Research on Formal Verification and Program Segment Testing for Software Reliability

RAO LEI

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Graduate School of Advanced Science and Engineering
Hiroshima University
June, 2024

Abstract

This dissertation explores the use of formal verification and Program Segment Testing (PST) to enhance software reliability. Initially, formal verification techniques, such as Event-B and Labeled Transition Systems (LTS), were proposed to ensure software correctness and reliability. These methods provide rigorous mathematical frameworks for modeling and verifying system behavior and are particularly suitable for safety-critical and completed systems or specifications. However, these techniques are less suitable for the iterative and evolving nature of software development, particularly in the context of Human-Machine Pair Programming (HMPP).

To address these limitations, this research introduces PST as a complementary technique. PST focuses on detecting runtime exceptions in both partial and entire programs during the software development process, providing real-time feedback without human intervention. Integrated within the HMPP framework, PST allows developers to identify and fix issues early, enhancing productivity and reducing debugging time.

The effectiveness of formal verification is evaluated through the detailed modeling and verification of the ARINC653 specification. Separately, the benefits of PST are demonstrated through experiments that showcase its ability to detect runtime errors early in the development cycle. This research highlights that while formal verification is powerful for ensuring system correctness in completed systems, PST offers significant advantages in iterative development environments by providing timely error detection.

This dissertation contributes to the field of software engineering by providing a comprehensive evaluation of formal verification and PST, highlighting their individual strengths and limitations, and proposing practical solutions for enhancing software reliability in different contexts.

Keywords: Combined Formal Method, Formal Verification, Human-Machine Pair Programming, Program Segment Testing, Software Reliability

Contents

Chapter 1

Introduction

1.1 Background and Motivation

Software reliability is a critical aspect of modern software development, ensuring that software systems perform correctly and consistently under various conditions. Software reliability refers to the probability of a software system performing its intended functions without failure over a specified period. This is particularly crucial in safety-critical systems, where software failures can lead to catastrophic consequences. Safety-critical systems include applications such as avionics, medical devices, and nuclear power plant controls, where failure or malfunction could result in serious harm to people, property, or the environment. Thus, improving software reliability is not just a technical necessity but also a societal imperative.

Traditional software verification techniques, such as testing and code reviews, often fall short in providing the necessary guarantees of correctness for these complex systems. These methods rely on sampling and heuristics, which may miss critical errors and do not offer formal guarantees about the system's behavior.

To address these challenges, formal methods have been developed. Formal methods are mathematically based techniques used to specify, develop, and verify software and hardware systems. They involve the use of formal logic and discrete mathematics to model and analyze system behavior, ensuring that the system adheres to its specifications. Formal verification, a process using formal methods to prove or disprove the correctness of a system's design with respect to certain formal specifications or properties, offers

a mathematically rigorous way to ensure software correctness. These techniques aim to eliminate ambiguities in system design and ensure that the system behaves as intended under all possible scenarios. Techniques such as Event-B and Labeled Transition Systems (LTS) are particularly effective for modeling and verifying the behavior of safety-critical and completed systems or specifications. Event-B, for example, allows for the incremental refinement of abstract models into detailed designs, ensuring correctness at each stage. LTS provides a graphical representation of system states and transitions, which is useful for analyzing concurrent and distributed systems.

Despite their strengths, formal verification techniques face significant challenges when applied to the iterative and evolving nature of software development. This is particularly true in the context of Human-Machine Pair Programming (HMPP), where software is developed in a collaborative and incremental manner. Formal verification can be resource-intensive, requiring substantial computational power and expertise. The process of creating and verifying formal models can be time-consuming and may not fit well with the fast-paced, adaptive nature of modern software development practices. As a result, these techniques are less practical for providing real-time feedback during the software development process.

To address these limitations, this research introduces Program Segment Testing (PST) as a complementary technique. A program segment refers to a sequence of statements derived from the current version of a program (a partial or completed program). PST focuses on detecting runtime exceptions within these segments during the development process. Integrated within the HMPP framework, PST provides real-time error detection without requiring human intervention, allowing developers to identify and fix issues early in the development cycle. This approach enhances productivity, reduces debugging time, and improves overall software reliability. PST's ability to work seamlessly in the background, continuously monitoring the software as it is developed, makes it particularly suited for modern, agile development environments.

The motivation for this research stems from the need to bridge the gap between the rigorous but resource-intensive formal verification techniques and the practical, real-time error detection provided by PST. By leveraging the strengths of both approaches, this research aims to enhance software reliability in a comprehensive manner, particularly for safety-critical and complex systems.

1.2 Research Objectives and Scope

The primary objective of this dissertation is to enhance software reliability through the evaluation and application of formal verification techniques and PST. The research aims to address the limitations of each method by leveraging their respective strengths in different phases of the software development process. Specifically, the research seeks to:

1. Assess the Effectiveness of Formal Verification Techniques

    - Evaluate the capabilities of formal verification techniques, such as Event-B and LTS, in ensuring software correctness and reliability for safety-critical and completed systems.

    - Investigate the application of these techniques to the ARINC653 specification, a critical avionics software standard, to demonstrate their effectiveness in modeling and verifying complex system behaviors.

2. Develop and Evaluate PST Methodology

    - Develop a robust PST methodology for detecting runtime exceptions in both partial and entire programs during the software development process.

    - Integrate PST within the HMPP framework to enable real-time error detection and feedback without human intervention, and evaluate its effectiveness through experiments and case studies.

3. Compare and Contrast Techniques

    - Conduct a comparative analysis of formal verification and PST to highlight their respective strengths and weaknesses.

    - Determine how PST can complement formal verification by addressing its limitations, particularly in terms of real-time application and iterative development processes.

4. Propose Practical Solutions and Best Practices

    - Based on the findings from the case studies and experiments, propose practical solutions and best practices for integrating formal verification and PST into the software development lifecycle.

- Suggest future research directions to further enhance the effectiveness of these techniques in ensuring software reliability.

This research focuses on evaluating formal verification techniques and PST independently to improve software reliability. The scope includes applying these techniques to specific case studies, such as the ARINC653 specification, and analyzing their effectiveness in detecting and correcting errors.

1.3 Limitations

While this research aims to provide a comprehensive evaluation of formal verification and PST, several limitations must be acknowledged:

1. Scalability Issues

Formal verification techniques, especially when dealing with large-scale and highly complex software systems, can face scalability issues. The computational resources and time required to model and verify such systems can be significant, potentially limiting the applicability of these techniques in real-world scenarios.

2. Domain Specificity

The findings from this research may not be directly applicable to all software domains without additional validation. The results may need adaptation to be relevant in different contexts.

3. Tool Support

While tools for formal verification are available, dedicated tools for PST are still under development. This research evaluates existing tools but does not propose improvements for them, acknowledging the ongoing need for better tool support to facilitate the use of PST in practice.

4. Partial Program and Entire Program Testing:

While PST is designed to detect runtime exceptions in both partial and entire programs, its focus on incremental and real-time error detection during development means it might not provide the same level of exhaustive verification as formal methods. This approach prioritizes practical and timely feedback over comprehensive validation.

5. Formal Method Expertise:

Implementing formal verification techniques requires a high level of expertise in formal methods and mathematical modeling. The steep learning curve and the need for specialized knowledge can be a barrier to widespread

adoption, particularly in development teams with limited experience in formal methods.

By acknowledging these limitations, this research aims to provide a balanced and realistic evaluation of the potential and challenges associated with formal verification and PST in enhancing software reliability.

1.4 Thesis Structure

This dissertation is structured as follows:

- Chapter 1: Introduction - Provides the background and motivation for the research, outlines the research objectives, scope, and limitations, and gives an overview of the thesis structure.
- Chapter 2: Literature Review - Reviews existing literature on formal verification techniques and PST, highlighting their development, applications, and the gaps this research aims to address.
- Chapter 3: Formal Verification for Software Reliability - Discusses the application of formal verification techniques, specifically Event-B and LTS, including a detailed case study on the ARINC653 specification.
- Chapter 4: Program Segment Testing for Software Reliability - Describes the PST methodology, its integration within the HMPP framework, and presents experiment results demonstrating its effectiveness in detecting runtime errors.
- Chapter 5: Comparative Analysis and Discussion - Compares the strengths and limitations of formal verification and PST, discusses their complementary nature, and provides insights into their combined use for enhancing software reliability.
- Chapter 6: Conclusion and Future Work - Summarizes the key findings, discusses the implications for theory and practice, and suggests directions for future research.

This structure ensures a comprehensive exploration of the research objectives and provides a clear pathway for understanding the contributions of both formal verification and PST in enhancing software reliability.

Chapter 2

Literature Review

2.1 Formal Verification

2.1.1 Formal Methods

"Formal methods" refer to various mathematical techniques used for the specification and verification of software. These methods utilize formal specification languages as modeling elements and rely on a set of tools to support formal checks and property verification [1]. With the support of formal methods, researchers can describe system requirements and behaviors using rigorous mathematical models and verify whether a given system or model meets the required properties[2]. Typical formal verification methods include model checking [3] and theorem proving [4].

Formal methods, known for their rigorous mathematical foundation, are considered one of the most promising approaches for future system verification. Currently, formal methods have been widely applied to the modeling and verification of various safety-critical systems, such as aerospace systems [5], railway systems [6], nuclear power management systems [7], and automotive electronic systems [8]. They are gradually extending into fields like the Internet of Things (IoT) [9], cloud computing [10], and artificial intelligence (AI) [11].

However, current formal methods often only model and verify certain aspects or attributes of a system. For instance, modeling languages based on model-checking methods, such as automaton and LTS, mainly focus on behavioral aspects. In contrast, formal systems based on theorem-proving methods, such as Event-B [12], Z notation [13], and VDM [14], primarily

focus on data refinement. To achieve comprehensive system verification, multiple attributes (e.g., timing, spatial efficiency, and energy consumption) must be considered. This not only requires ensuring data consistency but also verifying system behavior attributes.

To enhance the expressive power of certain formal methods, researchers often extend formal modeling languages based on actual requirements. For example, to improve the structural degree of a type system, languages like VDM++ [15], Obj-Z [16], and UML-B[17] have been developed. Event-B was proposed to meet the needs of event-driven systems, and CSP-OZ was introduced to visualize the control flow of abstract state machines[18]. To model and verify the temporal characteristics of systems, languages like Timed CSP [19], CSP-OZ-DC [20], and RT-Z [21] were proposed. These can all be seen as extensions of formal methods. Additionally, some research groups have proposed entirely new formal systems that fully incorporate the required expressive capabilities. For example, Circus [22] and OhCircus [23] combine features of CSP and Z.

However, in reality, there is no formal method that can express all aspects of a system's properties[24]. Moreover, learning to use multiple formal systems is often more practical than redefining a new one. Therefore, a more flexible and realistic approach is to combine various formal methods based on actual system modeling and verification needs. For example, to address the control flow modeling issues in the B-method and Event-B, researchers proposed CSP $\|$ B [25, 26, 27, 28, 29, 30, 31, 32] and $CSP \parallel Event - B$ [33, 34, 35, 36], which model CSP and B/Event-B separately and then integrate them into a single framework for analysis and verification. Similarly, to refine the temporal logic properties of Event-B models, S. Schneider and Thai Song Hoang proposed strategies that combine Event-B and LTL[37, 38, 39], which is more direct than using ProB.

The combined use of formal methods typically involves manual or automatic transformation, where elements from one formal system are mapped to corresponding parts in another. For instance, TLA+ models can be transformed into B models to facilitate verification [40], and glue code can be written to link ProB with the model checking tool LTSmin [41], enabling reachability analysis of B models [42]. This approach of using multiple formal systems for modeling and verification is known as combined formal methods [43, 44].

2.1.2 Formal Verification of Safety-Critical System

Baumann et al. used the VCC verification tool in an avionics project to verify all the function calls in the source code of the PikeOS partitioned operating system, ensuring the correctness of the kernel services provided to applications. They proposed a top-level abstract model and identified the simulation relation theorem between this model and the actual implementation of the operating system and its applications. This theorem helped in ensuring the overall correctness of the partitioned operating system kernel by identifying the properties that all components must possess [45, 46, 47]. Since safety-critical systems built on PikeOS depend on the correct implementation of spatial isolation mechanisms, verifying it must consider the correctness of memory isolation. Therefore, Baumann used the VCC tool to perform source-level verification of PikeOS's critical component, the memory manager [48].

Richards et al. conducted a security verification of Green Hills Software's commercial partitioned operating system, INTEGRITY-178B [49]. Their verification considered five key elements: (1) Formal specification of the system's relevant security properties; (2) Formal representation of the system's functional interfaces; (3) Semi-formal and abstract representation of the system's high-level design; (4) Semi-formal and detailed representation of the system's low-level design; (5)A model representing the correspondence between the above four elements. The system was modeled as a state transition system, which receives current and external inputs to produce new system states and external outputs. Using this approach, Richards verified the information flow security of INTEGRITY-178B's high-level model. The system's low-level design was modeled using the ACL2 theorem prover, ensuring that the ACL2 model corresponded with the C code.

The kernel of the ARM-based embedded partitioned operating system, PROSPER, consists of 150 lines of assembly code and 600 lines of C code. Dam completed the formal verification of PROSPER's information flow security by proving the bisimulation relation between the abstract specification and the kernel's binary code [50]. The system model only considers two partitions executing independently on two specific ARMv7 machines, communicating asynchronously via message passing. Dam ultimately verified that, apart from communication through designated channels, there is no direct or indirect influence between partitions. This was achieved by ensuring that partitions cannot read or write to each other's memory except through

explicitly using the pre-designated communication channels for message passing.

To ensure the information flow security of the Xenon partitioned operating system, Freitas and McDermott used Circus to create a formal model of Xenon [51] Murray et al. modeled and verified the security of the seL4 partitioned operating system kernel using Isabelle/HOL [52]. They used the specification of seL4 to verify the information flow security of the partitioned operating system. The specification defined a partition-based round-robin scheduling strategy, allocating fixed time windows to each partition.

The European EURO-MILS project aimed to precisely model PikeOS and its security policies using Isabelle/HOL, designing a general partitioned operating system model called CISK (Controlled Interruptible Separation Kernel) [53]. This model included several aspects of partitioned operating systems, such as interrupts and context switching between partitions, with detailed specifications suitable for the formal verification of industrial systems. Subsequently, they used the CISK method to verify the non-interference properties of PikeOS's inter-process communication API [54].

Sanan et al. used Isabelle/HOL in the ESA's IMA for Space project to construct a general partitioned operating system kernel functional model and security model [55]. The specification used ARINC 653 as the functional requirements and also covered hardware virtualization, CPU timers, and memory management. Zhao Yongwang et al. designed a top-level model of an ARINC 653-compatible partitioned operating system using Isabelle/HOL, considering partition management and other aspects of ARINC 653 [56].

2.1.3 Top-down Formal Modeling of Safety-Critical Systems

Z Notation, the B method, and the combined formal language Circus all support layer-by-layer refinement and correctness proofs, making them highly suitable for implementation-oriented formal modeling. Craig [57] used Z Notation to design and refine a relatively complete operating system kernel, proving that it could be directly translated into executable C code. Subsequently, he further designed and refined a partitioned operating system kernel, providing most of the functionalities of a partitioned operating system, including: (1) Tables for basic process management; (2)Allocation of partition memory spaces to non-overlapping addresses; (3) Inter-partition

communication through an asynchronous kernel-based messaging system; (4) Handling of temporal isolation between partitions using a non-preemptive scheduler and messaging system; (5) Clear interfaces and external process identifiers to maintain kernel protection, defining secure kernel exit operations in case of illegal interface access. (6) Exporting kernel resources as device processes. Craig used mathematical proofs to verify several basic properties of the kernel. His formal specification of the partitioned operating system is quite comprehensive, with a refinement level sufficient for direct translation into C or Ada code. However, Craig's work did not include a model for processes within partitions, and the hardware device models were relatively simplistic. All formal specifications and correctness proofs were completed entirely by hand.

Building on Craig's work, A. Velykis considered more security requirements and used automated Z Notation tools to further specify and verify the partitioned operating system [58, 59]. Using the Z/Eves automated theorem prover, Velykis formalized the specifications, eliminating syntax errors in Craig's model and verifying the feasibility and robustness of the API. The improved formal model of the partitioned operating system was entirely proven using automated theorem proving. However, Velykis's formal specification did not address the temporal and spatial isolation of partitions. Instead, the improved formal model focused on the core data structures of the partitioned operating system kernel, such as process tables, queues, and scheduling. The model mainly improved Craig's scheduler model and converted certain behavioral properties (e.g., deadlock analysis) from informal requirements to mathematical invariants for proof. Other components, such as message passing and memory management, were not proven using automated methods.

Critical Software's Andre used the B method to design a secure partitioning kernel (SPK) for a formal model of a secure partitioning operating system kernel [60]. They first fully developed the top-level model of SPK, completing the system architecture design, and used ProB [61] for simulation and verification. The top-level abstract SPK model consisted of memory management, scheduling, kernel communication, flow policies, and clock management. After verification, the top-level model was refined into a fully formalized SPK. As part of SPK, the information flow policy for partitions was refined to a level where C code could be automatically generated. This refinement process was completed with the assistance of Atelier B.

Kawamorita and colleagues also applied the B method to develop a se-

cure partitioning operating system kernel for embedded devices, named OS-K, and built a prototype on Intel's IA-32 architecture [62]. They used the B method as the development tool for the formal model and used Spin to verify the model's properties. The B4free tool was employed to generate and check proof obligations. Almost all 2700 proof obligations, including verifications, were automatically verified by B4free. The final partitioning operating system kernel provided several functionalities: partition management, inter-partition communication, access control for inter-partition communication, memory management, timer management, processor scheduling, device driver operations, and interrupt handling for I/O interrupt synchronization.

2.1.4 Formal Modeling and Verification of Safety-Critical System Specification

Gomes at the University of York used Circus to establish a formal model for the ARINC 653 standard in the IMA system [63]. Zhao Yongwang and his team at Beihang University proposed a refinement-based formal modeling and information flow security analysis method for ARINC 653 [64], using Isabelle/HOL for the formal specification and information flow security proof of ARINC 653. They reviewed industrial partitioning operating systems VxWorks 653 and two open-source partitioning operating systems, XtratuM and POK, using this formal specification, and discovered security vulnerabilities that could lead to information leakage [65].

Zhao Yongwang and his team also developed a formal model for the 57 services of ARINC 653 Part 1 using Event-B. They used the refinement structure of Event-B to gradually refine the abstract model of ARINC 653 and transformed the service requirements of ARINC 653 into the lower-level models. Zhao Yongwang utilized the formal reasoning capabilities of Event-B to identify three hidden errors and incompleteness in the specification. Their work included considering all factors required by a partitioned operating system, such as clocks, message queues, partition scheduling, and processes [65]. This comprehensive approach is a good example of using Event-B for the modeling and verification of complex systems.

Recently, Zhao Yongwang proposed a method for combining the Web Ontology Language (OWL2) and Event-B for modeling the ARINC 653 standard [56]. By using the ontology model of the partitioned operating system as an intermediary between the non-formal specification of ARINC 653 and

the formal specification in Event-B, they achieved automatic conversion from the ontology model to the Event-B model, resulting in a complete ARINC 653 specification described in Event-B. By introducing the ontology, the degree of automatic verification of the Event-B specification for ARINC 653 was further improved.

2.2 Program Segment Testing

2.2.1 Human-Machine Pair Programming

Wang proposes a systematic framework for detecting and fixing security vulnerabilities during code construction [66]. The framework uses attack trees to model various potential security threats in detail and generates vulnerability-matching patterns based on these models. These patterns can detect code vulnerabilities in real-time, providing detailed warning reports that include the location of the vulnerability, possible attack types, and suggested fixes. By leveraging the advantages of human-machine pair programming, developers and computers can work interactively to enhance the efficiency and accuracy of vulnerability detection and repair.

Then, he introduces a novel approach for identifying security vulnerabilities in software by using vulnerability nets, a special type of Petri net [67]. This method integrates data dependence graphs and control flow graphs to enhance the detection of taint-style vulnerabilities such as buffer overflows and injection vulnerabilities. The framework is tested on the Securibench Micro benchmark, demonstrating its capability to accurately identify various vulnerabilities with low false positive and false negative rates. The approach aims to combine the strengths of static analysis tools and manual audits by incorporating the expertise of security auditors into the vulnerability detection process.

The integration of artificial intelligence into HMPP has significantly transformed this methodology in recent years. The paper [68] investigates the potential for machine learning to enhance remote pair programming (RPP) by addressing common challenges such as pair incompatibility, imbalanced roles, and a preference for solo work. The authors collected a dataset of 3,436 utterances from 18 participants in a simulated RPP environment. They evaluated the effectiveness of machine learning algorithms in classifying dialogue acts, creativity stages, and pair programming roles. The study found that while

RPP dialogue poses challenges due to its unstructured nature, the choice of contextual dialogue features significantly improved the accuracy of machine learning classifiers. The results suggest that integrating machine learning agents into RPP could facilitate better coordination and collaboration in global software development and online computer science education.

Hannay et al. explore the impact of the Big Five personality traits on the performance of pair programmers, alongside other factors such as expertise and task complexity [69]. The study involved 196 software professionals from three countries, forming 98 pairs. The analysis, which included both confirmatory and exploratory parts, revealed that personality traits have a modest predictive value on pair programming performance compared to expertise and task complexity. The results suggest that factors such as programming skill and learning may be more influential. The study concludes that while personality traits have some effect, other human-related factors should be investigated to improve pair programming performance. The findings indicate that a focus on collaborative measures might be more beneficial for enhancing pair programming outcomes.

2.2.2 Runtime Exception

The paper [70] presents a comprehensive approach to identify and fix faults in Java programs that lead to runtime exceptions. This method addresses exceptions caused by incorrect value assignments, such as null pointer dereferences, arithmetic faults, and type faults. The technique combines dynamic analysis using stack trace information with static backward data-flow analysis. Starting from the point of the runtime exception, it traces back to the source statement where the erroneous value was assigned. The approach not only identifies the exact source of the fault but also provides context information to help developers repair the fault. The method is demonstrated through its application to null pointer exceptions, showing its effectiveness in locating and fixing faults compared to using static analysis or stack traces alone. The paper also discusses the implementation of this technique and presents empirical studies validating its advantages.

The paper [71] by Westley Weimer and George C. Necula, explores a dataflow analysis technique to identify and correct error-handling mistakes in programs. These mistakes often arise from improper resource management, such as failing to release resources or clean up properly along all execution paths. The analysis tracks obligations through the program paths, modeling

control flow in the presence of exceptions, and highlights violations of resource safety policies. The study identified over 800 error-handling mistakes in nearly 4 million lines of Java code, which commonly resulted in resource leaks, such as unclosed sockets, files, and database handles. The authors propose a programming language feature that ensures outstanding obligations are discharged by keeping track of them at runtime. This feature improves program reliability by systematically addressing the mistakes found through their analysis, resulting in more consistent and efficient error handling.

In [72], they present a machine learning approach to predict runtime exceptions in Java methods using static code analysis. The proposed method, D-REX (Deep Runtime EXception detector), leverages a neural network model called the Location Aware Transformer to predict runtime exceptions and identify exception-prone code elements. D-REX operates by constructing an Action-Context Token Sequence (ACTS) from Java methods, which captures key elements and their contexts within the code. This sequence is fed into the Location Aware Transformer model, which uses self-attention mechanisms to provide accurate predictions. The model is trained on a large dataset of Java projects from GitHub, achieving 81% Top 1 accuracy in predicting exception types and 75% Top 1 precision in identifying exception-prone tokens. The paper demonstrates the superiority of D-REX over other baseline models, such as Bi-LSTM and plain Transformer models, in both accuracy and precision. This method not only predicts the types of exceptions but also highlights the specific code elements likely to cause these exceptions, helping developers address potential issues proactively. The authors highlight the importance of handling runtime exceptions to prevent severe software failures and propose D-REX as a tool to assist developers in improving code reliability and robustness.

Sonal Mahajan et al. introduce a technique and prototype tool named Maestro [73]. Maestro aims to automatically recommend the most relevant Stack Overflow (SO) post for fixing runtime exceptions (RE) in Java code. The tool works by comparing the exception-generating scenario in the developer's code with scenarios discussed in SO posts. Maestro extracts relevant lines from SO posts' code snippets using Abstract Program Graph (APG) representations, which abstract and simplify the code structure for effective comparison. Maestro's evaluation on a benchmark of 78 Java runtime exceptions from top GitHub projects showed it could return highly relevant posts in 71% of cases, outperforming other state-of-the-art tools. A user study with 10 Java developers further validated its effectiveness, with partic-

ipants finding Maestro's recommendations relevant or highly relevant in 80% of instances.

2.2.3 Program Slicing

In [74], the authors introduce an innovative method for debugging Java runtime exceptions. This approach integrates program slicing, backward data flow analysis, and stack trace information to effectively identify the source of runtime exceptions caused by erroneous value assignments. The method begins with program slicing to narrow down the search scope to the relevant parts of the program. Subsequently, a backward data flow analysis is performed starting from the point where the exception occurred, using stack trace information to guide the analysis in determining the exact source statement responsible for the runtime exception. This combined approach addresses the imprecision of static techniques and reduces the performance overhead often associated with dynamic techniques.

Then, they present another method to identify the causes of null pointer exceptions in Java programs [75]. This approach begins with a backward program slicing from the dereference statement where the exception occurred, using stack trace data to guide the process. It then conducts null identification and alias analysis on the sliced program to accurately pinpoint the faulty statements responsible for the exception. By combining dynamic and static analysis, this technique mitigates the limitations of pure static methods, enhancing the precision of fault localization. Additionally, a visualization tool is provided to help developers understand and analyze the results. The implementation and results confirm that this hybrid approach outperforms static analysis alone in identifying null pointer exceptions.

Carlos et al. address the challenges of incorporating exception handling in program slicing [76]. The study demonstrates that the System Dependence Graph (SDG), commonly used in program slicing, can produce incorrect and incomplete slices when dealing with exception-handling constructs. The authors propose a new framework to correctly handle these constructs by representing all possible exception throwing and catching mechanisms. They introduce a new type of control dependence called conditional control dependence, which ensures more precise slices in the presence of catch statements. The proposed framework modifies the traditional construction of SDGs by creating an exception-sensitive version, termed ES-SDG. This new approach includes enhancements to control dependence computation, taking into ac-

count the unique behavior of exception handling constructs. The authors' solution addresses incompleteness issues seen in previous methods, ensuring that all relevant exception handling code is included in the slices when necessary. This method is applicable to most modern programming languages with exception handling capabilities, such as Java, C++, and JavaScript. By enhancing the accuracy of program slices, the proposed approach improves the effectiveness of program analysis and debugging, especially in scenarios where exception handling plays a critical role.

In [77], Matthew Allen and Susan Horwitz extend the existing program slicing techniques to accurately handle exception handling constructs in Java. The paper addresses the shortcomings of current slicing algorithms, which fail to account for the additional control and data dependencies introduced by exceptions. By incorporating try, catch, and throw constructs into the system dependence graph (SDG), the authors develop a method that correctly identifies all relevant program components affected by exceptions. This enhanced slicing technique ensures that both control and data dependencies are properly represented, providing a more accurate and complete slice of Java programs that include exception handling. Through this extension, the method supports better program understanding, debugging, and maintenance, especially in complex Java applications where exceptions play a critical role.

The paper [78] compares the effectiveness of statistical fault localization (SFL) and dynamic program slicing for identifying faulty code locations. In a large-scale study of 457 bugs across 46 open-source C programs, it was found that dynamic slicing was more effective for single faults, pinpointing faults 62% more accurately and requiring programmers to inspect fewer lines of code compared to the best performing SFL techniques. Conversely, SFL performed better for multiple faults. The paper advocates a hybrid approach, suggesting that starting with the top five most suspicious locations from SFL, followed by dynamic slicing, yields the best results. This combined method allows programmers to examine fewer lines of code, enhancing the efficiency and accuracy of fault localization.

2.2.4 Software Testing

The paper [79] by Saurabh Sinha and Mary Jean Harrold explores the impact of exception-handling constructs on various program analysis techniques, such as control flow, data flow, and control dependence. These techniques

are crucial for tasks like structural and regression testing, dynamic execution profiling, static and dynamic slicing, and program understanding. The paper highlights the challenges that arise when analyzing programs with explicit exception occurrences and presents new techniques and algorithms to construct accurate representations of these programs, ensuring correct analysis results. The study emphasizes the importance of considering exception-handling constructs in Java and C++ to avoid inaccurate analysis information that can lead to unreliable software tools. Empirical results from the study show the frequency and impact of exception-handling constructs in Java programs, underscoring their significance in various analyses. The authors propose algorithms to incorporate these constructs into control-flow and control-dependence analyses, and discuss their application in program slicing and structural testing. The paper concludes that accurate modeling of exception handling is essential for effective software analysis and testing, and provides a foundation for further research in this area.

In article [80], they discuss how AI and machine learning (ML) are revolutionizing the field of clinical microbiology. The use of AI is explored in various applications such as interpreting Gram stains, ova and parasite exams, digital plate reading of bacterial cultures, and advanced analysis of MALDI-TOF mass spectrometry and whole genome sequencing data. AI enhances the efficiency and accuracy of clinical microbiology by automating tasks that traditionally required significant human effort, thereby improving patient care. For instance, AI algorithms can automate the interpretation of blood culture Gram stains, assist in the identification and classification of parasites in stool samples, and streamline the analysis of bacterial cultures on agar plates. These AI tools use convolutional neural networks (CNNs) and other ML models to process and analyze complex datasets, offering significant advantages over manual methods.

Samer et al. explore the limitations and solutions for automating UI tests for dynamic web applications using the TestComplete tool [81]. The study highlights the challenges faced when using TestComplete's recorder tool for dynamic web pages, such as the tool's inability to recognize onscreen objects that change properties between test runs, leading to failed tests. To address these issues, the authors propose a methodology for writing robust test scripts using TestComplete's scripting API. By focusing on attributes that remain consistent across test runs, such as idStr, innerText, and ObjectType, testers can create more reliable and maintainable test scripts. The proposed solution involves writing scripts that ensure the web page is fully loaded be-

fore interacting with onscreen elements and using stable attributes to locate these elements. The authors conclude that while TestComplete's recorder tool has limitations, its scripting capabilities offer a powerful alternative for creating robust automated tests for dynamic web applications.

Hongyu Zhou et al. provide a comprehensive overview of the advancements in gesture recognition technologies and their applications in human-computer interaction (HCI) [82]. The study covers the principles and development of four primary gesture recognition methods: electromagnetic wave sensing, mechanical sensing, electromyographic sensing, and visual sensing. Each method's strengths and weaknesses are discussed in terms of dataset size, accuracy, biocompatibility, wearability, stability, and robustness. The authors highlight the improvements in sensor structures, signal processing algorithms, and the selection of characteristic signals that have enhanced the effectiveness of gesture recognition technologies. They also address the current challenges in gesture recognition, such as the biocompatibility of sensor materials, the adaptability and wearability of devices, and the stability and robustness of signal acquisition and analysis algorithms. The authors conclude that gesture recognition technology holds significant promise for applications in smart homes, medical care, sports training, and other fields, offering more natural and intuitive means of interaction compared to traditional keyboard and mouse interfaces. This systematic review serves as a valuable resource for researchers and developers working to innovate and improve gesture recognition systems for enhanced human-computer interaction.

2.3 Existing Work and Gaps

Research in the field of software reliability has produced a variety of methods and tools aimed at improving the correctness and robustness of software systems. This section reviews existing work on formal verification techniques, highlighting their development, applications, and the gaps this research aims to address.

2.3.1 Formal Verification Techniques

Formal verification has been a well-established field of research for several decades. Techniques such as Event-B and LTS have been widely used to model and verify critical systems. Event-B, with its emphasis on incremen-

tal refinement, allows developers to start with an abstract model and gradually introduce details while maintaining system correctness. This method has been successfully applied in various domains, including transportation, aerospace, and industrial control systems.

LTS provides a powerful framework for modeling the behavior of concurrent and distributed systems. By representing states and transitions, LTS helps in understanding complex interactions and verifying properties such as deadlock-freedom and reachability. Tools like the Label Transition System Analyzer (LTSA) facilitate the construction and verification of LTS models, making them accessible for practical applications.

Despite their strengths, formal verification techniques face several challenges:

1. Applicability to Safety-Critical Systems: - Formal verification techniques are well-suited for safety-critical systems and specifications. They provide a high degree of assurance that the system adheres to its specifications through rigorous mathematical proofs. However, these techniques are less suitable for use during the iterative and evolving phases of software development, such as in Human-Machine Pair Programming (HMPP).

2. Resource Intensity: - Formal verification requires significant computational resources and time, especially for complex systems. This can limit their practicality and make them less appealing for projects with tight deadlines or limited resources.

3. Expertise Required: - The steep learning curve and the need for substantial expertise in formal methods and mathematical modeling can be a barrier to their widespread adoption in the software development industry.

2.3.2 Program Segment Testing (PST)

PST is a novel approach proposed in this research to address the limitations of formal verification in the context of HMPP. Unlike formal verification, which aims to provide comprehensive proof of system correctness for completed systems, PST focuses on detecting runtime exceptions in both partial and entire programs during the software development process. PST is designed to work within the HMPP framework, enabling real-time error detection without human intervention.

The key advantage of PST is its ability to provide timely feedback on runtime errors, allowing developers to identify and fix issues early in the development cycle. This approach is particularly useful in agile development

environments, where rapid iteration and continuous integration are critical. By automatically monitoring the program and reporting errors in the background, PST enhances developer productivity and reduces the time spent on debugging.

2.3.3 Gaps in Existing Research

While formal verification offers valuable tools for improving software reliability in completed and safety-critical systems, there are several gaps in existing research that this dissertation aims to address:

1. Suitability for Iterative Development:

Formal verification is not well-suited for the iterative and evolving nature of software development in HMPP. There is a need for methods like PST that can provide real-time feedback and error detection during the development process.

2. Tool Support and Usability:

While tools exist for formal verification (e.g., Rodin for Event-B, LTSA for LTS), they often require significant expertise to use effectively. Additionally, there is a lack of dedicated tools for supporting PST, limiting its widespread adoption. This research evaluates current tools and highlights the need for improved tool support to facilitate the effective use of PST in real-world development environments.

3. Practical Applications:

While formal verification has been successfully applied in various domains, there is limited research on the practical application of PST in real-world development environments. This research will provide case studies and experiment evaluations to demonstrate the effectiveness of PST in detecting runtime errors and improving software reliability.

By addressing these gaps, this research aims to provide a comprehensive evaluation of formal verification and PST, highlighting their individual strengths and limitations, and proposing practical solutions for enhancing software reliability in complex systems.

Chapter 3

Formal Verification for Software Reliability

3.1 Combined Formal Method

The combined formal methods approach involves using multiple formal methods to complete the modeling and verification of various system attributes [83]. Clarke and others [24] pointed out in the 1990s that no single formal method could fully address the core issues of complex system modeling and verification in a completely satisfactory manner, suggesting the use of combined formal methods. Recently, Almeida and colleagues [2] have also argued that no current formal method can entirely meet the modeling and verification needs of complex systems. They believe that, given the current state of formal methods development, combining various methods and tools is an attractive solution, hence advocating for the use of combined formal methods.

Among the various combined formal methods, integrating model checking and theorem proving is considered one of the most promising approaches [24]. Ideally, this approach allows researchers to leverage the strengths of both methods while avoiding their respective limitations. However, their practical application still requires careful consideration of various factors. It is crucial to clearly define the objectives and principles of the combination, and to carefully choose the appropriate theorem proving frameworks and model checking tools. Otherwise, it may increase the cost of modeling and verification. To this end, we have summarized the typical views on the prin-

ciples and objectives of combined formal methods from existing research to guide the application of these methods in this paper.

1) Objectives of Combined Formal Methods

The objectives of combined formal methods clarify the capabilities that the newly obtained combined methods should possess after integrating two or more individual methods. According to literature [43], when combining model checking and theorem proving for system modeling and verification, the following goals should be achieved to some extent:

- G1: The combined method should enable a higher degree of automated verification for infinite state systems, minimizing the workload of interactive proof that requires human intervention.
- G2: The combined method should be able to verify larger state spaces than those that can be handled by using model checking alone, while also being capable of verifying complex control systems that are difficult to verify using theorem proving alone.
- G3: The combined method should be able to generate counterexamples (i.e., traces that violate properties) for infinite state systems, thus overcoming the limitation of theorem proving methods which can only provide a result (correct or incorrect) without offering counterexamples.

2) Principles of Combined Formal Methods

In reality, not all formal systems can be combined with each other. Therefore, when selecting two formal systems to be combined, the following principles should be adhered to [43]:

- C1: The transformation from one formal system to another should be correct, ensuring the behavioral semantic consistency between the model before and after the transformation.
- C2: The transformation process should be bidirectional.
- C3: The transformation should be applicable at every refinement level of the two heterogeneous models.
- C4: The theorem proving framework should be robust enough to be compatible with most model checking techniques.

3) How to Combine

Even after selecting two or more formal systems as the sources for combination, it is crucial to consider how to combine them effectively to achieve the desired goals. Clarke [24] suggests that when combining model checking and theorem proving methods, two key factors should be considered:

- F1: Choose an appropriate style to combine the formal methods. The chosen style should ensure that the advantages of each formal method are preserved. For example, languages like Z, B, and Event-B are known for their ease of understanding and low learning curve, making them accessible and easy to promote. When combining these languages with other methods, it is important to retain these advantages. Otherwise, the combination loses its significance.

- F2: Choose an appropriate meaning to combine the formal methods. This means finding a common mathematical foundation for the participating formal systems, such as LTS or automata. If the common foundation is not well-defined, the combination may just result in a simple mix of the two formal systems, without providing any additional benefits compared to using them separately. A clear interpretation of the combination allows for the formal modeling of different aspects of a system and facilitates the refinement and reasoning of the integrated views.

3.2 System modeling and verification framework based on combined formal methods

We adopt a modular decomposition approach to combine model checking and theorem proving. This involves decomposing the system model within the framework into verifiable fragments or components and then performing integrated verification. Based on the objectives and principles outlined in the previous section, and considering the strengths and weaknesses of various methods as well as the modeling and verification needs of safety-critical systems, this paper selects Event-B as the theorem proving framework and LTS [84] as the model checking method.

3.2.1 Event-B Theorem Proving Framework

Event-B [12] is currently one of the most software engineering-friendly formal languages. Its approach of gradual refinement and automatic code generation ensures the correctness and consistency of models while providing strong support for software engineering.

Based on the following reasons, this paper selects Event-B as the refinement framework for theorem proving methods:

- Event-B has a strict proof obligation generation mechanism and can be combined with many model checking tools for joint verification, which aligns with "Principle C4".
- Event-B models lack inherent behavioral semantics, allowing users to assign various behavioral semantics based on actual needs, such as LTS semantics, automata semantics, etc. This meets the requirements of "Factor F2" and "Principle C4".
- Event-B models can be easily transformed into LTS models for model checking. Literature [33] indicates that an Event-B model can be viewed as a complex LTS, and existing work [85, 86, 87] has demonstrated the feasibility of this transformation. This aligns with "Principle C1".
- Event-B has a "UML-like" front-end iUML-B, which uses a Statechart-like graphical representation to express Event-B model variables and their states, and supports the expression of refinement relationships. This is very beneficial for decomposing Event-B models into sub-models that can be verified by model checking tools. This meets the requirements of "Factor F2" and "Principles C2 and C3".

We adopts a modular decomposition approach to combine Event-B with other model checking methods. Specifically, the Event-B theorem proving framework serves as the main framework, with model checking methods assisting in property verification. This section first provides a brief overview of the basic concepts of Event-B, including its modeling elements, refinement framework, and proof obligations. It then analyzes the shortcomings of the Event-B method, providing a basis for selecting model checking methods to combine with the Event-B theorem proving framework.

1) Basic Modeling Elements

An Event-B model consists of two parts: the *Context* and the *Machine* [12]. The Context part includes the static elements of the model, while the Machine part contains the dynamic elements. Contexts can be extended by other Contexts and can be referenced by a Machine. Each Machine can be refined by other Machines. The main components and refinement principles of an Event-B model are shown in Figure 3.1.

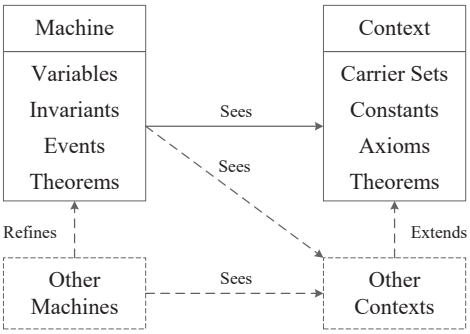


Figure 3.1: Main components and refinement principles of the Event-B model

The Context of a model can include definitions of *sets* and *constants*, as well as *axioms* describing the properties of these sets and constants. The Context can also contain theorems that must be proven to be consistent with the existing axioms. A Context can be extended by other Contexts and referenced by one or more Machines. Additionally, a Machine can indirectly reference a Context. For instance, if a Machine M can reference another Context $C1$ that extends Context C, then Machine M can indirectly reference Context C.

The Machine in an Event-B model contains the description of the dynamic behavior of the model. A Machine is composed of basic elements such as $Variables$, $invariants$, $Events$, and $Theorems$. Variables, like constants, correspond to simple mathematical objects such as sets, binary relations, functions, and numerical quantities. An invariant $I(V)$ is a logical expression defined over a set of variables, representing the properties that must hold true when the Event-B model performs various behaviors. Therefore, when the values of variables change, the invariants should always remain true. The preservation of invariants must be proven through the fulfillment of proof obligations.

An Event-B Machine can contain one or more Events, which define the possible state transitions of the model. Each event is composed of four el-

ements: the event name, event parameters, guards, and actions. As shown in Figure 3.2, *guards* are the necessary conditions for the event to be executed, while *actions* describe the changes in state variables when the event is executed.

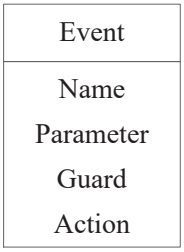


Figure 3.2: The components of event in Event-B

An event can only be executed if its preconditions (guards) are satisfied. When the guards of several events are true simultaneously, the specific event to be executed is nondeterministic, meaning only one of the events can be executed.

2) The Stepwise Refinement Framework of Event-B

In an ideal formal methods-based software development process (such as the Correct-By-Construction approach), it is desirable to gradually add design details from the highest level of abstract specification until the final implementation specification is achieved. This introduces an important concept: refinement consistency. Refinement consistency refers to how to derive an implementation of a system from an abstract specification so that it has behavior equivalent to the system specification. In other words, given a concrete specification, how to ensure that its behavior is the same as the system specification. This is a formal relationship problem between abstract specifications and concrete specifications.

Refinement [88, 89] is a mechanism that allows model developers to gradually add details to a system model until it becomes an implementable model, ensuring the consistency between the refined model and the abstract model. The main principle of refinement is that if the initial formal model is valid and the refinement process is correct, then the stepwise refinement process will yield a correct implementation model.

Refinement operators are mechanisms that support the transformation of system models. They provide rules for converting an abstract model into a

more concrete implementation model while maintaining the required properties [90]. Originally developed for the refinement of sequential programs, refinement operators were extended by Back [91] to handle distributed and concurrent system models through action system refinement.

Morgan [92] proposed a rule-based refinement method, which uses rules to automatically transform a model S from one form into another form S'. This transformation $S \sqsubseteq S'$(where S' refines S) is valid only if S' satisfies all the expected properties of S. Since this method is rule-based, it ensures that the concrete model is always a refinement of the abstract model.

Another refinement method is the "posit-and-prove" method. This approach involves rewriting the concrete model based on the abstract model (positing) and then using theorem provers or model checking tools to prove that the concrete model is a correct refinement of the abstract model (proving). This method requires the modeler to manually verify the correctness of the refinement, making it a non-automated approach.

Event-B and VDM use the typical "posit-and-prove" refinement method. As mentioned earlier, during the development of an Event-B model, the abstract model is continuously rewritten into a more concrete model, and it must then be proven that the concrete model is a correct refinement of the abstract model. The required properties are specified as invariants, which are predicates composed of state variables that must always hold true during the refinement process. If the model needs to follow certain LTL properties or timing constraints, these must also be specified as invariants and proven to remain true throughout the refinement process.

According to Abrial [12], refinements in Event-B can be classified into two categories: horizontal extension and vertical refinement. Horizontal extension involves introducing new objects into the model to meet system requirements that were not modeled at the previous level, thus deferring these needs to the next level. In a concrete Event-B model, this could mean introducing new variables to represent the state of these objects and events. For example, in an abstract model of a control system, only the controller components might be included initially. In the next level of horizontal extension, controlled objects and the environment are gradually introduced until the entire system is incorporated into the model.

Vertical refinement aims to add design and implementation details of specific objects or components to the abstract model, further describing how the system achieves a particular function. In Event-B models, this means decomposing an event into more sub-events (edge refinement) or concretizing

abstract state variables (node refinement). For instance, when modeling a file transfer protocol, an abstract model might include a "send message" event, which could be refined into "assemble message", "buffer message", and "transmit" events at a more concrete level.

In the actual system modeling process, both horizontal extension and vertical refinement are typically used simultaneously. For concurrent systems, horizontal extension is a very important method. It allows the modeler to avoid facing all the system objects at once by gradually introducing each object. Vertical refinement, on the other hand, allows for the behavior of each object to be detailed from an initial abstract description to more concrete implementation details.

Based on refinement operators, a theorem proving-based modeling language can provide a framework for the stepwise refinement of a system. This involves initially writing the highest-level abstract specification in mathematical language and then gradually refining the model step by step until it leads to an implementation model. Each refinement step makes the model increasingly concrete and closer to the actual implementation. In a theorem proving framework, numerous related proof obligations ensure that each refinement step is valid, meaning that the concrete model retains the properties of the abstract model. Therefore, assuming the original model is correct and each refinement step is proven to be valid, the system derived from or automatically generated by the final model will have a high degree of reliability.

The stepwise refinement process of Event-B is as follows: the initial specification S is the initial model M_0; then it can be refined into a more concrete model M_1, which is further refined into M_2, and so on, until the final implementation $M_n = E$. This is show in Equation:

$$S = M_0 \sqsubseteq M_1 \sqsubseteq M_2 \sqsubseteq M_3 \sqsubseteq ... \sqsubseteq M_n = E$$

3) Proof Obligations in Event-B

To ensure the well-formedness and correctness of refinements, the Event-B modeling and verification process involves generating a large number of proof obligations using theorem proving tools (such as Rodin [93]). These proof obligations must be correctly discharged to validate the model. Another use of proof obligations is to ensure that certain invariants are maintained, thereby guaranteeing that the system satisfies specific properties. The Rodin proof obligation generator creates several types of obligations:

- Well-Definedness (WD): Ensures that the axioms, invariants, guards,

and actions in the model are well-defined.

- Invariant Preservation (INV): Ensures that invariants are not violated during state changes.
- Guard Strengthening (GRD): Ensures that the guards of concrete events are a correct refinement of the corresponding guards in the abstract model.
- Action Simulation (SIM): Ensures that the actions of concrete events correctly refine the actions of the abstract events.

4) Limitations of Event-B

Although Event-B has many advantages, as Clarke pointed out, no single formal system can meet all modeling and verification needs. Overall, the following limitations of Event-B make it challenging to fully meet the modeling and verification requirements of partitioned operating systems:

(1) The Event-B Refinement Process Cannot Guarantee LTL Properties

From the analysis of the Event-B refinement process and proof obligations in the previous section, it is clear that Event-B's refinement is primarily at the event level. This means the "Action Simulation (SIM)" relationship only ensures that if there is an event EA in the abstract model, there must be a corresponding refined event ER in the refined model. The correctness condition for both the abstract and refined models is simply that the guards in the refined model are stronger than those in the abstract model.

However, there is an issue here: Does refining an event equate to refining behavior? Clearly not, because behavior is a sequence of events. In the Event-B refinement process, even if the model fully complies with the Event-B refinement process and all proof obligations are discharged, it cannot guarantee behavioral equivalence between the abstract and refined models. Here, behavioral equivalence specifically refers to maintaining LTL properties.

As Schneider [37, 39] and Thai Song Hoang [38] pointed out, the Event-B theorem proving framework does not specifically provide proof obligations for maintaining LTL properties. Typically, behaviors are ensured by adding the required properties to the invariant section and proving they are not violated during the modeling and refinement process. Alternatively, one can use model checking tools like ProB to verify certain temporal logic properties. Therefore, tools are needed to ensure the behavioral equivalence between the

lower-level and upper-level models, thereby ensuring that LTL properties and timing constraints are maintained throughout the refinement process.

(2) Event-B is Not Well-Suited for Expressing and Verifying Control Flow

As a modeling language based on first-order predicate logic, Event-B inherently struggles to express the control flow of a system—the sequence of events that occur. Although there has been extensive research on expressing control flow in Event-B, such as the AD/ERS method, Flow method, and $CSP \parallel B$ method, these approaches have not been as intuitive as graph-based formal systems for representing control flow. Essentially, an Event-B machine consists of a "flat" collection of events. Moreover, Event-B lacks inherent behavioral semantics. While Butler has given Event-B behavioral semantics using a behavioral systems approach, and Hoang has assigned refinement semantics using the CSP method, these efforts still fall short of providing direct behavioral formalism like LTS.

The lack of behavioral semantics poses a significant challenge to verifying LTL properties in Event-B models. Extracting the behavioral semantics directly from an Event-B model can require extensive learning and analysis. For instance, the MBT (Model-Based Testing) method [94, 95, 96] acquires the behavioral semantics of an Event-B model through model learning. However, this approach does not directly provide behavioral semantics within the model. Providing explicit control flow constructs during the process of assigning behavioral semantics to Event-B models offers a more reasonable path.

(3) Event-B Models Do Not Support Composition and Decomposition of Concurrent Objects

Composition and decomposition have long been classic methods for addressing model state space explosion. Although theorem proving methods can theoretically handle infinite state systems, the cost of proofs increases dramatically as the scale reaches the limits of automatic tools. Therefore, researchers have proposed various methods to support the composition and decomposition of theorem proving languages, such as Obj-Z, VDM++, and the Decomposition method in Event-B [97]. However, the Decomposition method struggles to achieve true decomposition, meaning that multiple subsystem models can execute in parallel within the same machine.

Researchers have also developed the Modularisation method for Event-B [98, 99, 100], but this approach is limited to interface-level decomposition. The fundamental issue with these methods is their difficulty in converting to automata-based formal models that support composition and decomposi-

tion. Colin Snook's UML-B [101, 102, 103, 104] and the subsequent iUML-B method [105] use state machines to represent the transitions of concurrent components or objects, which is closest to the model checking approach of system modeling. However, current research on UML-B primarily focuses on data refinement [106], with less emphasis on model composition and decomposition.

Based on the above analysis, although Event-B provides a powerful theorem proving framework, it has significant limitations in modeling and verifying safety properties and supporting composition. These limitations make it difficult to meet the modeling and verification requirements of safety-critical systems. Therefore, it is necessary to complement Event-B with model checking methods.

3.2.2 Choice of model checking methods

This paper chooses Finite State Process (FSP)[107] as the model checking component in the combined formal methods framework.

1) LTS Modeling Language FSP

FSP, like CSP, is a formal specification language based on process algebra. It provides a concise way to describe LTS using syntax similar to CSP, rather than directly describing the system as a list of states and transitions between them. The reasons for choosing LTS as the behavioral modeling language for Event-B in this paper are as follows:

(1)Firstly, the foundation of model checking is the transition system [3]. The essence of model checking is to exhaustively search to determine whether all states of a transition system satisfy (or do not satisfy) certain behavior properties expressed in some form of temporal logic. Although researchers have proposed numerous formal systems based on automata or process algebra to model system behaviors, such as CSP [108], CCS [109], and Interface Automata [110], these formal systems typically use LTS as their behavioral semantics model. Therefore, this paper uses LTS as the behavioral semantics model for Event-B, fundamentally addressing the limitations of other methods (such as $CSP \parallel B$) that cannot be universally applied. Another advantage of using the combination of LTS and Event-B is the ability to further utilize variants of LTS, such as IOLTS and TIOTS, to achieve more powerful analysis and verification capabilities.

(2)FSP was originally invented for modeling the behavior of multithreaded, concurrent systems, making it highly suitable for the behavioral modeling of

safety-critical systems. Like CSP, the FSP model is an event-based model, allowing the system to be described as a set of interacting components, each modeled as a state machine. The visual state transition diagrams of FSP provide valuable insights and support for understanding the control flow in Event-B models. Using the FSP modeling language, two or more LTS can be combined at any level of granularity. This is of great significance for modeling control flow in the stepwise refinement process of Event-B models and for the separate analysis of control flows between components.

(3)The FSP model provides a foundation for a wide range of automated analysis techniques, particularly deadlock analysis, model simulation, and model checking. Variants of linear temporal logic, such as Fluent Linear Temporal Logic (FLTL) [111], can be used to check various temporal isolation properties, including liveness and safety properties, of the LTS system models.

3.3 Preliminary of Event-B and LTS

3.3.1 LTS and its Combinations

LTS belongs to a specific category of automaton, which is widely used to model and analyze the behavior of concurrent and distributed systems [112]. LTS is a state transition system in which the transitions are marked as actions. The set of actions of the LTS is called its communication alphabet [113]. The following is the formal definition of LTS and its composition.

Definition 1 (LTS [1]): Let $States$ represent a universal set of states, $Acts$ represent a universal set of actions, and then an LTS P is defined as a quaternion $P = \langle Q, \Sigma, \Delta, q \rangle$ where:

- $Q \subseteq States$, representing the state set of P;
- $\Sigma = \alpha P (\alpha P \subseteq Acts)$, representing the action set of P;
- $\Delta \subseteq Q \times \Sigma \times Q$, representing the transition relation in P, these transitions are labeled with the elements in Σ;
- $q \subseteq Q$, representing the initial state of P

If $P^{'} = \langle Q, \Sigma, \Delta, q^{'} \rangle$, and $(q, a, q^{'}) \in \Delta$, LTS P can be converted to LTS $P^{'}$ by action $a (a \in Acts)$, denoted as $P \xrightarrow{a} P^{'}$.

We need to use the parallel composition of LTSs to express the interaction between multiple LTSs. The following gives the definition of LTS parallel composition.

Definition 2: Parallel composition of LTSs: The parallel composition of two LTS $M = \langle Q_1, \Sigma_1, \Delta_1, q_1 \rangle$ and $N = \langle Q_2, \Sigma_2, \Delta_2, q_2 \rangle$ is expressed as $LTS(M \parallel N) = \langle Q_1 \times Q_2, \Sigma_1 \times \Sigma_2, \Delta, (q_1, q_2) \rangle$, where $\parallel$ is a commutative and associative operator, which means:

$$LTS(M \parallel N) = LTS(N \parallel M)$$

In addition, Δ is the minimum relation that satisfies the following constraints:

$$\frac{M \xrightarrow{a} M'}{M \parallel N \xrightarrow{a} M' \parallel N} \; a \notin \alpha N \tag{1}$$

$$\frac{N \xrightarrow{a} N'}{M \parallel N \xrightarrow{a} M \parallel N'} \; a \notin \alpha M \tag{2}$$

$$\frac{M \xrightarrow{a} M', N \xrightarrow{a} N'}{M \parallel N \xrightarrow{a} M' \parallel N'} \; a \neq \tau \tag{3}$$

where $a \in \Sigma_1 \cup \Sigma_2$, τ denotes an action that is internal to a subsystem, and therefore unobservable by its environment.

3.3.2 Event-B, iUML-B State Machine and Its Combination

Event-B is a formal modeling language evolved from the B method. An Event-B model consists of two parts: machine and context. The context describes the static elements of the system, including sets, constants, axioms, and theorems. The machine uses variables and events to describe the dynamic behavior of the system. In Event-B, an event consists of guards and actions, which can usually be expressed as:

$$e := WHEN\ guards\ THEN\ actions\ END$$

When the guards of the event are satisfied, the event can be triggered, and the expression in the actions part describes the change in the state variable when the event occurs.

However, since Event-B is based on set theory and first-order logic, there is inevitably a problem that modeling is not intuitive enough. Therefore, C. Snook invented a UML-like Event-B graphics front-end called UML-B [2]. UML-B uses common class diagrams and state diagrams to describe the state and actions of the system. The system model represented by UML-B can generate the corresponding Event- B code directly on the Rodin platform [93]. As UML-B is continuously applied and expanded, UML-B has evolved into iUML-B.

Each iUML-B state machine can automatically generate some code and embed it into the Event-B model. When there are multiple iUML-B state machines in an Event-B model, the behavior of the model is affected by all automatically generated code. In order to facilitate the analysis and verification of the behavior of these state machines, this article uses "$\otimes$" to represent the combination of iUML-B state machines. Its definition is as follows:

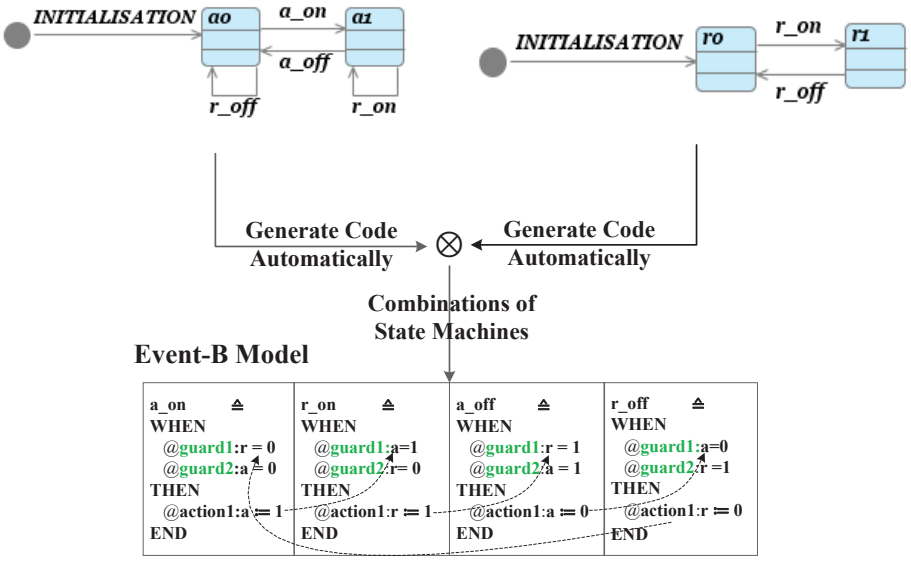


Figure 3.3: Combination of iUML-B state machines

Definition 3: Combination of iUML-B state machines: The Event-B model generated by the combination of two iUML-B state machines Stm_1

and Stm_2 is symbolically expressed as:

$$M = Stm_1 \otimes Stm_2$$

Further, the Event-B model generated by the combination of N iUML-B state machines $Stm_1, Stm_2, ..., Stm_N$ is symbolized as

$$M = \otimes_{i=1}^{N} Stm_i$$

For example, the Event-B model shown in Figure 3.3 is the combination of the state machine in the upper left corner and the state machine in the upper right corner.

3.4 Methodology

In order to combine Event-B and LTS in the process of system modeling and verification, we first work out the differences and connections between the event-based refinement process of Event-B and the state-based refinement process of LTS. Then we propose to use the graphical front-end iUML-B of Event-B to obtain a unified representation with LTS, and verify the bisimulation equivalence between them. Finally, we briefly discuss the improvements of this method.

3.4.1 Refinement Process in Event-B and LTS

1) Event-based Refinement Process

Refinement is a technology in which engineers build abstract models for software based on requirements documents in requirements analysis, and the process of modeling will build a series of more and more accurate models of software. Therefore, the refinement process is a process of increasing the function of the system and adding details. The establishment and refinement of the entire system model are completed by the decomposition and addition of events. Take Figure 3.4 as an example, in general, the refinement of the Event-B model follows the following process:

- Define an abstract event (E_{1_0})
- Perform one or more of the following two operations:

- Refine abstract events into (one or more) concrete events (E_{1_0} is decomposed into E_{1_1} and E_{2_1}) and add order constraints between concrete events (E_{1_1} occurs before E_{2_1}); or
- Add a new event (E_{3_2}) and add a constraint relationship between the new event and the original event (E_{3_2} occurs before E_{1_2}).

• Repeat the second step until the final refined model is obtained.

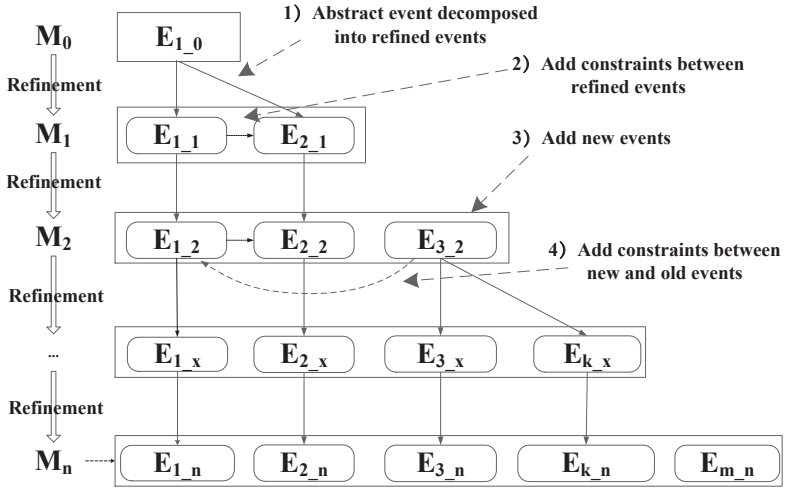


Figure 3.4: Event-based refinement process

This approach is very beneficial for expressing event-based systems, but it also brings about several problems: First, the refinement process is not clear, and there is no clear distinction between horizontal expansion (i.e., adding new objects) and vertical refinement (i.e., decomposition of the original object), for example, it is difficult to distinguish whether E_{2_1} is decomposed from E_{1_0} or a newly added event in the model. Second, it is difficult to model and analyze concurrent objects. In Figure 3.4, the newly added objects and the interaction and constraints between these new objects and the original objects cannot be seen. Third, the event-based refinement process is difficult to smoothly transform into a state-based expression.

2) State-based Refinement Process

The refinement of an LTS refers to a transformation from an abstract state machine into a concrete machine. An LTS model can be constructed by

means of a number of refinement steps. Specifically, such a refinement process includes the following actions:

- Establish an abstract behavior model of the system, usually an object containing a few states (an object O_1 containing states A and B).
- Perform one or more of the following two operations:
  - Decompose the state (transition) in the object into several concrete states (transitions) (state A is decomposed into D and C, and event e is decomposed into e_1, e, e_2), and/or
  - Add new states, which is to add new objects (an object O_2 composed of state G, I and K) and add behavioral constraints between new and existing objects (some constraints between objects O_1 and O_2).
- Repeat the second step until the final refined model is obtained.

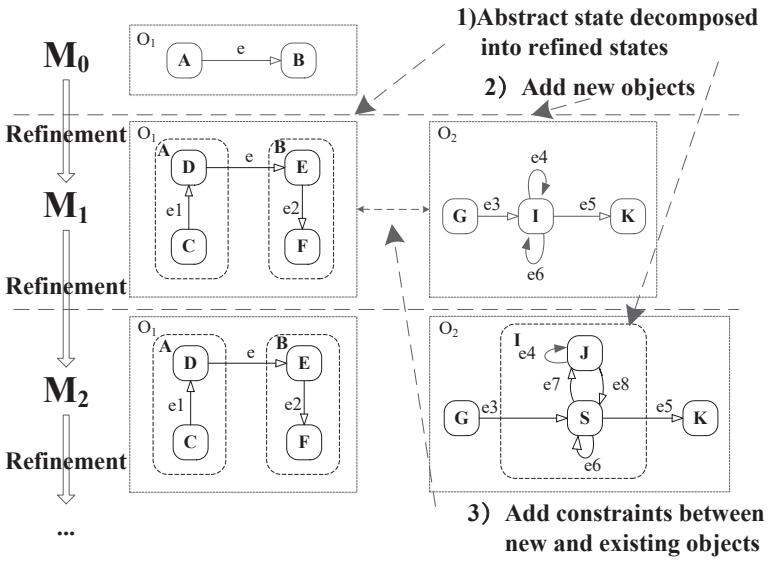


Figure 3.5: State-based behavior refinement process

The state-based refinement process is depicted in Figure 3.5. It can be seen that this refinement process is actually the decomposition and addition of the state in the object. The advantage of this expression is the ability

to clearly express the objects that make up the system and the interactions between the objects. The weakness is that the refinement relationship between the abstract model and the refined model is not clear. People cannot understand the correspondence between transitions or states in the refined model, and corresponding elements in the abstract model even by analyzing the model. For example, it is difficult to distinguish the correspondence between the states C, D, E, F in M_1 and A, B in M_0 in the model.

It can be seen that event-based refinement and state-based refinement have their own advantages and disadvantages. The former is strong in maintaining the vertical refinement relationship between the refined model and the abstract model, while the expressive ability in the addition of concurrent objects is weak. The latter is just the opposite. It can clearly express the behavioral interactions between objects, but lack the ability to express the refinement relationship of the model. Although studies have been conducted on how to transform an Event-B model to LTS, neither has been able to fundamentally resolve the syntactic and semantic gap between the two expressions, making it difficult to achieve a unified representation [4].

3.4.2 Unified Representation

This section uses the graphical front-end iUML-B of Event-B to obtain a unified representation with LTS, so as to reduce the syntactic and semantic gap between LTS and Event-B, thus realizing the purpose of achieving an appropriate combination. We achieve the combination by taking the following actions:

(1) The iUML-B state machine is used as a bridge between Event-B and LTS. We use the iUML-B state machine to express the state transition and time-lapse of an object, and use the combination of these state machines to express the behavioral interaction of concurrent objects. Here, we use the slightly modified BRP protocol model in Abrial's article as a case for an explanation [5]. As shown in Figure 3.6, this model contains three objects: $sender, receiver, and\ timer$. The *sender* is responsible for sending data to the channel, the *receiver* is responsible for receiving the data in the channel, and the *timer* is used to record the time. We model them separately. When they are combined, some behavioral interactions occur. For example, after the *sender* sends the data (*send_data*), at this time, each time the *tick_tock* event is triggered, the timer t will increase by one time unit. If the *receiver* receives the data within five time units, it means the reception is successful

($rcv\_success$), so the guard of this event is "$t \leq 5$", which is the prerequisite for the event to occur. However, if there is no reception within five time units, it means the reception failed ($rcv\_failure$). Therefore, the precondition for this event to occur is "$t > 5$". Regardless of whether the reception is successful or unsuccessful, the timer will be reset, so the action of $rcv\_success$ and $rcv\_failure$ is "t:= 0", which is the result of the event.

In this way, we use the iUML-B state machine to simulate the expression of LTS in system modeling, that is, to separate the objects in the system and model them separately, which provides a basis for completing the smooth transition from Event-B to LTS.

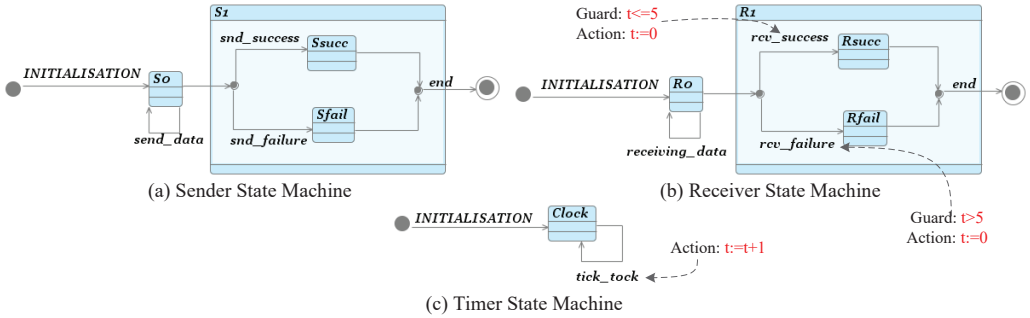


Figure 3.6: Timer State Machine

(2) In the vertical refinement direction, the node refinement of the iUML-B state machine is used to represent the node refinement and edge refinement in the state transition system [6], that is, state decomposition and transition decomposition. Figure 3.7(a) is an initial state transition model. If we want to refine the event $part$ into an event sequence $enter \rightarrow loop^* \rightarrow part$ (* means this event can occur from 0 to countless times), then we can adopt node refinement and edge refinement respectively to complete.

Node refinement is to replace a state in the abstract model with a super state in the refined model, then add new states within the super state, and add events between the states. As shown in Figure 3.7(b), the state I in the abstract model is replaced with the super state I , in which a state I with the same name and a new state J are added, two opposite edges are added between state I and J to represent the events $enter$ and $part$, and add a reflexive edge to the state J to get the required sequence of events and complete the refinement.

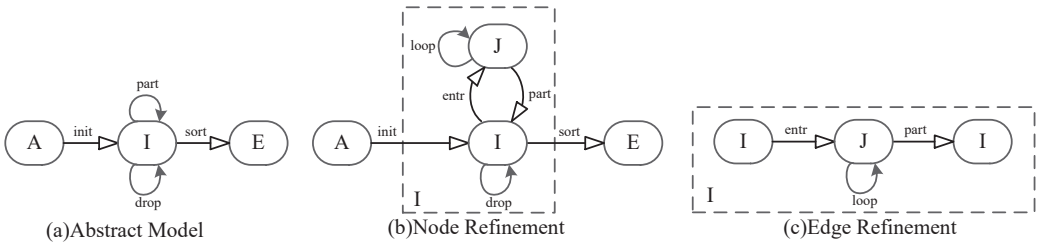


Figure 3.7: State refinement and edge refinement

Edge refinement is to replace the edge between the source state and the target state with a super state, and add some intermediate states to satisfy the target event sequence. As shown in Figure 3.7(c), first replace the edge *part* with the super state I , then add an intermediate state J, and finally use the edge marked by the event to connect the states to complete the refinement.

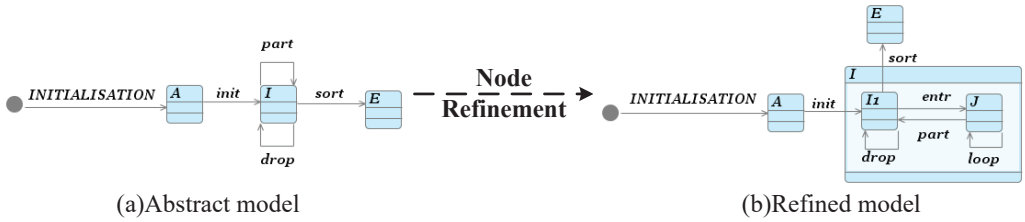


Figure 3.8: Node refinement of iUML-B state machine

For a state transition system like LTS, both state-based refinement methods can be uniformly represented by the node refinement of the iUML-B state machine. After node refinement of the abstract model in Figure 3.8(a), the refined model shown in Figure 3.8(b) is obtained.We retained the original I state and made it a super state. Since the state with the same name is not allowed in the iUML-B state machine, we add an I_1 node to replace the original I node, and the rest of the states and transitions are constructed as shown in Figure 3.7(b), that is, the refined model is obtained.

This means that all vertical refinement of LTS can also be achieved by the iUML-B state machine. Since the iUML-B state machine model is consistent with its automatically generated Event-B model, it can ensure the consis-

tency between the event-based vertical refinement process of the Event-B model and the state-based vertical refinement process of the LTS model.

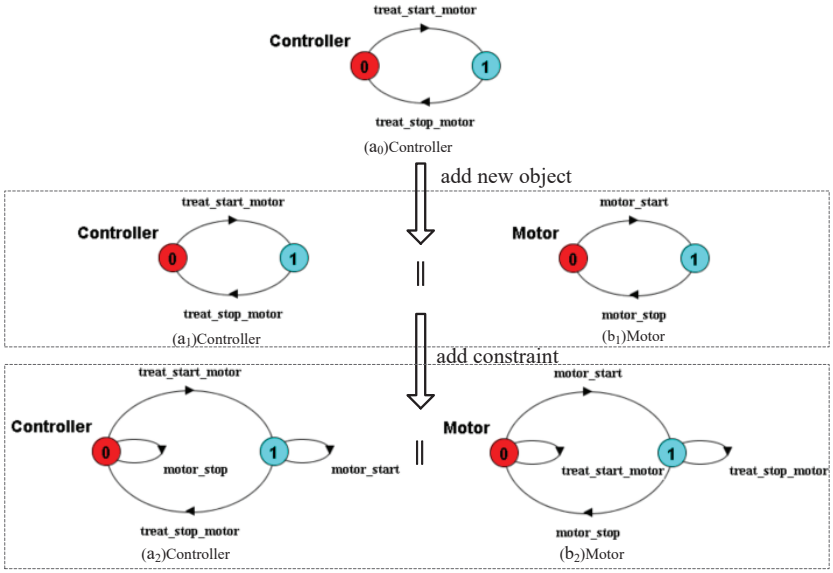


Figure 3.9: Combination of LTSs

(3) In the horizontal expansion direction, we achieve the purpose of adding new objects in the state transition system by adding new iUML-B state machines, and adding transition edges between states to add constraints between new objects and existing objects. The *Press* system [13] includes multiple components such as *controller* and *motor*. In the abstract model, there is only the *controller* LTS, and correspondingly only the controller iUML-B state machine, as shown in Figure 3.9(a0) and Figure 3.10(a0) respectively. When we want to add a *motor* object, the first is to add the *motor* iUML-B state machine, then consider adding the constraints between the *motor* and the *controller*. For example, if the *controller* issues a *start* command, the *motor* must respond. To put it bluntly, after the *treat_start_motor* event occurs, the *motor_start* event must occur immediately. In order to meet this constraint, we can add a reflexive edge "*motor_start*" to the "*ma_working*" state in the *controller* iUML-B state machine. Other constraints can be added one by one using similar operations, and finally we get an iUML-B model with the same behavior as the LTS model.

The horizontal expansion of LTS can be realized by adding new states and events in iUML-B. Similarly, because the iUML-B state machine model

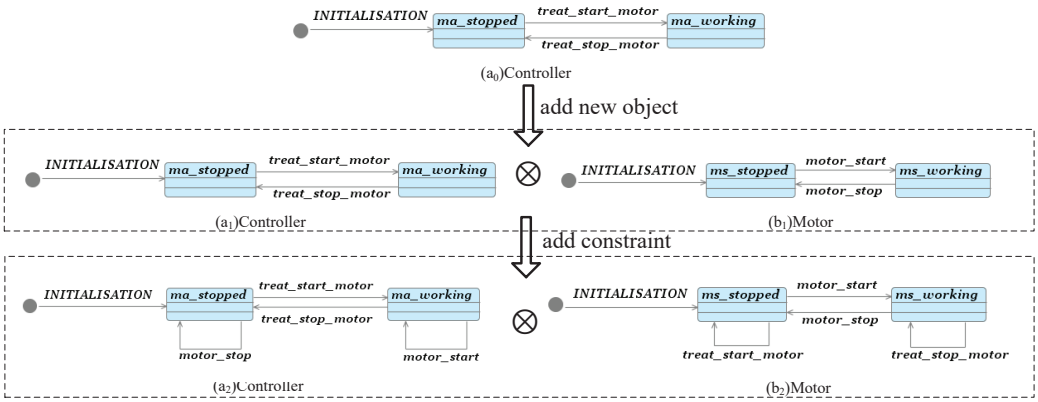


Figure 3.10: Combination of iUML-B state machines

and the Event-B model are consistent in behavior, it also means that the horizontal expansion of the Event-B model is supported, thereby ensuring the consistency of the event-based horizontal expansion process of the Event-B model and the state-based horizontal expansion process of the LTS model.

In the introduction, we mentioned that one of the main problems of the current integrated formal method is that there are gaps in the syntax and semantics of different formal methods, and that the system is modeled from different perspectives by modeling objects separately. There is no guarantee that the final system models will meet the same system requirements. Therefore, we consider first establishing the iUML-B model of the system, and then transforming it to the LTS model.

Since the iUML-B state machine itself is only an expression of the Event-B, which cannot directly obtain the corresponding LTS model. Therefore, we need to transform the iUML-B state machine model into the LTS model. When constructing the LTS behavioral semantic model, the central idea is to treat the Event-B model as a combination of all state variables, and at the same time treat each state variable of the Event-B model as an atomic LTS. Therefore, the LTS behavioral semantics model of the Event-B model is a parallel combination of all atomic LTS.

According to the description in our previous work [7], we use the following transformation rules:

*Rule*1: Each variable in the Event-B model is modeled as an atomic LTS,

and all possible values for this variable form the state space of this atomic LTS.

*Rule*2: For each atomic LTS P, if an event e in the Event-B model changes the value of its corresponding variable from s_1 to s_2, then we add an element $(s_1 \xrightarrow{e} s_2)$ to the transition set of this atomic LTS.

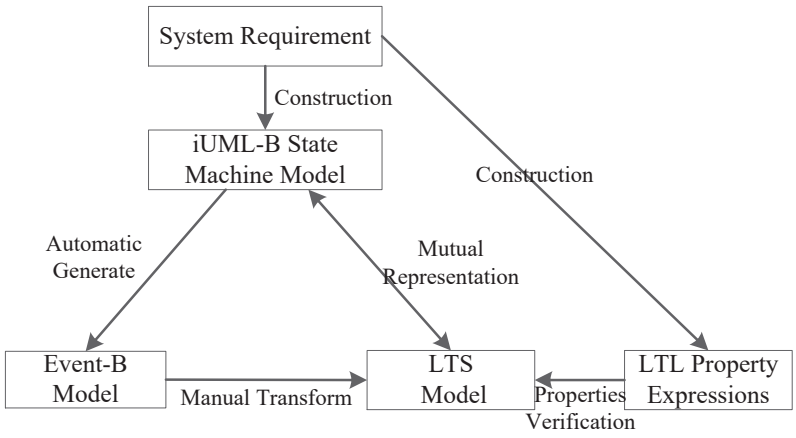


Figure 3.11: System modeling and verification process

During the construction of the model, Rodin will generate a large number of proof obligations for the Event-B model. Only when these proof obligations are correctly proved can the correctness of the model be guaranteed. In addition, invariant proofs are also used to verify whether the system violates the corresponding properties, including safety properties and liveness properties. However, the Rodin platform's ability to automatically prove properties is weak, which means that interactive proofs are required to manually derive property expressions, which consumes a lot of time and energy, and requires researchers to have a high mathematical foundation. LTS can use LTSA,1 a highly automated tool, to perform the lineal temporal logic (LTL) property expressions constructed according to the system requirements to verify whether the model meets the required properties. Therefore, we consider the property verification of the Event-B model to be done indirectly by verifying the properties of the LTS model. The premise is that there is equivalence between the Event-B model and the corresponding LTS model, which we prove in Section 3.4.

The application of these rules can be illustrated by a simple example as shown in Figure 3.11. When we model and verify the actual system, we first

establish its iUML-B state machine model according to the system requirements, secondly use the Rodin tool to automatically generate its Event-B model, then convert it to the corresponding LTS model according to the transformation rules. Finally, we verify whether the LTS model satisfies the necessary properties.

3.4.3 Proof of Equivalence

In order to prove that our method is correct, we give a proof of the bisimulation equivalence between the LTS model and the Event-B model, so as to ensure that the Event-B model generated from iUML-B model is consistent with the LTS model translated from the same iUML-B model. First, we give the definition of bisimulation [8].

Definition 4 (Bisimulation Equivalence): Let $LTS_i = (Q_i, \Sigma_i, \Delta_i, q_i)$, i = 1,2, be labeled transition systems over the actions set Σ. A bisimulation for $(LTS_1,\ LTS_2)$ is a binary relation $R \subseteq Q_1 \times Q_2$ such that

- for the initial state q_1 and q_2, $(q_1, q_2) \in R$
- for any $(s_1, s_2) \in R$, it holds that
  - if $s_1 \xrightarrow{a}_1 s_1^{'}$, then $s_2 \xrightarrow{a}_2 s_2^{'}$ with $(s_1^{'}, s_2^{'}) \in R$ for some $s_2^{'} \in Q_2$
  - if $s_2 \xrightarrow{a}_2 s_2^{'}$, then $s_1 \xrightarrow{a}_1 s_1^{'}$ with $(s_1^{'}, s_2^{'}) \in R$ for some $s_1^{'} \in Q_1$

LTS_1 and LTS_2 are bisimulation equivalent, denoted as $LTS_1 \sim LTS_2$, if there exists a bisimulation R for (LTS_1, LTS_2). The bisimulation equivalence relationship is transmitted, i.e., $LTS_1 \sim LTS_2 \wedge LTS_2 \sim LTS_3 \longrightarrow LTS_1 \sim LTS_3$.

In the previous section, we mentioned that a system LTS is composition of multiple atomic LTS_s:

$$LTS(System) = \|_{i=1}^{n}\ AtomicLTS_i \tag{4}$$

where i is the sequence number of the atomic LTS, and n is the total number of atomic $LTSs$.

The following is the proof process:

(1) We first establish an atomic LTS $AtomicLTS = \langle Q, \Sigma, \Delta, q \rangle$ based on the atomic iUML-B state machine. The construction process is as follows.

(a) An "atomic iUML-B state machine" is defined as $AtomicStm = \langle Node, E, Edge, InitNode \rangle$, where $Node$ represents a set of nodes in the iUML-B state machine; E represents the set of events that linked on the edges of the iUML-B state machine; $Edge \subseteq Node \times E \times Node$ represents the set of edges in the iUML-B state machine; InitNode represents the initial node of the iUML-B state machine, which is the target node of the edge that is linked to the Initialization event.

(b) In the process of establishing the atomic LTS, let $Q = Node, \Sigma = E, \Delta = Edge, q = InitNode$. For example, if there is a node s_1 (edge e) in atomic iUML-B state machine, a state s_1 (transition t) is also added in the corresponding atomic LTS. This mapping process is very easy to operate, and we will not explain it further.

(c) Generate the Event-B code from the iUML-B state machine using the automatic code generation tool Rodin.

(2) In the following, we prove that the Event-B model generated by AtomicStm is bisimulation equivalent to the $AtomicLTS$ translated from the same $AtomicStm$.

We define an Event-B model as $M = \langle V, Event, Guard, Action, V_{init} \rangle$, where V represents variables set of M, $Event$ represents event set of M, $Guard$ represents guard set of M, $Action$ represents action set of M, and V_{init} represents the initial value set for each element in the V. We define the LTS corresponding to the M as $LTS(M) = (Q_M, \Sigma_M, \Delta_M, q_M)$.

We named an Event-B model generated by $AtomicStm$ as $M_A = \langle V_A, Event_A, Guard_A, Action_A, V_{initA} \rangle$. It should be emphasized that at this time there is only one element var in V_A, and V_{initA} is the initial value of this element, because an $AtomicStm$ only describes the change of one variable. $Event_A$ represents events that modify the value of var, and $Guard_A$ represents those guards that contain var in the $when$ clause of an event. Similarly, $Action_A$ represents actions that modify the value of var. We assume that the type of var is D, that is, $var \in D$, then the state space of var is D.

Since the code of M_A is generated by $AtomicStm$, we have $LTS(Atomic\ Stm) \sim LTS(M_A) = (Q_{MA}, \Sigma_{MA}, \Delta_{MA}, q_{MA})$. If an Event-B model M is generated by a combination of multiple atomic iUMLB states machines, then we have

$$LTS(M) = \|_{i=1}^{n} LTS(M_{Ai}) \tag{5}$$

where i is the sequence number of the atomic iUML-B state machine, and n is the total number of atomic iUML-B state machines.

We explain the equivalence between $AtomicLTS$ and M_A according to the rules of Rodin for generating Event-B code from the iUML-B state machine.

(a) First, Rodin will generate a variable based on one $AtomicStm$ and automatically generate a SET which contains all possible values of this variable. For example, an $AtomicStm$ named $node$ which contains n nodes (e.g., $s_1, s_2, ..., s_n$) will generate

$$partition(Node, \{s_1\}, \{s_2\}, ..., \{s_n\}) \tag{6}$$

which means $node \in Node$. As we mentioned in (1) (b), $Q = Node$. Therefore, the state space D of variable $node$ of Event-B model M_A is equal to Q, and then $Q_{MA} = Q$.

(b) Secondly, Rodin generates the following code based on the edge that links the $Initialization$ event:

$$INITIALISATION := BEGIN\ node = s_1\ END \tag{7}$$

Since $s_1 = q$ (in the construction process (1) (b)), we have $V_{init} = q$, and then $q_{MA} = q$.

(c) Rodin will generate an event named "$event\_i$" in the Event-B model M_A according to the event "$event\_i$" which has been linked on the edge of iUML-B state machine, and will generate the following code according to each edge from the node s_i to s_j (where s_i and s_j are the node name):

$$event\_i := when\ node = s_i\ then\ node := s_j \tag{8}$$

Therefore, for a transition $s_i \xrightarrow{a} s_j$ in $AtomicLTS$, there will be a corresponding transition $s_i \xrightarrow{event\_i} s_j$ in LTS (M_A). At the same time, for each action a in AtomicLTS, there will be an event $event\_i$ corresponding to it in the Event-B model M_A. So, we have $\Delta_{MA} = \Delta$.

(d)The reverse mapping process from M_A to $AtomicLTS$ is similar, and we will not repeat them here.

(e) We can define a mapping relationship R so that $AtomicLTS$ and LTS (M_A) comply with the requirement of bisimulation equivalence. In fact, this R can be a renamed function, such as $R(Sender) = sender$. Now we get

$$LTS(AtomicStm) \sim LTS(M_A) \sim AtomicLTS \tag{9}$$

(3) Finally, we use the theorem in the literature [8].

Lemma 5 (Congruence w.r.t LTS Composition): For labeled transition systems LTS_1 and $LTS_1^{'}$ over Σ_1, LTS_2 and $LTS_2^{'}$ over Σ_2, and $H \subseteq \Sigma_1 \cap \Sigma_2$, it holds that

$$LTS_1 \sim LTS_1^{'}$$

$$LTS_2 \sim LTS_2^{'}$$

$$impliesLTS_1 \parallel_H LTS_1^{'} \sim LTS_2 \parallel_H LTS_2^{'} \tag{10}$$

According to expression (4) and expression (5) and Lemma 1, we have

$$\parallel_{i=1}^{n} LTS(M_{Ai}) \sim \parallel_{i=1}^{n} AtomicLTS_i \tag{11}$$

That is, the LTS model LTS (System) of the system and the LTS model LTS(M) of Event-B model obtained according to the mapping rule of (1) are bisimulation equivalent:

$$LTS(M) \sim LTS(system) \tag{12}$$

3.4.4 Discussion about the Ability of the Method

Having presented the proposed method, we need to discuss about the ability of the method in modeling and verification. We focus on the following four points for the discussion:

(1) LTS does not support model refinement, but the combination with Event-B gives it the ability for refinement.

(2) Use iUML-B state machines to model the behavior of concurrent objects and make up for the defects of Event-B in the expression of control flow.

(3) The iUML-B state machine is used to simulate the process of modeling concurrent objects in LTS. In the simulation, the combination/decomposition of the Event-B model is transformed into the combination/decomposition of the iUML-B state machine to avoid the learning of some complex methods in Event-B, such as Decomposition [9].

(4) iUML-B expresses the interaction between concurrent objects by means of constantly adding new state machines, which solves the problem that Event-B is difficult to express the interaction semantics of concurrent objects.

3.5 Experiment

To demonstrate the practicality of the method proposed in the previous section, we use an ARINC653 specification example to describe how it is used in practical applications in this section.

3.5.1 Introduction to the ARINC653 Specification

The ARINC653 software specification describes the kernel and related services of the standard APEX (Application Executive) [10]. These services are supported by a variety of safety-critical real-time operating systems (RTOS) for use in avionics. The APEX services specified in ARINC653 Part1 mainly include partition management, process management, time management, intra-partition (inter-process) communication, inter-partition communication, health monitoring, etc. These services determine the highly concurrent features of a Partition Operating System.

- Partition management services: In the ARINC653 specification, only the mode of the partition and the service of obtaining/setting the partition mode are specified. The partition mode includes IDLE, COLD START, WARM START, and NORMAL four working modes. The specification does not specify how the partition mode switching is implemented. Therefore, in the specification level modeling, only the phenomenon needs to be concerned, and there is no need to consider the object that triggers the phenomenon.

- Process management services: these services include creation, suspend, resume, stop, start, get process identifier and state, disable/allow process scheduling, and other functions. When modeling the state transition of the process, it is necessary to consider the mode in which the partition is located. For example, a process can start only when the partition is in NORMAL mode.

- Time management services: these services include services such as delayed waiting, periodic waiting, increasing process time budget, and obtaining current time value, etc.

- Inter-partition communication services: these services are mainly composed of queue port services and sampling port services, including ser-

vices such as creation, read/write, and status acquisition of these two types of ports.

- Intra-partition communication services: the intra-partition communication specifies various services that can be used for synchronization and mutual exclusion between processes within a partition. It is mainly designed around four objects, including buffers, blackboards, semaphores, and events.

- Health monitoring services: the health monitoring specifies the fault response and processing mechanism of the partition operating system, including three monitoring levels: system level, module level, and partition level.

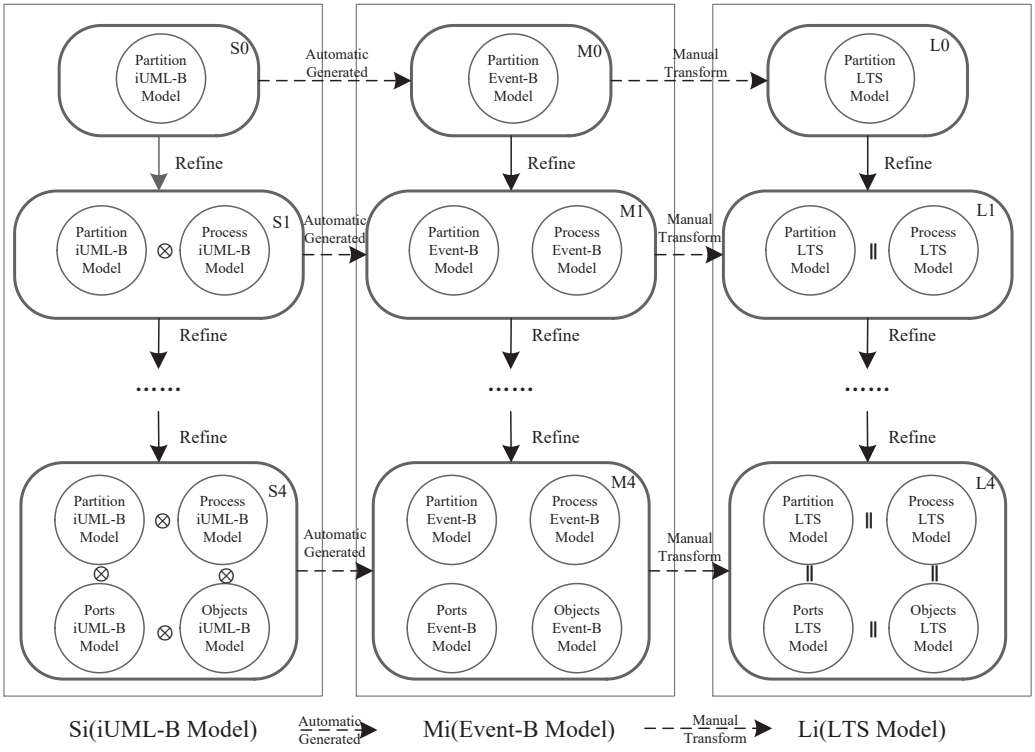


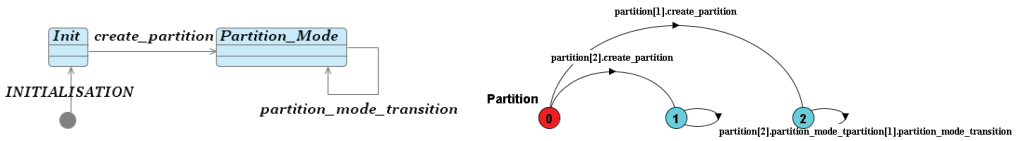
Figure 3.12: Modeling process of the ARINC653 model

3.5.2 ARINC653 Specification Modeling

In this section, we use our method to model the ARINC653 specification. Separate the concurrent objects that produce phenomena in the specification in the horizontal direction and model the interaction between these objects. In the vertical direction, we gradually refine the behavior of concurrent objects to ensure the behavioral consistency between the concrete model and the abstract model.

In order to reach the goal of making concurrent objects separated, we divide the objects involved in the specification into partitions, processes, inter-partition communication objects (queue port and sampling port), intra-partition communication objects (buffer, blackboard, event, semaphore), and use the method proposed in the previous section to complete the modeling of ARINC653 specification model, as shown in Figure 3.12.

We start with building the abstract model that only contains partitions, and then take four refinement steps to complete the modeling of all the required services except the health monitoring service. Figure 3.12 shows the final specification containing the models of partitions, processes, intra-partition communication objects and inter-partition communication objects. Next, in this paper, we use partitions and processes as an example to describe our modeling process.



(a) Partition initial iUML-B model

(c) Partition initial LTS model

create_partition ≙
STATUS
ordinary
ANY
PartId
WHERE
isin_Init : part(PartId) = Init
THEN
enter_Partition_Mode : part(PartId) ≔ Partition_Mode
END

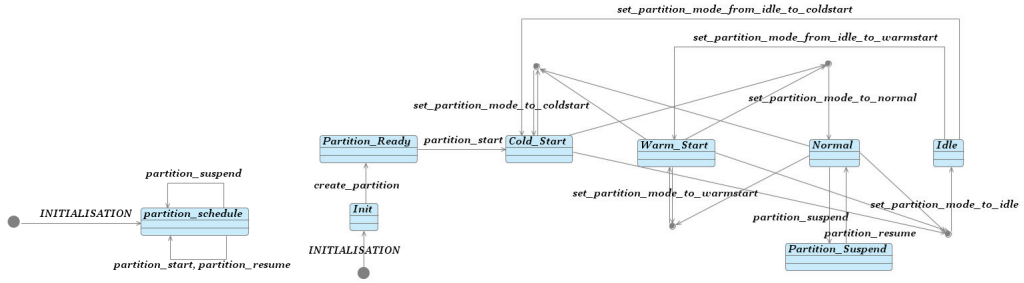
partition_mode_transition ≙
STATUS
ordinary
ANY
PartId
WHERE
isin_Partition_Mode : part(PartId) = Partition_Mode
THEN
skip
END

(b) Partition initial Event-B model

Figure 3.13: Partition initial model

Partition

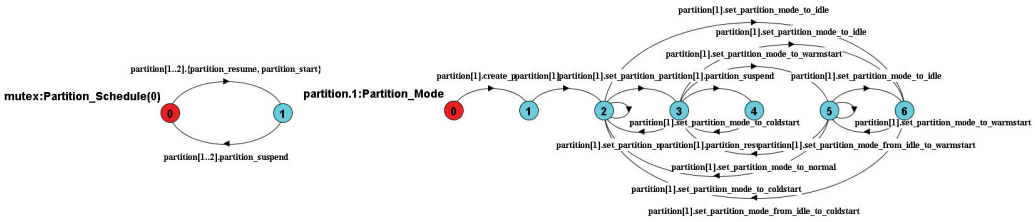
In the abstract model M0 of the system, we introduce the initial partition model, including the established partition iUML-B state machine model, the automatically generated partition Event-B model, and the transformed partition LTS model, as shown in Figure 3.13. It should be mentioned that in order to more easily and effectively simulate the model, we define two partitions in the model.



（a）Partition_Schedule　　（b）Partition_Mode

Figure 3.14: Partition model in iUML-B

After a partition is created (*create_partition*), the partition performs mode transition (*partition_mode_transition*). However, in the initial model, the mode of the partition is only modeled as one state (*Partition_Mode*), paving the way for being split into multiple partition mode states in subsequent refinements.



（a）Partition Scheduling LTS　　（b）Partition Mode Transition LTS

Figure 3.15: Partition model in LTS

The initial model is refined layer by layer to obtain the final partition iUML-B state machine model and the transformed LTS model, as shown in

Figure 3.14 and Figure 3.15, respectively. Since the code of the Event-B model is too large, we will not present it. The model consists of two parts, *Partition_Schedule* controls the scheduling of the partition, and *Partition_Mode* shows the mode transition of a single partition.

At any one time, only one partition can be scheduled. As shown in Figure 3.14 (a), when a partition is scheduled for the first time, the scheduler schedules the partition to the running state by the *partition_start* event. When the time window allocated to the partition ends, the scheduler sets it to the suspended state by activating the *partition_suspend* event, and waits for the next time window of the partition to arrive and then schedules the partition again by the *partition_resume* event.

In Figure 3.14 (b), one partition has four modes: $WARM\_START$, $COLD\_START$, $NORMAL$, and $IDLE$. After a partition is scheduled in a specified time window, it can be transformed between four modes in different ways. For example, when the partition is in $COLD\_START$ or $WARM\_START$ mode, it can be transformed into $NORMAL$ mode by the *set_partition_mode_to_normal* event.

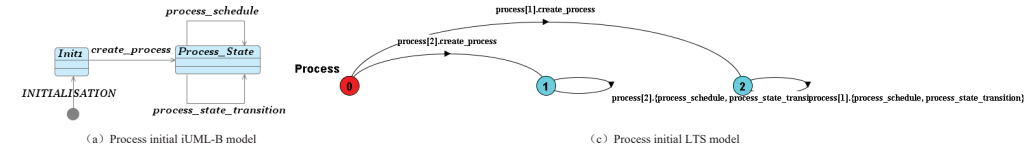
Process

The operating system views the execution of a process as a transition between a series of continuous process states. In order to complete the modeling of the process, we introduce the initial model of the process in the first-level refined model $M1$ and obtain the following iUML-B model, Event-B model, and LTS model, as shown in Figure 3.16. For the same reason, we only define two processes in the model.

The final process iUML-B state machine model and LTS model obtained by refinement describe the more specific process state and state transitions conforming to the partition mode, as shown in Figure 3.17 and Figure 3.18, respectively.

Combination of Partition and Process

After completing the construction of the partition model and the process model respectively, we need to consider how to add the constraints brought about by their combination in the model. For example, only when the partition is in *Normal* mode, can the process perform the startup operation. In other words, all events related to process startup in the process model can



(a) Process initial iUML-B model

(c) Process initial LTS model

create_process ≙
STATUS
ordinary
ANY
ProcId
WHERE
isin_Init1 : proc0(ProcId) = Init1
THEN
enter_Process_State : proc0(ProcId) ≔ Process_State
END

process_schedule ≙
STATUS
ordinary
ANY
ProcId
WHERE
isin_Process_State : proc0(ProcId) = Process_State
THEN
skip
END

process_state_transition ≙
STATUS
ordinary
ANY
ProcId
WHERE
isin_Process_State : proc0(ProcId) = Process_State
THEN
skip
END

(b) Process initial Event-B model

Figure 3.16: Process initial model

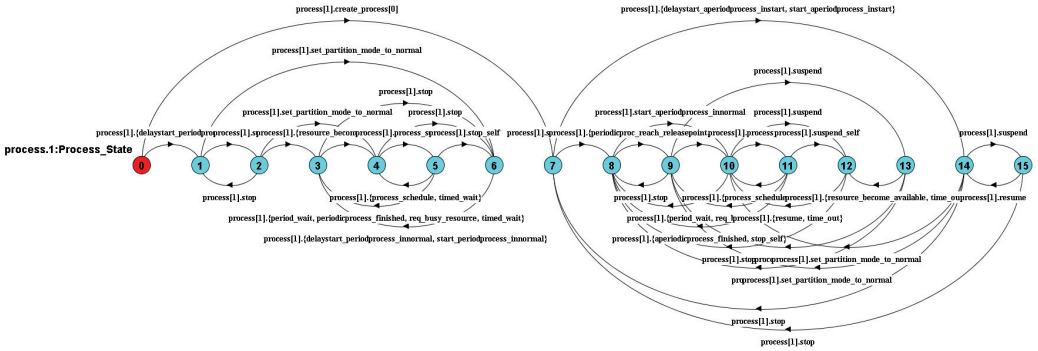


Figure 3.17: Process state transition model in LTS

only occur when the partition is in *Normal* mode. In order to meet this condition, we need to add the reflexive edges of these events to the Normal state in the partition model, as shown in Figure 3.19. The remaining constraints can be added one by one using similar operations until the correct behavior model is constructed.

3.5.3 Property Verification of the ARINC 653 Specification Model

Labeled Transition Systems Analyzer (LTSA) is a tool that can be used to check whether the specification of a concurrent system satisfies the required

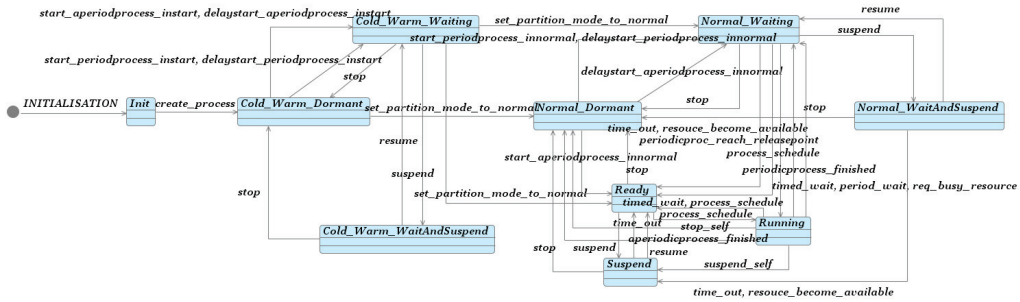


Figure 3.18: Process state transition model in iUML-B

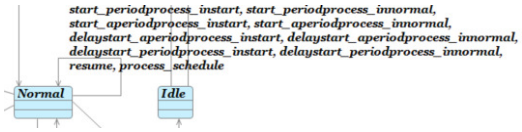


Figure 3.19: Constriants from combination

behavioral properties. It models LTS and properties as state machines, and then performs compositional reachability analysis between them to exhaustively search for violations of the desired properties. In this section, we use LTSA to perform the written LTL property expressions for property verification on the established system LTS model, so as to ensure that the model we built satisfies the requirements of the system. Table 3.1 presents the statistics for the properties that are automatically proved in the model.

Table 3.2 gives the informal description of some properties that need to be verified, and their formalization is given in Table 3.3. The symbols "□" and "♢" in Table 3 represent "Always" and "Final" in LTL, respectively,

Table 3.1: Number of properties that are automatically proved

Refinement Layer	Safety Properties	Liveness Properties
L0	1	1
L1	3	6
L2	12	17
L3	18	29
L4	29	44

while the symbols "$\neg$" and "$\rightarrow$" represent "negative" and "implication" in propositional logic, respectively.

We use LTSA to combine these property expressions with the constructed system LTS model to verify the behavior of the system. If there is a behavior in the model that violates these properties, LTSA will give the sequence of events corresponding to the behavior. On the contrary, it means that the model satisfies these properties.

Table 3.2: Informal description of properties to be verified in the system model

The serial number of requirements	Informal description of requirements
SAF_1	Two or more processes cannot write into the same buffer at the same time
SAF_2	Two or more partitions cannot be scheduled at the same time
$LIVE\_3$	As long as the process is waiting for resources, it will eventually get resources or timeout
$LIVE\_4$	As long as the process requests the port, it will eventually get the port or timeout
$LIVE\_5$	As long as the start process event occurs, the stop event of the periodic process or aperiodic process will eventually occur
$LIVE\_6$	As long as the *time_wait* event occurs, the *time_out* event will occur

After verification, we do not get any information that violates the properties. In other words, as far as the current situation is concerned, the model we have established is no error, but it does not mean that it is correct. However, what we can confirm that as long as we write more property expressions and no errors are reported after verification, the reliability of the system model will be higher.

Table 3.3: The formal description of properties to be verified in the system model

The serial number of requirements	Formal description of requirements
SAF_1	$\Box(\neg(Process[1].send\_buffer \wedge Process[2].send\_buffer))$
SAF_2	$\Box(\neg(Partition[1].schedule \wedge Partition[2].schedule))$
$LIVE\_3$	$\Box(Process[i].req\_busy\_resource \rightarrow$ $\Diamond(Process[i].receive\_buffer\_available \vee$ $Process[i].time\_out))$
$LIVE\_4$	$\Box(Process[i].send\_buffer\_withfull \rightarrow$ $\Diamond(Process[i].receive\_buffer\_needwakeupsendproc \vee$ $Process[i].time\_out))$
$LIVE\_5$	$\Box(Process[i].start \rightarrow$ $\Diamond(Process[i].periodicprocess\_finished \vee$ $Process[i].aperiodicprocess\_finished))$
$LIVE\_6$	$\Box(Process[i].timed\_wait \rightarrow \Diamond Process[i].time\_out)$

3.6 Results and Discussion

In this section, we compare our work with the work of Zhao et al., who used Event-B alone to establish the system model of the ARINC653 specification in Section 3.5.2 [11]. The simulation results on the Rodin platform show that in each layer, the event traces of the model obtained using our method are consistent with theirs. In their work, all Event-B code is hand-written by the research team, which involves a large number of *variables*, *invariants*, *guards*, *actions* and other elements. Such a huge workload requires a lot of time and energy, and text-based programming also makes the code extremely prone to errors.

The growth trend of these elements is shown in Figure 3.20. From the perspective of statistical data, we can find a phenomenon: as the layer of refinement increases, the number of codes grows extremely fast. The total number of *variables*, *invariants*, *guards* and *actions* is 1374 when refinement reaches the 4th layer. Moreover, it can be seen that the growth rate of *guards* is the fastest, followed by *actions*. For more complex multi-level control systems, this phenomenon will be more obvious, which means that the slope of the two curves will be greater.

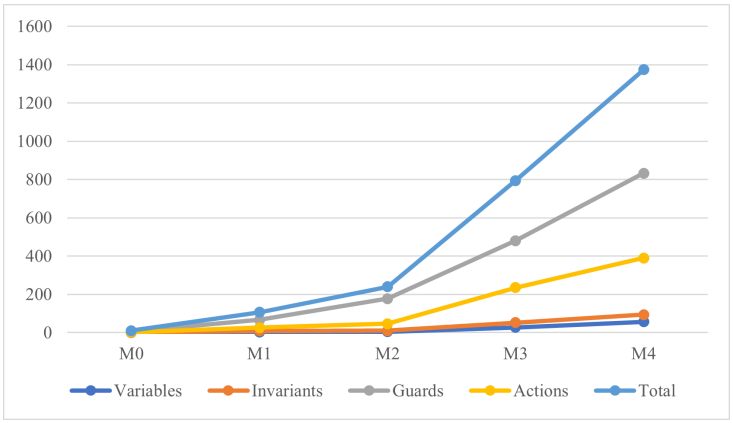


Figure 3.20: Statistics in the Event-B model of the ARINC653 speicification

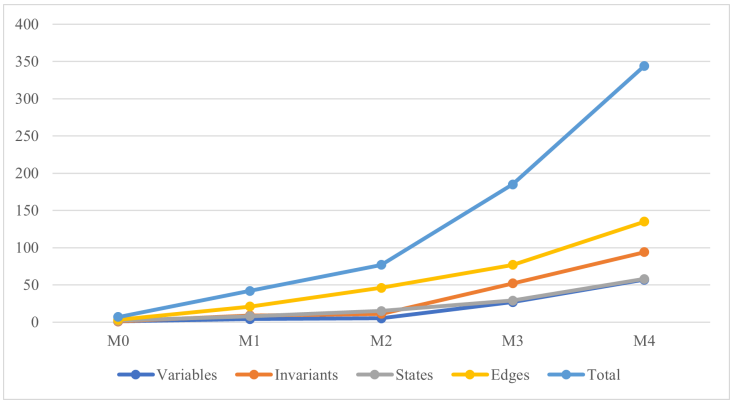


Figure 3.21: Statistics in the iUML-B model of the ARINC653 speicification

In our work, we first use the states and edges linked with events in the iUML-B to build system models, and use the pattern state machine proposed in our previous work to improve reuse rate and programming efficiency [20], and then use Rodin tools to automatically generate the Event-B model. The growth trend of the main elements in the iUML-B model with the increase of the refinement layer is shown in Figure 3.21. Since one edge can link multiple events, the number of edges is greatly reduced, and modeling becomes easier [12]. Although the variables and invariants in the code still need to be written manually, the guards and actions with the highest proportion can be automatically generated. In other words, the percentage of manual coding we can save is roughly $(833 + 390)/1374 = 89\%$.

Another benefit of our method is that the control flow and behavioral interaction become visible. Compared with Event-B's event-based textual representation, iUML-B state machine can express and analyze the event order of the system easily.

Finally, we pointed out in [13] that we choose LTS as the behavioral semantic model of Event-B, and convert the Event-B model to the corresponding LTS model, so that the behavioral properties of the Event-B model can be analyzed and verified. From the point of view of behavioral semantic verification of event-based method, this is an advantage.

Chapter 4

Program Segment Testing for Software Reliability

4.1 Overview of Program Segment Testing

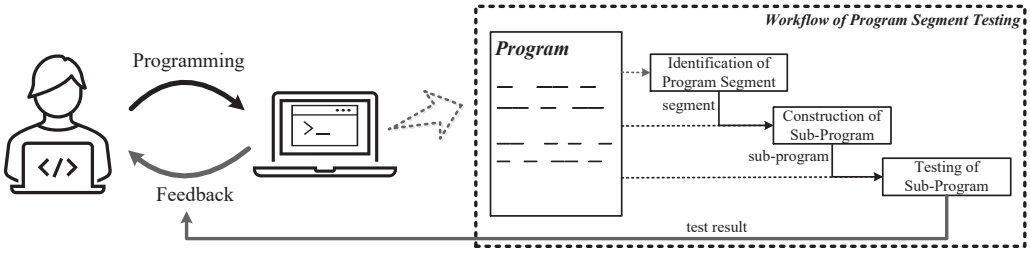


Figure 4.1: Workflow of Program Segment Testing

The PST consists of three steps, as shown in Figure 4.1. During the programming process, the developer works on the machine while the machine automatically monitors the code to identify program segments with the potential to cause corresponding runtime exceptions. The program slicing technique is then employed to construct a sub-program that creates a suitable testing environment for the identified segments. Subsequently, the testing phase takes place to determine whether the program segment will cause a runtime exception. The test results are promptly provided as feedback to the programmer to help them correct the bug, if any. This entire process is carried out by the machine in the background, ensuring it does not

interfere with the ongoing activities of human developers involved in software construction.

As bugs are discovered and eliminated, the subsequent bugs are continuously exposed, indicating an iterative process in the utilization of this technique. This iterative approach is dedicated to developing programs that ultimately do not have bugs leading to runtime exceptions.

Let us provide a detailed definition of the relevant concepts involved in PST and the Algorithm 1 that implements it.

Definition 1 (Segment). Let P be the program under construction, S_i

Algorithm 1 Implementation of Program Segment Testing

Input: Program P
Output: Test Result

$P \leftarrow program$
$SEG \leftarrow empty\ set$
$VAR \leftarrow empty\ set$
$SUB \leftarrow empty\ set$
while P not ended **do**
  $SEG \leftarrow ANTLR\_Detect(P)$
  for each seg in SEG **do**
    $VAR_{seg} \leftarrow Extract\_Variables(seg)$
    $SUB_{seg} \leftarrow empty\ set$
    for each v in VAR_{seg} **do**
      $SUB_{seg} \leftarrow SUB_{seg} \cup Program\_Slice(P, v)$
    end for
    if $Has\_Input(SUB_{seg})$ **then**
      $Input\_Vars_{seg} \leftarrow Identify\_Vars(SUB\_seg)$
      $T \leftarrow Fuzz\_Test\_Generation(Input\_Vars_{seg})$
      for each t in T **do**
        $Result \leftarrow Analyze\_Test(t)$
        if Result **then**
          $Report\_Error(t, Result)$
        end if
      end for
    end if
  end for
end while

the i-th statement in P. Then SEG_i is a segment of continuous statements in the program that starts with the i-th statement:

$$SEG_i = \{S_i, S_{i+1}, \ldots, S_{i+n_i-1}\}$$

where n_i is the number of statements encapsulated within this segment.

Our practical observations have led us to recognizing that a segment typically encompasses a single statement. Nonetheless, in scenarios where adjacent lines of code contain operations that could possibly trigger the same type of exception, we opt for efficiency by grouping these lines together as a single segment. We will provide a more detailed introduction to this approach in Section 4.2.2. Once the target segment is identified, the next step is to create a suitable environment for testing these segments.

The initiation of exceptions within a segment frequently stems from incorrect assignments to specific variables, which we identify as our slicing criteria of interest. Sole reliance on the program slice that pertains to a particular variable to assess the likelihood of an exception being triggered proves to be insufficient. This insufficiency arises because other variables in the code segment might influence the value of the target variable, potentially precipitating an exception. For example, the array access operation *arr[i]* could lead to an exception if the index variable i is improperly assigned. However, the significance of i is not fully appreciated unless it is evaluated in conjunction with the array variable *arr*. In isolation, slicing the program based solely on the variable i does not provide the comprehensive insight required for our analysis.

To address this issue, we undertake program slicing for each variable within the code segment. We then amalgamate the statements from these slices, eliminating any duplicates and reordering them by their line numbers to construct a sub-program tailored specifically for testing. This sub-program encapsulates the collection of interactions and dependencies critical for understanding the behavior of the code segment under test. A detailed proof of the feasibility of such an operation will be given in Section 4.2.2. Subsequently, we will provide a formalized definition of the sub-program derived from a code segment.

Definition 2 (subProgram). Let P be the program under construction, S_i the i-th statement in P, $V_{s_i}^j$ the j-th variable in S_i, SEG_i is a segment of continuous statements in the program that starts with the i-th statement, SL_i^j is a slice obtained by performing program slicing on P with $\langle S_i, V_{s_i}^j \rangle$ as

the slicing criterion. Then, a sub-program for SEG_i is

$$subProgram_i = \bigcup_{i}^{i+n_i-1} SL_i^j,\ where\ j = 1, 2, \ldots, num_i$$

where num_i denotes the number of variables in S_i, n_i represents the number of lines of statement in SEG_i.

To generate specific test cases for the subprogram designated for segment SEG_i, we employ fuzz testing. This method uses a fuzzer to automatically generate a large volume of random or semi-random input data for the input variables. These test cases are designed to exercise different scenarios and edge cases within $subProgram_i$. Mathematically, we can describe this process as $T = Fuzz\_Test\_Generation(Input\_Vars_{seg})$.

To observe if an exception occurs during the execution of the generated test cases T, we define an exception function $Analyze\_Test$: $T \rightarrow \{true, false\}$ that maps the executed test case to the occurrence of an exception. $Analyze\_Test$ takes a test case $t \in T$ and returns true if an exception occurs, and false otherwise.

From the analysis above, it is evident that to implement PST effectively, we must address the following three key challenges:

- RQ1: How to identify program segments that have the potential to cause runtime exceptions?
- RQ2: How to construct sub-programs to provide a suitable testing environment for these segments identified?
- RQ3: How to perform the testing to determine whether these segments will trigger runtime exceptions?

Each runtime exception can be located using different methods to identify the corresponding suspicious code segments. Additionally, generating test cases requires various techniques. Therefore, we will use index out-of-bounds exceptions and arithmetic exceptions as representative examples to illustrate the application of PST in handling runtime exceptions and addressing these research issues.

4.2 PST for Arithmetic Exception

4.2.1 Preliminary

Program Slicing

Program slicing is a powerful technique in software engineering, originally introduced by Mark Weiser in 1981, that focuses on extracting relevant subsets of a program for specific computations or debugging tasks [114]. This method simplifies the process of analyzing software by creating a System Dependency Graph (SDG) which illustrates the flow of data and control between various program statements [115]. By analyzing this graph, program slicing identifies subsets of statements and variables that influence a particular computation or behavior—referred to as the slicing criterion, which could be a variable, event, condition, or any program element.

The resultant subset, known as a program slice, encompasses all statements that directly or indirectly contribute to the value of a specified variable or influence a specific execution point in the program. This creates a concise view of the program's behavior, reducing the code size and complexity needed for analysis or debugging purposes.

Program slicing is highly useful in several key areas of software development. During program comprehension, it allows developers to focus on specific functionalities or understand the impact of changes on a particular part of the program [116]. In debugging, slicing helps isolate the code segments that are likely connected to a bug, thus enabling developers to focus on a smaller portion of the code to efficiently locate and resolve issues [117]. Additionally, in the context of software maintenance, slicing can demonstrate how changes in one section might affect other parts of the program, helping to prevent unintended consequences [118]. It is also beneficial in test case generation, where identifying relevant code segments for specific testing scenarios can significantly streamline the testing process [119].

Let us use a short program to demonstrate program slicing. In static slicing, the slicing criterion has the form of $\langle i, v \rangle$ where i is the serial number of a statement in the program and v the variable set. We use $\langle 10, product \rangle$ to perform static backward slices on the program on the left in Fig. 4.2, and the slice on the right will be obtained, in which statements irrelevant to *product* are excluded.

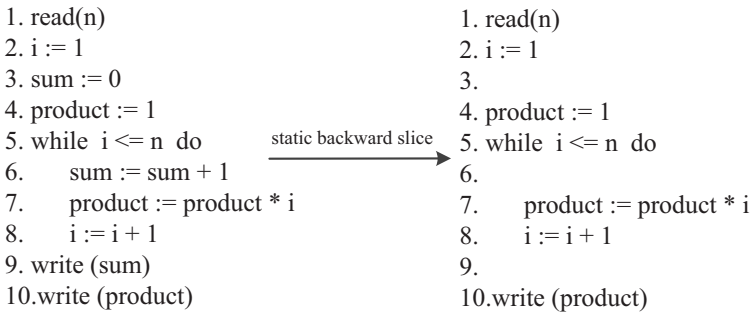


Figure 4.2: An example program for static backward slicing

Fuzz Testing

Fuzz testing, also known as fuzzing, was introduced by Barton Miller at the University of Wisconsin in 1988 during his course experiments [120]. The core concept of fuzz testing involves inputting automatically or semi-automatically generated random data into a program and monitoring for exceptions such as crashes, assertion failures, to identify potential programming errors like memory leaks.

Modern fuzz testing follows a structured procedure that begins with selecting a corpus of "seed" inputs to test the target program [121]. The fuzzer repeatedly mutates these inputs and assesses the program's behavior. If an input produces "interesting" behavior, such as a crash or uncovering a new execution path, the fuzzer preserves this input for future tests and documents the findings. Fuzzing ends either upon achieving a specific goal, like identifying a particular bug, or after a predefined timeout.

Different fuzzers vary in their observational methods when executing programs. In "black box" fuzzing, the only observation made is whether the program crashes. "Grey box" fuzzing captures intermediate information about the execution, such as the sequence of basic block identifiers, which helps trace execution paths. "White box" fuzzing delves deeper by analyzing the source or binary code of the application, enabling more sophisticated observations and adjustments based on the semantics of the code. This deeper analysis adds overhead but aims to enhance the effectiveness in bug detection.

The ultimate goal of a fuzzer is generally to generate an input that causes the program to crash. Depending on the configuration, a fuzzer might terminate upon detecting a crash or continue to explore for additional crashes.

For example, libfuzzer [122] typically stops when a crash is found, whereas AFL continues in an attempt to identify multiple crashes[123]. Observations of prolonged execution times may also indicate potential vulnerabilities due to algorithmic complexity [124]. The results from fuzzing, including specific inputs and configurations, allow software developers to confirm, reproduce, and debug the identified issues.

4.2.2 Case Study

Example Program for Explanation

Listing 4.1: An example program with arithmetic exceptions

```
main () {
    Scanner scanner = new Scanner(System.in);
    int p = 12;
    int q = scanner.nextInt();
    int r = 2;
    System.out.println(p/r);
    int result = testme (p, q, r);
    System.out.println(result);
}
int testme (int p, int q, int r){
    int s = twice(q);
    int t = p/(s-r);
    int result = p/(t-r);
    return result;
}
int twice(int s){
    return s*2;
}
```

Our method is effectively integrated into the construction process, though it currently lacks automated tool support, necessitating manual implementation. Before detailing the workflow of the PST, let us consider a case study program to illustrate its application. Imagine a program that defines three integer variables: p, q, and r. The variables p and r are initialized to 12 and 2, respectively, while the value of q is provided by the user at

runtime. Additionally, a function named *twice* is defined to return double the value of its input parameter. Another function, *testme*, which performs a series of operations on its input parameters and returns a result, also needs to be defined. The program requirements include outputting the result of p divided by r, as well as the result of passing p, q, and r as arguments to the *testme* function. We assume the program is developed sequentially from top to bottom as illustrated in Listing 4.1, with particular attention to the automated handling of parentheses in development environments such as IntelliJ IDEA[125] and Eclipse[126]. For example, when a developer types an opening parenthesis, these tools automatically insert the corresponding closing parenthesis, allowing the developer to continue coding within them. For example, the parenthesis on line 9 is typically completed in conjunction with the completion of line 1. Thus, for the purposes of this discussion, we presume the program does not contain syntax errors related to missing parentheses.

Determination of Suspicious Segments for RQ1

The first step involves using ANTLR (Another Tool for Language Recognition) to identify the target program segments in which potential exceptions may occur during execution. ANTLR is a powerful tool designed to process and parse programming languages, construct lexical analyzers and parsers, and generate Abstract Syntax Trees (ASTs). This process is further detailed in several steps as illustrated in the accompanying Figure 4.3.

The initial step involves defining grammar rules, which includes creating a ".g4" file containing lexical rules (for identifying identifiers, keywords, literals, etc.) and syntax rules (defining the grammatical structure of the language). This file establishes the rules for converting source code text into an AST. Using the ANTLR tool, lexer and parser code for Java is then generated based on the ".g4" grammar file. This results in a series of Java classes capable of processing the source code and constructing its AST. Following this, the generated lexer and parser are utilized to parse the target source code file. This parsing process reads the source code, conducts lexical and syntactic analysis according to the rules defined in the ".g4" file, and ultimately produces the corresponding AST. An AST is a tree-like data structure representing the hierarchical structure of the source code, where each node reflects a construct in the code, such as declarations and expressions. The final step involves writing code to traverse the AST, employing the Visitor

Pattern—a design pattern that allows operations to be performed on elements of an object structure without changing the classes of the elements. We extend the visitor class provided by the ANTLR-generated parser to implement our custom traversal logic. During traversal, this logic is employed to identify code constructs that could trigger runtime exceptions, such as array access operations and division operations.

In the programming process, once a code fragment that may give rise to an exception is identified, the statement containing that fragment is determined and labeled as a suspicious segment. Note that the term "suspicious" is used because while there is a possibility of an exception being thrown, the authenticity of this possibility needs to be confirmed through subsequent testing. However, the process does not end there; we continue to monitor the program's development. If the subsequent statement also contains a fragment that could potentially trigger the same exception, it is merged into the previously established suspicious segment. This merging continues until a statement is encountered that no longer contains such a fragment. This approach is based on our experience that consecutive lines of code containing fragments potentially triggering the same exceptions are usually closely related. For example, lines 12 and 13 in Listing 4.1, and common scenarios in various sorting algorithms where multiple lines of code might trigger array index out-of-bounds exceptions during element swapping. The purpose of

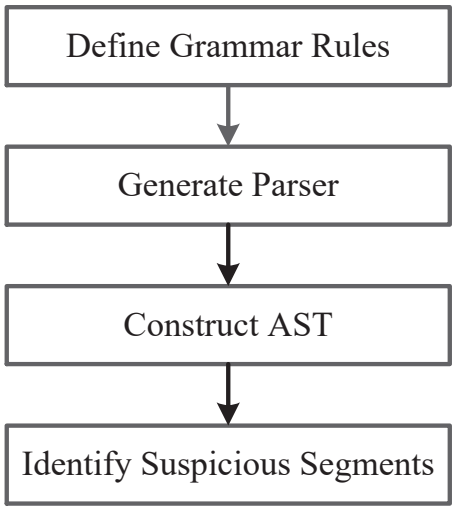


Figure 4.3: Process of determination of suspicious segments

this merging analysis operation is to enhance efficiency. For instance, testing line 13 independently would inevitably involve the code in line 12.

In the development process of the program depicted in Listing 4.1, lines 6, 12, and 13 were identified as suspicious segments because they each contained division operations that could potentially lead to arithmetic exceptions. Moreover, according to Definition 1 regarding the segment, they are recognized respectively as seg_6, seg_{12}.

Construction of Sub-program for RQ2

As previously discussed, our objective in this phase centers around employing program slicing techniques to create a sub-program that serves as a testing environment for each identified suspicious segment. This step can be further broken down into detailed processes as illustrated in Figure 4.4.

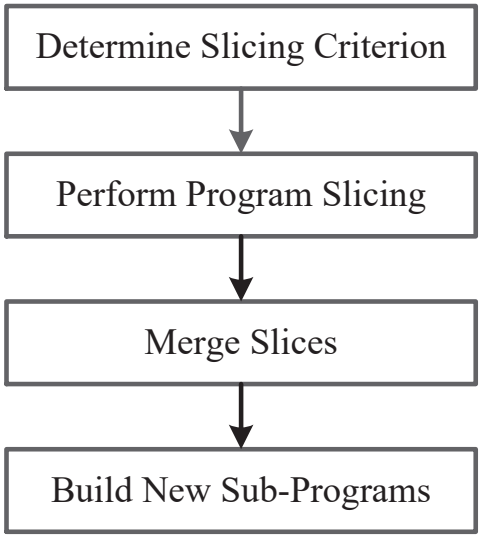


Figure 4.4: Process of construction of sub-program

Initially, the target variables and statements for slicing need to be identified. In this context, the target statements are those that might trigger arithmetic exceptions, such as line 12 in Listing 4.1. The target variables are those whose inappropriate values could lead to the triggering of exceptions, such as the variables s and r in divisor. However, since the ultimate goal is to execute a sub-program for testing, other variables in that line of code, such

as the denominator variable p and the division result variable t, which do not have a direct causal relationship with the exception, must also be included to ensure the executability of the sub-program. This means that the target variables for a suspicious segment include all variables within that segment. The next step involves performing program slicing on the original program using these slicing criteria to obtain the program slices. In this paper, we utilize JavaSlicer—a Java program slicer based on SDG—to accomplish this task [127]. After obtaining program slices for different variables, the next task is to consolidate these slices. In the consolidated slices, there may be many repetitive statements, especially those sharing the same dependencies. It is necessary to identify and remove these duplicate statements, keeping only unique instances to form a new, smaller sub-program. This consolidation process also poses some challenges: whether the sub-program is executable and maintains the original program's behavior. Next, we provide a mathematical proof to address these issues.

To validate that the merged slices result in an executable sub-program, it is crucial to define a few key concepts and build a logical framework for proof. Here are the definitions and an outline of the proof:

Key Definitions

- System Dependence Graph(SDG): An SDG is a directed graph $G = (V, E)$, where each node $v \in V$ represents a statement in the program, and each edge $e \in E$ represents a dependency within the program, including data dependencies and control dependencies.

- Program Slice: Given an SDG $G = (V, E)$ and a slicing criterion (typically a pair $(v, Vars)$, where v is a node in the SDG and $Vars$ is a set of variables), a program slice is a subgraph $G^{'} = (V^{'}, E^{'})$ of G. $V^{'}$ includes all nodes that are relevant to the slicing criterion, i.e., all nodes that can be reached by tracing back through E starting from v, along with all their interdependencies $E^{'}$.

- Executability: A sub-program is executable if it can run independently of the original program without causing syntax or runtime errors.

Proof Outline

To prove that the merged slices are executable, we need to demonstrate the following:

- Preservation of All Necessary Dependencies: This means that for any statement or expression in the slice, all its data dependencies and control dependencies are still satisfied in the merged slice.
- No Introduction of Syntax or Runtime Errors: This means that the merging process does not lead to issues such as variable scope errors, type mismatches, and maintains the program's control flow structure.

Mathmetical Proof Framework

- Preservation of Dependencies:
  - Suppose there are two program slices $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$, each corresponding to variables V_1 and V_2 related to the same exception trigger.
  - The merged slice $G^{'} = (V^{'}, E^{'})$ is the union of G_1 and G_2, where $V^{'} = V_1 \cup V_2$ and $E^{'} = E_1 \cup E_2$.
  - Proof: For any node v in G_1 and G_2, its dependencies are preserved in $G^{'}$ since $E^{'}$ includes all dependencies from E_1 and E_2. Thus, any dependency from nodes in V_1 or V_2 is retained in $G^{'}$.
- No Introduction of Syntax or Runtime Errors:
  - Proof: Since each slice G_1 and G_2 is extracted from the same SDG G, they adhere to the syntax and runtime rules defined in G. The merging operation merely combines these slices without altering any internal logic of nodes or introducing new nodes, thereby not introducing new syntax or runtime errors.

Conclusion

Based on the proof framework outlined, it can be concluded that the merged sub-program is executable. This ensures that the program can run without new errors and maintains all necessary dependencies, validating the effectiveness of our slicing and merging strategy in maintaining program integrity while isolating critical segments for error analysis.

Continuing with the example from Listing 4.1, we illustrate the steps involved in this phase. When we type out line 6 in the program, responsible for outputting the result of variable r divided by p, it is identified as a suspicious statement, labeled as seg_6. Subsequently, line 7 does not reveal any

fragments that could trigger an arithmetic exception, thus seg_6 includes only line 6. Through analysis, it is found that seg_6 includes only variables r and p, setting the slicing criteria as $\langle 6, r\rangle$ and $\langle 6, p\rangle$. Using these slicing criteria, program slices for variables r and p are derived separately. Once the slices for these two variables are obtained, the next step is to merge them and eliminate any redundant statements, resulting in a new, executable sub-program for seg_6. This resulting sub-program, which is devoid of redundancies and organized in a sequential manner, is illustrated in Listing 4.2. Since variable q does not influence the values of p and r, the original lines 2 and 4, which read the user input for q, are excluded from the sub-program.

Listing 4.2: The slice obtained from the segment seg_6

```
1main () {
2    int p = 12;
3    int r = 2;
4    System.out.println(p/r);
5}
```

The process outlined above clearly illustrates how a sub-program is generated from a suspicious segment that has been identified. Continuing with this approach, when programming reaches seg_{12}, unlike seg_6, it includes both lines 12 and 13, which involve five variables: t, p, s, r, and $result$. When the slicing criterion $\langle 12, s\rangle$ is applied to the original program to perform program slicing, it is discovered that the value of variable s is the return value of the function *twice*. However, at this stage, the *twice* function has not yet been written, meaning the sub-program generated is not executable. Therefore, seg_{12} must be temporarily suspended until the function is implemented, after which the process can be restarted.

This situation introduces a challenge: how to determine when the necessary function has been completed in practice. Our current approach involves monitoring the programmer's ongoing coding activity. When the cursor moves into the function we are concerned with, it indicates that the function is being written. If the cursor moves out of the function for some time or begins coding other parts, it may suggest that the function is complete, at which point we can restart testing the suspicious segment. Of course, it is also possible that the function may still be incomplete, so to prevent infinite loops, we set a limit on the number of retests. Once this limit is reached, further attempts will cease.

As mentioned earlier, since PST is performed by machines in the background, it does not interfere with our current programming activities. When programming reaches line 18, after the completion of the *twice* function, seg_{12} will be retested for the first time. After merging all slices obtained for the five variables, the resulting sub-program lacks lines 6 and 8 compared to the original program.

The comparison between the sub-program and the original program demonstrates that this method can significantly reduce the scale of the test program, thus enhancing testing efficiency. This is particularly evident in large-scale programs, as shown by the systematic experiments discussed in Section 4.2.3.

Implementation of Testing for RQ3

In the process of testing a new sub-program, the initial action is to determine the presence of inputs. If no inputs exist, the program can be executed directly to assess its behavior. Conversely, if the program does accept input, the next crucial step involves identifying the input variables involved in the program's operation, as shown in Figure 4.5.

Once the input variables are recognized, the focus shifts to the generation of test cases, employing fuzz testing techniques. As discussed in Section 4.2.1, Fuzz testing is an automated technique that involves providing data as inputs to the program. For our purposes, we have chosen AFL++ as our fuzzer. The primary goal of fuzz testing in this context is to uncover potential exceptions or unexpected behavior, such as arithmetic errors when the divisor could be zero.

After generating the test cases, the actual testing is conducted. This stage involves running the program with all the generated test cases and closely monitoring the software's behavior for any runtime exceptions. Analyzing the test results is crucial; if any of the test cases cause an exception to be thrown, it indicates the presence of bugs. Identifying and recording the specific conditions and input values that trigger exceptions is vital for understanding the reasons behind these exceptions and guiding future code repair recommendations.

If the developer needs to modify the code, such analysis helps pinpoint the exact issues needing attention. Conversely, if no bugs are found and the program operates as expected across all test scenarios, no feedback is necessary. This approach ensures that feedback provided to programmers is constructive, specifically enhancing the robustness of the program under

various input conditions, thereby contributing to the development of more reliable and fault-resistant software.

The sub-program derived from the segment seg_6, as depicted in Listing 4.2, does not require any user input and executes without throwing any arithmetic exceptions, suggesting that this segment is secure within the context of this program. Thus, no further testing or feedback to the developer is necessary for this segment.

Conversely, the sub-program from the segment seg_{12}, which does involve user input for the variable q, highlights a critical aspect of testing. In this sub-program, the only input variable is q, which becomes the focus of our test cases. For instance, when q is set to 1, the value of s becomes 2, leading to a division by zero in line 12, thus triggering an arithmetic exception. Similarly, when q is set to 4, the value of s becomes 8, and the value of t becomes 2, resulting in another division by zero in line 13 and again triggering an arithmetic exception. This indicates that a single segment can have multiple test values that can lead to exceptions, reinforcing our choice of AFL++ as our fuzzer, as it is designed to exhaustively identify all test values that could cause the program to crash.

Following these findings, the system prompts the programmer to modify the program to prevent q from taking values that could lead to runtime crashes, such as 1 and 4. However, it's important to note the inherent limitations of fuzz testing: while it is effective in randomly generating test values, it may not always produce specific values like 1 and 4 within the constraints

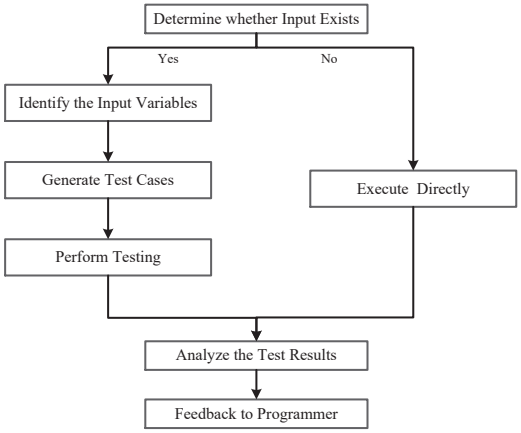


Figure 4.5: Process of implementation of testing

of time and resources available. This limitation means that sometimes, fuzz testing might not detect all potential exceptions, thus not guaranteeing the absence of defects that could trigger exceptions in the program.

4.2.3 Experiment

In this section, we acknowledge that the tools required to fully support the PST are still under development. Consequently, a comprehensive performance evaluation is not feasible at this time. Instead, we will manually assess the effectiveness of the PST through a series of code evaluations. We will also describe our experiment setup, present our findings, and analyze the results.

Experiment Design

We conduct the experiments manually with the help of four doctoral students. For simplicity in this discussion, they will be referred to as Student 1, Student 2, Student 3, and Student 4. Student 1 is responsible for analyzing our experiment program to determine the actual number of effective code lines in each program used for the experiment. Student 2 is tasked with using our predefined logic to identify suspicious segments within the programs. Student 3 takes charge of using program slicing to construct a sub-program that serves as a testing environment for each identified suspicious segment. Student 4 is responsible for generating the corresponding test cases, executing the tests, and recording the test results. To mitigate potential biases in the evaluation of experiment results, we ensure that the outcomes are always reviewed under the supervision of an independent party. Specifically, we choose an 'academic peer'—a colleague from a different academic department who has no involvement in our project. This step is crucial to maintain objectivity and ensure that the results are interpreted accurately and impartially. To lend credibility to our experiment, we chose several programs from the Software Artifact Infrastructure Repository for analysis [128]. These programs are considered "real" in the sense that they are non-trivial applications developed by experienced software programmers to address actual problems, rather than being crafted specifically for research or educational purposes. This choice was made because these programs are commonly used in fault localization and software testing studies, emphasizing their relevance and importance in our study [129, 130, 131, 132].

Table 4.1: Overview of the experiment results

Program Name	*LOC_OP*	Suspicious Segment	*LOC_SP*	*IV_OP*	*IV_SP*	Expected Result	Actual Result
AccountSubType	124	68	69	2	2	×	×
AllocationVector	127	77	36	4	1	×	×
BinaryHeap	76	189	63	1	1	✓	✓
		195	70		1	✓	✓
BinarySearchTree	209	261	192	2	2	✓	✓
BoundedBuffer	88	76	25	5	5	×	×
		86	29		5	×	×
		136	39		5	✓	×
CruiseControl	262	309	65	0	0	×	×
		313	52		0	×	×
		317	48		0	×	×
		319	59		0	×	×
Disjoint	77	130	42	2	2	✓	✓
LinkedList	176	16	13	2	2	✓	✓
OrdSet	230	56	17	5	1	✓	✓
Piper	85	28	27	4	4	×	×
		35	32		4	×	×
		48	34		4	×	×
ProducerConsumer	134	75	32	4	4	×	×
		104	32		4	×	×
RaxExtended	160	22	38	2	1	×	×

Result Analysis

The findings from our experiment, summarized in Table 4.1, offer a detailed view of the different programs we analyzed, and the results obtained through our application of the PST. LOC_OP represents the number of effective code lines in each original program, excluding blank and comment lines, as meticulously identified by Student 1. As identified by Student 2, *Suspicious Segment* refers to segments within the program that could potentially trigger arithmetic exceptions. For clarity in the table, these segments are represented by the subscript. LOC_SP, also determined by Student 1, represents the number of effective code lines in each sub-program constructed for the suspicious segments.

The box plot depicted in Figure 4.6 encapsulates the ratio of LOC in sub-programs relative to their respective original programs, providing a statistical summary of the distribution of efficiency gains from slicing. The median ratio, marked at approximately 0.49, indicates that the central tendency of LOC reduction is just under half of the original program size. This is supported by the mean, signified by the 'X', which is slightly above the median, suggesting a distribution that's mildly right-skewed—there are a few instances where slices retain a larger proportion of the original LOC. The interquartile range, representing the middle 50% of the data, is quite narrow,

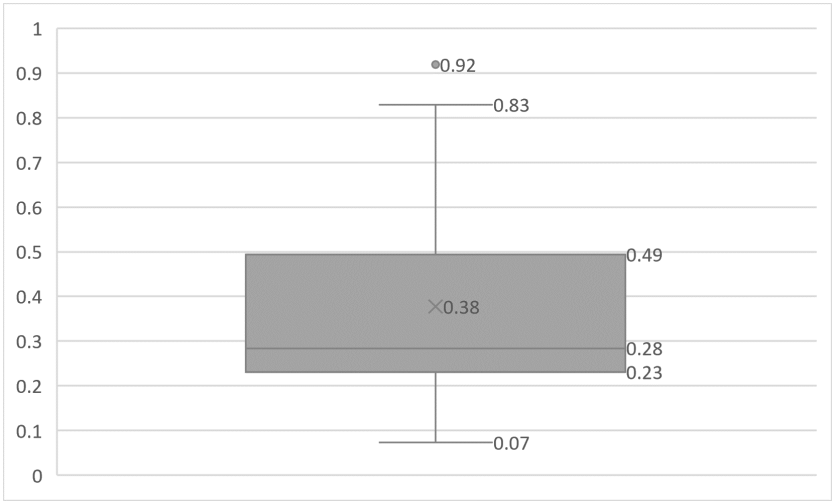


Figure 4.6: The ratio of LOC in sub-programs relative to their respective original programs

between roughly 0.23 and 0.28, indicating consistency in slicing efficiency across various programs; most slices tend to be about a quarter of the size of the full program. The outliers at 0.83 and 0.92 show exceptional cases where the slices have a higher LOC ratio, possibly indicating complex dependencies that result in less reduction. The extent of the upper whisker up to 0.92 and the lower whisker down to 0.07 demonstrates the variability in the efficiency of program slicing. A whisker reaching down to 0.07 denotes that in some cases, slices can be extremely efficient, reducing the original program's LOC to just 7%. The absence of outliers below the lower whisker further emphasizes the consistency of achieving significant LOC reduction through slicing. Overall, the box plot suggests that our method is generally effective at reducing the size of code to a significant extent.

In addition to the reduction in size of the sub-programs compared to their original programs, our analysis also revealed that the number of input variables contained in the programs sometimes decreased. This is indicated by two metrics: IV_OP, which represents the number of input variables in each original program, and IV_SP, which denotes the number of input variables in each sub-program. For instance, in the case of the *OrdSet* program, the number of input variables was reduced from five in the original program to just one in the sub-program. This reduction occurred because the target suspicious segment did not involve the use of the other variables. Therefore, these unrelated input variables were excluded during the construction of the sub-program. The reduction in the number of input variables also implies a geometric decrease in the number of test cases required, which significantly reduces the time needed for testing. This efficiency gain from size and variable reduction not only streamlines the testing process but also enhances the focus on the critical parts of the code that are more likely to affect the program's behavior, allowing for more targeted and effective testing.

Expected Result refers to whether the suspicious segment is expected to actually trigger an arithmetic exception, as determined by the lead researcher. A mark of "×" indicates that the segment will not trigger an exception in the current context of the program, whereas a "✓" suggests the opposite. Normally, exceptions may be triggered by incorrect values of certain variables. However, there are special cases where the context of the program prevents an exception from being triggered. For example, consider the synchronized *put* method in the *ProducerConsumer* program as displayed in Listing 4.3, which is used to place objects into a buffer. The sixth line of code updates the *in* variable, indicating the next position to place an object. Theoreti-

cally, an arithmetic exception could be triggered when the buffer size *size* is zero. However, if *size* is zero, it means there is no space in the buffer, no interaction between the producer and consumer occurs, and the count of objects remains zero. As the *while* loop in the second line indicates, the thread will be suspended indefinitely, and the sixth line of code will never execute, hence the exception will never be triggered.

Listing 4.3: The synchronized put method in the ProducerConsumer program

```
1public synchronized void put(Object o) {
2    while (count==size)
3        wait();
4    buf[in] = o;
5    ++count;
6    in=(in+1) % size;
7    notify();
8}
```

Actual Result represents the outcome obtained from testing the subprogram with our method, performed by Student 4. As previously mentioned, the programs used in our experiments are non-trivial, created by experienced software programmers to address real-world problems, and typically involve relatively small input values. To enhance efficiency, we generally limit the range of generated test cases to between -100 and 100. Additionally, due to the resource constraints, we set the testing period to 24 hours. Of course, testers can adjust the timing flexibly based on actual requirements. After comparing with the *Expected Result*, we found, for instance, that in the *BoundedBuffer*, Seg_{136} was expected to trigger an exception, but the actual test results showed it did not. This was due to the inability to generate test cases within the limited time that could trigger the exception. Yet, we believe that with sufficient time to generate enough test cases, we would eventually produce cases that cause the expected exception.

Although our testing method shows some limitations under conditions of constrained resources and time, it generally succeeds in accurately predicting program behavior in most cases. This indicates that our testing strategy is fundamentally effective. While it is only testing and cannot provide the exact certainty of model checking or formal verification, it suffices for most practical scenarios.

4.3 PST for index out of bounds exceptions

4.3.1 Preliminary

Index out of Bounds Exceptions in Java

An index out of bounds exception is an error that occurs when trying to access an element with an index that is outside the bounds of a defined array, collection, or string. When this happens, the system may return a random value, leading to unpredictable results. In addition, out of bounds errors often result in buffer overflows. The empirical study in [133] indicates that approximately 34% of buffer overflows are caused by index out of bounds errors. Unfortunately, almost all programming languages do not perform bounds checking to improve program execution speed, which can lead to crashes at runtime. The Common Weakness Enumeration (CWE) TOP 25 is a list of the most dangerous software weaknesses that provide attackers with a way to take over a system, steal data, or prevent applications from working [134]. Out of bounds are the most common weakness on this list. Therefore, it is crucial to identify the bugs that have the potential to cause them in order to prevent them from arising at runtime.

Index out of bounds exceptions in Java can be roughly classified into three categories, including array index out of bounds exceptions, collection index out of bounds exceptions, and string index out of bounds exceptions. These exceptions are primarily instigated by unauthorized access attempts on elements with arrays, collections, and strings.

ArrayIndexOutOfBoundsException. This occurs when attempting to access to an element of an array with an index that is outside its permissible range. Java arrays adhere to a zero-based indexing convention, where the valid index range for an array named *arr* is from 0 to *arr.length* -1. Access is typically performed using the syntax *arr[i]*, where i is the index. If i is set to a value less than 0 or greater than or equal to *arr.length*, an *ArrayIndexOutOfBoundsException* is triggered.

CollectionIndexOutOfBoundsException. In Java's collection framework, the Collection interface is the root interface with *List* and *Set* as its primary sub-interfaces. These interfaces provide specific methods to manipulate collection elements. Methods such as *remove*, *get*, *set*, and *subList* are commonly used with lists and require an index parameter. Providing an index beyond the collection's current size results in an *IndexOutOfBound-*

sException. It is crucial to note that while the *List* interface supports index-based operations, the *Set* interface does not, reflecting its inherent unordered nature.

StringIndexOutOfBoundsException. Strings in Java, along with their mutable counterparts, *StringBuilder* and *StringBuffer*, necessitate the use of built-in methods for character manipulation. For the *String* class, methods like *charAt* and *substring* can lead to exceptions if the index provided is outside the string's length. Similarly, *StringBuilder* and *StringBuffer* classes offer methods such as *delete*, *replace*, *insert*, and *setCharAt*. These methods, while providing mutable operations on the string, also carry the risk of throwing an exception if the index is improperly used.

Boundary Value Analysis

To introduce the concept of boundary value analysis, it is first necessary to explain the method of equivalence class partitioning. Equivalence class partitioning is a black box testing technique that involves dividing the input domain of a program into distinct data classes, from which test cases are derived [135]. The input set is partitioned into subsets or classes, each of which represents a set of test inputs with similar features and specifications. By testing one condition from each class, we can assume that all conditions in that class will be treated in the same way by the software, thereby reducing the number of test cases required.

The validity of each subset or class depends on the input constraints. For instance, a program that only accepts one-digits integers would have a valid equivalence class of integers ranging from -9 to 9, while the invalid equivalence classes would be integers less than -9 and greater than 9.

During software development, dealing with the boundary values of equivalence classes can be a source of errors for programmers. To address this challenge, boundary value analysis is another black box testing method that is used to create test cases that exercise the boundaries of input and output classes [136]. This method involves identifying both valid and invalid boundaries within the equivalence classes and deriving test cases from them. To perform BVA effectively, the following guidelines should be followed:

1. If an input condition specifies a variable with a range of values between m and n, test cases should be designed with four values and divided into four cases as shown in Table 4.2.

2. If an input condition specifies multiple values, the minimum and maximum values, and values just above the maximum value and below the minimum value should be used as test cases.

$$Test\ Cases = \{Min, Max, Min - 1, Max + 1\}$$

Table 4.2: Input conditions and their corresponding test cases

Input Condition	Test Cases
$var \in (m, n)$	{m, n, m + 1, n - 1}
$var \in [m, n)$	{m, n, m - 1, n - 1}
$var \in (m, n]$	{m, n, m + 1, n + 1}
$var \in [m, n]$	{m, n, m - 1, n + 1}

Looking back at the aforementioned program that accepts only one-digit integers, whose valid equivalence class is integers from -9 to 9, that is, $input \in (-10, 10) \cap \mathbb{Z}$. However, if the restriction added to the input is erroneously written as $input \leq 10$ && $input \geq -10$, then the test case will be $Test\ Cases = \{-11, -10, 10, 11\}$. This error leads to the program accepting invalid inputs. Furthermore, when the input is given -10 or 10, no error is reported, which violates the requirements. This illustrates that there is a design flaw in the program that does not conform to the specific requirements.

4.3.2 Case Study

Example Program for Explanation

Assuming that such a program is implemented now, in which an array of *int* type with a length of 10 is defined. The variable i and j are both index variables of the array, the value of i is initialized to 3, while the value of j is input by the user during runtime. Moreover, a function called *swap* has been defined to exchange the position of two elements in the array. It is required to output the i-th element of the array and then output the array after swapping the i-th element with the j-th element by calling the *swap* function. Such a program realized according to the above requirements is shown in Listing 4.4.

Listing 4.4: An example program with array index out of bounds exceptions

```
main () {
    int[] arr = new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int i = 3;
    int j = ?;
    print arr[i];
    int[] newarr = swap (arr, i, j);
    print newarr;
}
int[] swap (int[] arr, int i, int j){
    int t = 0;
    if(i >=0 && i<=arr.length && j >=0 && j<=arr.length){
        t = arr[i];
        arr[i] = arr[j];
        arr[j] = t;
    }
    return arr;
}
```

Determination of Suspicious Segments for RQ1

In addition to ANTLR, we can also use regular expressions to identify the target suspicious segments .

Table 4.3: Regular expressions corresponding to arrays, collections, and strings

Type	Pattern	Regular Expression
Array	ArrayName[index]	`(?<!new )\b[a-zA-Z_]+\w*\[[^]]+\]`
Collection	CollectionName. MethodName(index, ...)	`\b[a-z_]+\w*\.(get|set|remove|subList)\(((\((?<c>)|[^()]|\)(?<-c>))+(?(c)(?!))\)(?! \(| {)`
String	StringName. MethodName(index, ...)	`\b[a-z_]+\w*\.(charAt|substring|delete|replace|insert|setCharAt)\(((\((?<c>)|[^()]|\)(?<-c>))+(?(c))\)(\[.+\])?`

As mentioned in Section 4.3.1, that an index out of bounds exception is thrown means that there is an operation that attempts to access an element in an array, collection, or string using an unexpected index. We have discussed common methods with the potential to cause such exceptions. In fact, identifying suspicious segments involves searching for statements that contain these methods in the program. In this paper, we make use of regular expressions corresponding to arrays, collections and strings shown in Table 4.3 to match these methods.

The program depicted in Listing 4.4 is subjected to matching, and the analysis reveals that there exist four statements (i.e., fifth, twelfth, thirteenth and fourteenth statements) containing array element access operations. Subsequently, according to the rules described in Algorithm 1, these four statements are identified as two suspicious segments to wait for further analysis, which gives seg_5, seg_{12}.

Construction of Sub-program for RQ2

As discussed earlier, our objective in this phase is to create a sub-program that serves as a testing environment for each identified suspicious segment. Continuing with the example of the program in Listing 4.4. The segment seg_5 is responsible for outputting the i-th element of the array. To effectively analyze seg_5, we identify the variables engaged in this operation, which are *arr* (the array) and i (the index). We then apply the slicing criterion $\langle 5, arr \rangle$ to derive a program slice for the variable *arr*. This slice is comprised of all the statements that have a direct or indirect impact on the value of *arr*. Analogously, we utilize the slicing criterion $\langle 5, i \rangle$ to generate a program slice for the variable i. Once we have acquired the program slices for both *arr* and i, we proceed to remove any overlapping code between the two slices. The goal is to consolidate these slices into a single, cohesive sub-program that encompasses all necessary elements to facilitate the testing of seg_5. The resulting sub-program, which is devoid of redundancies and organized in a sequential manner, is then illustrated in Listing 4.5.

Similarly, for the segment seg_{12}, which involves exchanging the position of the i-th and j-th elements in the array *arr* in the formal parameter, we use the variables involved in the segment as slicing criteria to slice the program and obtain another sub-program depicted in Listing 4.6. By replicating this procedure for each suspicious segment, we can systematically generate sub-programs for all identified segments.

Listing 4.5: The slice obtained from the segment seg_5

```
main(){
    int[] arr = new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int i = 3;
    print arr[i];
}
```

Listing 4.6: The slice obtained from the segment seg_{12}

```
main () {
    int[] arr = new int[]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int i = 3;
    int j = ?;
    int[] newarr = swap (arr, i, j);
}
int[] swap (int[] arr, int i, int j){
    int t = 0;
    if (i>=0 && i<=arr.length && j>=0 && j<=arr.length){
        t = arr[i];
        arr[i] = arr[j];
        arr[j] = t;
    }
    return arr;
}
```

Implementation of Testing for RQ3

For index out of bounds exceptions, we utilize a specialized test case generation method. It also begins with determining whether there is any input from the user in the segment. If not, it will be executed directly. Otherwise, a series of steps as shown in Figure 4.7 will be taken to complete the testing process. The first step is to identify the input variables and the restrictions on them by means of the lexical analysis of the code, and then determine their boundary values. Next, specific test cases are defined according to the generation rules outlined in Section 4.3.1. However, in some cases, it is not always possible to obtain the boundary values of the input variables. In this case, fuzz testing is needed to randomly generate test cases. Then, the test-

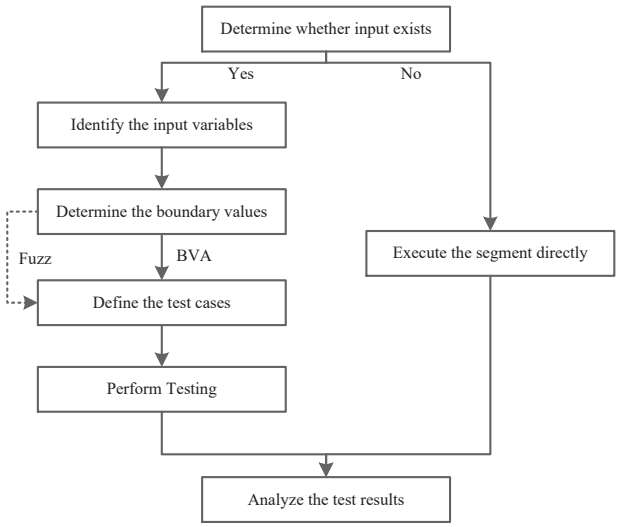


Figure 4.7: The process of the testing

ing is conducted, which runs the program with all the generated test cases. Finally, these results are analyzed to identify bugs or defects.

The sub-program obtained from the segment seg_5 in Listing 4.5 does not involve any user input. Upon execution, no array index out of bounds exceptions is thrown, leading to the conclusion that this segment is exception-safe in this program. Therefore, there is no necessity to perform any further operations on it or provide any feedback to the developer.

For the sub-program from the segment seg_{12} in Listing 4.6, in which the value of the variable j is input by the user when the program is executed. Variable j is the index variable of the array *arr*, whose restriction in the program is $j \geq 0$ && $j \leq arr.length$. Therefore, it can be seen from the generation criteria in Table 4.2 that test cases should be designed with values 0 and *arr.length*, as well as values just below 0 and above *arr.length*, that is, $Test\ Cases = \{-1,\ 0,\ arr.length,\ arr.length + 1\}$.

When the test case is -1, 0 and *arr.length*+1, the program runs normally or outputs an incorrect result but no exception will be thrown. However, when j is assigned the value of *arr.length*, which exceeds the bound of the array, it will cause the program to throw an exception. At this time, the system requires the developer to modify the program so that the program will not be terminated during runtime.

4.3.3 Experiment

This section introduces our study to evaluate PST and compare it with three other existing fault detection techniques using a set of programs. We will first provide an overview of the three techniques and then describe our experiment. Finally, we will present the results of the experiment and provide an analysis of them.

Java Fault Detection Tools

Checker Framework. The Checker Framework significantly enhances Java's type system, providing developers with tools to detect and prevent errors in their Java code more effectively than Java's built-in type system [137]. It includes a variety of compiler plug-ins, known as checkers, which are designed to identify bugs or verify their absence. These checkers cover a wide range of error types, from null pointer exceptions to concurrency flaws. Moreover, the framework allows for the creation of custom checkers, enabling developers to enforce specific correctness properties unique to their applications.

SpotBugs is an analysis tool for Java, which uses a series of ad-hoc techniques designed to balance precision, efficiency and usability [138]. One of the main techniques SpotBugs uses is to examine a class or JAR file, comparing the bytecode to a set of bug patterns to find potential vulnerabilities. For example, SpotBugs checks whether creating an IO stream object in a Java program using classes such as FileInputStream and InputStream is accompanied by a close operation, which is considered a safe operation in most cases. In some cases, SpotBugs also employs data flow analysis to check for bugs. For example, SpotBugs uses simple intraprocedural data flow analysis to check for null pointer references.

SpotBugs allows users to configure detection rules, that is, users can determine the types of bugs reported after program detection, such as bad practice and malicious code vulnerability. Futhermore, SpotBugs can be extended by implementing unique validation rules, although this requires inheriting its interface when customizing a specific bug pattern.

ESC/JAVA, the Extended Static Checking system for Java, performs formal verification of properties of Java source code depending on theorem proving [139]. ESC uses an automatic theorem-prover to reason about the

semantics of the program, which allows ESC to give static warnings for errors caught at runtime. ESC allows program developers to record design decisions in Java, and issues warning when the program violates these design decisions. Moreover, ESC/Java is designed so that it can produce some useful output even without any specification, which is how we use it in our study. In this case, ESC/Java is able to detect runtime exceptions such as null pointer dereferences, index out of bounds.

Experiment Design and Analysis

We conduct the experiments with the assistance of four doctoral students and an assistant professor, who are experienced in software development, software fault detection, and the PST method respectively. For brevity, they will be referred to as student 1,2,3 and 4 in the following discussion. Student 1, with a background in software development, is tasked with employing regular expressions to identify suspicious segments within the programs. Student 2 analyzes the code to ascertain the actual effectiveness of these suspicious segments. Student 3 is responsible for injecting bugs into the original programs, while the assistant professor conducts tests on them using our PST method. To mitigate potential biases in the evaluation of test results, we ensure that the outcomes are always reviewed under the supervision of the authors. To lend credibility to our experiment, we chose several programs from the Software Artifact Infrastructure Repository for analysis. These programs were selected due to their common use in fault localization and software testing, highlighting their relevance and importance in our study [129, 130, 131, 132].

Table 4.4 offers a comprehensive summary of the various programs utilized in our study and presents the findings obtained through our method. *LOC* stands for the Lines Of Code in each program, specifically referring to any text line that is neither a blank line nor a comment line, as identified by the first author. The term *Matched Suspicious Segment* (MSS) denotes the number of segments identified by Student 1 that could potentially cause index out of bounds exceptions, as detected using the regular expression techniques described in Section 3.1. However, the number of segments that actually result in exceptions may differ from those identified through matching due to various factors. Consequently, Student 2 evaluates the program to provide the actual count of *Effective Suspicious Segments* (ESS), reflecting the true number of problematic segments.

Table 4.4: Overview of the experiment program and experiment results using PST

Program Name	LOC	Matched Suspicious Segment	Effective Suspicious Segment	Injected Bugs	Faulty Version	Successful Detection	Failed Detection
Account	146	12	12	2	2	2	0
AlarmClock	338	12	12	2	3	3	0
ArrayPartion	63	8	8	1	1	1	0
LinkedList	138	13	13	3	4	3	1
SleepingBar	154	2	2	1	1	1	0
Lang	2448	71	71	5	10	8	2
ProducerConsumer	193	11	11	2	2	2	0
Vector	454	21	29	4	7	7	0

For example, in the *AlarmClock* program, both the MSS and ESS figures are initially twelve, which might seem consistent. Yet, upon closer examination, we find that the *elementAt* method, flagged for its suspicious operation of accessing an array element, is overridden in the program but never invoked, thus it would not trigger an exception. This oversight suggests that the ESS should be adjusted to eleven. Nevertheless, the program includes a *removeElement* method, akin to Java's *remove* function, which is employed once. This method, not originally flagged as suspicious, was missed in the regular expression match, leading to a false negative. Accordingly, the accurate count for ESS is corrected back to twelve. Similarly, in the *Vector* program, the discrepancy of eight additional ESS compared to MSS is attributed to the inclusion of *getElement* and *modifyElement* methods. These methods are designed to obtain and modify a specific element in the vector respectively, and are invoked eight times within the program.

The Injected Bugs column in our data illustrates the number of bugs that Student 3 introduced into various programs, each designed to trigger an exception during runtime. These modifications resulted in different versions of the programs, each harboring one or more deliberate bugs. A version is deemed to have achieved *Successful Detection* if all introduced bugs are identified; otherwise, it falls under *Failed Detection*, as determined by the assistant professor. The data reveals that our method did not completely detect the bugs in two versions of the *Lang* program and one version of the *LinkedList* program. This outcome underscores a limitation in our approach when applied to entire programs that contain multiple bugs, a topic we plan to explore further in Section 4.5.

The *Detected Bugs* column reflects the count of distinct bugs discovered in all faulty versions of the programs, as verified by Student 4. A comparative analysis between this column and the *Injected Bugs* column reveals a discrepancy in the *Vector* program, where one less bug was detected than the number initially injected. Specifically, a bug introduced into the previously discussed *getElement* function was not flagged as a suspicious statement during the matching process, culminating in a false negative.

Experiment Results and Analysis

In Figure 4.8, we present the comparative outcomes concerning suspicious segments that could potentially lead to exceptions within the programs, as delineated in Table 4.4. The illustration indicates that our method did not

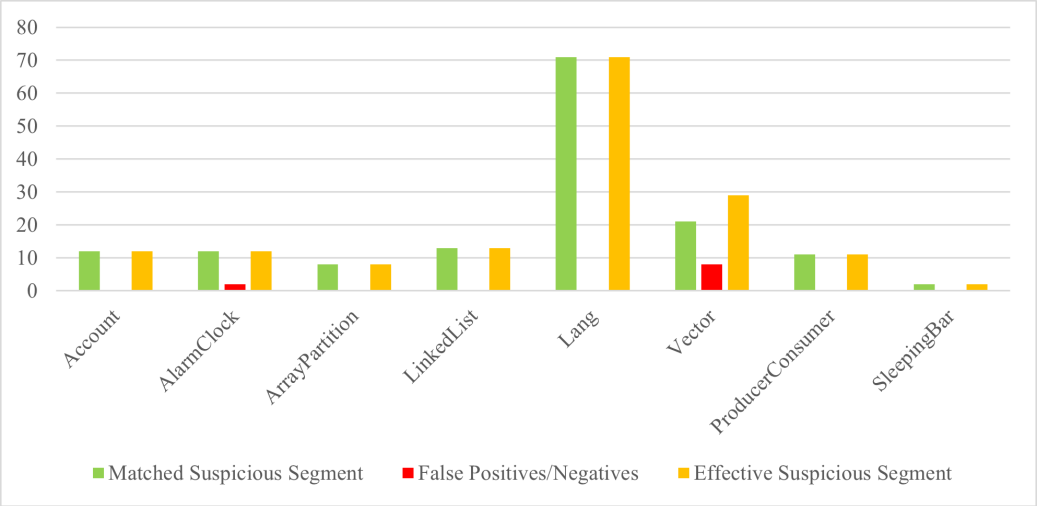


Figure 4.8: Comparison results of suspicious statements

seamlessly match all suspicious segments, evidenced by the discrepancies between the MSS and the ESS. Particularly, in programs such as *AlarmClock* and *Vector*, there are noticeable instances of false positives/negatives, which highlight the challenges in accurately identifying all problematic segments of code. Nonetheless, the overall results demonstrate a substantial degree of accuracy. Specifically, our method successfully identified a significant proportion of the true positives, which is indicative of the precision of the regular expressions used and the effectiveness of the subsequent analysis. Despite the challenges, an accuracy rate of 150 out of 158, or 94.9%, provides a compelling indication of the method's reliability. This high accuracy rate is critical because it lays a robust foundation for attaining a significant fault detection rate in the ensuing phases of our research. Moreover, it also suggests that with further refinement, particularly in reducing the false positives/negatives as seen in the red columns of the chart, the method could be made even more effective.

The experiment results comparing the injected bugs with the actual detected bugs will be discussed in the next section when we compare our tool with others.

Experiment Comparison

In this section, we engage in a comparative analysis of bug detection results using established and widely adopted tools as compared to those obtained via our method, a process undertaken by the second author. Consistent with standard practice, we evaluate and compare these tools based on their effectiveness and efficiency, as cited in references [131, 140]. However, given that our method is not yet supported by an automated tool and is implemented manually, the comparison in terms of efficiency, that is, the time required to perform the detection, is of no reference value. Therefore, our comparison focuses solely on effectiveness, which we measure by the count of successfully detected injected bugs. A higher number of detected bugs is indicative of a tool's greater effectiveness. The findings of this evaluation are organized in Table 4. This table features several columns for easy reference: the first column specifies the names of the programs tested, the second column enumerates the number of bugs injected into each program, and the subsequent columns (third to sixth) present the detection results garnered by PST and the other tools that were introduced earlier in Section 4.1.

Spotbugs is a powerful tool capable of detecting bugs across ten major categories, each containing dozens or even hundreds of more precise bug types. For example, it can identify *Security* issues, such as the use of unsafe external inputs; *Multithreaded Correctness*, which includes code that might lead to deadlocks; and *Malicious Code Vulnerability*, where the code is at risk of being attacked by malicious software, such as returning a mutable reference stored in an object field, among others. Of particular interest is the *Correctness* category, which points out code that may cause runtime errors, such as index out of bounds. Unfortunately, despite these capabilities, the

Table 4.5: Experiment comparison results using PST and three other techniques

Program Name	Injected Bugs	Program Segment Testing	Index Checker	SpotBugs	ESC/Java
Account	2	2	2	0	1
AlarmClock	2	2	0	0	0
ArrayPartition	1	1	1	0	0
LinkedList	3	3	0	1	0
SleepingBar	1	1	0	0	1
Lang	5	5	3	1	1
ProducerConsumer	2	2	2	0	1
Vector	4	3	0	1	0

performance of Spotbugs in analyzing the results is not always satisfactory.

ESC/JAVA employs an automatic theorem-prover to perform formal verification of properties of Java source code, which allows it to give static warnings for potential runtime errors. Analysis of the ESC detection results reveals that the warnings primarily fall into three types: (1) The prover cannot establish an assertion; (2) Precondition conjunct is false; (3) Associated declaration. The first type can be further subdivided into multiple sub-types, such as *PossiblyNegativeIndex*, which indicates that the index of the array may be negative, *PossiblyTooLargeIndex*, which suggests that the value of the index may be too large, such as exceeding the upper bound of the Int type variable, and *ArithmeticOperationRange*, which implies that the prover cannot verify the loop condition of a loop structure when it contains arithmetic operations. Upon close examination of these warnings, it is evident that ESC will issue a warning for any operation that accesses array elements, much like the operation of identifying suspicious statements in our method. However, ESC's focus is limited to arrays, which constrains its utility for methods that access elements in strings or collections. Furthermore, ESC predominantly performs tests in extreme cases, such as determining if the value of a variable exceeds the upper and lower bounds of the *Int* variable or 0. It lacks the capability to verify more nuanced conditions, such as whether a variable is less than a fixed value like the length of an array. The experiment results clearly indicate that only when exceptions that the tool is capable of handling occur in the program, can it manage to provide a report, albeit an imprecise one. Consequently, the number of exceptions detected is quite sparse.

The Checker Framework utilizes a series of custom type systems that operate as plugins for the javac compiler to find and prevent various bugs within programs. The Index Checker, specifically designed to warn against potential out-of-bounds accesses to sequence data structures, prevents *IndexOutOfBoundsException* that may arise if an index expression is likely to be negative or equal to or greater than the sequence's length. This is achieved by writing annotations to indicate which expressions are indices for which sequences. For example, if a variable i is used as an index for an array *myArray*, we annotate the type of i with *@IndexFor("myArray")* during its declaration. This annotation ensures that the variable i is always non-negative and less than the length of *myArray* during runtime, that is, $0 \leq i < myArray.length$. The Index Checker prohibits any operations that could potentially violate these properties and utilizes them when verifying index operations.

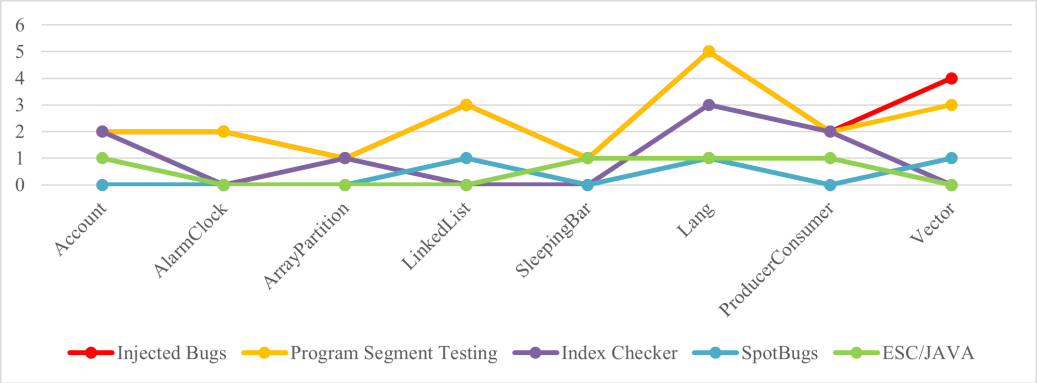


Figure 4.9: Experiment comparison results using PST and three other techniques

However, the Index Checker is limited to checking only fixed-size data structures, whose sizes cannot change after creation, such as strings and arrays. This implies that it is not suitable for mutable data structures like lists, which can alter their size through methods like *add* and *delete*. This limitation was clearly confirmed in programs such as *LinkedList* and *AlarmClock*. Additionally, the Index Checker does not check for arithmetic overflow; thus, if there is a potential risk of overflow in the expressions of a program, the Index Checker will not provide warnings. While this is unlikely to be a concern in most practical developments, we also confirmed this limitation in the *SleepingBar* program. In other scenarios, the Index Checker can provide warnings as long as appropriate annotations are placed correctly within the program. This presupposes familiarity with the various annotations of the checker and their meanings, ensuring that the framework's annotations are correctly understood and applied. However, this can be daunting for large codebases that lack prior type annotations.

In conclusion, to present the comparison results more intuitively, we translate the data from Table 4.5 into a line graph, as shown in Figure 4.9. In this graphical representation, the line corresponding to our method appear higher than those of all other methods. This visual elevation of the PST line in the graph clearly illustrates its effectiveness and superiority, corroborating the analytical findings previously discussed.

4.4 Threats to Validity

Addressing threats to validity is crucial in ensuring the reliability of experiment results in software engineering research. Here we discuss various measures implemented to mitigate such threats during our experiment.

1. Selection and Diversity of Target Programs

One potential threat involves the number and complexity of the target programs used for testing. To enhance the reliability of our results, we selected programs from the Software Artifact Infrastructure Repository. These programs are commonly employed in studies involving fault localization and fault detection, covering typical issues encountered in software development, such as producer-consumer problems and sorting algorithms. While the programs used may not suffice for an exhaustive evaluation of our method due to their limited scope, they serve well for initial observations and analyses in smaller-scale scientific research experiments. Although no instances were encountered where our method underperformed or proved invalid, extending our approach to a broader array of programs might reveal different outcomes. Therefore, conducting larger-scale empirical studies is essential for a more comprehensive evaluation of our method in diverse software environments.

2. Human Factors

Human factors significantly threaten the validity of experiment results due to the potential for bias and subjective influences. To mitigate these threats, we adopted a blind experiment design. In this setup, individuals involved in specific tasks were kept unaware of the activities of others participating in the experiment. This method helps prevent biases or preconceived notions from affecting the outcomes, as experimenters do not have prior knowledge about what the expected results should be.

Moreover, to further reduce the impact of subjective human errors, all experiment interactions and evaluations were recorded and periodically reviewed by an independent party not involved in the experiment phases. This process of external review helps to identify any inconsistencies or biases in the handling or interpretation of data collected during the experiments. By incorporating these checks, we aim to ensure that our experiment conclusions are based on objective observations and are not unduly influenced by the experimenters' expectations or personal experiences.

3. Experiment Controls and Repetition

To further solidify the integrity of our findings, we maintained strict control over the experiment conditions. This included consistent use of equip-

ment, settings, and methodologies across all test scenarios. Repeating experiments under the same conditions helps in identifying any anomalies or inconsistencies in the results, thereby providing a more robust set of data to support our conclusions.

By addressing these potential threats to validity, we aim to ensure that our findings are not only insightful but also dependable and applicable in real-world settings. Future studies could expand upon this foundation, exploring larger datasets and more complex program structures to fully validate and potentially enhance the efficacy of our proposed method.

4.5 Discussion about PST

PST is a dynamic and automated testing methodology specifically designed to identify and handle runtime exceptions during the software development process. Initially developed for Java, PST focuses on detecting common runtime exceptions such as arithmetic errors, which are prevalent during the execution phase of software.

1. Expansion to Other Exception Types

The fundamental approach of PST, which involves identifying potentially problematic code segments, setting up a targeted testing environment, and conducting tests, can indeed be expanded to detect other types of runtime exceptions. For instance, adaptations of PST could effectively pinpoint issues like null pointer exceptions. These are typical errors that can also cause significant disruptions in software operations.

To extend PST to cover these additional exception types, the detection logic and testing mechanisms need to be appropriately modified. This involves adjusting the criteria for segment selection and enhancing the testing algorithms to recognize and react to the specific nuances of each new type of exception.

2. Adaptability Across Programming Languages

Moreover, the core principles of PST are not confined to Java and can be seamlessly adapted to other programming languages. This adaptability is crucial given the diverse programming environments and the specific error-handling mechanisms inherent to each language.

- C++: Adapting PST to C++ would involve integrating with its exception handling and memory management features. C++ programs may

face unique issues related to pointer arithmetic and memory allocation, which PST could be tailored to monitor and test.

- Python: For Python, adapting PST would mean focusing on dynamically typed variable issues and runtime errors common in a scripting environment, such as type errors or index errors in lists.
- JavaScript: In the context of JavaScript, PST could be adapted to handle errors typically found in asynchronous programming and event-driven architectures, such as callback errors and promises rejections.

Chapter 5

Comparative Analysis and Discussion

5.1 Introduction

This chapter provides a comparative analysis of formal verification techniques and PST, highlighting their respective strengths, limitations, and complementary nature. The aim is to understand how these techniques can be leveraged together to enhance software reliability and to identify scenarios where each method is most effective.

5.2 Strengths of Formal Verification Techniques

Formal verification techniques, such as Event-B and LTS, offer several strengths:

1. Rigor and Precision:

Formal verification provides a mathematically rigorous way to ensure software correctness. By using formal methods, developers can create precise models of system behavior, eliminating ambiguities and reducing the risk of errors.

2. Proof of Correctness:

These techniques allow for the comprehensive proof of system properties, ensuring that the software adheres to its specifications. This is particularly important for safety-critical systems, where failures can have severe consequences.

3. Modeling Complex Interactions:

Techniques like LTS are effective in modeling and analyzing concurrent and distributed systems. They provide a clear representation of system states and transitions, helping to identify potential issues such as deadlocks and race conditions.

5.3 Limitations of Formal Verification Techniques

Despite their strengths, formal verification techniques face several limitations:

1. Resource Intensity:

Formal verification can be highly resource-intensive, requiring significant computational power and time to model and verify complex systems. This can limit its practicality, especially for large-scale projects with tight deadlines.

2. Expertise Required:

The steep learning curve and the need for substantial expertise in formal methods and mathematical modeling can be a barrier to widespread adoption. Many development teams may lack the necessary skills to effectively use these techniques.

3. Limited Real-Time Feedback:

Formal verification is typically performed after the software design is complete, making it less suitable for providing real-time feedback during the development process. This can delay the detection and correction of errors.

5.4 Strengths of PST

PST offers several advantages that complement the strengths of formal verification:

1. Real-Time Error Detection:

Integrated within the HMPP framework, PST provides real-time feedback on runtime exceptions. This allows developers to identify and fix issues early in the development cycle, enhancing productivity and reducing debugging time.

2. Applicability to Iterative Development:

PST is particularly well-suited for agile development environments, where software is developed incrementally. It can be applied to both partial and entire programs, providing continuous error detection as the software evolves.

3. Reduced Expertise Requirement:

PST does not require the same level of mathematical and formal methods expertise as formal verification. This makes it more accessible to a broader range of development teams.

5.5 Limitations of PST

PST also has its limitations:

1. Scope of Error Detection:

While PST is effective in detecting runtime exceptions, it may not provide the comprehensive proof of system correctness that formal verification offers. It focuses on identifying specific types of errors rather than verifying all system properties.

2. Tool Support:

Dedicated tools for PST are still under development, limiting its widespread adoption. The effectiveness of PST is heavily influenced by the availability and usability of these tools.

5.6 Complementary Nature of Formal Verification and PST

Formal verification and PST have distinct strengths that make them complementary:

1. Enhanced Reliability:

By combining the rigor of formal verification with the real-time error detection capabilities of PST, developers can achieve higher software reliability. Formal verification ensures that the software adheres to its specifications, while PST provides continuous monitoring and immediate feedback during development.

2. Improved Development Workflow:

The integration of both techniques into the software development lifecycle can streamline the process. Formal verification can be used to validate the

initial design and critical components, while PST can be employed to monitor and test the software as it is being developed.

3. Balanced Resource Utilization:

Combining both techniques allows for a balanced approach to resource utilization. Formal verification can be applied to the most critical parts of the system, while PST can be used more broadly across the entire development process, providing a practical and efficient solution.

5.7 Discussion

The comparative analysis highlights that formal verification and PST are not mutually exclusive but rather complementary techniques. By leveraging the strengths of both methods, developers can enhance software reliability in a comprehensive and efficient manner. The key is to integrate these techniques thoughtfully into the software development lifecycle, using formal verification for critical components and design validation, and PST for continuous monitoring and error detection during development.

Chapter 6

Conclusion and Future Work

6.1 Summary of Key Findings

This dissertation set out to enhance software reliability through the evaluation and application of formal verification techniques and PST. The research aimed to address the limitations of each method by leveraging their respective strengths in different phases of the software development process. The key findings from this research are summarized as follows:

1. Effectiveness of Formal Verification Techniques:

Formal verification techniques, specifically Event-B and LTS, have been shown to be highly effective in ensuring software correctness and reliability for safety-critical and completed systems. The application of these techniques to the ARINC653 specification demonstrated their ability to rigorously prove system properties and ensure adherence to specifications.

2. Challenges and Limitations of Formal Verification:

Despite their strengths, formal verification techniques face significant challenges in dynamic and iterative development environments such as HMPP. These challenges include resource intensity, the need for substantial expertise, and limited real-time feedback capabilities.

3. Development and Evaluation of PST:

PST has been developed and evaluated as a complementary technique to formal verification. Integrated within the HMPP framework, PST provides real-time error detection without human intervention. The experiments and case studies demonstrated PST's effectiveness in identifying runtime exceptions early in the development cycle, enhancing productivity and reducing

debugging time.

4. Comparative Analysis:

The comparative analysis highlighted the complementary nature of formal verification and PST. Formal verification provides rigorous proof of system correctness, while PST offers practical, real-time error detection during development. Together, these techniques can significantly enhance software reliability.

6.2 Implications for Theory and Practice

The findings from this research have several important implications for both theory and practice:

1. Theory:

This research contributes to the theoretical understanding of software reliability by demonstrating the complementary strengths of formal verification and PST. It provides a framework for integrating these techniques in a way that leverages their respective advantages.

2. Practice:

For practitioners, the research offers practical insights into how formal verification and PST can be integrated into the software development lifecycle. By applying formal verification to critical components and using PST for continuous monitoring, developers can improve the reliability of their software in a resource-efficient manner.

6.3 Future Research Directions

Based on the findings and limitations identified in this research, several future research directions are proposed:

1. Tool Development:

Continued development and refinement of tools to support PST is essential. These tools should be made more accessible and user-friendly to facilitate widespread adoption.

2. Integrated Frameworks:

Research should focus on developing integrated frameworks that seamlessly combine formal verification and PST. Such frameworks would provide a cohesive solution for enhancing software reliability across various domains.

3. Empirical Studies:

Further empirical studies are needed to validate the effectiveness of combining formal verification and PST in different domains and development environments. These studies would help to generalize the findings and provide additional evidence of their benefits.

4. Education and Training:

Enhancing education and training programs to equip developers with the necessary skills to effectively use both formal verification and PST is crucial. This would help to bridge the expertise gap and promote the adoption of these techniques.

6.4 Conclusion

This dissertation has demonstrated that formal verification and PST are powerful techniques for enhancing software reliability. While formal verification provides rigorous proof of system correctness, PST offers practical, real-time error detection during development. By leveraging the strengths of both techniques, developers can achieve higher software reliability and improve the efficiency of the development process. The insights gained from this research provide valuable guidance for future research and practical implementation, ultimately contributing to the development of more reliable software systems.

Chapter 7

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor, Professor Shaoying Liu. His unwavering support, guidance, and encouragement throughout my doctoral research have been invaluable. Not only did he provide critical academic insights, but he also offered advice on personal development and career planning.

I am also grateful to all the faculty and staff of the School of Advanced Science and Engineering for their support and assistance during my PhD studies.

A special acknowledgment goes to the members of my dissertation committee for their valuable feedback and suggestions throughout my research plan and dissertation writing process. Without their help, this dissertation would not have been possible.

I would like to extend my heartfelt thanks to my fellow students and friends, who have provided immense support and encouragement both academically and personally. The discussions and exchanges with them have given me many valuable insights and assistance.

Lastly, I am profoundly grateful to my family. Their unconditional support and love have been the driving force behind the completion of this research. Special thanks to my parents, whose constant encouragement and support have enabled me to persevere.

Once again, I extend my sincere gratitude to everyone who has helped and supported me throughout this journey. Your support and assistance have made it possible for me to complete this challenging yet rewarding task.

Reference

[1] N. Plat, J. van Katwijk, and H. Toetenel, "Application and benefits of formal methods in software development," *Software Engineering Journal*, vol. 7, no. 5, pp. 335–346, 1992.

[2] J. B. Almeida, M. J. Frade, J. S. Pinto, S. Melo de Sousa, J. B. Almeida, M. J. Frade, J. S. Pinto, and S. Melo de Sousa, "An overview of formal methods tools and techniques," *Rigorous Software Development: An Introduction to Program Verification*, pp. 15–44, 2011.

[3] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[4] J. M. Schumann, *Automated theorem proving in software engineering*. Springer Science & Business Media, 2001.

[5] J. Souyris, V. Wiels, D. Delmas, and H. Delseny, "Formal verification of avionics software products," in *FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings 2*, pp. 532–546, Springer, 2009.

[6] T. Lecomte, T. Servat, G. Pouzancre, *et al.*, "Formal methods in safety-critical railway systems," in *10th Brasilian symposium on formal methods*, pp. 29–31, 2007.

[7] J. Yoo, T. Kim, S. Cha, J.-S. Lee, and H. S. Son, "A formal software requirements specification method for digital nuclear plant protection systems," *Journal of Systems and Software*, vol. 74, no. 1, pp. 73–83, 2005.

[8] S. Siegl, K.-S. Hielscher, R. German, and C. Berger, "Formal specification and systematic model-driven testing of embedded automotive sys-

tems," in *2011 Design, Automation & Test in Europe*, pp. 1–6, IEEE, 2011.

[9] M. Mohsin, Z. Anwar, G. Husari, E. Al-Shaer, and M. A. Rahman, "Iotsat: A formal framework for security analysis of the internet of things (iot)," in *2016 IEEE conference on communications and network security (CNS)*, pp. 180–188, IEEE, 2016.

[10] T. Grandison, E. M. Maximilien, S. Thorpe, and A. Alba, "Towards a formal definition of a computing cloud," in *2010 6th World Congress on Services*, pp. 191–192, IEEE, 2010.

[11] D. Poole, A. Mackworth, and R. Goebel, "Computational intelligence: a logical approach. 1998," *Google scholar google scholar digital library digital library*, 1998.

[12] J.-R. Abrial, *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[13] J. M. Spivey, *Understanding Z: a specification language and its formal semantics*, vol. 3. Cambridge University Press, 1988.

[14] P. G. Larsen, K. Lausdahl, N. Battle, J. Fitzgerald, S. Wolff, S. Sahara, M. Verhoef, P. W. Tran-Jørgensen, T. Oda, and P. Chisholm, "Vdm-10 language manual," *Internet: overturetool. org/documentation/manuals. html [Oct. 5, 2020]*, 2013.

[15] E. Durr and J. Van Katwijk, "Vdm++, a formal specification language for object-oriented designs," in *CompEuro 1992 Proceedings Computer Systems and Software Engineering*, pp. 214–219, IEEE, 1992.

[16] R. Duke, G. Rose, and G. Smith, "Object-z: A specification language advocated for the description of standards," *Computer Standards & Interfaces*, vol. 17, no. 5-6, pp. 511–533, 1995.

[17] C. Snook and M. Butler, "Uml-b: Formal modeling and design aided by uml," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 15, no. 1, pp. 92–122, 2006.

[18] C. Fischer, "Csp-oz: a combination of object-z and csp," *Formal Methods for Open Object-based Distributed Systems: Volume 2*, pp. 423–438, 1997.

[19] O. U. T. C. Group, "Timed csp: Theory and practice," in *Real-Time: Theory in Practice: REX Workshop Mook, The Netherlands, June 3–7, 1991 Proceedings*, pp. 640–675, Springer, 1992.

[20] J. Hoenicke and E.-R. Olderog, "Csp-oz-dc: A combination of specification techniques for processes, data and time," *Nord. J. Comput.*, vol. 9, no. 4, pp. 301–334, 2002.

[21] C. Sühl, "Rt-z: An integration of z and timed csp," in *IFM'99: Proceedings of the 1st International Conference on Integrated Formal Methods, York, 28-29 June 1999*, pp. 29–48, Springer, 1999.

[22] J. Woodcock and A. Cavalcanti, "The semantics of circus," in *International Conference of B and Z Users*, pp. 184–203, Springer, 2002.

[23] A. Cavalcanti, A. Sampaio, and J. Woodcock, "Unifying classes and processes," *Software & Systems Modeling*, vol. 4, pp. 277–296, 2005.

[24] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996.

[25] M. Butler and M. Leuschel, "Combining csp and b for specification and property verification," in *FM 2005: Formal Methods: International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005. Proceedings*, pp. 221–236, Springer, 2005.

[26] M. Butler, "csp2b: A practical approach to combining csp and b," *Formal Aspects of computing*, vol. 12, pp. 182–198, 2000.

[27] S. Colin, A. Lanoix, O. Kouchnarenko, and J. Souquières, "Towards validating a platoon of cristal vehicles using csp—— b," in *Algebraic Methodology and Software Technology: 12th International Conference, AMAST 2008 Urbana, IL, USA, July 28-31, 2008 Proceedings 12*, pp. 139–144, Springer, 2008.

[28] A. A. McEwan and S. Schneider, "Modelling and analysis of the amba bus using csp and b," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 8, pp. 949–964, 2010.

[29] S. Schneider and H. Treharne, "Csp theorems for communicating b machines," *Formal Aspects of Computing*, vol. 17, no. 4, pp. 390–422, 2005.

[30] S. Schneider and H. Treharne, "Changing system interfaces consistently: a new refinement strategy for csp ‖ b," *Science of Computer Programming*, vol. 76, no. 10, pp. 837–860, 2011.

[31] S. Schneider, H. Treharne, and N. Evans, "Chunks: Component verification in csp ‖ b," in *Integrated Formal Methods: 5th International Conference, IFM 2005, Eindhoven, The Netherlands, November 29-December 2, 2005. Proceedings 5*, pp. 89–108, Springer, 2005.

[32] H. Treharne and S. Schneider, "Using a process algebra to control b operations," in *IFM'99: Proceedings of the 1st International Conference on Integrated Formal Methods, York, 28-29 June 1999*, pp. 437–456, Springer, 1999.

[33] S. Schneider, H. Treharne, and H. Wehrheim, "A csp approach to control in event-b," in *Integrated Formal Methods: 8th International Conference, IFM 2010, Nancy, France, October 11-14, 2010. Proceedings 8*, pp. 260–274, Springer, 2010.

[34] S. Schneider, H. Treharne, and H. Wehrheim, "Bounded retransmission in event-b ‖ csp: a case study," *Electronic Notes in Theoretical Computer Science*, vol. 280, pp. 69–80, 2011.

[35] S. Schneider, H. Treharne, and H. Wehrheim, "Stepwise refinement in event-b csp. part 1: safety," 2011.

[36] S. Schneider, H. Treharne, and H. Wehrheim, "A csp account of event-b refinement," *arXiv preprint arXiv:1106.4098*, 2011.

[37] S. Schneider, H. Treharne, and H. Wehrheim, "The behavioural semantics of event-b refinement," *Formal aspects of computing*, vol. 26, pp. 251–280, 2014.

[38] T. S. Hoang, S. Schneider, H. Treharne, and D. M. Williams, "Foundations for using linear temporal logic in event-b refinement," *Formal Aspects of Computing*, vol. 28, pp. 909–935, 2016.

[39] S. Schneider, H. Treharne, H. Wehrheim, and D. Williams, "Managing ltl properties in event-b refinement. arxiv: 1406: 6622," *to appear IFM2014*, 2014.

[40] D. Hansen and M. Leuschel, "Translating tla+ to b for validation with prob," in *Integrated Formal Methods: 9th International Conference, IFM 2012, Pisa, Italy, June 18-21, 2012. Proceedings 9*, pp. 24–38, Springer, 2012.

[41] S. Blom, J. van de Pol, and M. Weber, "Ltsmin: Distributed and symbolic reachability," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*, pp. 354–359, Springer, 2010.

[42] J. Bendisposto, P. Körner, M. Leuschel, J. Meijer, J. van de Pol, H. Treharne, and J. Whitefield, "Symbolic reachability analysis of b through prob and ltsmin," in *Integrated Formal Methods: 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings 12*, pp. 275–291, Springer, 2016.

[43] T. E. Uribe, "Combinations of model checking and theorem proving," in *International Workshop on Frontiers of Combining Systems*, pp. 151–170, Springer, 2000.

[44] S. Berezin, *Model checking and theorem proving: a unified framework*. PhD thesis, Carnegie Mellon University, 2002.

[45] C. Baumann, B. Beckert, H. Blasum, and T. Bormer, "Formal verification of a microkernel used in dependable software systems," in *International Conference on Computer Safety, Reliability, and Security*, pp. 187–200, Springer, 2009.

[46] C. Baumann and T. Bormer, "Verifying the pikeos microkernel: first results in the verisoft xt avionics project," in *Doctoral Symposium on Systems Software Verification (DS SSV'09) Real Software, Real Problems, Real Solutions*, p. 20, 2009.

[47] C. Baumann, B. Beckert, H. Blasum, and T. Bormer, "Better avionics software reliability by code verification," in *Proceedings, embedded world Conference, Nuremberg, Germany*, 2009.

[48] C. Baumann, T. Bormer, H. Blasum, and S. Tverdyshev, "Proving memory separation in a microkernel by code level verification," in *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pp. 25–32, IEEE, 2011.

[49] R. J. Richards, "Modeling and security analysis of a commercial real-time operating system kernel," in *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pp. 301–322, Springer, 2010.

[50] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz, "Formal verification of information flow security for a simple arm-based separation kernel," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 223–234, 2013.

[51] L. Freitas and J. McDermott, "Formal methods for security in the xenon hypervisor," *International journal on software tools for technology transfer*, vol. 13, pp. 463–489, 2011.

[52] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "sel4: from general purpose to a proof of information flow enforcement," in *2013 IEEE Symposium on Security and Privacy*, pp. 415–429, IEEE, 2013.

[53] F. Verbeek, S. Tverdyshev, O. Havle, H. Blasum, B. Langenstein, W. Stephan, Y. Nemouchi, A. Feliachi, B. Wolff, and J. Schmaltz, "Formal specification of a generic separation kernel," *Archive of Formal Proofs*, vol. 2014, no. 2014-07-18, 2014.

[54] F. Verbeek, O. Havle, J. Schmaltz, S. Tverdyshev, H. Blasum, B. Langenstein, W. Stephan, B. Wolff, and Y. Nemouchi, "Formal api specification of the pikeos separation kernel," in *NASA Formal Methods: 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings 7*, pp. 375–389, Springer, 2015.

[55] D. Sanán, A. Butterfield, and M. Hinchey, "Separation kernel verification: The xtratum case study," in *Working Conference on Verified Software: Theories, Tools, and Experiments*, pp. 133–149, Springer, 2014.

[56] Y. Zhao, D. Sanán, F. Zhang, and Y. Liu, "Formal specification and analysis of partitioning operating systems by integrating ontology and refinement," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 4, pp. 1321–1331, 2016.

[57] I. D. Craig, *Formal models of operating system kernels*. Springer Science & Business Media, 2007.

[58] A. Velykis, "Formal modelling of separation kernels," *Master's thesis*, 2009.

[59] A. Velykis and L. Freitas, "Formal modelling of separation kernel components," in *Theoretical Aspects of Computing–ICTAC 2010: 7th International Colloquium, Natal, Rio Grande do Norte, Brazil, September 1-3, 2010. Proceedings 7*, pp. 230–244, Springer, 2010.

[60] A. Passos, J. M. Faria, and S. M. de Sousa, "Assessing the formal development of a secure partitioning kernel with the b method," *Critical Software, ADCSS*, vol. 2009, 2009.

[61] M. Leuschel and M. Butler, "Prob: A model checker for b," in *FME 2003: Formal Methods: International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003. Proceedings*, pp. 855–874, Springer, 2003.

[62] K. Kawamorita, R. Kasahara, Y. Mochizuki, and K. Noguchi, "Application of formal methods for designing a separation kernel for embedded systems," *International Journal of Computer and Information Engineering*, vol. 4, no. 8, pp. 1349–1357, 2010.

[63] A. Oliveira Gomes, *Formal specification of the ARINC 653 architecture using circus*. PhD thesis, University of York, 2012.

[64] Y. Zhao, D. Sanán, F. Zhang, and Y. Liu, "Refinement-based specification and security analysis of separation kernels," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 1, pp. 127–141, 2017.

[65] Y. Zhao, Z. Yang, D. Sanán, and Y. Liu, "Event-based formalization of safety-critical operating system standards: An experience report on arinc 653 using event-b," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 281–292, IEEE, 2015.

[66] P. Wang, S. Liu, A. Liu, and F. Zaidi, "A framework for modeling and detecting security vulnerabilities in human-machine pair programming," *Journal of Internet Technology*, vol. 23, no. 5, pp. 1129–1138, 2022.

[67] P. Wang, S. Liu, A. Liu, and W. Jiang, "Detecting security vulnerabilities with vulnerability nets," *Journal of Systems and Software*, vol. 208, p. 111902, 2024.

[68] P. Robe, S. K. Kuttal, Y. Zhang, and R. Bellamy, "Can machine learning facilitate remote pair programming? challenges, insights & implications," in *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1–11, IEEE, 2020.

[69] J. E. Hannay, E. Arisholm, H. Engvik, and D. I. Sjoberg, "Effects of personality on pair programming," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 61–80, 2009.

[70] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold, "Fault localization and repair for java runtime exceptions," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, pp. 153–164, 2009.

[71] W. Weimer and G. C. Necula, "Finding and preventing run-time error handling mistakes," in *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pp. 419–431, 2004.

[72] F. Farmahinifarahani, Y. Lu, V. Saini, P. Baldi, and C. Lopes, "D-rex: Static detection of relevant runtime exceptions with location aware transformer," in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 198–208, IEEE, 2021.

[73] S. Mahajan, N. Abolhassani, and M. R. Prasad, "Recommending stack overflow posts for fixing runtime exceptions using failure scenario matching," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1052–1064, 2020.

[74] S. Jiang, H. Zhang, Q. Wang, and Y. Zhang, "A debugging approach for java runtime exceptions based on program slicing and stack traces," in *2010 10th international conference on quality software*, pp. 393–398, IEEE, 2010.

[75] S. Jiang, W. Li, H. Li, Y. Zhang, H. Zhang, and Y. Liu, "Fault localization for null pointer exception based on stack trace and program slicing," in *2012 12th International Conference on Quality Software*, pp. 9–12, IEEE, 2012.

[76] C. Galindo, S. Pérez, and J. Silva, "Program slicing with exception handling," in *11th Workshop on Tools for Automatic Program Analysis*, 2020.

[77] M. Allen and S. Horwitz, "Slicing java programs that throw and catch exceptions," *ACM SIGPLAN Notices*, vol. 38, no. 10, pp. 44–54, 2003.

[78] E. Soremekun, L. Kirschner, M. Böhme, and A. Zeller, "Locating faults with program slicing: an empirical analysis," *Empirical Software Engineering*, vol. 26, pp. 1–45, 2021.

[79] S. Sinha and M. J. Harrold, "Analysis and testing of programs with exception handling constructs," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 849–871, 2000.

[80] K. P. Smith, H. Wang, T. J. Durant, B. A. Mathison, S. E. Sharp, J. E. Kirby, S. W. Long, and D. D. Rhoads, "Applications of artificial intelligence in clinical microbiology diagnostic testing," *Clinical Microbiology Newsletter*, vol. 42, no. 8, pp. 61–70, 2020.

[81] S. Al-Zain, D. Eleyan, and J. Garfield, "Automated user interface testing for web applications and testcomplete," in *Proceedings of the CUBE International Information Technology Conference*, pp. 350–354, 2012.

[82] H. Zhou, D. Wang, Y. Yu, and Z. Zhang, "Research progress of human–computer interaction technology based on gesture recognition," *Electronics*, vol. 12, no. 13, p. 2805, 2023.

[83] N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe, "Step: Deductive-algorithmic verification of reactive and real-time systems," in *Computer Aided Verification: 8th*

International Conference, CAV'96 New Brunswick, NJ, USA, July 31–August 3, 1996 Proceedings 8, pp. 415–418, Springer, 1996.

[84] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.

[85] D.-H. Vu, Y. Chiba, K. Yatake, and T. Aoki, "Checking the conformance of a promela design to its formal specification in event-b," in *Formal Techniques for Safety-Critical Systems: Third International Workshop, FTSCS 2014, Luxembourg, November 6-7, 2014. Revised Selected Papers 3*, pp. 110–126, Springer, 2015.

[86] D.-H. Vu, Y. Chiba, K. Yatake, and T. Aoki, "A framework for verifying the conformance of design to its formal specifications," *IEICE TRANSACTIONS on Information and Systems*, vol. 98, no. 6, pp. 1137–1149, 2015.

[87] D. H. Vu, Y. Chiba, K. Yatake, and T. Aoki, "Verifying osek/vdx os design using its formal specification," in *2016 10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pp. 81–88, IEEE, 2016.

[88] W.-P. De Roever and K. Engelhardt, *Data refinement: model-oriented proof methods and their comparison*. Cambridge University Press, 1998.

[89] A. Edmunds and M. Butler, "Linking event-b and concurrent object-oriented programs," *Electronic Notes in Theoretical Computer Science*, vol. 214, pp. 159–182, 2008.

[90] D. Carrington, "Vdm and the refinement calculus: a comparison of two systematic design methods," tech. rep., Citeseer, 1993.

[91] R.-J. Back, "Refinement calculus, part ii: Parallel and reactive programs," in *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness: REX Workshop, Mook, The Netherlands May 29–June 2, 1989 Proceedings*, pp. 67–93, Springer, 1990.

[92] C. Morgan, *Programming from specifications*. Prentice-Hall, Inc., 1990.

[93] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin, "Rodin: an open toolset for modelling and reasoning in

event-b," *International journal on software tools for technology transfer*, vol. 12, pp. 447–466, 2010.

[94] I. Dincă, "Multi-objective test suite optimization for event-b models," in *International Conference on Informatics Engineering and Information Science*, pp. 551–565, Springer, 2011.

[95] F. Ipate, A. Stefanescu, and I. Dinca, "Model learning and test generation using cover automata," *The Computer Journal*, vol. 58, no. 5, pp. 1140–1159, 2015.

[96] I. Dinca, F. Ipate, and A. Stefanescu, "Model learning and test generation for event-b decomposition," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pp. 539–553, Springer, 2012.

[97] T. S. Hoang and J.-R. Abrial, "Event-b decomposition for parallel programs," in *Abstract State Machines, Alloy, B and Z: Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings 2*, pp. 319–333, Springer, 2010.

[98] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, and T. Latvala, "Supporting reuse in event b development: modularisation approach," in *Abstract State Machines, Alloy, B and Z: Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings 2*, pp. 174–188, Springer, 2010.

[99] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, P. Väisänen, D. Ilic, and T. Latvala, "Verifying mode consistency for on-board satellite software," in *Computer Safety, Reliability, and Security: 29th International Conference, SAFECOMP 2010, Vienna, Austria, September 14-17, 2010. Proceedings 29*, pp. 126–141, Springer, 2010.

[100] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, and T. Latvala, "Developing mode-rich satellite software by refinement in event-b," *Science of Computer Programming*, vol. 78, no. 7, pp. 884–905, 2013.

[101] C. Snook and M. Butler, "Uml-b and event-b: an integration of languages and tools," in *The IASTED International Conference on Software Engineering - SE2008*, (Innsbruck, Austria), February 12-14 2008.

[102] F. Bernin, M. Butler, D. Cansell, S. Hallerstede, K. Kronlöf, A. Krupp, T. Lecomte, M. Lundell, O. Lundkvist, M. Marchetti, *et al.*, "The uml-b profile for formal systems modelling in uml," *UML-B specification for proven embedded systems design*, pp. 69–84, 2004.

[103] C. Snook and M. Butler, "Uml-b: A plug-in for the event-b tool set," *ABZ 2008: The First International Conference on Abstract State Machines, B and Z*, vol. 5238, 2008.

[104] C. Snook, V. Savicks, and M. Butler, "Verification of uml models by translation to uml-b," in *Formal Methods for Components and Objects: 9th International Symposium, FMCO 2010, Graz, Austria, November 29-December 1, 2010. Revised Papers 9*, pp. 251–266, Springer, 2012.

[105] T. S. Hoang, C. Snook, L. Ladenberger, and M. Butler, "Validating the requirements and design of a hemodialysis machine using iuml-b, bmotion studio, and co-simulation," in *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016, Proceedings 5*, pp. 360–375, Springer, 2016.

[106] M. Y. Said, M. Butler, and C. Snook, "A method of refinement in uml-b," *Software & Systems Modeling*, vol. 14, pp. 1557–1580, 2015.

[107] J. Magee and J. Kramer, *State models and java programs*. wiley Hoboken, 1999.

[108] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[109] R. Milner, *A calculus of communicating systems*. Springer, 1980.

[110] L. De Alfaro and T. A. Henzinger, "Interface automata," *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 5, pp. 109–120, 2001.

[111] A. Badica and C. Badica, "Fsp and fltl framework for specification and verification of middle-agents," *International Journal of Applied Mathematics and Computer Science*, vol. 21, no. 1, p. 9, 2011.

[112] K. K. Sheridan-Barbian, *A survey of real-time operating systems and virtualization solutions for space systems*. PhD thesis, Monterey, California: Naval Postgraduate School, 2015.

[113] G. Gigante and D. Pascarella, "Formal methods in avionic software certification: the do-178c perspective," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pp. 205–215, Springer, 2012.

[114] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[115] N. Walkinshaw, M. Roper, and M. Wood, "The java system dependence graph," in *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 55–64, IEEE, 2003.

[116] A. De Lucia, A. R. Fasolino, and M. Munro, "Understanding function behaviors through program slicing," in *WPC'96. 4th Workshop on Program Comprehension*, pp. 9–18, IEEE, 1996.

[117] T. Gyimóthy, A. Beszédes, and I. Forgács, "An efficient relevant slicing method for debugging," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 303–321, 1999.

[118] K. B. Gallagher, *Using program slicing in software maintenance*. University of Maryland, Baltimore County, 1990.

[119] V. M. Vedula, J. A. Abraham, and J. Bhadra, "Program slicing for hierarchical test generation," in *Proceedings 20th IEEE VLSI Test Symposium (VTS 2002)*, pp. 237–243, IEEE, 2002.

[120] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[121] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pp. 2123–2138, 2018.

[122] L. Project, "Libfuzzer – a library for coverage-guided fuzz testing." `https://llvm.org/docs/LibFuzzer.html`.

[123] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[124] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pp. 2155–2168, 2017.

[125] JetBrains, "Intellij idea: The capable and ergonomic java ide." `https://www.jetbrains.com/idea/`.

[126] E. Foundation, "Eclipse foundation." `https://www.eclipse.org/`.

[127] JavaSlicer, "https://github.com/mistupv/javaslicer."

[128] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact.," *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[129] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, H. N. A. Hamed, and M. D. M. Suffian, "Test case prioritization using firefly algorithm for software testing," *IEEE access*, vol. 7, pp. 132360–132373, 2019.

[130] H. K. Wright, M. Kim, and D. E. Perry, "Validity concerns in software engineering research," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pp. 411–414, November 2010.

[131] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 273–282, 2005.

[132] A. Dutta, R. Jain, S. Gupta, and R. Mall, "Fault localization using a weighted function dependency graph," in *2019 International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering (QR2MSE)*, pp. 839–846, IEEE, 2019.

[133] T. Ye, L. Zhang, L. Wang, and X. Li, "An empirical study on detecting and fixing buffer overflow bugs," in *2016 IEEE International*

Conference on Software Testing, Verification and Validation (ICST), pp. 91–101, IEEE, 2016.

[134] A. Summers, "Common weakness enumeration." https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html, 2022.

[135] B. W. Boehm, "Software engineering," *IEEE Trans. Computers*, vol. 25, no. 12, pp. 1226–1241, 1976.

[136] A. Bhat and S. Quadri, "Equivalence class partitioning and boundary value analysis-a review," in *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, pp. 1557–1562, IEEE, 2015.

[137] M. Kellogg, V. Dort, S. Millstein, and M. D. Ernst, "Lightweight verification of array indexing," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 3–14, July 2018.

[138] S. Developers, "Spotbugs." https://github.com/spotbugs/spotbugs, 2024. Accessed: [2024.4.25].

[139] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pp. 234–245, May 2002.

[140] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for java," in *15th International symposium on software reliability engineering*, pp. 245–256, IEEE, 2004.

Chapter 8

Publication List of the Author

- Conference

  (1) Lei Rao, Shaoying Liu, and Ai Liu. "Testing Program Segments to Detect Runtime Exceptions in Java." International Workshop on Structured Object-Oriented Formal Language and Method. Cham: Springer International Publishing, 2022: 93-105.

  (2) Lei Rao, Shaoying Liu. Program Segment Testing for Software Fault Prevention. Proceedings of the Software Engineering Symposium 2021, 2021: 191-195.

- Journal

  (1) Lei Rao, Shaoying Liu, and Ai Liu. "Testing Program Segments to Detect Software Faults during Programming." International Journal of Performability Engineering, 17 (11), 2021. (Scopus)

  (2) Lei Rao, Shaoying Liu, and Han Peng. "An integrated formal method combining labeled transition system and Event-B for system model refinement." IEEE Access 10 (2022): 13089-13102. (SCI)

  (3) Lei Rao, Shaoying Liu, and Ai Liu. "Program Segment Testing for Human-Machine Pair Programming". International Journal of Software Engineering and Knowledge Engineering. (Has been accepted for publication) (SCI)

  (4) Han Peng, Lei Rao, et al. "A Time Refinement Framework Based on iUML-B State Machine". Scientific Programming, 2021: 1-21. (SCI)