claim on devices, from all manufacturers, that fit a particular category. There is no complete discussion, in [67], on what properties or attributes requirements should address. For our situation, reliability and robustness are two primary properties that should necessarily be imposed on functionality
Figure 3.2: Goal Graph Structure
Table 3.2: Table for Component *Fluid Delivery*

<table>
<thead>
<tr>
<th>Keyword(s)</th>
<th>Fluid Delivery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute List</td>
<td>Delivery Mode (Basal or Profile)</td>
</tr>
<tr>
<td>Component(s) Referred</td>
<td>User Interface</td>
</tr>
<tr>
<td>Functionality</td>
<td>Manage the infusion of desired fluid; Program the infusion instruction sequence; Execute infusion programs; Monitor the infusion status</td>
</tr>
</tbody>
</table>

Table 3.3: Table for Requirement *Safe Infusion*

<table>
<thead>
<tr>
<th>Keyword(s)</th>
<th>Infusion, Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement</td>
<td>The device shall provide features to guarantee the programmed infusion session can be executed correctly and accurately as instructed and expected. Also the device shall also provide clear and safe means to deal with typical clinical practices, e.g. modifying program during infusion.</td>
</tr>
<tr>
<td>Component(s) Referred</td>
<td>User Interface</td>
</tr>
<tr>
<td>Contribution Method</td>
<td>AND-node</td>
</tr>
<tr>
<td>Support Methods</td>
<td>Design Precaution, Black Box Testing</td>
</tr>
<tr>
<td>Typical Failure</td>
<td>Infusion of air into patient, Under-infusion</td>
</tr>
</tbody>
</table>

of medical devices. Regulators must be convinced, before they clear a device, that reliability properties are met by the device. An example of such a property is that patients can expect to receive a designed treatment within directed duration by using the device. A robustness property, on the other hand, enforces that a legal device will cause no injury or death to patients due to variations in its operating environment, alteration or loss of functionality.

A realistic *Goal Graph* should also consider and incorporate a wide range of other properties of importance. These include properties regarding quality of devices. An example of such properties is a characterization, based on human factors analysis, of the efficacy of interface built in a class of devices; the importance of the property is underscored by the fact that user mistakes in input contribute to a significant portion of infusion pump failures [68].

Irrespective of what the property is, a requirement is rendered in our *Goal Graph* model as a node at the *Requirement Hierarchy* level. A requirement node may be connected to its subordinate nodes with directed links, reflecting how a property is decomposed to form a tree of subordinate requirements. This decomposition can be repeated until a level
is reached such that all requirements at the leaves are directly analyzable.

Table 3.3 illustrates how information is stored inside requirement nodes. The core of such a table is the Requirement field, which presents a textual description of the corresponding requirement. Auxiliary information relating to a requirement can, although not required, be introduced into the table. For instance, Support Method may be used to provide information on methods that are necessary for fulfilling the requirement. Clearly, such auxiliary fields allow better comprehension of a requirement.

Searching a requirement in a complex Goal Graph is facilitated by two additional fields in the Requirements table: the field Keyword highlights subjects and attributes that a requirement addresses, and Typical Failure enumerates common problems that could arise as a consequence of violating the requirement. The later may also be used in post-mortem analysis to help identify requirements that might have been violated when malfunction is reported. The violated requirements can then be traced back to the offending code, which purportedly satisfies the requirement.

A requirement can be affected by its subordinate requirements in two typical ways. One possibility is that the violation of one of its sub-requirement invalidates the main/parent requirement (denoted as AND-node in Table 3.3). The second possibility is that a parent requirement is unmet only if all of its sub-requirements are not met (also called OR-nodes). The prevalence of the second possibility, however, in regulatory practices is doubtful; this is because a set of subordinate requirements is typically not complete and manufacturers may always add to this list based on features they have implemented. Currently we only introduce the Contribution Method field in Table 3.3 to record the way a sub-requirement may affect its parent, and leave it as future work to study how interaction among requirements can be modeled in Goal Graphs.

3.1.4 Pre-Conditions

Special care is necessary to accommodate, in a Goal Graph, a variety of existing infusion pumps in the market. These products are different from each other not only in their implementation mechanism, but also in features and functionalities they provide. Products that entered the market more recently tend to have more powerful functionalities in order to attract interest from users. For example, infusion pumps introduced in late 1990s allow monitoring of vital signs of patients and its use in infusion adjustment. Clearly
such features cannot be found in older pumps. Consequently, besides being able to capture functional requirements in a *Goal Graph* a reviewer should be able to communicate validity and reliability requirements of advanced features in a *Goal Graph*.

We introduce the notion of **Pre-Condition** to deal with differences between infusion pumps. To guarantee our *Goal Graph* work for all manufactures, such *Pre-Condition* elements possess the *semantics* that if the *Pre-Condition* is satisfied by a certain infusion pump (i.e. the pump provides the functionality that the *Pre-Condition* specifies), all subsequent requirements and constraints of the *Pre-Condition* are required to be satisfied by the pump. For instance, if an infusion pump allows infusion programs to be received over network, as specified by the *Pre-Condition* *Are infusion Instructions Sent Via Network* in Figure 3.5, the pump is required to provide a secure and safe network connection to prevent intrusion attacks; such a requirement is illustrated by the *Requirement* *Safe Network Connection* in Figure 3.5.

It should not be surprising that an infusion pump allows users to customize a feature available for only a certain set of clinical practices. For example, Keep-Vein-Open (KVO) infusion is implemented in most infusion pumps to execute a moderate infusion immediately after normal infusion sessions. However, this mechanism is keep disabled until an explicit activation is received from a medical personnel. Clearly, to model the requirement associated with this dynamically customizable feature we will have to precisely identify the configuration in which it becomes effective.

Considering KVO infusions as an example, it is only meaningful to require that the KVO infusion rate stay within a safe range provided it has been activated. A more complicated situation arises if a feature may be affected by a set of configurations, especially when all features of the device are organized in a layered structure. Consider, for example, a syringe infusion pump that will allow entry of a patient’s weight only after the medical personnel enables *Square Bolus Infusion* and then chooses the programming mode as *Patient-Weight-Based*. In this case, the pre-condition for the requirement on input of patient weight should mention both the required settings.

In *Goal Graphs*, *Pre-Condition* elements are introduced articulating configuration issues mentioned above, given the fact that an assertion *Requirement X should be satisfied when Feature Y is enabled in current configuration*, can be rephrased as *Feature Y is enabled in current configuration is the Pre-Condition that Requirement X should be satisfied*. Furthermore, when a complicated device is under consideration, identifying all
Table 3.4: Table for Constraint *Safe KVO Rate*

<table>
<thead>
<tr>
<th>Keyword(s)</th>
<th>KVO, Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limit(s)</td>
<td><strong>Max KVO Rate, Min KVO Rate</strong></td>
</tr>
<tr>
<td>Referred</td>
<td>KVO Rate</td>
</tr>
<tr>
<td>Parameter(s)</td>
<td><strong>Min KVO Rate ≤ KVO Rate ≤ Max KVO Rate</strong></td>
</tr>
<tr>
<td>Typical Failure</td>
<td>Over- or Under-infusion</td>
</tr>
</tbody>
</table>

configurations affecting a certain requirement can be implemented as collecting all *Pre-Condition* nodes along the path from the root to the requirement in our *Goal Graph*.

### 3.1.5 Constraints

Requirements contained in a *Goal Graph* model can not necessarily be turned into properties that are directly analyzable; the textual representation of requirements prohibits (semi-)automatic manipulations that are desirable for either post-market review or the validation phase in the software development life cycle. However, our experience indicates that a subset of properties can be formalized as logic formulae (in an appropriate theory) and hence be suitable for automatic analysis procedures. Instances of such properties are accuracy, consistency and controllability, all of which share the common characteristic of restricting the behavior of a device to predefined ranges\(^1\). This characteristic allows a user to state requirements uniformly as assertions on relationship between parameters of the device and some constants. For example, we can phrase a controllability property of KVO rates in generic infusion pumps as:

*The KVO rate, if a KVO infusion is enabled, should be no less than the minimum KVO rate, and no greater than the maximum KVO rate that the pump allows.*

The difficulty of formalizing assertion-like requirements comes from the fluctuation essence possessed by entities involved in the property. Studying the assertion above, two types of entities can be identified:

- Entities abstracting device limits, irrespective of whether they are or not configurable by users. Minimum and maximum KVO rates in the statement given above are examples.

\(^1\)In assurance case area, claims corresponding to such properties are categorized as *value-or-relation* claims in [67].
- Entities that discuss performance parameters, such as the KVO rate above.

It is obvious that device performance parameters will vary over time, and that device limits may also change between system runs due to user actions to modify configuration settings. Therefore, no specific values work for these entities in all situations. Considering this, we introduce symbolic names to abstract all possible values to which these entities might be evaluated. For example, symbolic names $KVO_{Rate}$, $Min_{KVO_{Rate}}$ and $Max_{KVO_{Rate}}$ are used to abstract KVO rate in any KVO infusion session, the minimum and maximum KVO rate that the pump allows respectively. With the help of symbolic names, the above assertion can be translated into a formula as follows:

$$Min_{KVO_{Rate}} \leq KVO_{Rate} \leq Max_{KVO_{Rate}}$$

**Constraints** elements in our Goal Graph model provide a scheme to encapsulate assertion-like requirements, in which both information formalized as formulae and auxiliary indexing information are stored in a tabular format as shown in Table 3.4.

Because of **Constraints** elements, our Goal Graph model is similar to safety case patterns [16], in the sense that a Goal Graph can be used for arbitrary analysis task by instantiating it based on information relevant to the task. More specifically, a Goal Graph can be reused in one of two ways:

1. Manufacturers can obtain a program invariant from a **Constraints** element by replacing symbolic names with corresponding variables in their implementations. Thus, checking whether a device satisfies the **Constraints** is accomplished by verifying whether the program invariant holds for an implementation through either formal testing or static analysis.

2. During a post-market review a **Constraints** element can be transformed to a logical formula with symbolic names replaced by specific limits of the device under the specific configuration when a failure occurs. Such a formula can then be used in forensic analysis [69], where reasons for reported failures can be isolated.
3.2 Connecting Goal Graphs and Assurance Cases

Provided a Goal Graph, the process of building the corresponding assurance case is straightforward, and can be characterized as a three-step process:

**Step 1** For any requirement $R$ in the Goal Graph, if all preconditions on the path from the root requirement to $R$ are satisfied, then goto step 1.1, otherwise ignore $R$.

1.1 Introduce into the resultant assurance case a claim node $C$ The device satisfies $R$, as well as directed edges connecting from all claim nodes that correspond to $R$’s parent requirements to $C$. If $R$ has no sub-requirement. Provide evidences and arguments that can justify $C$, otherwise goto step 1.2.

1.2 If $R$ is not covered by all its subordinate requirements, i.e. the satisfaction of all its subordinate requirements cannot directly imply the satisfaction of $R$, provide evidences and arguments that are sufficient enough to justify $R$ being satisfied under the assumption that all $R$’s sub-requirements have been satisfied.

**Step 2** Introduce into the resultant assurance case a claim node The device has
achieved adequate assurance., which has out-going edges destining at all claims that correspond to top-level assurance requirements in the Goal Graph.

**Step 3** For any claim $C$, if it has more than one sub-claims $C_1, C_2, ..., C_n$, then introduce into the resultant assurance case a strategy node **The Claim C is substantiated by sub-claims $C_1, C_2, ..., C_n$** and then redirect all edges from $C$ to $C_i$ as from the strategy node to $C_i$.

Figure 3.3 presents in part how corresponding claims, evidences and arguments are organized by following the organization of the requirement tree. In the figure, dashed directed edges indicate the one-to-one relationships between requirements and claims.

The feasibility and efficiency of the above algorithm, however, is greatly depended on the degree of confidence that evidences attached can substantiate its corresponding claims. For a claim that cannot be established on the basis of an evidence alone, strategies and complicated arguments are often necessary gluing a set of evidences together to form an objective proof for the claim. In this case, the automatic generation of an assurance case is obviously impossible.

### 3.3 Discussions

A Goal Graph is useful only if it is concise and consistent. We should also give equal attention to sufficient coverage of assurance objectives and simplicity of representation; the former establishes a desirable level of confidence that adequate assurance is realized in the device and the latter eases the comprehension of the graph. It is also important that the use of Goal Graphs should not impair the freedom that manufacturers have to design and develop their product. In this section, we explore issues relating to improving the quality and usefulness of Goal Graphs.

1. Completeness: Ideally, a Goal Graph should consist of a complete set of requirements standing for real world expectations. Absolute assurance, however, is not practical for medical devices that involve principles from several engineering disciplines. Therefore, regulatory agencies pay more attention to robustness-related assurances, hoping to save the public from being adversely impacted by malfunctioning medical devices. No further restrictions are imposed by regulators on the functioning of medical device except that it should accomplish tasks claimed in their manuals.
As there are no standard measures to verify either the completeness of a Goal Graph or that a medical device is free of errors, knowledge in the field serves as the ultimate help that a reviewer can depend upon. For robustness-related assurance, however, the following two types of documentation can be used to address completeness, with which potential risks or harms relating to devices under review could be exhaustively enumerated:

- Documentation of Failure Model and Effect Analysis (FMEA) from previous devices, which strongly suggests risks that might be encountered under regular uses of devices in directed circumstances.

- Surveillance reports, which might expose behavior patterns that users undertook leading to typical failure or malfunctions involved with target devices.

2. Extensibility: Although our Goal Graph model discussed so far is designed for regulating infusion pumps, it can be customized for other practices. To be more specific, fields can be freely added or deleted from instrumented tables for Component, Requirement and Constraint nodes, if doing so can facilitate regulation processes or ease comprehension of the customized Goal Graph. However, a principle that any possible customization should comply with is to make resultant tables as compact as possible, otherwise unnecessary formality and unusually large amount of information might be burdensome on regulators.

3. Level of Details: The quality of a Goal Graph, to a large extent, depends on what level of details the graph has addressed. Here, by level of details, we mean how large set of functionalities have been presented and how accurately the interaction and cooperation between functionalities have been described. Therefore, the depth of the functionality tree in a Goal Graph can work as a direct measurement of the level of details.

A straightforward benefit brought by introducing more details into a Goal Graph is that requirements are described very precisely and strictly, making the task of constructing evidence and necessary arguments easier. Too many details, on the other hand, makes pre-market review harder; in fact, it could cause rejection of an application for a devices because it has been implemented in an alternative though
Thus, the level of details contained in the Goal Graph should be constantly adjusted based on regulation experience such that a smooth trade-off between applicability and precision can be achieved. When a Goal Graph is initially constructed, however, a rule of thumb to be followed is that decomposition of functionalities should proceed conservatively and terminate when differences in device functionalities of the same class emerge, even though Pre-condition nodes can be used to facilitate accommodating a variety of implementations. The abuse of Pre-condition nodes may span the entire functionality tree, making it less possible that the requirements are consistent and understandable.

3.4 Related Work

In the Requirements Engineering (RE) field, the concept of goal refers to an objective the system under consideration should achieve [70], while the term requirement specifies how a goal should be accomplished by a proposed system, especially through the cooperation of components in the system [71]. Therefore, the term requirement used in our Goal Graph model, which defines a particular attribute or property to be satisfied by all devices in a same category, shares a similar meaning with goal, rather than requirement, adopted in RE.

A number of techniques, such as [72, 73, 74], have been proposed to specify goals, either formally or semi-formally. Generally, semi-formal specifications declare goals in terms of their types, attributes, and connections with others. Moreover, goals in [74, 75] are represented as parameters of a set of basic verbs with predefined semantics, allowing a lightweight analysis of goals. Formal specifications, on the contrary, assert the goal formulation in a logic system amenable to reason. For example, the KAOS tool allows users to write goals in real-time linear temporal logic formulae.

Since one of primary objectives of the Goal Graph model is to improve the communication between regulators and manufacturers, we avoided formal requirement specifications in the model, saving stakeholders, often non-experts, from understanding intricate logic formulae. Instead, each requirement in the model is assigned with a tabular specification, in which the information relevant to the requirement while critical to regulation tasks is also included.

Many different types of links have been introduced in the literature to relate goals
with each other and with other entities of requirement models. Links between goals formalize the way goals could support other goals. AND/OR graphs [73] are aimed at capturing the decomposition relations between a goal to a set of subgoals. AND (resp. OR) links means that satisfying all (resp. one of) subgoals in the decomposition is sufficient for satisfying the parent goal. When regulating different systems in a same class, however, the decomposition of a goal might result in different sets of subgoals. In such situations, our Goal Graph model chooses to document only the common subset of subgoals that a goal might be possibly decomposed into for all systems under concern. To denote such special version of decomposition links between goals, we borrow weaker versions of AND/OR links introduced in [72]. The semantic rules are now as follows: if a goal is AND-decomposed into subgoals and all subgoals are satisfied, then the parent goal is satisfiable.

Beside inter-goal links, goals are in general also linked to operations [74], scenarios [73], and objects and agents [73, 76]. However, due to various categorization rules, systems that fall into a same category might implement totally different mechanisms, possess distinct behaviors, and perform diverse styles of interactions with users. Therefore, it is often impossible to extract common scenarios and to identify an identical set of entities for all systems in the category. As a result, links relating goals to elements in requirement models are not suitable for our Goal Graph model.

The elicitation and elaboration of goals and requirements for a family of systems have been studied by the Commonality and Variability Analysis (CVA) [77] techniques. However, the effectiveness and efficacy of such techniques rely greatly on whether the re-development, oracle, and organization hypotheses can be met by all member systems in the family. These hypotheses essentially require requirement engineers and developers to be capable of predicting variations existing in all members of the family, identifying what they have in common, and reusing the common aspects in producing the variations. One perfect family of systems are manufactures from the same product line [78], where variations between manufactures are configurable and controllable. However, the prediction of variations between all systems in the same category might not often possible in regulation practices. For example, infusion pumps submitted would be considered by regulators as generic Large Volume Infusion pumps (LVP) if they can deliver over 10000 milliliter of fluid in one session, no matter what delivery mechanisms they implement inside. The prediction of variations between all LVP pumps is impossible and meaningless. In our work, we consider only commonalities, i.e. the common functionalities that all these systems should
achieve, between a class of systems, and leave out various designs and implements that these systems might have. Only in this way can we identify the common set of safety requirements for a class of systems and help regulators to evaluate these systems against a minimal assurance standard.

Techniques that have been proposed to assist the development of high-quality assurance cases focus on constructing, organizing, validating and maintaining assurance arguments [79]. One established instance of these techniques is Goal Structuring Notation (GSN) [15], which helps users to explicitly represent individual elements of any assurance arguments and the relationships that exist between these elements with graphical notations. By allowing users to denote the assurance levels for arguments, GSN can also automatically calculate the overall assurance level of the system under consideration. Our Goal Graph model, intending to provide an initial set of goals upon which assurance cases can be developed, can therefore be considered as orthogonal to these techniques for assurance cases.

3.5 A Case Study on Large Volume Infusion Pumps

An infusion pump is a typical safety-critical device in which software is deployed to control the infusion set and to ease the use of the pump. According to commonly accepted categorization, an infusion pump is considered to be large-volume (therapeutic/analgesic) pump (LVP) if it can deliver more than 1 liter of fluid to a patient in a single session. As

![Figure 3.4: Top Level Components of LVP Software](image)
LVPs have been involved in numerous incidents causing serious injuries or deaths, techniques to ensure that such pumps work well is of interest to all stake-holders. The primary focus, of our work, is on enhancing the comprehension of assurance objectives relating to reliable and robust functioning of LVPs, and hence ensuring quality design and development of their software components.

In this section, we present a Goal Graph constructed for LVPs, as proof of the flexibility and sufficiency of our model in real practice. In our case study, assurance-related requirements are retrieved from three primary sources: previous pre-market submissions from various manufacturers that have been proven as safe and efficient in realistic practices, purported failures and deficiencies collected from surveillance reports reasons of which can be attributed to software defects, and generalized guidelines including normative documents from the regulatory agency and international standards (e.g. ISO14971 and IEC60601).

The entire goal graph thus developed for LVP devices could be divided into two subtrees: the component hierarchy subtree was organized in 3 layers with 25 sub-component nodes inside; the requirement subtree that contained 223 requirement nodes locating in 8 layers and 24 constraint nodes in total. The graph was developed using the Adelard Safety Case Editor (ASCE) \[80\], and a style of notation \(^2\) similar to the Goal Structure Notation \[15\]. The notation was borrowed to distinguish between four categories of nodes in the graph.

The following subsections describe the construction process, as well as underlying developing methodology, in a top-down fashion starting from the very top level device node Generic LVP, and ending up at the bottom level of safety requirements relevant to infusion safety. For brevity, we narrow the description down to requirements tightly bounded to the reliability and safeness of infusion sessions, and omit tables associated with these requirements to keep the presentation simple.

### 3.5.1 Top Component Hierarchy

The core functionality of a LVP is to deliver an appropriate type and volume of medication to patients as directed by a treatment plan, where the treatment plan, which could be a prescription for a single or a series of infusion sessions, could either be transmitted

---

\(^2\)In our notation, **Component** elements are expressed in diamond boxes, **Requirement** and **Constraints** nodes are written in rectangle boxes subscribed with **Goal** and **Cond** respectively, and **Pre-Condition** nodes are depicted as ellipse shapes marked with **Cond**.
over network or be programmed by users on-site. In our Goal Graph, this functionality is formalized by the Delivery Mechanism component node, as shown in Figure 3.4.

Whenever medical device software interacts with human users, the regulatory agency enforces simple while unambiguous logic to be implemented in the software, in order to reduce the number of device failures caused by human misoperations. In our graph, we formalize this functionality in Alarm Management and User Interface components, the first of which takes care of configuring and controlling of audible or visual alarm/alert systems upon requests. The second, on the other hand, takes inputs from users and check its correctness and integrity. Moreover, the User Interface component displays out succinctly and clearly information on devices, infusion procedures and patients.

Accountability is an important issue when a medical device encounters a failure or system crash. Therefore, a LVP, if going into the market, should provide the functionality of logging configurations of facilities that are used for infusion programming and execution activities. Such a log is necessary for the recovery of context and scenario when a failure occurred, and can help analysts to determinate to whom the failure should be ascribed. The Alarm Management component in Figure 3.4 is hence introduced to characterize such a functionality.

3.5.2 Delivery Mechanism

To accomplish its primary task, a reasonable Delivery Mechanism component is required to provide necessary auxiliary functionalities, such as it should be able to log medication infused (Component Prime) and to optionally provide recommendations for treatment plans (Component Decision Support), as explained in Figure 3.5. However, we concentrate on its critical task of administering delivery of fluid (Component Fluid Delivery). Clearly, it is not practical to allow implementations of LVPs to influence derivation of assurance requirements for the Fluid Delivery component, given the fact that implementations can be realized in various ways. For example, the power driving the LVPs can come from springs, pistons or rolling carriers. Therefore, we chose to overlook implementation details and treat LVPs as black boxes. As a result, all requirements derived either focus on the delivery of treatment to be performed as expected (Requirement Reliable Infusion), or restricts that the delivery should not impair patients if abnormalities happen (Requirement Robust Infusion).
Figure 3.5: Goal Sub-Graph for Fluid Delivery
Essentially, the implication of a reliability requirement is that desirable consequences can be anticipated if acceptable inputs are fed to the device. Hence, reliability requirements derived for a device can not be claimed as complete unless all acceptable inputs have been covered. Therefore, we consider in our case study both situations of how an infusion program can be imported into the pump, either programmed locally or transmitted via network. For the later situation, an extra requirement (Requirement Safe Network Connection) is enforced, to prohibit corruption or tampering on the transmitted program.

Once all possible inputs have been identified, the reliability requirement is coded and decomposed into two sub-requirements: (Requirement The Program Should Pass Safety Check) and (Requirement Safe Program Execution). The former prescribes that a safety check on the input should be carried out before the beginning the process of infusion. The later, on the other hand, can be further broken down into two sub-requirements – that the process be safe, (Requirement Safe Process), and that an appointed volume of drug be infused into a patient at directed rate (Requirement Safe Infusion).

### 3.5.3 Correctness of Infusion Process

A technique widely adopted in software design is to depict software products as state machines. With this technique, the concept of state is introduced to abstract certain operational status of the device, and functioning of the device is characterized as transitions between states. If the state machine is available for a device, requirements can be stated that codify the stability of states and that transitions indeed cause a change of state, and the meaning they carry.

We borrowed the concept of Finite State Machine (FSM) to characterize status-correctness requirements for regular infusion processes of LVPs, in which infusion sessions were generalized as a series of transitions among the following states:

- **Ready**: no infusion is in process and an infusion program has been imported.
- **Infuse**: the infusion is on process.
- **Halt**: the ongoing infusion is paused.

Transitions between these states and associated correctness requirements can be summed up as follow (details of related requirements are illustrated in Figure 3.6):
1. **Start**: transitions from state **Ready** to **Infuse**.

   **Requirements**:
   
   - The device should ask users to confirm that desired infusion programs have been loaded before the infusion can start.
   
   - If the infusion is programmed to start at a future point in time, audible notifications should be presented to users when the time for infusion arrives.

2. **Stop/Cancel**: transitions from state **Infuse** to **Ready/Halt**.

   **Requirements**:
   
   - The pump should have a safe measure to manipulate the last mechanical stroke according to the stop request.
   
   - The VTBI value should be adjusted correctly based on the volume of drug infused during the last stroke.

3. **Restore**: transitions from state **Halt** to **Infuse**.

   **Requirements**:
   
   - The restoration of a paused infusion session should not proceed unless the validity of the restoration has been ensured.
   
   - Moreover, infusion parameters of the paused session should be kept after the restoration.

   In order to protect ongoing infusions from unwanted interventions, LVPs are required to provide means to lock the keypad or any other interface that accepts input from users during infusions. Consequently, the **Infuse** state can be divided into two sub-states **Locked** and **Unlocked**. Safety requirements on transitions between these sub-states can hence be summarized as follows:
   
   - The lock or unlock request from users should not be accepted unless the authority of users has been verified.
   
   - The input interface should be unlocked automatically in emergency situations.
   
   - The lock request should not be accepted when failures occur and only a persistent press of the unlock button should cause the input interface to become unlocked.
3.5.4 Reliability of Infusion Execution

As opposed to the correctness requirements in last section, the Safe Infusion sub-tree consists of a set of requirements characterizing the functioning process of a device. Essentially, these requirements enforce a set of principles to be obeyed by the system’s performance while it is functioning. The principle behind a particular performance parameter could be either quantitative, as setting up a threshold for the parameter, or qualitative, typically described by a rule that the parameter should follow during the entire process. For instance, the requirement Safe Post Rate in Figure 3.7 uses a quantitative principle; it states that the infusion rate should be confined to the range of the programmed infusion rate and the KVO rate. On the other hand, the requirement No Change to Patient Info sets up a qualitative principle for the Patient Information parameter, prohibiting it from being changed irrespective of any change made to the current infusion program.

The derivation of reliability requirements can proceed in two steps:

- We first identify performance parameters critical to the quality of the functioning process, and then
- collect principles that the parameters need to obey.

Considering our case, we observed that the quality of an infusion execution is basically based on the infusion rate and the volume of medication infused. We then extracted principles for these two parameters to obey (as illustrated in Figure 3.7). Listed here are requirements just associated with the infusion rate; the requirements for infusion volume can be derived in an analogous way.

1. Quantitative requirements for the infusion rate:

   **Requirement Rate Accuracy:** The actual infusion rate should follow the programmed rate at an acceptable precision range.

   **Requirement Update Rate:** The rate shall be updated as the newly programmed rate at the end of infusion if another session starts immediately.

   **Requirements Safe Post Rate/ Safe Rate Switch:** Otherwise, the pump shall transit into KVO mode when the infusion ends and the new flow rate should be reset as the smaller of the KVO rate and the previous rate.
Requirement **Safe KVO Rate**: KVO rates shall stay in a safe range specified by current configuration.

Requirement **Safe Recalculation**: The infusion rate should be recalculated as \((\text{VTBI/Infusion Duration})\) if either of these values are modified during the infusion.

2. Qualitative requirements for the infusion rate:

   Requirement **Single Rate**: The infusion rate should be unique anytime during the infusion.

   Requirement **Keep Rate**: The infusion rate should be unchanged when the infusion program being executed is modified.

   A point worthy of mentioning is that regulators can freely organize reliability requirements in arbitrary ways to facilitate an easy understanding of the **Goal Graph**. They can also glue related requirements into an intermediate conceptual requirement, based on factors other than performance parameters. For example, in Figure 3.7, a requirement **Safe Modification** is introduced to highlight all requirements related to the modification of infusion programs; this requirement is based on the perspective of clinical practice.
Figure 3.6: Goal Sub-Graph for Safe Process
Figure 3.7: Goal Sub-Graph for Safe Infusion
Chapter 4

Abstraction for Program Comprehension

Post-mortem analysis – or the process of tracing software failure to source code – is the most challenging activity after a software system has been released to the market. When a software failure is reported, the reviewer, usually coming from a regulatory agency or a third party different from the original developers, has to spent a great chunk of effort to understand the code and locate the cause of a failure. The task of post-mortem analysis is more complicated than corrective maintenance, because the former is usually accomplished in a situation of sparse documentation and inaccessibility to the original developers. Moreover, when carrying on post-mortem analysis, the reviewer does not always have access to execution code or implementation environments. This restricts the reviewer to seek help to only static analysis techniques.

Traditionally, program slicing techniques have been used to perform post-mortem analysis [81]. However, slicing techniques provides only a limited, scaled-down view of the program. When the connection between functionalities and code is obscure to the reviewer, a wider-scope, highly-abstract view of the program has to be provided to the reviewer for his complete understanding of the system. Another problem with static slicing techniques is that slices obtained by such techniques are generally too large to be of much use. This problem gets exacerbated when the system being analyzed is large and complicated in structure.

Abstraction-driven slicing [82] addresses these two problems by combining abstrac-
tion with static slicing. With the aid of visualization, abstraction in this framework helps obtain a highly abstract cognition model from each slice constructed. This cognition model saves the reviewer from manually scanning the slices for confirming his hypothesis regarding the connection between functionalities and the implementation, and hence improves the reviewer’s comprehension of the system. Our case studies shows that the integration of slicing and abstraction can be leveraged to be a more efficient approach to post-mortem analysis.

The rest of this chapter describes the basic procedure of the framework, as well as specific effort on integrating abstraction and slicing smoothly. Moreover, our attempts to improving the usability of the framework is also presented. Hence, later sections in this chapter are arranged as follows: first of all, a high level overview of the abstraction-driven slicing approach is sketched in section 4.1. Also included in this section is the program comprehension model underlying the integrated framework. Section 4.2 provides the algorithm to automatically generate abstraction criteria based on slices and slicing criterions. We will describes in section 4.3 on how static slicing is implemented directly on an intermediate representation of C program to allow the iterative interaction between abstraction and slicing. After that, the algorithm to extract a hierarchical graph representation from the abstract model is presented in section 4.4. Lastly, in section 4.5 we summarize our experience of applying the framework onto realistic post-mortem analysis.

4.1 Abstraction-Driven Slicing

The concept of program comprehension is firstly defined by deLucia et al. [84] 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. Research has been proposed to improve analysts’ comprehension of programs, and at the same time save them from the burden of manually scanning the entire code. Based on how the comprehension bridges the higher and lower level description of the system, research in this area[85, 86, 87] can be categorized into three styles:

**Top-down Style:** In applications where the code under consideration is familiar and a description of the conceptual components of the application domain is available, the

---

1 All of the work in this chapter is original contribution of the author. Some of the work was carried in collaboration with R. P. Jetley, which was reported in [82]. A version of Sections 4.1, 4.3, 4.4, and 4.5 also appear in [82] and [83].
analyst exploits his knowledge of the code to formulate hypotheses about the meaning of the program segments being analyzed, and confirms the hypothesis by scanning the code construct relevant. The confirmation will produces new sub-goals to be verified, and leads to scanning code again. Program slicing falls into this style serving as an effective code scanning technique that allows the analyst to verify and refine hypotheses.

**Bottom-up Style:** In applications where the code under consideration is completely new, the analyst starts the comprehension process by firstly exploiting his knowledge about functionalities of basic structures in the code, e.g., code blocks and procedures. A more abstract model is built up for the code under analysis by chunking micro-structures into macro-structures, and via cross-referencing. The process can continue recursively until the connection between the knowledge of real-world problem domain and the code is clearly discovered. Abstraction works as a good facilitator for bottom-up comprehension by automatically extracting abstract models from source code.

**Integrated Style:** Mayrhauser et al. [87] suggest to conduct program comprehension to be accomplished through a process switching continuously between the top-down and bottom-up directions. Our abstraction-driven slicing approach is actually an instance of program comprehension methods in such a style, where 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.

The process in our approach to facilitate program comprehension is given in Algorithm 1. 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, $M$, which is highly abstract to begin with keeps getting more and 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.

Based on the Algorithm 1, we implement a semi-automatic tool, called CAdS, based upon the CWolf architecture. The tool is semi-automatic because the analyst can
**Algorithm 1 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 $S$.
- 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**

intercept the iterative process by specializing slicing criterion and refining abstraction criteria. The most important benefit of CAdS can bring to the comprehension process is to save the significant amount of effort that is required for the analyst to define abstraction criteria and to map abstract models to slices.

CAdS consists of three major components: a model generator (CWMB) extracting abstract models from source code, a CWI file slicer implementing static slicing, and an abstraction map generator generating graphic representation from the abstract model. An overview of the CAdS tool is illustrated in Figure 4.1.

In order to seamlessly integrate abstraction into our tool, three problems have to be solved:

1. How to automatically generate a sufficiently rich, but not too precise, abstraction criteria from the source code?

2. How to realize successive slicing given the fact that slices obtained by most static slicing tools are no longer compilable, which prohibit them from being sliced again?

3. How to generate a representation of the abstract model to make it easier for the analyst to browse and understand, given the fact that the abstract model is usually too large to be followed for large systems?

Later sections in this chapter will describe our solutions to these problems.
Figure 4.1: The CAdS System Architecture
4.2 Automatic Generation of Abstraction Criteria

A key concern of applying abstraction is to select the level of details that abstraction should appropriately keep. This selection should be carefully made by considering the tradeoff between the overhead of exploring the abstract model and the details this model can provide. However, deciding whether the abstraction is ‘appropriate’ is totally application-oriented, and it is not possible to design an automatic algorithm that can be universally used in any application domain to find out the best abstraction. Abstraction Refinement [12] is a semi-automatic approach that, on termination, can discover an abstraction approaching the best one. Starting from an initial coarse abstraction, Abstraction Refinement continuously refines abstractions iteratively until sufficient details are discovered.

Our framework follows the Abstraction Refinement paradigm to find out the optimal abstraction. After each call to extracting the abstract model from the slice, the analyst decides whether the abstraction selected is sufficient enough to solve the problem, i.e., to form a clear judgement that the slice considered is or is not related to the error under consideration. The judgement, once made, asserts that current abstraction is optimal. Otherwise the reason why the judgement cannot be made contributes clues to refine the current abstraction. In CAdeS, users are allowed to refine abstraction by providing a updated Abstraction Map to specify more precise abstraction criteria. However, when and how to refine abstraction depends on the user’s wisdom. One contribution that CAdeS can make in this process is to save the user from checking and updating the abstraction criteria for every variable. The user can identify certain variables as being interesting; subsequential analysis provides updated criteria for them to be abstracted. The abstraction map generator component in CAdeS will scan the structure of the slice and update abstraction criteria for other variables based on the information the user provides.

One important reason why the analyst cannot obtain a clear understanding of the slice from the abstract model lies in the existence of spurious paths in the model. Spurious paths are those paths that do not belong to the system behavior but get introduced into the abstract model because of a coarse level of abstraction. Based on our observation obtained from our case studies, we infer that spurious paths are introduced when branching statements are interpreted very conservatively. In order to reduce the number of spurious paths introduced, our framework enforces an appropriate type of abstraction for these variables occurring in conditionals. The particular types of abstraction that is associate with
variables appearing in conditionals depends on the preliminary abstraction criteria the user provides, for such variables, and the format of conditionals.

An ideal type of abstraction for a variable occurring in some conditional in the program is one that can keep sufficient information to discriminate whether or not this conditional can be evaluate to specific values (such as true or false) that leads to different choice of branches. Given a list of interesting values, the Partition abstraction implemented in CWolf is an ideal type of abstraction.

The Partition abstraction divides the integer domain $\mathbb{Z}$ into a set $T$ of disjoin interval domains, which must cover all integer values.

By making a subset of ranges, each of which contains only one integer, the partition abstraction can identify a set of interesting value for some data variable. Figure 4.2 illustrates one such application of partition abstraction. In the example, the variable $ch$ is read from input, and at state 1, right after the execution of `getc` statement, no conclusion can be made on $ch$’s value. By analyzing the program, we can infer that only the set of values 0, 97, 98 of $ch$ will effect the switch statement in the program, and that other values are not important. Thus, $ch$ can be abstracted using partition abstraction as $\{< -\infty, -1>, <0, 96>, <97, 97>, <98, 98>, <99, +\infty>\}$. Using the chosen Partition abstraction as a prototype of abstraction for each interesting variable $x$ (i.e., variables occurring in slicing criterion or conditionals), the task of associating it with an appropriate type of abstraction is converted to identifying the set $IVL(x)$ of interesting values for it.

Since CWolf can only abstract integer-typed values, the only formats of conditionals our approach can deal with are:

1. Predicates having only one integer-typed variable;
2. Predicates that can be converted to the form $x \ rel \ c_1 y + c_2$ where $x$ and $y$ are both integer-typed variables, $\ rel$ is a relational operator and $c_1$ and $c_2$ are both integer constants.
3. Formulae obtained by combining, using logical connectives, predicates satisfying 1 and 2.

Suppose we already know the set $IVL(y)$ of interesting values for variable $y$, we may still not be able to evaluate the truth value of predicates. For example, we can decide the truth value of $x \ == \ 2y + 1$ if the abstraction type we choose for $x$ does not keep the
Figure 4.2: An Example of Partition Abstraction

information whether the value of $x$ falls into the set of $L' = \{2v + 1 \mid v \in L\}$. To solve this problem, we should include $L'$ into $IVL(x)$. Therefore, the information we collect for $y$ is used to update the information we collect for $y$. In another words, information flows from the right-hand-side of the equality to its left-hand-side. The information flow in the other direction can also be accomplished if we can detect the set of interesting values for $x$, and not for $y$.

However, the information flow in an (in-)equality in both direction might lead to infinite increase in the size of interesting value set for the variable on either left- or right-hand-side. For the above example, suppose we have $IVL(x) = [1, 2]$ and $IVL(y) = [1]$ for $y$. We first update $IVL(x)$ to $IVL(x)' = [1, 2, 3]$ based on $IVL(y)$, then update $IVL(y)$ to $IVL(y)' = [1, 2]$ based on $IVL(x)'$, and then update $IVL(x)'$ to $IVL(x)'' = [1, 2, 3, 5]$ based on $IVL(y)'$, and so on.

Our answer to this problem is to first choose an appropriate rank value for each

```c
#include <stdio.h>

int main() {
    char ch;
    int state = 0;
    while (1) {
        ch = getc(stdin);
        if (ch == 'a' || ch < 0)
            return 0;
        switch(state) {
            case 0:
                if (ch==b') ++state
                    else if (ch != 'a') re
                        break;
            case 1:
                if (ch != 'b') return 1;
            break;
            default: break:
        }
    }
}
```
Table 4.1: Information Types

<table>
<thead>
<tr>
<th>Information Types of x</th>
<th>Rank (R(x))</th>
<th>Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dom</td>
<td>1</td>
<td>x occurs in a predicate of the form x rel c, where rel is a relational operator and c is a constant</td>
</tr>
<tr>
<td>Selected</td>
<td>2</td>
<td>x is specified in the slicing criterion</td>
</tr>
<tr>
<td>Relative</td>
<td>3</td>
<td>x occurs in a predicate that can be converted to x rel c1y ± c2, where y is of Dom type, and c1 and c2 are constants</td>
</tr>
<tr>
<td>Ignore</td>
<td>4</td>
<td>Otherwise</td>
</tr>
</tbody>
</table>

side of the (in-)equality, and then only allow information to flow from the side with lower to the other side with higher rank. The process of determine the rank value for an expression works as follows:

1. First the rank for each variable in the program is determined based on where it occurs. In particular, the slicing criterion file is first scanned and all variables specified in it are associated with a rank value of 2 and an information type Selected indicating their contribution to the slicing criterion. Secondly a syntax-based search is made over the code. During the search, we decide, using the rules shown in Table 4.1, the information type and rank of x (denoted as T(x) and R(x) respectively) for each of its occurrence in conditionals. The final rank for a variable x is the minimum rank among these having been determined for x according to all its occurrences in the code and in the slicing criterion file.

2. The rank of an expression can then be deduced based on its structure, as shown in Table 4.2. Moreover, we calculate the interesting value set for an expression e (denoted as IVL(e)) based on its structure and the interesting value sets of variable occurring in it. Details of this calculation is also shown in Table 4.2.

<table>
<thead>
<tr>
<th>e</th>
<th>Rank (R(e))</th>
<th>IVL(e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>0</td>
<td>[c]</td>
</tr>
<tr>
<td>x</td>
<td>R(x)</td>
<td>IVL(x)</td>
</tr>
<tr>
<td>¬e₁</td>
<td>R(e₁)</td>
<td>{1 − v</td>
</tr>
<tr>
<td>ce₁</td>
<td>R(e₁)</td>
<td>{cv</td>
</tr>
<tr>
<td>e₁ ± e₂</td>
<td>max(R(e₁), R(e₂))</td>
<td>{v₁ ± v₂</td>
</tr>
<tr>
<td>Otherwise</td>
<td>4</td>
<td>∅</td>
</tr>
</tbody>
</table>
The process to collect the set of interesting values for each variable and propagate it all over the program is a recursive one, i.e., the syntax of a module in the program might be scanned multiple times because the new information collected from other modules should be propagated in it. This recursive process terminates only when the interesting value sets for all variables are stable. To expedite the termination of this process, we implement it in a modular-based style. In detail, all functions in the program are first grouped into Strongly Connected Components (SCCs for short) based on the function call graph of the program, and all SCCs are sorted into a list $SCC_1, SCC_2, ..., SCC_n$ (In Algorithm 2, we refer such a list as a well-ordered list), s.t. any function in $SCC_i$ cannot call functions from $SCC_j$ if $i < j$. Provided with such a list, the process of finding the sets of interesting values for all interesting variables is divided into sub-steps: information is first collected and propagated in a lower-indexed $SCC_i$, and the process will not proceed to a higher-indexed one unless the information we can collect in $SCC_i$ gets stable. Details of the process is shown in Algorithm 2.

```
1 #include <stdio.h>
2 int main() {     
3     int i, j, k;
4     k = 0;
5     scanf("%d", &j);
6     switch(i) {
7         case 0: j = 1; break;
8         case 1: j = 2; break;
9         default: j = 6;
10     }
11     if (i == 1) printf("i is either 0 or 1"");
12     if (j == 0 && j < i) k = i;
13     if (k > 0)           
14         printf("k is positive\n");
15     return 0;
16 }
```

```
Original Abstract Map
file "part.c"
    fun main () : top ()

Slicing Criterion

Updated Abstract Map
file "part.c"
    fun main () : top ()
        var i : part(0, 1, 2);
        var j : part(0, 1, 2, 3);
        var k : part(0, 1);
```
Based on abstraction criteria that is refereed as *Original Abstraction Map*, CWolf extracts an abstract model with 15 states and 19 transitions. Inside this model, the statement at line 14 is executed by 12 possible execution paths. Starting from the *Original Abstraction Map*, our approach detects interesting value sets for $i$, $j$ and $k$, and assigns appropriate types of abstraction for them, which is shown in the *Updated Abstraction Map* in Figure 4.3. With the updated map, the truth values of $i == 0 \&\& j < i$ and $j == i + 1$ can be tracked correctly. The resulting abstract model for such a criteria posses 34 states and 38 transitions, as illustrated in the right column of Figure 4.4. However, no execution within the model pass through the statement at line 14.

### 4.3 Slicing over CWI Representations

Following the statement-deletion paradigm to produce slices, most static slicing tools face the problem that slices they produce are usually no longer compilable. This prohibits our CAdS to employ such tools as subroutines for slicing. Considering this, we
Algorithm 2 Derivation of Abstraction Criteria

Input: a slice \( S \), an Abstraction Map \( AM \), the slicing criterion \( SC \), and the well-ordered list \( SL \) of SCCs in \( S \)

Output: the updated Abstraction Map \( AM' \)

\( fun\_set = \emptyset \)

for all variables \( x \) in \( S \) do
  if \( x \) is specified as partition\((l)\) in \( AM \), \( IVL(x) = l \), otherwise \( IVL(x) = \emptyset \)
end for

while \( SL \) is not empty do
  \( fun\_set = \) head of \( SL \); \( SL = \) tail of \( SL \)
  while \( fun\_set \) is not empty do
    for all statement \( st \) in any function \( f \) in \( fun\_set \) do
      if \( st \) is if cond then ... else.. then
        for all predicate \( p \) in cond do
          if \( p \) is a variable \( x \) or \( \neg x \) then
            add 1 into \( IVL(x) \)
          else
            if \( p \) can be converted to \( x \) rel \( c_1 y \pm c_2 \) and \( R(y) < R(x) \) then
              if \( rel \ is > \) or \( <= \) then
                \( IVL(x) = IVL(x) \cup \{v + 1 \mid v \in IVL(c_1 y \pm c_2)\} \)
              else
                \( IVL(x) = IVL(x) \cup IVL(c_1 y \pm c_2) \)
              end if
            end if
          end if
        end for
      else
        if \( st \) is switch (cond) ...case \( v_1\):... case \( v_n\).. then
          collect all \( v_i \) into a list \( l \)
        end if
        if \( cond \) can be converted to \( x \pm c \) then
          \( IVL(x) = IVL(x) \cup \{v \mp c \mid v \in l\} \)
        else
          if \( cond \) can be converted to \( x \pm (c_1 y + c_2) \), where \( R(y) < R(x) \) then
            \( IVL(x) = IVL(x) \cup \{v_1 \mp v_2 \mid v_1 \in l, v_2 \in IVL(c_1 y + c_2)\} \)
          end if
        end if
      end if
    end for
  end while
  if no interesting value set is updated for any variable then
    \( fun\_set = \emptyset \)
  end if
end while

for all variables \( x \) in \( S \) do
  In the updated \( AM \), set the abstraction type for \( x \) as \( part(IVL(x)) \) if \( IVL(x) \), otherwise set as \( Top \).
end for
implemented a slicer working directly over the CWI file, the intermediate representation CWolf. 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 [88] to evaluate slices. A System Dependency Graph (SDG) is used to define the data-flow and control-flow dependencies between the basic blocks. Once done, the task of finding statements affecting the value of an interesting variable at a specified location is accomplished by identifying all statements whose corresponding vertexes in the SDG are reversely reachable by the vertex standing for the location.

The control dependencies between basic blocks are computed following the approach given in [89]. 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 especially important for applications where only part of code is available. If the definition of a function is missing, a summary still can be generated for this function if the documentation describes the interface for it. For functions that we can access their definitions, their summaries are calculated using a working list algorithm to calculate intra-procedural data-dependency for each function in turn. Whenever a statement calling some function is encountered during the calculation, the current version of summary for 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 out 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. In order to deal with pointer aliasing, a map recording the possibility of two pointer-type variables might point to each other is propagated when the
intra-procedural data-dependency calculation proceeds between blocks. 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 the resultant slice is obtained as follows:

1. For each block, the list of statements it possesses is updated by deleting unmarked statements from the list.

2. For each block, if its updated list of statements is empty, it is removed from the slice. The edge from its predecessors to it are redirected to all of its successors. If no such successor exists, a new block containing only the Halt statement is introduced into the slice, and all edges from its predecessors are redirected to this newly introduced block.

3. For each function, set the immediate successor of its start block as its entry block. If no such a block exists, remove its declaration from the declaration depositary that the CWI file keeps.

4.4 Abstracting Models

It can be a discouraging task for an analyst to comprehend the source code by browsing abstract models (LTSs actually) that CWMB generates, because such models might be quite large in size for complicated systems. In order to relieve the burden of
browsing LTSs for the analyst, we implement in CAdS a component, called Graph Builder, to extract models from LTSs. Such further-abstracted models are constructed from LTSs by gluing micro-structures in LTSs, i.e. states and instances of function calls, that contribute to the same conceptual functionality, into a macro-structure.

In the bottom-up style program comprehension process, an analyst forms his knowledge about how a conceptual functionality is realized in the implementation from his previously obtained knowledge on how sub-functionalities are realized. The extraction of further-abstracted models from LTSs, essentially expedites this comprehension process by aiding the analyst to move his focus onto how sub-functionalities are organized and how they interact with each other.

For ease of presentation, we refer to the act of collecting micro-structures in a LTS that contribute to the same conceptual functionality into a macro-structure as an aggregation operation. From an implementation point of view, an aggregation operation implies two activities: removing all related micro-structures from the model, and introducing a new vertex representing the macro-structure into the model. Moreover, the aggregation operation requires the newly introduced vertex to inherit all in-coming and out-going edges from the aggregated micro-structures to the rest of the model.

Aggregation operations are invoked in two situations:

1. All states or macro-structures within an instance call to a function are aggregated to a new macro-structure, referred as FunctionCall.

2. All states or macro-structures within a strongly connected component should be ag-
The purpose of performing aggregations in the first situation is self-explanatory. Moreover, our observation on how strongly connected components are brought into LTSs supports our decision to invoke aggregation in the second situation. Basically there are two important sources of strongly connected component in the LTS: (a) a loop structure in the code gets expressed as a strongly connected component, and (b) invocations to functions that work together for the same functionality introduce a strongly connected component. Therefore the aggregation of a strongly connected component in the LTS helps to delineate the part that realizes a conceptual functionality from the rest of the model.

The graph builder implements a depth first search algorithm traversing each branch in the LTS to identify all situations where an aggregation operation is necessary. Basically, the kind of state encountered indicates the necessity for aggregation. If the state encountered is generated because of the interpretation of a function call statement, the traversing can be predicated to enter into an instance of function call. Therefore a FunctionCall is pushed into the stack and the depth first search process proceeds to identify all states and macro-structures inside the instance of function call. Having identified all such states and macro-structures, strongly connected components are calculated and all components are aggregated into Component macro-structures. Once done, the FunctionCall is popped out from the stack, and all remaining states and macro-structures within the instance of the function
call is aggregated into FunctionCall. Therefore, the extraction a further-abstracted model from a LTS is essentially a process in which depth-first search and aggregation is interleaved.

To enable produce a visualization for the further-abstracted model, it 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 Table 4.3. The DTD defines the layout of the model and specifies the format for storing states or macro-structures and edges between them.

The CAdS user interface is used to display the XML file generated and facilitate a bottom-up understanding of the slice. The interface makes use of the Java JGraph API [90] 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. See Figure 4.5 for an illustration.

### 4.5 Case Studies

We carried out two case studies to evaluate the effectiveness of our CAdS tool. The first one tracks errors in XiO system and the second analyzes PCA software.
Table 4.4: Effort Required for Analyzing the XiO System

<table>
<thead>
<tr>
<th></th>
<th>Static Slicing</th>
<th>CAdS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Effort Required</td>
<td>221 person-hr</td>
<td>178 person-hr</td>
</tr>
<tr>
<td>Effort per Error</td>
<td>8.50 person-hr</td>
<td>6.84 person-hr</td>
</tr>
</tbody>
</table>

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 provides access to planning functions. The post-mortem analysis of the XiO system was carried out on the basis of bug reports received from the field. For the purpose of evaluating our approach, we limited ourselves to the software interface for the teletherapy module, and the errors detected therein. The module that was considered was 40K lines of C code.

This second case study was based on an implementation of a Generic Patient Controlled Analgesic (GPCA) infusion pump. The GPCA infusion pump is a standardized medical device system being developed at the Center for Devices and Radiological Health (CDRH), FDA. The specifications for the GPCA pump were collected by aggregating the behavior of various real-world PCA infusion pumps and were consolidated using the Jumb1 model builder developed at the University of Tennessee. The specification were provided to us in the form of tables and spreadsheets listing the observations for each situation. The code for the GPCA pump was in C language, and implemented the various situations and events defined. Supplemental code relating to specific pump functionality was added to facilitate source compilation and to ensure a more complete implementation. The total size of this software consists of approximately 20000 lines of code.

For both case studies, we carried on the analysis with the help of the CAdS tool, and compared the result with those obtained through slicing [83].

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 4.6 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
Figure 4.7: The Number of Iterations for Analyzing the PCA Prototype

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 was lesser at 8.92 per error for the CAdS tool. As shown in Figure 4.7, the number of iterations required to trace an error to source code were significantly higher for static slicing when compared with that required during abstraction-driven slicing using CAdS. The average number of iterations needed when using only static slicing was 4.39 per error, while this number dropped to 3.77 when using CAdS.

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 4.8 shows the comparison between both approaches in analysis of the XiO system with respect to the average size of the slices considered. The comparison in PCA analysis is shown in Figure 4.9. As can be seen from both figures, the average size of successive slices kept reducing in each of the approaches, as the slicing criteria were refined. As noted in the figure, the slices used in the abstraction-driven approaches were consistently smaller over successive slices.

The effort required for both approaches is compared in Table 4.5. Post-mortem analysis with the aid of CAdS was found to be more efficient than pure static slicing. When using CAdS, the effort required was 19.4% lesser than that for static slicing.
Figure 4.8: Comparison of Slice Sizes for the XiO Analysis

Figure 4.9: Comparison of Slice Sizes for the PCA Analysis
Chapter 5

Error Report Driven Software Failure Analysis

When applied to post-mortem analysis, slicing faces a common complaint about the need to specify an exact location in the program as the slicing criterion. In typical practice, however, the analyst (or reviewer) is unable to relate the error reported to a clear location in the implementation, because he often comes from a third-party and has no intimate knowledge of the system being reviewed. This situation gets exacerbated when the analyst has access only to parts of the code and has sparse documentation. On the other hand, error reports from the field, while not providing clear clues for slicing criterion, do possess information depicting the scenario in which the system fails. This information, if they can be made use of, can assist the process of tracing failures.

We already know that the abstract model extracted from the system under consideration, by virtue of exploring all possible execution paths of the system, is suitable for finding a system’s executions satisfying certain properties pre-defined. Therefore, if we looked at a particular error report as a special property that specifies a temporal order between events triggering the occurrence of the reported failure, it is not hard to see that a abstract model will also be suitable for isolating those system execution sequences that satisfy the error report, i.e., the execution sequence contains the usage scenario specified by the report. Providing that it is possible to recover the failure scenario, the task of identifying the accurate cause to the failure will reduce to a relatively simple task.

Given a property, an abstract model has the advantage over concrete models that it
allows relatively short execution sequences to be isolated as an evidence of the property. The other side of the coin, however, is that the existence of abstraction might lead to spurious execution sequences being isolated. Since our primary concern is to ease the reviewer’s comprehension, either on a piece of code or on a particular execution, it is preferable to trace a failure over an abstractly constructed model if this model does not cause unacceptably high false positive rate.

In this chapter, we will explain our framework that uses error reports to guide the failure tracing process in post-mortem analysis. In our framework, error reports under consideration are formalized as temporal properties, and tracing failures is accomplished by finding evidences for these properties in an abstract model. Furthermore, the abstraction used for constructing the model could be refined to yield an appropriate level of precision in an iterative way.

Our framework adopts program slicing as an additional option to facilitate the reviewer’s comprehension on isolated abstract execution sequences. However, slicing procedures invoked by our framework are directed by explicit criteria elicited from abstract execution sequences. In this way, the reviewer is released from the responsibility of hypothesizing slicing criteria, and the cost of unnecessary iterations of slicing procedures is saved.

The rest of this chapter is organized as follows: in Section 5.1, we first introduce a uniform representation, called Maluse Case, to encapsulate useful information in error reports, and then in Section 5.2 we discuss the essence of Maluse Cases by providing its explanation in temporal logic. Our framework that utilizes error reports in failure tracing processes is detailed in Section 5.3. Section 5.4 presents a divide-and-conquer algorithm to extend our framework to analyzing large-scale systems. Lastly, we evaluates the efficacy of our framework in Section 5.5.

### 5.1 Maluse Case

Although sparse in details, bug reports from users do contain useful information from which we can restore (partially) an error scenario. It is very common for end users to describe briefly in an error report the system context under which they perform a particular task leading to the error; a short action history during the task; and the symptom from which they identify the occurrence of the error. Table 5.1 sketches an example error report.
Table 5.1: An Example Error Report

**SSR Number**: f16517

**Detail**: If an MLC has been leaf edited, Port→Mirror Port shapes it to the mirror image of the last auto or drawn MLC shape rather than mirroring the leaf positions. It also leaves the MLC shape displayed.

In this report, the phrase *an MLC has been leaf edited* describes the system context, i.e. an MLC did exist and it had been previously edited before the error. Actions of pressing *Port* and *MirrorPort* buttons sequentially can be extracted as the history of what users did before the error happened. Finally *shapes it to the mirror image of the last auto or drawn MLC shape* specifies the error symptom.

A number of problems prevent error reports from being directly used by post-mortem analysis tools. Written in a natural language, error reports can not be parsed automatically and partitioned accurately into meaning groups. However this partition is crucial for us to isolate user actions and assemble them in the proper order. More important is that users can only describe an error using entities they can interact with, e.g. system devices or interface items. Entities of concern to analysts or reviewers, on the other side, are modules, functions or variables. Constructing a connection between these two categories cannot be done automatically, and often requires intensive communication with users.

Our approach doesn’t address the above problems. We simply assume that sufficient documents are available for forensic analysis. Thus after manual effort, it is always possible to relate entities used in error reports to these in source code. This assumption is reasonable in real applications. For example, when an error is reported to US FDA about a medical device, the vendor is required to provide necessary design and implementation document to FDA for post-mortem analysis. Our target is to design a formal format to characterize context, action history and error symptom information underlying error reports. Moreover the format should be generic enough to accommodate various styles users compose error reports in. This point is important in that users might miss part or all of context and action history. Therefore, it is impossible to build up a complete scenario from the report.

*Maluse Case* is a formal representation of error scenarios. Each *Maluse Case* consists of four components *Start Function S*, *Prerequisite P*, *Input Sequence I* and *Error Condition E*, where *S* indicates the entry point of the module to be considered...
and $P$, $I$ and $E$ directly reflect the context, action history and error symptom of an error scenario respectively. The grammar of Maluse Case is shown in Table 5.2.

**Start Function** $S$ is introduced to endow analysts the freedom of starting the analysis process anywhere inside source code, which allows them to focus on the portion of code most likely related to the error. Based on appropriate selection of **Start Function**s, our error analysis can be decomposed into a series of sub-processes each of which gets started with points from lower-layered modules in system architecture to higher-layered ones; details will be presented in Section 5.4.

The intention of including the **Prerequisite** component into a Maluse Case is to recover the environment setting described in the error report. More formally, **Prerequisite** defines a set of predicates $C$ over global variables or formal parameters $S$. Then the precondition $\bigwedge_{c \in C} c$ restricts the initial evaluation of variables at the beginning of the erroneous behavior of the system. Essentially this format of **Prerequisite** acts to specify the particular subset of system executions over which the subsequent analysis should carry on. One point that should be noticed is that users would not clearly specify all system configuration for the error. Therefore we cannot make assertions on the initial evaluation of some variables.

**Input Sequence** $I$ is represented in a Maluse Case as a sequence of input actions $a_1a_2...a_n$. Each action $a_i = \langle \text{loc}, c(x) \rangle$ simulates the users action of setting the variable $x$ at the program site $\text{loc}$ by using constraint $c(x_i)$ to restrict the value $x_i$ may take. Finding out the correspondence between input statements in source code and the users action should not be difficult when the system documentation available is sufficient. For a users’ action, the procedure in source code dealing with it can be isolated based on the

<table>
<thead>
<tr>
<th>Table 5.2: The Maluse Case Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MaluseCase</strong>: Start Function Prerequisite Input Sequence Error Condition</td>
</tr>
<tr>
<td><strong>Start_Function</strong>: Function</td>
</tr>
<tr>
<td><strong>Prerequisite</strong>: Constraint*</td>
</tr>
<tr>
<td><strong>Input_Sequence</strong>: [Location Constraint]*</td>
</tr>
<tr>
<td><strong>Error_Condition</strong>: CallCond</td>
</tr>
<tr>
<td><strong>CallCond</strong>: SimpleCall</td>
</tr>
<tr>
<td><strong>SimpleCall</strong>: ‘Call’ (Function)</td>
</tr>
<tr>
<td><strong>ConstrainedCall</strong>: ‘Call’ (Function (Pars)) [&amp;&amp; Constraint]*</td>
</tr>
<tr>
<td><strong>Constraint</strong>: Variable Relation Const</td>
</tr>
<tr>
<td><strong>Pars</strong>: Variable ‘,’ Variable*</td>
</tr>
</tbody>
</table>
Table 5.3: Maluse Case for the Example Error Report

<table>
<thead>
<tr>
<th>Start Function</th>
<th>ui_port_locfuncbar.c :: DoPortMnu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisite</td>
<td></td>
</tr>
<tr>
<td>g.port.ex_info → plan → cur_subp → subp → cur_beam → port_flag== FALSE</td>
<td></td>
</tr>
<tr>
<td>g.port.ex_info → plan → cur_subp → subp → cur_beam → mlc_flag== TRUE</td>
<td></td>
</tr>
<tr>
<td>Input_Sequence</td>
<td></td>
</tr>
<tr>
<td>ui_port_locfuncbar.c :: 572 (DoPortMnu :: itm == ePMNU_MIRR_M)</td>
<td></td>
</tr>
<tr>
<td>Error Condition</td>
<td>Call(port_mlc_util.c :: UpdateMlcAndColl)</td>
</tr>
</tbody>
</table>

system documentation or with the help of auxiliary data structures, e.g. concern graphs [91]. Syntax searching tools like grep can then be used to localize the exact input statement inside the procedure, given the fact that input statements possess fixed formats. The order of actions in $I$ is crucial in that it correlates with steps that users take leading to the error. Any system execution claimed as the error cause should preserve this order.

Rather than stick to the place where the error symptom is identified, Error Condition moves the attention of reviewers to conditions satisfied by the state at which the error occurs. We refer to such conditions as error triggers. If the module exhibiting an error symptom is complicated, multiple locations inside it might be responsible for the reported occurrence of the symptom. Previous techniques had to consider each of such locations individually. Error triggers, on the other hand, consider all possible locations responsible for the symptom at one time. In our approach, error triggers are categorized into three kinds: (a) a specific program statement gets executed, (b) a particular procedure is invoked unexpectedly or (c) the system execution under consideration hits a pitfall where the values of variable satisfy predefined constraints.

For the above example error report, simulating the effect of an MLC has been leaf edited is quite complicated. From the implementation documents, however, we found the port_flag subfield in the global variable g.port.ex_info is set to FALSE, the mlc_flag subfield is set to TRUE once a MLC is edited. So we can use such constraints to approximate the system configuration after an MLC has been leaf edited. The only action the user took, as described in the error report, is pressing Port → MirrorPort button, which corresponds to reading a const value ePMNU_MIRR_M from users and assigning it to the variable item in function DoPortMnu. Finally the error condition is specified as an unexpected call to the function UpdateMlcAndColl, which takes care of passing updated MLC data to the display function. The exact Maluse Case is shown in Table 5.3.
5.2 Logical Explanation of Maluse Cases

In this section, the essence of Maluse Cases is demonstrated by presenting its explanation in a temporal logic, particularly, in CTL. As discussed in last section, the order of input actions is crucial for error detection. This characteristic can be formalized using temporal logics, like CTL [92]. Encoded into a formula in CTL, a Maluse Case can be formalized as a time-dependent property \( p \), which allows such property to be reasoned over the model extracted from the system implementation by static analysis or model checking tools. First we give a brief and informal introduction to CTL. We refer to [92] for the formal syntax and semantics for CTL. Formulas in CTL are built from path quantifiers, modal operators, and standard logical operators. The path quantifiers are \( A \) (for all paths) and \( E \) (for some path). The modal operators are \( X \) (next step), \( F \) (eventually), \( G \) (always) and \( U \) (until). For a standard logic proposition \( f \), we write \( M, s \models f \) if a state \( s \) of the system model \( M \) (in out framework, LTS) satisfies \( f \), and write \( M, \pi \models f \) if we have \( M, s_1 \models f \) for the a path \( \pi \) in \( M \) with \( s_1 \) as its starting state. For the ease of expression, we use \( \pi^i \) to denote the suffix of \( \pi \) starting at the \( i \)th state of \( \pi \). Hence, whether a state \( s \) in the system model \( M \) satisfies a formula \( f \) in CTL can be defined inductively as shown in Table 5.4.

Finally we say a model \( M \) satisfies a formula \( f \), denoted as \( M \models f \) if and only if for the initial state \( s_{\text{init}} \) of \( M \) we have \( M, s_{\text{init}} \models f \)

In order to encode a Maluse Case, special attention should be paid to function calls and error triggers mentioned in it. We introduce a predicate \( \text{Exe}(\text{loc}) \) to deal with the trigger \( \text{a specific program statement at loc gets executed} \). A state \( s \) in the system model is said to satisfy \( \text{Exe}(\text{loc}) \) only when it is generated while the system execution reaches \( \text{loc} \). Also we introduce a predicate \( \text{Call}(p) \) to handle the trigger \( \text{a particular function p gets called} \). The meaning of a state \( s \) satisfies \( \text{Call}(p) \) is straightforward. With the help of
Call predicates, error triggers like a unary function \( f \) gets called, and the actual parameter passed to \( f \) is less than 0 can be encoded as \( \text{Call}(f) \land (\forall x \ f(x) \land (x < 0)) \). This encoding can be extended pairwise for functions with multiple parameters. Finding the formula corresponding to a Maluse Case is basically a two-step process: first each component of the Maluse Case gets encoded into a proposition, and then all propositions are folded to form the desired formula. It is easy to encode Start Function, Prerequisite, Input Sequence and Error Condition components in a Maluse Case. We denote by \( P_S := \text{Call}(s) \) if the start function is \( s \). For all predicates \( p_i \) in Prerequisite constraining the initial evaluation of global variables or formal parameters of \( s \), we define \( P_P := \bigwedge p_i \). Moreover, let \( t^i \) be the set of propositions encoding all error triggers listed in the Error Condition section, \( P_E := \bigvee t^i \) is the proposition we define for the Error Condition component. The underlying meaning of the \( \mathbf{U} \) operator in CTL makes it ideal to reflect the partial order between input actions described in the Input Sequence component. Suppose \( i^1, i^2, \ldots, i^n \) is the sequence of input actions in Input Sequence. For each input action \( i_i \) like read a variable \( x \) at a location \( \text{loc} \), and the value of \( x \) read in should satisfy a predicate \( p(x) \), we use \( I_i := \text{Exe}(\text{loc}) \land p(x) \) to encode it. Therefore, the proposition \( P_I \) to encode the Input Sequence component can be defined as \( P_I = IU_1 \), where \( IU_i := I_i \mathbf{U} IU_{i+1} \) for all \( 1 \leq i \leq n - 1 \) and \( IU_n := I_n \). Having obtained propositions for each component in a Maluse Case together, the CTL formula for the Maluse Case can be folded as

\[
P_S \land P_P \land \mathbf{E}(P_I \mathbf{F} P_E).
\]

Thus, isolating a system execution that matches the Maluse Case can then be transformed to finding a witness of the system model satisfying the CTL formula described above. And this task can be accomplished automatically by any model checking tool that accepts CTL formulae and is capable of extracting models from source code. One instance of such model checking tools is SMV [93]. The problem with explicit model checking tools is that these tools can only detect an error execution at one time. However it is possible that multiple error executions are related to the reported error. Abstraction, on the other hand, can aggregate multiple system executions into one abstract path if the level of approximation selected is appropriate. Therefore, the number of calls of detecting possible error paths before the real error cause is identified will be reduced when such detections are performed over an abstract model of the system. Another problem with most of such model checking tools is that they often make optimization over the mechanism to store states and transitions inside
the model. For example, SMV implements Binary Decision Diagram (BDD) as the format to encode and store states. Once an error execution is detected by such model checking tools, it is often difficult to restore a visualization model recognizable to the reviewer from the execution for these tools. Motivated by these observations, we propose in our approach to make use of CWolf as a framework to extract abstract models from source code, and to implement a detection scheme over CWolf to isolate possible error paths based on Maluse Cases. Since CWolf outputs LTS as the abstract model for the target system, error paths extracted from such model are easy for the reviewer to browse, simulate and understand. Also, the presence of abstraction deepens the reviewer’s comprehension of error causes. The performance comparison between detecting error executions either by conventional model checking tools or by our approach is left to future work.

5.3 Framework Description

As explained in the last section, a Maluse Case can be reduced to a temporal logic formula. Therefore, isolating system executions that match a Maluse Case can be accomplished by using any tool capable of checking CTL formulae against system descriptions. However, we choose to use abstract interpretation in our implementation. With an appropriate level of precision, an error trace isolated by abstraction would be as suitable as a concrete execution for simulation and for bug detection. Moreover, such abstract error traces are shorter and more recognizable, compared to those isolated from concrete models.

With a Maluse Case as input, our framework explores system executions based on abstract interpretation and reports those matching the behavior pattern prescribed by the Maluse Case as (potential) error traces. More strictly, a system execution $P$ is said as an error trace if:

1. there exists in $P$ a subset of states $s_1, s_2, ..., s_n$, such that $s_i$ can match the $i^{th}$ action of Input Sequence in the Maluse Case for all $1 \leq i \leq n$, and $n$ equals to the length of Input Sequence. Moreover,

2. Error Condition of the Maluse Case is satisfied by the last state of $P$.

---

In the rest of this thesis, we say a state $s$ satisfies a (proposition-typed) error condition or a (proposition-typed) error condition is satisfied by a state $s$ by meaning this error condition can be evaluated to true in $s$. 

---

1In the rest of this thesis, we say a state $s$ satisfies a (proposition-typed) error condition or a (proposition-typed) error condition is satisfied by a state $s$ by meaning this error condition can be evaluated to true in $s$. 

---


Once the number of error traces exceeds a pre-defined threshold or the exploration of system behaviors terminates, all error traces detected so far are output to analysts. As output error traces are often in great length, our framework provides three options for reviewers to ease their understanding of these traces:

- Every individual detected error traces are output as XML files, the format of which is explained in Table 4.3. Therefore, reviewers can walk through the output XML file and discover what sequence of statements that have been executed along the trace.

- Reviewers can focus on the sequence of function invocations along a detected trace by choosing to illustrate the trace on the graphic interface mentioned in section 4.4.

- Finally, based on Error Conditions specified in the Maluse Case, our framework is capable of automatically deriving a slicing criterion for each detected error trace. As such criteria are elicited directly from the Maluse Case, reviewers can use it with confidence to guide static slicing over detected traces, and hence can narrow down the range of code they need to consider without doubt.

Our experiences, as discussed later in section 5.5, indicates that all of the above options are useful in improving reviewer’s comprehension of error traces and facilitating reviewers to justify whether detected traces recover the purported error scenario or not. Any evidence convincing a reviewer to dispute a detected trace can contribute to refining the abstraction used previously. With this refined abstraction, the error tracing process could be invoked again, while with a higher level of precision.

Figure 5.1 illustrates the basic procedure of our framework. It is worthy to point out that the framework can also be invoked iteratively to narrow the cause of the error down to a particular segment of code. During successive iterations, new Maluse Cases can be made up by refining components in the old Maluse Case or by using the knowledge about error trace detected in last iteration. Essentially, the above iterative procedure is a top-down program comprehension process: the reviewer first obtains an overview of the error scenario, and then improves his knowledge about the code’s contributions to each component of the scenario.

Another option to trace down the exact error cause is to iteratively partition Maluse Cases. Here by partitioned Maluse Cases, we mean these Maluse Cases contain none or part of input action sequence defined by the Input Sequence in the previous Maluse
Figure 5.1: Error Tracing Process
Table 5.5: Abstraction Types Supported by CWolf

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>All information about the value of a variable is discarded.</td>
</tr>
<tr>
<td>minmax</td>
<td>The value of a variable is within an interval of integers.</td>
</tr>
<tr>
<td>mod($k$)</td>
<td>Values are abstracted to a set of remainders modulo $k$.</td>
</tr>
<tr>
<td>int</td>
<td>All information of the value of a variable is preserved.</td>
</tr>
<tr>
<td>free</td>
<td>The abstraction type for a variable is chosen dynamically.</td>
</tr>
<tr>
<td>part($a_1, a_1, ..., a_k$)</td>
<td>The value of a variable locates in set of partitions $[-\infty, a_1 - 1]$, $[a_1, a_2 - 1], ..., [a_k, \infty]$</td>
</tr>
<tr>
<td>$\tau$ array</td>
<td>The value of a variable is an array of values in $\tau$ abstraction type.</td>
</tr>
</tbody>
</table>

Case. At the mean time, Error Conditions for these Maluse Cases are required to be the particular constraint that makes the rest of the error trace feasible, instead of from error triggers. We will discuss details of partitioning Maluse Cases in section 5.4.

For the purpose of automatic manipulation, our approach actually parses a Maluse Case into a tuple $< p_{c_0}, p, Input, e >$, where $p_{c_0}$ indicates the entry of Start Function, $p$ and $e$ are propositions generated from Prerequisite and Error Condition recognizable by the abstract model builder tool (CWolf actually, in our approach). Moreover Input is an array, where its $i^{th}$ cell is generated from the $i^{th}$ action ($loc, c(x)$) of Input Sequence. In details, Input[$i$] is a tuple ($pc_i, prop_i(x)$), where $pc_i$ corresponds to the $loc$ specified by the $i^{th}$ action and $prop_i(x)$ is a proposition corresponding to $c(x)$ while recognizable by CWolf.

5.3.1 Abstract Model Construction

In our framework, we employ CWolf as the engine to extract a cognition model for the target system. In the abstract model (actually LTS) that CWolf extracts from source code, each state $s$ is represented as a tuple ($pc, env$), where $pc$ indicates current program point when the state is generated, and $env$ is current configuration mapping variables to abstract values.

The obligation of abstracting integer variables in the source program is left to users by requiring them to provide an abstraction map file binding particular abstractions with interesting variables. Types of abstraction that CWolf supports is shown in table 5.5. However, a too coarse abstraction would make the evaluation the truth value for predicates would no longer result in true or false. For example, for a variable $x$ with top abstraction type, we don’t preserve any information about $x$’s value, hence the truth value for predicate
Table 5.6: Base Abstraction Types for Variables in Predicates

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Abstraction type assigned to $x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x == c$ or $x != c$</td>
<td>part($c, c+1$)</td>
</tr>
<tr>
<td>$x &lt; c$ or $x &gt;= c$</td>
<td>part($c$)</td>
</tr>
<tr>
<td>$x &gt; c$ or $x &lt;= c$</td>
<td>part($c+1$)</td>
</tr>
</tbody>
</table>

$x < 0$ is Unknown.

The above situation is not acceptable when we evaluate predicates contained in the Error Condition section, as a high confidence in the identification of correct final states would endorse a relatively low false positive rate for our framework. For this purpose, we present an automatic solution to assist analysts obtain base abstraction types for these variables in error predicates. Any abstraction coarser than base ones, if assigned to these variables, would affect the precision of error detection.

Intuitively, our solution utilizes information in the Error Condition section to suggest useful basic abstraction types. In our solution, predicates in the Error Condition section are first scanned to collect variables inside and integer-typed constants that confine them. Base abstraction types for variables can then be decided based on constants collected, as shown in table 5.6. Certainly, analysts are allowed to refine base abstraction types for these variables.

For error analysis, the Maluse Case of the format $< pc_0, p, Input, e >$ is passed forward to CWolf. However, the information contained in conventional states that CWolf generates is not sufficient for error tracing. In order to isolate the trace $P$ from the model simulating the error scenario, information should also be recorded in states about attempts we make trying to match Input Sequence so far. In our approach, each state $\tilde{s}$ in LTS model is a triple $(pc, env, index)$, where $pc$ and $env$ share the same meaning as previously mentioned, and the extra component $index$ means that the next action to be matched from Input Sequence is stored in $Input[index]$. An $index$ also implies that there exists $\tilde{s}_{i_1}, \tilde{s}_{i_2}, ..., \tilde{s}_{i_{index-1}}$ along the trace from the start state to $\tilde{s}$, such that $\tilde{s}_{i_j}$ is matched with the $j^{th}$ action in Input Sequence.

To accommodate the above extension, we define the mechanism of generating successors for one state as follows:

- If a state $s_1 = (pc', env')$ is a successor of $s = (pc, env)$, then the extended state $\tilde{s} = (pc, env, i)$ also has one successor $\tilde{s}_1 = (pc', env', i)$ if the statement at $pc$ is not
matched by Input[i].

- Otherwise, if the statement at pc is \( \text{read}(x) \) and \( \text{Input}[i] = (pc, \text{prop}(x)) \), \( \tilde{s} \) has two successors: then unmatched one \( \tilde{s}_1 = (pc', env_1, i) \) and the matched one \( \tilde{s}_2 = (pc', env_2, i + 1) \), where \( env_1 = env_{x=\top} \) and \( env_2 = env_{x=v}; \) s.t. \( \text{prop}(v) == \text{TRUE} \) \( \land \) \( (\forall v' \; \text{prop}(v') == \text{TRUE} \Rightarrow v' \preceq v) \).

Intuitively, if current statement is an input matched by the \( i^{th} \) action in \text{Input Sequence}, we consider both cases of the \( i^{th} \) action being matched or not being matched. For the former case, \( x \) is evaluated to an abstract value that has the least degree of precision while at the same time satisfies corresponding constraint, and the \( i + 1^{th} \) action becomes the first input action expected to be matched for the rest of execution. On the contrary in the later case a top value is assigned to \( x \) meaning no assumption can be made over \( x \)’s value after the input. Moreover, the \( i^{th} \) action is still waiting to be matched later. In other situations, the generation of successors just proceeds as previously.

Based on the modified definition of successor generation mechanism, we present a Depth-First-Search procedure to construct abstract model for the system under consideration, as shown in Table 5.7. In this procedure, the model construction process starts from the initial state \( s_0 \) in which the precondition \( p \) in \text{Maluse Case} is satisfiable. The EnvRefine reproduces the initial environment for \( s_0 \) by confining abstract values for each variable occurred in \( p \). Since the set of possible values of these variables can take is finite, the call of EnvRefine can always stop in finite time by trying every possible values of each variable. During the model construction process, reachable abstract states are explored. Here, a state \( s = (pc, env, i) \) is reachable if the path from \( s_0 \) to it respects the segment of actions \( \text{Input}[1], \text{Input}[2], ..., \text{Input}[i] \). The whole procedure stops whenever a state \( s_e \) satisfying the error condition \( e \) is found and the path from the initial state to it matches the input action sequence. In this case, a backtracking procedure BackTrack is called to extract the path \( P \) from \( s_0 \) to the error state \( s_e \), and \( P \) is output to slicer for further analysis.

Our framework allows analysts to define error condition \( e \) as a complex logical combination of predicates \( p_i \), where \( p_i \) checks the value of some variable against an integer const. In this case, having to recalculate the truth value of \( e \) after interpreting each statement will greatly affect the efficiency and applicability of our framework. To address this issue, our framework assign a temporary variable for each predicate in \( e \), and the env com-
Table 5.7: The Model Generation Process

Input: prog, < pc0, p, Input, e >
Output: Error trace P if exist

ModelGenerator(){
  cur_env = EnvRefine(TopEnv, p);
  s0 = (pc0, cur_env, 0);
  Q = {s0};  \ \ \ \ \ \ \ \ \ \ Q is the queue of unvisited states
  H = {};  \ \ \ \ \ \ \ \ \ \ H is a heap structure storing visited states
  While Q is not empty {
    Delete the first state s = (pc, env, i) from Q;
    Fetch the statement st at pc;
    If st is specified by e and all input actions are matched so far
      return(BackTrack(s, H));
    else
      [s1, s2, ...] = SuccessorGenerator(s, st);
      If there exist some si in [s1, s2, ...] s.t. e is evaluated to TRUE in Env(si)
        return(BackTrack(s, H));
      else
        Add [s1, s2, ...] into Q;
        Add s into H;
  }
}

ponent of every regular state will store values of these temporary variables. The truth value of e is not recalculated unless the statement encountered is an assignment one and its interpretation changes the truth value of some pi. Then the recalculation of e is done by using new value of pi and fetching from current env values of temporary variables corresponding to other predicates. In some sense, this solution combines data- and predicate-abstraction together in that predicate abstraction also contributes to the composition of states.

Given the existence of abstraction, it is possible that multiple error paths can be isolated from the partial model constructed. To handle this situation, the above procedure can be extended to continue exploring states until the number of error paths found exceeds a threshold. This extension allows aggregate code along all error paths, and output what would be obtained from a slicer. Therefore, the cost of data- and control-flow analysis needed by slicer can be saved for the common part of error paths. However, the analyst would take the risk that setting the threshold too high would force the construction of the entire model.
5.3.2 Slicing Criterion Derivation

The likelihood that an error trace detected in abstraction stage is a real one depends on two factors:

1. the level of precision we choose to approximate the system behaviors with and,

2. the correctness of error condition analysts define to characterize error symptom.

Theoretically a semi-automatic algorithm can always be presented to refine the abstraction we use. However, justification of error condition requires human effort and wisdom. For this reason, our framework leaves to analysts the task of identifying whether an error trace is a real error cause. To aid the analyst in understanding the error path, which can be quite long in length for large-scale systems, slicing can provide an option for reducing the analysis effort by eliminating irrelevant details to the occurrence of the error symptom. Moreover, when an error trace is spurious, slicing helps the analyst sketch the skeleton of the error trace detected, hence making it easier for him to find out points where the error trace deviates from real system behaviors. These points contribute an important clue to refine abstraction for next iteration of fault analysis.

Having obtained an error trace $P = s_0, s_1, ..., s_e$, slicing is accomplished in two steps:

1. Firstly all statements not encountered along $P$ are eliminated;

2. Secondly backward static slicing is performed over the portion of code remained starting from the location where $s_e$ is detected.

The first step saves our analysis from conducting costly semantic analysis over code irrelevant to $P$. For the second step, our framework is open to any slicer tool, e.g. CodeSurfer [94], that is capable to perform backward slicing over C programs. In backward static slicing, a criterion is provided as $(loc, S)$ and the resultant slice is the part of code affecting values of variables in $S$ at the program site $loc$. For our failure analysis backward slicing can serve as an effective tool to obtain the part of code that affects the occurrence of the error symptom discovered by our abstract model constructor. The error state $s_e$ detected, therefore, is a natural source of slicing criterion. We design an algorithm, as shown in Table 5.8, to derive slicing criterion based on the error state $s_e$ and the Error Condition $e$ defined in the Maluse Case. In our algorithm, the derivation of criterion is straightforward when
Table 5.8: Derivation of Slicing Criterion

<table>
<thead>
<tr>
<th>Input: error state ( s_e = (pc, env, t) ), error condition ( e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: slicing criterion, ((pc', [x_1, x_2, ...]))</td>
</tr>
</tbody>
</table>

\[
\text{SlicingCriterionDerivation}()\
\begin{align*}
&\text{fetch the statement } st \text{ at } pc; \\
&\text{switch}(e)\
&\quad\text{case } \text{EXEC}(pc): \text{return } (pc, []); \\
&\quad\text{case } \text{CALL}(f()): \text{return } (pc, []); \\
&\quad\text{case } \text{CALL}(f(pars)) \land \text{prep}(pars): \\
&\quad\quad\text{if } st \text{ is } f'(args) \\
&\quad\quad\quad\text{find the bounding } m : pars \rightarrow args; \\
&\quad\quad\quad\text{collect the set } S \text{ of names occurring in prep(pars)}; \\
&\quad\quad\quad S' = \{v \mid \exists n \in S (m(n) = v)\}; \\
&\quad\quad\quad\text{return}(pc, S'); \\
&\quad\quad\text{else exit}(0); \\
&\quad\text{case } p(V): \\
&\quad\quad\text{collect the set } S \text{ of variables occurring in } p(V); \\
&\quad\quad\text{return}(pc, S); \\
&\end{align*}
\]

\( e \) specifies an statement gets executed unexpectedly, a proposition is satisfied or an unconditional call to a function, because for such cases, we cannot assume the importance of any variable. If \( e \), in the format of logical combination of predicates, makes assertions about values of a particular set of variables, all variables in \( e \) should be put into the slicing criterion, because tracing the values of such variables is an important approach for the analyst to understand why the reported failure occurred. Finally when \( e \) declares a conditional call to a function \( f \), we choose to include into the slicing criterion only variables that occur in real arguments whose corresponding formal parameters are constrained by \( e \). We make this decision because the analyst’s judgement about the importance of all \( f \)'s formal parameters has to be respected.

5.4 Decomposition of Maluse Cases

Although abstraction allows to construct finite models for software systems, the time cost of building such models is usually unacceptable for extremely large systems. Moreover when an error state is hit, the cost to backtrack the corresponding error trace
is proportional to the model size, which is quite expensive. To cope with this complexity issue, we propose a methodology to partition the input sequence in a Maluse Case, and then proceed our fault analysis in a modular-based way. For a software system with the set $F$ of functions, its function call graph (FCG) is constructed as: each function $F$ in the system is assigned with a vertex $v_f$, and a directed edge is inserted from vertex $v_f$ to $v_g$ only if function $f$ calls $g$. Therefore, by calculating Strongly Connected Components (SCC) in FCG, a label function $L : F \rightarrow \text{Nat}$ satisfies:

1. if $f$ and $g$ are in the same SCC, $L(f) = L(g)$.
2. $\forall f, g \in F \ L(f) < L(g) \iff \text{No path can be found in FCG from } g \text{ to } f$.

Have defined the label function $L$, the label function $L'$ for all input actions $act_i$ specified in Input Sequence of the Maluse Case can subsequently defined as $L'(act_i) = L(f)$ where the input action locates inside $f$. One important use of $L'$ is to check whether the order of all $act_i$ is correct. One Input Sequence $I = act_1, act_2, ..., act_n$ is correct if it satisfies:

1. $\forall 1 \leq i, j \leq n \ i < j \implies L'(act_i) \leq L'(act_j)$
2. $L'(act_i) \leq L(f_0)$, where $f_0$ is the start function.

The meaning of this rule is obvious: if an action $act_i$ is defined before another $act_j$, there must exist at least one execution path from the function that $act_i$ locates to the one that $act_j$ does.

Given the start function $f_0$ and a correct input sequence $I = act_1, act_2, ..., act_n$, we define function $f$ is a dominating function for one action $act_i$ in $I$, or in another word, $act_i$ is dominated by $f$, if $L'(act_{i-1}) \leq L(f)$ and $f$ lies on all paths in FCG from $f_0$ to the function $g$ where $act_i$ locates in. It is obvious that $act_i$ is dominated by the function it lies in, thus it is always possible to find a dominated function $f$ for any input action over $I$. Consequently, We can always partition $I$ into two segments $I_1 = act_1, ..., act_i$ and $I_2 = act_{i+1}, ..., act_n$ for any $1 \leq i \leq n$ and find the dominating function $f$ for $act_i$. Thus one Maluse Case $MC = (f_0, p, I, e)$ can be decomposed correspondingly into $MC_1 = (f_0, p, I_1, e')$ and $MC_2 = (f, TRUE, I_2, e)$ for some $e'$. We can conclude that such an $e'$ always exists if we can detect an error trace for $MC$ in the abstract model.
Proof Suppose the error trace $P = s_0s_1...s_n$ is detected. Since $f$ dominates $act_i$, there must exist some $s_j$ on $P$ such that the statement $st$ associated with $s_j$ is $f(arg_1, arg_2, ..., arg_k)$, where $arg_1, arg_2, ..., arg_k$ are arguments passed to $f$. With the $env$ component of $s_j$, we denote $v_i$ for any $1 \leq i \leq k$ as the value $arg_i$ evaluated in $env$, then $e'$ can be defined as $CALL(f(a_1, a_2, ..., a_k))$ and $\bigwedge_{1 \leq i \leq k} a_i = v_i$. Clearly $s_0, s_1, ..., s_j$ is the error trace for $MC_1$ and $s_{j+1}, ..., s_n$ is the one for $MC_2$. □

The reverse direction is also correct: if we can detect $s_0, s_1, ..., s_j$ as an error trace for $MC_1 = (f_0, p, I_1, CALL(f(a_1, a_2, ..., a_k))$ and $Pred)$, and also we can detect $s_{j+1}, ..., s_n$ for $MC_2 = (f, TRUE, I_2, e)$, then the trace $P = s_0s_1...s_n$ is an error trace for $MC = (f_0, p, I, e)$. The proof for it can be converted to proof there must exist an edge in the abstract model from $s_j$ to $s_{j+1}$, because if so, the trace $P = s_0s_1...s_n$ is obviously an error trace for $MC$ considering our definition for error traces. Now we proof there must exist an edge from $s_j$ to $s_{j+1}$:

Proof Suppose $s'$ is a successor of $s_j$ in the abstract model, its corresponding statement must be the entrance of $f$. Then any of its concrete counterpart $s$ i.e., a concrete state $s$ that share the same $pc$ with $s'$ and for any variable $x$ in the program we have $\alpha(Env_s(x)) \succeq \overline{Env}_{s'}(x)$, must be a counterpart of $s_{j+1}$, because for any variable $x$, we have $\overline{Env}_{s_{j+1}}(x) = \top$, and hence $\alpha(Env_s(x)) \succeq \top$. Finally, we know that CWolf uses over-approximately extracts abstract models from C programs[95], which means there must exist a transition from $s_j$ to $s_{j+1}$. □

Since the above conclusion stands for any $Pred$, we can divide our fault analysis as following, and this division can be performed recursively.

Step1 : break $MC$ into $MC_1$ and $MC_2$ leaving $e'$ undefined, and then detect the error trace $P'$ for $MC_2$;

Step2 : collect path constraints $C$ along $P'$ relating to formal parameters of $f$. By path constraints, we mean the decision we make to choose a particular branch for a conditional statement. For examples, provided a statement $if(x < 2)\{\} else\{\}$, if we choose the true branch to generate the successor for current state, the path constraint is $x < 2$. $e'$ is defined as $CALL(f(a_1, a_2, ..., a_k))$ and $\bigwedge_{a_i \in s} s_i$;

$^2Env_s(x)$ is the value $v$ of $x$ in $s$, and $\alpha(Env_s(x))$ is the abstract value corresponding to $v$. $\overline{Env}_{s'}(x)$ is what $x$ will be evaluated to in the abstract environment that $s'$ posses. By $v_1 \succeq v_2$, we mean the set of concrete values that $v_2$ can be interpreted to is a subset of that for $v_2$. 
Step3: detect the error trace $P''$ for $MC_1$. The concatenation of $P''$ and $P$ is the error trace expected.

5.5 Case Study

To evaluate the effectiveness of our framework, we carried on our case study over the XiO system, as mention in section 3.5. To compare our framework with the abstraction-driven slicing approach, we also limit ourselves to the software interface for the teletherapy module, and the errors detected therein.

5.5.1 Program Instrumentation

Since our framework relies on the abstract model construction process to isolate possible error traces, we have to make the code under consideration executable. Before we carried on our case study, the code was instrumented as follows:

- There are a bunch of functions in the code being used in the code while their definitions are unaccessible to us. We enforced the return values from such functions to be $\top$, meaning we considered all possible consequence of invoking them.

- Our framework requires that we should be able to access the mechanism that connects procedures taking users’ input to corresponding handlers. However, such mechanism was missed from the part of code we obtained for the XiO system. We instruments the code with functions that randomly take input from users and call corresponding handlers based on the documentation we had and the information contained in the code.

- 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 $\text{DoPortMnu()}$. Also the function $\text{InitTelePulldowns()}$ initializes all tabs and menu items. Therefore we instrumented the module we consider with a function $\text{main()}$ that firstly calls $\text{InitTelePulldowns()}$ to initialize the interface and then continuously calls $\text{DoPortMnu()}$. 

5.5.2 A Running Example

In this section we demonstrate how our framework can be used to assist failure tracing process. Taking the following error report on the XiO system as an example:

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.

The primary difficulty behind formalizing the above-mentioned reports lies in that it is hard to find directly logical representation of the should prompt clause. From function interfaces, however, we discovered that function ActivatePortToMLC() is the one that responses to requests on converting aperture of current active beam. Therefore, we used the initial Maluse Case, illustrated in Table 5.9, to recover the error scenario, and to study what elicited the system’s behavior deviates from expectation.

Taking such an initial Maluse Case as input, our framework identified and re-reported an error trace that could match the input case, at the cost of generating 955 states and 1138 transitions during the construction of the abstract model.

Figure 5.2 points out the most suspicious part of the detected trace,. From this hierarchy, we could discover that function InitTelePulldowns() was responsible for enabling the Port-¿Convert aperture under the unexpected situation. Tracing back to design documents, we could also find that function EnableMnuItem() configures menu items according to users’ input. As a result, we composed a new Maluse Case, as shown in Table 5.10, to detect why the attribute Port-¿Convert aperture is enabled by InitTelePulldowns():

Similar to the last step, an error trace was detected this time based on the updated Maluse Case. However, the resultant trace was too long to be easily understood. To simply
Table 5.10: Second Maluse Case for the Example

<table>
<thead>
<tr>
<th>Start Function</th>
<th>InitTelePulldowns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prerequisite</td>
<td></td>
</tr>
<tr>
<td>Input Sequence</td>
<td></td>
</tr>
<tr>
<td>Error Condition</td>
<td></td>
</tr>
<tr>
<td>Call(EnableMnuItem(a, b, c) &amp; b == ePMNU\text{CONVERT} &amp; c == TRUE)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.2: Hierarchy of the Detected Error Trace

The trace, the following slicing criterion was generated as a by-product of our error detection process, indicating what we should focus on to comprehend the detected trace:

\text{line 23}@\text{ui\_port\_locfuncbar.c, plan\_has\_photon\_nonseg\_port\_or\_mlc}

Figure 5.3 outlines the slice we obtained after slicing source code under the guidance of the above criterion. We used the CWI2C component of CWolf to translate the resultant slice represented in the CWI format back to a C program. This translation preserves the semantics of the original program, while at the same time makes the output C program not compact enough. Taking this slice as evidence, it is trivial to conclude that
the actual cause of the purported failure can be attributed to the confusion that the system has on recording current active beams. In particular, Convert aperture will be enabled 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.

5.5.3 Experimental Results

We performed error tracing analysis over the same set of error reported as mentioned in Section 3.5. Table 5.5.3

<table>
<thead>
<tr>
<th>Error Report</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total States</td>
<td>1.4K</td>
<td>1.6K</td>
<td>1.8K</td>
<td>52K</td>
<td>73K</td>
<td>64K</td>
<td>3.5K</td>
<td>2K</td>
<td>4K</td>
<td>5K</td>
</tr>
<tr>
<td>Total Transitions</td>
<td>1.8K</td>
<td>2.1K</td>
<td>2.7K</td>
<td>54K</td>
<td>81K</td>
<td>72K</td>
<td>4.8K</td>
<td>3.6K</td>
<td>5.1K</td>
<td>7K</td>
</tr>
<tr>
<td>Iteration Numbers</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

In our framework, the number of iterations for an error-tracing task is defined as how many error detection procedures are invoked before the real error cause is isolated. On the contrary, in abstraction-driven slicing, the number of iterations is defined as how many successive slicing procedures are called. Figure 5.4 illustrates the comparison between...
the number of iterations both approaches used for each error report. Our approach costs less iterations for every error. However, during each iteration, our approach requires a little more time to construct the abstract model, compared to the abstraction-driven approach. In abstraction-driven approach, the abstract model construction process is invoked to improve the analyst’s comprehension over a particular function or a module. In some cases relatively coarse abstraction is acceptable if the analyst focuses on a high-level overview about the interaction between functions. In our approach, abstraction is invoked to detect error traces. The size of the abstract model constructed in our approach, and hence the time cost to generate this model, depends on the distance between the start function specified in the Maluse Case and the error location detected.

The cost of effort in our framework comes from three sources:

- The effort required to form the initial Maluse Case.
- The effort required to instrument the program and to assign abstraction types to functions and variables in the code.
- The effort required for generating successive error detection and slicing tasks.

For the first two situations, the effort cost for each error reported will decrease if multiple errors are reported about the same module. The total effort in our case study turns out to
be 53 person hours (approximately 7 working days), averaging 5.3 person hours for an error report. Compared to 6.84 person hours for each error in the abstraction-driven approach [82], the effort required by our framework is approximately 23% less.
Chapter 6

Conclusions and Future Work

In this thesis, we have given solutions to facilitate the reviewer’s comprehension of a software artifact both in pre- and post-market review practices. Although we targeted our work at regulating medical device software, approaches presented in this thesis can be easily applied in other safety-critical industries to improve the comprehension of software for all key stake-holders, such as system developers, independent assessors or certification authorities.

To conclude this thesis, we will summarize our work in Section 6.1, and discuss three possible directions for future research in Sections 6.2, 6.3, and 6.4.

6.1 Summarization

6.1.1 A Conceptual Requirement Model

We presented in Chapter 3 a conceptual requirement model - Goal Graph - as a step towards expediting error-free pre-market regulation practices. The structured hierarchy inherent the model reduces the manufacturers’ workload of composing convincible assurance cases. We also proposed a straightforward method to create assurance cases according to a particular Goal Graph. Assurance cases thus created permit reviewers a fast comprehension on applicant products, by presenting to them explicit links between requirements that they insist, designs that manufactures adopt to fulfil these requirements, and actual implementations.

Once a Goal Graph has been proved to be sound and sufficient by experts in
the field, we advocate it being used by manufacturers as the starting point for composing assurance cases for their products. In this way, pre-market review submissions from various manufacturers can be organized in a uniform format and a quick agreement can be reached between regulators and manufacturers on issues they are commonly concerned with, both of which will reduce the workload of regulators and accelerate regulation processes.

6.1.2 Improving Software Comprehension in Post-market Analysis

We have presented in Chapter 4 and 5 two approaches to improve the efficiency of post-mortem review process. Both of these approaches aim at improving the review’s comprehension on the accurate cause of a purported error by narrowing his attention down to the piece of code relevant to a purported software failure. The difference between these two approaches lies in: the first approach only provide an integrated recognition model for the reviewer to comprehend a program and requires the reviewer himself to recover the failure cause, while the second approach deviate the reviewer directly onto the program’s execution that recovers the failure scenario.

The first approach integrates abstraction into the abstraction-driven slicing framework serving to provide the reviewer a bottom-up recognition model of the program. This recognition model helps the reviewer comprehend the program and derive better slicing criteria. Specially, for this approach we provided our solutions to integrate abstraction into the abstraction-driven slicing framework seamlessly by answering questions: How to automatically generate a sufficient while not too precise abstraction criteria from the source code? How to realize successive slicing? How to generate a representation from the abstract model to make it easier for the reviewer to browse and understand?

For the second approach we introduced Maluse Case as a uniform representation to formalize information in error reports that can be used to expedite the error detection process. Moreover, we presented a framework to let Maluse Case guide our error detection process, in which abstraction is employed as a simulation and detection engine to isolate possible causes from the abstract model. For the applications of finding executions matching the Maluse Case in a large-scale, complicated system, we provided a divide-and-conquer algorithm to decompose the task into a set of similar subtasks based on the modular hierarchy of the system. Having accomplished each of these subtasks, error executions obtained from these subtasks can be concatenated together to form possible error executions for the
6.2 Enhancing The Goal Graph Model

Our case study on applying Goal Graphs onto generic large-volume infusion pumps demonstrates that the Goal Graph model is expressive and flexible enough to be applied in realistic regulation exercises. However, a set of issues, while not addressed in our current model, are brought to our attention. These issues, as shown in the following, represent significant needs in realistic regulating activities, and should be considered by our model in the future:

- **Assumption of Environments**
  Investigation should be conducted on enriching the format of Goal Graphs, such that they are capable of clarifying slight differences in clinical environments that would affect and differentiate the functioning of medical devices even in the same category.

- **Acceptability**
  Not all requirements demand absolute satisfaction, especially those relating to tolerable risks. Therefore, research is necessary on incorporating acceptability metrics into Goal Graphs, with which manufacturers are able to adopt wise implementation plans by considering economic costs and whether requirements are met.

- **Automation**
  New methodology is necessary on the derivation of requirements, such that requirements obtained becomes more checkable and suitable for automatic analysis.

The general acceptance of the Goal Graph model by both regulators and developers would depend on regulation policy issues, as well as comprehensive case studies that show the advantage of communicating a formalized requirement model among different parties. Atentions should be equally paid during case studies on both testifying the viability of the model in a wide range of applications and identifying critical information that the model should preserve when facing various types of devices, component, and requirements.

To realize the communication of Goal Graphs among regulators, a database supporting automatic input and intelligent search of Goal Graphs is necessary. In order to accommodate Goal Graphs with different formats of underlying tables, a front-end should
also be implemented for the database that could accept regulators’ customizations of Goal Graphs for different regulation purposes and populate arbitrary tables accordingly.

6.3 Improving The Error Report Driven Failure Analysis

Based on our experience gained in tracing software failures in complicated medical devices, we believe that making appropriate use of the information inherent in error reports can greatly reduce the effort required in the post-mortem analysis. System executions that faithfully recover the failure scenario can shift the reviewer’s attention onto events occurred before, during and after the failure, and hence ease the review on identifying the accurate cause to the failure.

Our work of formalizing error reports and utilizing them in post-mortem analysis is only a first step in this direction. There still still exists certain types of information, such as the system context information, contained in error reports that our approach cannot deal with. In our framework, we restrict the only allowed format of predicates that can be declared in a Maluse Case as an assertion on the relationship between a variable and an integer const. This restriction is made because CWolf, upon which we implement our framework, can only deal with non-relational data abstraction for integer-typed variables only. Although this restriction has no effect on formalizing user actions, this format of predicates is not sufficient to characterize context information contained in an error report, which means the loss of information and the increase of complexity we can detect error execution paths.

Therefore, it is worth to conduct research on enriching the format of Maluse Cases. One possible extension to the format of Maluse Cases is to allow the format of predicates used in Maluse Cases be able to assert on the relationship between two variables. However, such an extension requires the relationship between variables being propagated and updated during the model construction process, which will increase the size of state space that needs to be explored. Investigation is necessary on balancing the tradeoff between usability and efficiency in going from non-relational to relational abstractions.

For the prototype tool we developed to realize the framework, more comprehensive case studies are necessary to evaluate its efficacy. In our case study, we limited ourselves to the user interface of the teletherapy module in the XiO system. Therefore, error paths detected for 10 Maluse Cases did not cross module boundaries.
Moreover, our prototype tool has not been proven for its efficiency, because there is no existing verification tools that can take Maluse Cases as input. However, as explained in section 5.2, a Maluse Case can be encoded into a CTL formula. Therefore, detecting error execution paths matching this Maluse Case can be accomplished by any model checking tool that is capable of checking CTL formulas over the system under analysis. MAGIC [96] is an interesting instance of such tools, because it extracts abstract models from C programs based on predicate-abstraction. Comparison between using our framework with using MAGIC over the same case study can suggest which type of abstraction is suitable for our problem domain. Such comparison might also disclose the possibility to develop a more efficient tool by integrating different types of abstractions.

One observation on the case study indicates that extra iterations of the error detection process will not be invoked unless all error execution paths detected in the previous iteration have been proven as spurious. Spurious error execution paths are introduced in the abstract model either because the abstraction employed in the previous iteration is too coarse or because such error paths are obtained by concatenation of sub-paths. For the former case, we rely on the reviewer’s wisdom to refine abstraction. For the later case, on the other side, the number of spurious paths can be reduced if we can increase the precision of partition of the Maluse Case.

A brief idea to reduce the number of iterations is to increase the precision of information we can obtain from error executions detected in lower-layer components, which can lead to a more strict Error Condition in MaluseCases against with higher-layered components will be checked. A possible solution to realize the idea is applying reverse symbolic execution on error executions in order to obtain a constraint over the values of actual parameters and global variables passed to the entry of the component as strict as possible, such that the constraint can help compose a stricter Error Condition in the sub-Maluse-Case for higher-layered components, resulting in less number of system executions can satisfy the Error Condition.

Besides aiding the post-mortem analysis, our framework can also be extended to detect a family of errors that are similar\(^1\) to the one reported. With such an extension, our framework can provide useful information for the generation of test cases that can filter out the family of errors in the implementation. Fortunately, such an extension is possible, given

\(^1\)Here, by similar errors, we mean errors that might have the same symptom as the reported error, but obtained under different sequences of user actions.
the fact that an error execution $p$ matching the error report, if real, must possess a state $s$ that contributes to the cause of the error, and all executions in the abstract model that share with $p$ the same suffix staring from $s$ must commit similar errors.

### 6.4 Connecting Pre- and Post-Market Review with Goal Graphs

Although the Goal Graph model can be customized in arbitrary ways for intended quality regulating applications, it is always more practical to associate each requirement node inside the resultant model with its typical failure symptoms. A direct good of doing this is that the associated set of failure symptoms could facilitate the reader of a Goal graph to understand the requirement from risk-based perspective, and at the mean time, provides the reader a means to assess the requirement’s criticality.

![Diagram](image)

**Figure 6.1: Connecting Pre- and Post-Market Review**

A not so clear good, brought by associating requirements with typical failure symp-
toms, is that the Goal Graph thus documented can be treated as a function mapping a failure symptom to a set of corresponding requirements. Hence, it is possible, in the event of a software failure, to identify through a top-down search within the graph the requirement whose violation corresponds to the failure. The so-found requirement implies that its justifying evidences that manufacturers presented and documented in relevant assurance cases are not sufficient enough to support their claims of the requirement having been satisfied.

It should be noted that problematic evidences, usually in the format of testing results, mean either that manufacturers focus on wrong aspects of the code to compose test cases, or that test cases they devised do not cover all possible inputs. Both cases, however, indicate the portion of code that is most likely contain defects leading to the failure. Moreover, arguments that manufacturers use to link their claims and ill evidences, although not entirely correct, do expose information on the mechanism manufacturers adopt to fulfil the corresponding requirement. The information underlying these evidences and arguments can assist our error tracing framework to compose accurate Maluse Cases in less effort. For example, the link connecting requirements, evidences and arguments could narrow down the range of implementation that our framework should consider, and reduces the difficulty of relating error reports to the implementation.

Moreover, based on a throughout study on typical ways how requirements are structured and how manufacturers react to them, it is conceivable that more intricate connections underlying requirements and implementations can be found. Hence, the problem of formalizing failure reports can be tackle down to the problem of correlate the terminology that users have and the one that regulators possess.

Figure 6.1 illustrates a possible solution to the above idea of utilizing the Goal Graph model and its corresponding assurance cases in post-mortem analysis. The solution relies on knowledge in the domain, formalized as Domain Knowledge Base in Figure 6.1, to connect entities commonly used in these reports and typical failure symptoms. In order to acquire sufficient domain knowledge, it is necessary to conduct comprehensive ontology study on surveillant/error reports in the field.
Bibliography


