A MODEL-BASED TESTING TECHNIQUE FOR COMPONENT-BASED REAL-TIME EMBEDDED SYSTEMS

by

Jing Guan
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

____________________	Dr. Jeff Offutt, Dissertation Director
____________________	Dr. Daniel A. Menasce, Committee Member
____________________	Dr. Sam Malek, Committee Member
____________________	Dr. Kris Gaj, Committee Member
____________________	Dr. Stephen Nash, Senior Associate Dean
____________________	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering
Date: ________________	Summer Semester 2015 George Mason University Fairfax, VA

A Model-Based Testing Technique for Component-Based Real-Time Embedded Systems

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Jing Guan
Master of Science
University of Maryland at College Park, 1999
Bachelor of Science
Northwestern Polytechnical University, 1997

Director: Dr. Jeff Offutt, Professor
Department of Computer Science

Summer Semester 2015
George Mason University
Fairfax, VA

Dedication

I dedicate this dissertation to my dearest children William Wang and Patrick Wang for your understanding. You both have grown up along with my PhD study. Your sweetness makes everyday filled with sunshine. I dedicate this dissertation to my loving parents Fengqin Wang and Decai Guan for your encouragement, trust and support. Your unconditional love gives me strength to work hard and always try my best endeavor on everything. I dedicate this dissertation to my husband Tao Wang for your patience. Your optimistic character helped me through many difficult times in this journey. You motivated me to start, continue and finish my PhD study.

Acknowledgments

My biggeset thanks goes to my dear advisor, Dr. Jeff Offutt. This dessertation could not have been started and completed without him. At my very first class in George Mason, he told me "You are in the right place!". That sentence changed my whole life. He not only taught me how to do a research and write research papers but also helped me find the best of myself. He always believes in me and gives me confidence when I feel I am not good enough to move forward. I would like to thank him for his patience and his encouragement to guide me through every gloomy moment. You are the person who truly understands how hard it's been for me and offers your support at any times.

I would like to thank all my other committee members for offering their time and for delivering valuable knowledge and insight. I would like express my greatest respect to Dr. Daniel Menasce for not only providing me valuable suggestions on my research and dissertation, but also guiding me through the darkest time. I will never be able to describe how much your encouragement meant to me. Your profound knowledge and noble character will influnce and benefit me forever! I sincerely thank Dr. Sam Malek for challenging me so I can come to a whole new level of the research. Your suggestions not only opened my eyes on the academic area but also made me a stonger person. As a result, this journey becomes more unforgettable and worthy. I truely appreciated Dr. Kris Gaj for stepping up to serve as my committee member at the time when I needed it the most.

I would also like to thank my company Lockheed Martin for providing me all the valuable resources and financial support. I have the best colleague I could ask for. They've shown their support for my PhD study in all kinds of ways.

Table of Contents

List of Tables

List of Figures

Abstract

A MODEL-BASED TESTING TECHNIQUE FOR
COMPONENT-BASED REAL-TIME EMBEDDED SYSTEMS

Jing Guan, Ph.D.

George Mason University, 2015

Dissertation Director: Dr. Jeff Offutt

The growing complexity of modern real-time embedded systems makes component-based software engineering (CBSE) technology more desirable. Although many ideas have been proposed for building component-based real-time embedded software, techniques for testing component-based real-time systems have not been well developed. A typical component-based embedded system consists of multiple user tasks, as well as hardware, middleware and software layers. Interaction problems between different components can cause system failures in field applications. The challenges not only come from the integration of multiple components through their interfaces, but also include the composition of extra-functional properties. In an embedded system, extra-functional requirements are as important as functional requirements. A real-time embedded system needs to achieve its functionality under the constraints caused by its extra-functional properties. Since the time at which the system actions take place is important, correct functional behavior with regard to timing properties is essential to real-time embedded systems. Therefore, this research is intended to help detect both functional and temporal faults during the integration of component-based real-time embedded software.

This dissertation presents a test model that depicts both inter-component and intra-component relationships in component-based real-time embedded software and indentifies key test elements. The test model is realized using a family of graph-based test models in which not only are the functional interactions and the dependence relationships illustrated, but also the time-dependent interaction among components, are illustrated. Time dependednt behavior is modelled by means of timers and clocks. A notion of timed events define constraints in timed scenarios. I use the graph-based test model to develop a novel family of test adequacy criteria that help generate effective test cases. In this test model, I use timing-constraint to specify the duration and execution cost of inter-component and intra-component interactions as well as state transitions and activities. I also present new algorithms to facilitate automate generation of the test cases. To increase the observability of system behavior, I instrument related operations to generate trace data including task id, operation, time stamp, and execution state from program execution, where the dynamic information gathered is used to check against the expected results. The experiments showed that the proposed approach effectively detected various kinds of integration faults and optimized the balance between budget and quality in an industrial product software testing.

Chapter 1: Introduction

1.1 General Introduction

The increased complexity of real-time embedded systems leads to increasing demands with respect to software engineering. This complexity makes the development of such systems very expensive. To reduce development costs, component-based modeling is increasingly being used to develop for embedded systems. It is expected to bring several advantages to embedded systems, such as rapid development time, the ability to re-use existing components, and the ability to compose sophisticated software [28].

Component-based technology has been extensively used for many years to develop software systems in desktop environments, office applications, and web-based distributed application [8]. The advantages are achieved by facilitating the reuse of components and their architecture, raising the level of abstraction for software construction, and sharing standardized services. Real-time embedded systems have requirements not found in most desktop systems such as timeliness requirements and resource efficiency. So, it is more difficult to adapt component-based software engineering (CBSE) to real-time embedded system than desktop systems [35].

The components may have been written in different programming languages, execute in various operational platforms, and be distributed across vast geographic distances. Some components may be developed in-house, while others may be third party or commercial off-the-shelf components (COTS) [86]. A benefit of this kind of design is that software components can be analyzed and tested independently. At the same time, this independence of components means that significant issues cannot be addressed until full system integration testing. As a result, complex behaviors are observed when components are integrated and several kinds of faults may arise during integration. Thus, testing each component independently does not eliminate the need for system integration testing. Many possible interactions between components need to be tested to ensure the correct functionality of the system [36].

As the size and complexity of software systems increase, problems stemming from the design and specification of overall real-time embedded system structure become more significant. The result is that the way groups of components are arranged, connected, and structured is crucial to the success of software projects. Problems in the interactions can affect the overall system development and the cost and consequences could be severe [45].

Testability for real-time embedded software is a challenging problem due to the low observability and controllability of embedded systems [80]. Embedded software often gets its inputs from hardware and generates outputs for hardware rather than for users, so it is more difficult to control and observe. CBSE increases the complexity, which affects the observability and controllability.

Real-time embedded platforms are also complex integrated systems where multiple real-time tasks execute in multi-tasking environments. In a component-based embedded system, each individual component may run several tasks, so the number of messages between tasks across components is dramatically increased [57]. When integrating these components together, unexpected results may occur if interactions between components are not fully tested. Interactions between components may give rise to subtle errors that could be hard to detect.

Additionally, any component in a system can be updated at any time, so the dependencies between the existing and the modified components can change dynamically. Therefore, adequate and extendable testing methods for continuous validation of integrated component systems is essential to the success of a component-based real-time embedded system. Despite its value, component-based modeling introduces new problems when testing real-time embedded software systems [50]. This dissertation describes research to develop a new systematic software testing technique to thoroughly examine the component interaction behavior during system integration testing activities. The technique is based on models created from the software architectures and design specification, which specify the primary components, interfaces, dependences, and behaviors of software systems.

The faults that occur during component integration are different from those found during unit and component testing [14]. Faults that cannot be detected during unit, module, and component testing are often faults in the way the software components are structured or in how they communicate. Correctly implementing interactions can be difficult because, unlike the components of a software system, the interactions are rarely isolated in a single, independent runtime structure. Instead, interaction is typically spread across the components involved in the interaction. To make matters more difficult, this interaction code is often tightly integrated with the code associated with the component's functionality.

A fundamental problem of test data generation is that the only way to ensure complete correctness is to test with all possible inputs [64]. However, the number of possible inputs to a given program is effectively infinite, so testers must accept partial results by finding a finite number of test cases that will provide a high level of confidence that the program's behavior is satisfactory. Software architecture can provide a framework for understanding system components and their interrelationships, especially those attributes that are consistent across time and implementations. At the software architecture level, software systems are presented at a high level of abstraction where a software system is viewed as a set of compositional components, interactions among these components, and the configuration of the system. Implementation details are suppressed and the independence of system components is increased. The software architecture specifications provide a description of the software system that could be used to generate tests at the integration level. This lets developers abstract away unnecessary details and focus on the assignment of software components, interfaces to hardware components, and how they model their important aspects throughout and especially early in development.

A software architecture design specification captures the system level details of components, interactions and context. For example, the Unified Modeling Language (UML) provides a variety of diagramming notations for capturing architecture and design information from different perspectives [6] [38]. In a component level sequence diagram, the interaction between components is defined explicitly. In recent years, researchers have invented ways to use UML models as a source of information in software testing [54] [86] [66] [12]. Many UML design artifacts have been used in different ways to perform different kinds of testing. For instance, UML statecharts have been used to perform unit testing [78] [67] [20], and interaction diagrams (collaboration and sequence diagrams) have been used to test class interactions [76] [59] [10] [34].

A software architecture design specification precisely describes how the software is expected to behave in a high level way that can easily be used by automated methods.

Evaluating and testing software systems based on software architecture and design specifications can allow tests to be created earlier in the development process, therefore substantially reducing the costs of any problems and errors. Currently, there is a lack of testing techniques for testing component-based real-time embedded system at the software architecture level. In this dissertation, I present research in the area of software architecture and design based testing to create a general testing technique at this level.

1.2 Goals and Scope of This Research

Testing integration of components becomes an important activity in CBSE [69]. Testing methods for component-based embedded real-time software is a relatively new research field [60]. The informality of the usual testing methods makes it difficult to measure the quality of testing, leads to a lack of repeatability in the process and results, and means that the tester cannot be confident in the efficiency of the testing. As a result, errors may severely impact the software in ways that are costly to fix and causes delays and failures when deploying systems in the real world. To improve both functional and real-time testing of component-based real-time embeded software, tester need a time-dependent interaction model between the embedded real-time components.

The objective of this research is to provide a new model-based testing approach to help component-based real-time embedded software testing become more structured. In turn, this will lead to software systems that have fewer faults.

1.2.1 Problem Statement

Component-based real-time embedded system development brings new challenges to traditional embedded engineering. In particular, the current method of testing this type of software is inadequate. We need stronger techniques to test this type of component-based real-time embedded systems.

Manual and specification testing techniques do not work effectively for component-based real-time embedded systems since they cannot detect many of the faults that are novel to this type of software system.

To solve the problem, we need to consider the unique characteristics of this type of software. This research project has addressed the problem of finding a better way to test component-based real-time embedded software.

1.2.2 Thesis Statement

This thesis addresses the problem of testing component-based real-time embedded software by inventing formal test criteria for software architecture level integration testing in component-based real-time embedded systems. The research includes techniques to design effective test cases based on a novel family of test adequacy criteria and analyze results to detect both functional and temporal faults during the integration of component-based real-time embedded software.

1.2.3 Scope of Research

This research focuses on studying efficient and practical solutions for modeling and deriving integration tests for component-based real-time embedded software. Due to the tight relationships between its functionality and the constraints caused by its extra-functional properties, component integration testing in an embedded system cannot be solely achieved by examining components' behavior through their interfaces. Among extra-functional properties, this dissertation only focuses on the timing requirements and software/hardware dependencies. In this research, periodic tasks are limited to periodic events triggered by periodic timers controlled by components; Aperiodic tasks are activated by external events. Other embedded constrained resources, such as resource allocation, amount of memory, processor speed and power consumption, are not in the scope of this research. Task scheduling, tsk priorities that change over time and clock-based deadlines are also out of scope of this research.

This dissertation makes the following contributions:

1. A technique that enhances the integration testing of components by accounting for states of collaborating components in an interaction. This is important as interactions may trigger correct behavior for certain states and not for others. To achieve such an objective the proposed technique includes a novel intermediate test model called the Component-based Real-time Embedded Architecture-based Test Graph (CREATEG) model from component level sequence diagrams and state diagrams of the components involved in the component interactions.

2. Timing notations to specify the duration and execution cost of inter-component and intra-component interactions as well as state transitions and activities. Timing characteristics of the execution infrastructure are taken into account in the test model. A functional behavior determinism with regard to time is realized.

3. Test criteria for generating integration tests from the CREATEG models. The CREATEG models all possible paths for component state transitions that a message sequence may trigger. These criteria can be used both to guide the architecture designers and to help the testers generate meaningful and effective test cases.

4. Timed input values and timed output results to utilize timed message sequence. Trace information is unified in the program instrumentation to include task id, operation,

time stamp, and execution state. Non-deterministic outputs, caused by real-time and concurrency features in embedded systems, are better observed and analyzed.

5. Algorithms to automatically create test requirements. These algorithms are based on CREATEG models.

6. An experimental tool to generate test cases automatically from CREATEG models. The test generator uses CREATEG to generate test paths whose test cases attempt to detect faults that may arise due to invalid component states during interactions.

7. Empirical validation of the ideas. This dissertation reports on an empirical study to assess the effectiveness and cost of the test adequacy criteria based on the CREATEG test model. The architecture-based testing technique was applied to an industrial software system. The results are compared with results from using two other testing methods. The goal of this process is to determine whether the new testing technique can effectively detect faults.

1.3 Solution Strategy

To find solutions to our research problems, I first discuss issues of testing component-based real-time embedded systems, then list a set of properties to test based on software architecture design artifacts and models. This helps us decide what to test when testing at the integration level. Then we define architecture relations at the architectural level, and formally define these relations. Three graphical representations are introduced for testers to visualize the testing technique and for possible analysis and simulations. Test criteria based on the architecture relations are then presented. These criteria are formally defined. We then apply the technique to an industrial system, and develop algorithms to transform the architecture and design models to graphical representations. An empirical evaluation of the technique is carried out using an industrial software system, and its results are discussed.

1.4 Unique Contributions of the Research

Major contributions of this dissertation are:

1. New test criteria for testing component-based real-time embedded software systems
2. A novel formal model of architecture relations in component-based real-time embedded software systems
3. A new architecture modeling technique based on UML
4. A timed model with timing notations and constraints
5. Formal definitions of transformation rules for translating UML models to architecture models
6. A unified form of timed instrumentation for increasing observability of component interactions
7. A prototype tool for generating test cases based on architecture models

1.5 Dissertation Organization

Chapter 2 describes background. Chapter 3 reviews issues and related research. Chapter 4 discusses the architecture based testing technique for general component-based real-time embedded software. Chapter 5 presents a proof-of-concept-tool. Chapter 6 discusses an empirical validation of the technique. Finally, Chapter 7 concludes the dissertation research and discusses future research directions.

Chapter 2: Background

This chapter provides background information and concepts required to understand the work in this dissertation. Section 2.1 describes background on embedded systems, including its architecture and properties. Section 2.2 discusses embedded software testing. Section 2.3 describes component-based real-time embedded software. Section 2.4 and Section 2.5 discuss component-based embedded software architecture and design.

2.1 Real-Time Embedded System

An *embedded system* is a special-purpose computer system built into a larger device [71]. Usually there is no disk drive, keyboard or screen. Broekman and Notenboom [22] define embedded systems as a generic term for a broad range of systems covering, for example, cellular phones, railway signal systems, hearing aids, and missile tracking systems. They specify that all embedded systems have a common feature in that they interact with the real physical world, controlling hardware. The term embedded system can encompass a variety of devices and systems.

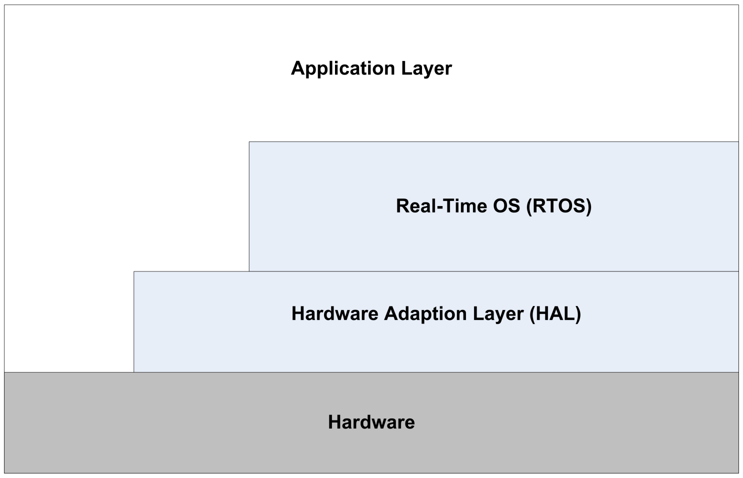


Figure 2.1: Structure of an embedded system

Embedded systems are designed to perform specific tasks in a particular computational environment consisting of software and hardware components. Figure 2.1 illustrates the typical structure of such a system in terms of four layers.

An application layer consists of software that satisfies application requirements; applications layers that use services from underlying layers, including the Real-Time Operating System (RTOS) and Hardware Adaptation Layer (HAL).

An RTOS consists of task management, interrupt handling, inter-task communication, and memory management facilities [71] that allow developers to create embedded system applications that meet functional requirements and deadlines using provided libraries and APIs.

The hardware adaptation layer is a runtime system that manages device drivers and provides hardware interfaces to higher level software systems–applications and RTOSs.

Interactions between different layers play an essential role in embedded system application execution. An application layer consists of multiple user tasks that execute in parallel, sharing common resources like CPU, bus, memory, device, and global variables. Interactions between application layers and lower layers, and interactions among the various user tasks that are initiated by the application layer, flow the information created in one layer to others for processing. Faults in such interactions could result in execution anomalies.

Multitasking is one of the most important aspects in embedded system design, and is managed by the RTOS. Due to thread inter-leaving, embedded systems employing multiple tasks can have non-deterministic output, which complicates the determination of expected outputs for a given input.

Embedded system often functions under real-time constraints, what means the request need be completed within a required time interval from the triggering event [25]. In the real-time domain, the temporal behavior [48] is as important as the functional behavior. Embedded systems can be classified into hard real-time and soft real-time depending on their requirements. Their main difference lies in the cost or penalty for missing their deadlines. Hard real-time embedded systems have strict temporal requirements, in which failure to meet a single deadline may lead to catastrophic outcomes. On the other hand, soft real-time embedded systems are not required to satisfy hard real-time constraints; missing deadlines lead to performance degradation.

2.2 Testing Embedded Software

According to statistical studies [37], software accounts for as much as 80% of the functionalities in embedded real-time systems such as home appliances, information appliances, personal assistants, telecommunication gadgets, and transportation facilities. Software is also much more complex than hardware due to its inherent flexibility. It is often found that an on-market real-time embedded system fails due to some simple software glitches, which could have been avoided if the software was thoroughly verified. All these facts show that verifying the correctness of software is a demanding and important issue in the design phase of an embedded real-time system [44].

These features greatly affect software testability and measurability in embedded systems. The term *testability* in software testing can be considered from various viewpoints [12] [17] [23]. While some consider the architectural viewpoints [47] [51], few describe techniques for more effective design for testability at the architectural level [11] [13]. However, increasing testability is commonly identified as an important goal in software testing research [27].

In embedded software systems, two main viewpoints of testability are considered from the architectural viewpoint: controllability and observability [17]. To test a component, we must be able to control its input, behavior and internal state. To see how this input has been processed, we must be able to observe the components output, behavior and internal states. Finally, the system control mechanisms and observed data must be combined to form meaningful test cases.

Embedded software often generates outputs for hardware to function rather than for users to interact with, so the observability is significantly low. It is easy to control software input values that are entered from a keyboard. But an embedded program that gets its inputs from hardware is more difficult to control. Testability is an important factor in embedded real-time systems, especially when the systems have additional properties that increase the complexity.

Faults in embedded systems can produce effects on program behavior or state that, in the context of particular test executions, do not propagate to output, but do surface later in the field. To detect internal program faults when testing embedded systems is the primary focus of this work.

2.3 Component-based Real-time Embedded Software

The embedded systems industry is under competitive pressure to continually shorten its time-to-market, increase product differentiation, and at the same time offer more customer value. As a result, (i) embedded systems are becoming increasingly software intensive, and (ii) individual components integrate increasing functionality over different projects and reuse cycles. Integrating more functions into a single component gives rise to increasingly varying behavior.

During the last decade advances have been made in component-based development for desktop and internet applications. Embedded and real-time systems have requirements not found in desktop systems, including real-time requirements and resource efficiency. This is one reason why embedded and real-time systems have more difficulties adapting to component-based software engineering (CBSE) than desktop systems [35]. A few de-facto standards have completely transformed the way such software is developed. These standards are mainly Microsoft's .NET [1], SUN's Enterprise Java Beans [2] and OMG's CORBA Component Model (CCM) [3]. Component models for embedded systems are usually designed with very domain specific requirements in mind [63]. There is a large set of different component technologies that approach different problems in different ways such as the SCA-based Component Framework for Software Defined Radio [50] and many more. For embedded systems it seems difficult to define de-facto standards due to highly diverging requirements in different industrial segments [27].

Software Defined Radio (SDR), developed with the Software Communications Architecture (SCA), is an example of such an effort. SDR refers to reconfigurable or reprogrammable radios that can have different functionality with the same hardware. Because the functionality is defined in software, a new technology can easily be implemented on a radio by updating its software. So a radio can be built to meet the need for continuously changing technology. In an SDR, multiple waveforms can be implemented in software using the same hardware. One software defined radio can communicate with many different radios, with only a change in software parameters. This increases interoperability among different military units, emergency units, and coalition forces. Also new technologies can be adapted to quickly, easily, and with a much lower cost than with the traditional method.

SCA is a common, well-defined open architecture. It is used to build radios that support operations in a wide variety of domains without losing the ability to communicate with each other. It can help radio vendors improve interoperability by providing the ability to share waveform software between radios, and reduce development time through software reuse. This architecture also facilitates scalability and technology insertion.

As an emerging technology in embedded software development, the SCA presents a new paradigm, and it affects the entire embedded software development cycle including analysis, specification, design, implementation, verification, validation and maintenance. However, it introduces a new environment to embedded communication engineers, filled with system software concepts such as object oriented programming, portable operating system interfaces, and middleware using the Common Object Request Broker Architecture (CORBA) [4].

2.4 Component-based Real-time Embedded Software Architecture

Software Communications Architecture (SCA) [7] is a component-based software architecture specifically designed for real-time embedded communications devices. The SCA structure is composed of an application layer and an operation environment (OE) layer. The software architecture detailed view is shown in Figure 2.2.

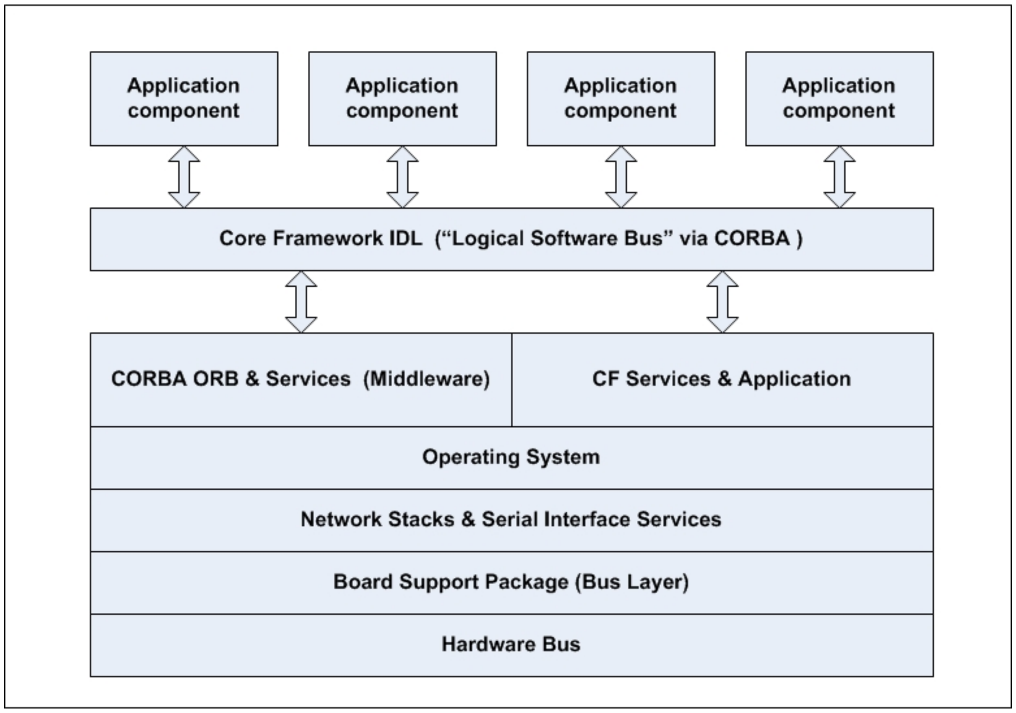

Application component
Application component
Application component
Application component
Core Framework IDL ("Logical Software Bus" via CORBA)
CORBA ORB & Services (Middleware)
CF Services & Application
Operating System
Network Stacks & Serial Interface Services
Board Support Package (Bus Layer)
Hardware Bus

Figure 2.2: SCA software architecture detailed view

The OE consists of a Core Framework (CF), a CORBA middleware and a POSIX-based Operating System (OS). Since the SCA uses the CORBA middleware, application programs are basically composed of CORBA objects that conform to the SCA core framework. The SCA core framework is composed of the specification of interfaces and a domain profile. A domain profile is composed of XML descriptor files that describe the hardware and software configuration information of a SCA system domain. The OE specifies the services and interfaces that the applications use from the environment. The interfaces are defined by using the CORBA Interface Definition Language (IDL), and graphical representations are made by using UML [38]. Figure 2.3 shows the relationships between the OS, the application, and the OE.

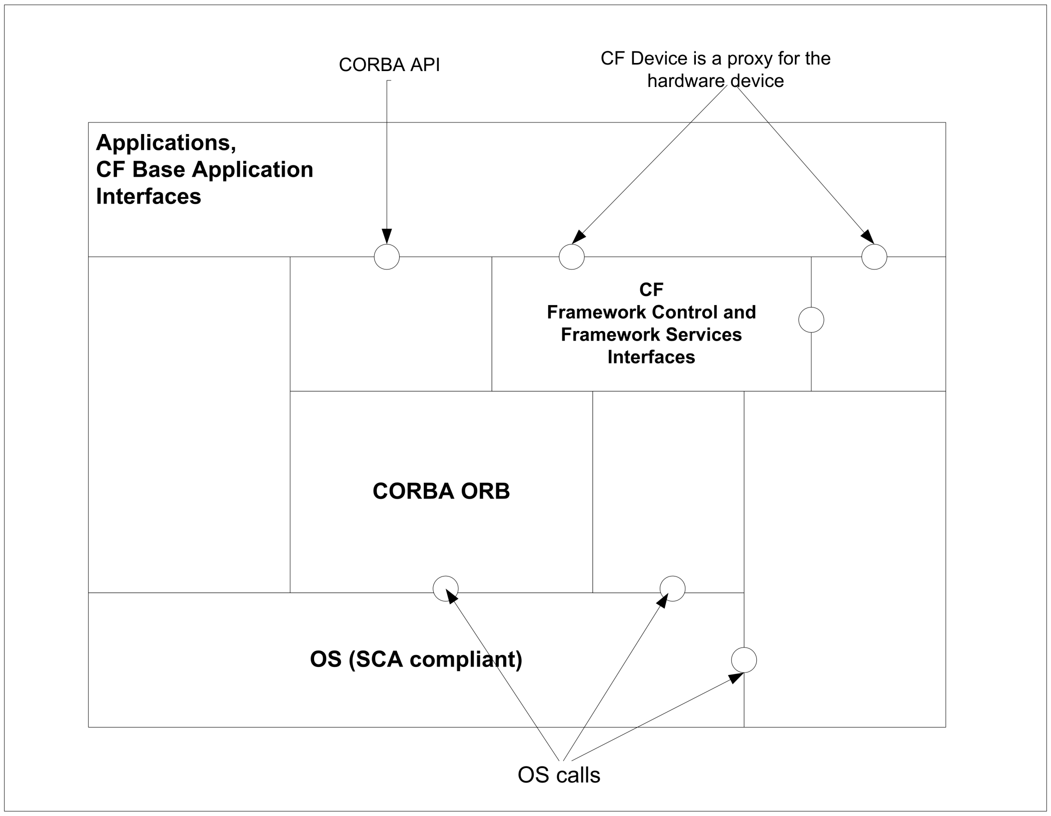

CORBA API
CF Device is a proxy for the hardware device
Applications, CF Base Application Interfaces
CF Framework Control and Framework Services Interfaces
CORBA ORB
OS (SCA compliant)
OS calls

Figure 2.3: Relationship between SCA components

Any component on a radio can be replaced or upgraded, and the download process can be made transparent to the user. In the SCA context, a radio application is known as a *waveform*, which is defined as the set of transformations applied to information that is transmitted over the air and the corresponding set of transformations to convert received signals back to their information content. The core framework defines a common mechanism to manage and control waveforms and their components. Therefore, components can come from different sources and still use the same mechanisms to be deployed, connected, and managed.

Port provides a specialized connectivity to a component. It is used to set up and tear down connections between application components in the CF domain. ***Port*** is a logical element that enables components to exchange data. Ports are classified into ***Uses*** ports (clients) and ***Provides*** ports (servers). Provides port of a component is used to retrieve an object reference for a server object contained in the component. ***Uses*** port of a component is used to retrieve an object reference for a proxy object connected with a server object contained in another component. ***Port*** interface also provides components with connect and disconnect functionalities, which are necessary to assemble waveforms. Figure 2.4 illustrates the connection of components via ports in SCA model.

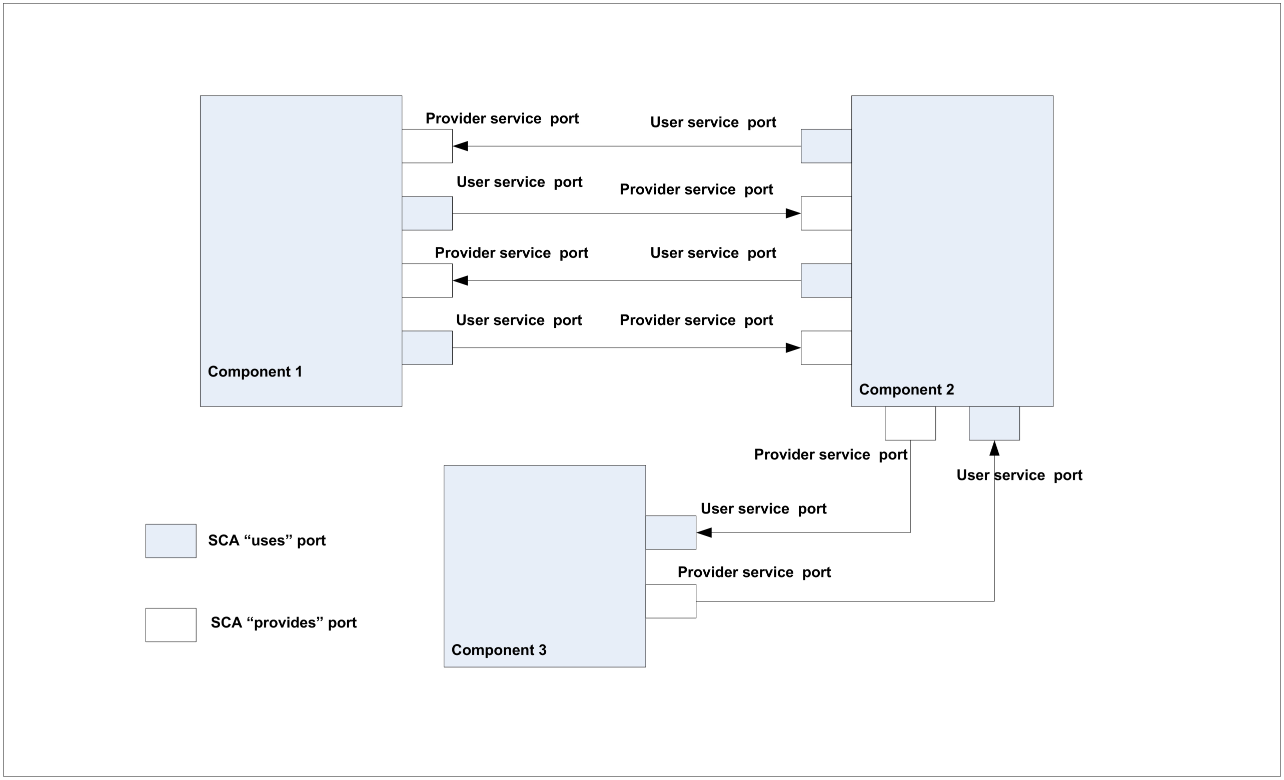


Figure 2.4: Connection of components via ports

Significant efforts are being carried out to facilitate SCA-based SDR software development on integrated development environments, reusable software modules, and implementations of software architectures. For example, the Open-Source SCA Implementation::Embedded (OSSIE) [82], developed at Virginia Tech for research and education, is an open SCA implementation that can reduce the entry cost of SCA development and training.

But less attention has been brought to finding effective and efficient testing strategies to make such systems more robust and reliable.

2.5 Component-based Real-time Embedded Software Design Models

The Unified Modeling Language (UML 2.0) [6] [38] offers a great opportunity to describe component-based embedded systems. It provides constructs to deal with varying levels of modeling abstraction to visualize and specify both the static and dynamic aspects of systems. And with the stereotypes, tagged values, and constraints, the semantics of model elements can be customized and extended. The Object Management Group (OMG) has proposed the Model Driven Architecture (MDA) approach, which aims to allow developers to create systems entirely with models. It provides a set of guidelines to structure specifications expressed as models. Therefore, it is useful and significant to combine the MDA approach and UML models with the component technique to develop software for embedded systems [59].

A model is a formal specification of the function, structure and behavior of a system within a given context, and from a specific point of view (or reference point). A model is often represented by a combination of drawings and text, typically using a formal notation such as UML, augmented where appropriate with natural language expressions.

MDA itself is not a new OMG specification but rather an approach to software development that is enabled by existing OMG specifications such as the Unified Modeling Language (UML), the Meta Object Facility (MOF), and the Common Warehouse Metamodel (CWM).

Chapter 3: Related Work

3.1 Related Work

Although a great deal of research has addressed the overall process of component-based software engineering (CBSE) on requirements engineering, design and evaluations, we do not have as much reserach on testing CBSE. Testing CBS is a challenging area of research. Existing knowledge in this field shows that CBSE introduces new problems for testing and maintaining software systems and we need new ways to validate software components, especially when they are integrated into new enviroments [42] [83] [87].

There are a number of component-based testing methods and techniques which have different paradigms, characteristics and perspectives. This section comprehensively reviews research work related, to provide a key foundation of the literature review. It also identifies the important problems and limitations in the existing component-based integration testing techniques.

Liang et al. [42] proposed a testing technique that is based on analysis of component-based systems from component-provider and component-user perspectives. The technique makes use of complete information from components for which source code is available and partial information from those for which source code is not available. Their approach separated the testing of the component-provider from the testing of the component-user, so it presented two different techniques for each category. Valentini et al. [79] developed a framework based on contract-checkers. It verifies the information between the component producer and the user. This technique was leveraged by Zheng and Bundell [90] with UML-based testing at the modeling level to design model-level test contracts.

Machado et al. [61] presented a UML based approach to integration testing using UML diagrams including the Object Constraint Language. Zheng and Bundel [89] further extended the work to develop a model-based approach using three techniques, scenario-based, contract-based and component test mapping. Hartmann [43] presented a design-based testing approach to generate test inputs from UML state machines. It models the behavior of each component, specifies component interactions, and annotates the state machines with test requirements to construct a global behavioral model of the composed statecharts. Then, test cases are automatically derived from the annotated statecharts and global behavioral model, and executed to verify component conformance behavior. State machines of individual components are combined and then used to design tests.

Briand et al. have published several research papers on state-base testing with UML statechart diagrams [20] [21] [70]. They proposed to use class diagrams, collaboration diagrams, or OCL to derive test requirements [20]. They also proposed a methodology to automate the derivation of test cases from UML statechart diagrams for a given set of transition test sequences [21]. Their results show that, in most cases, state-based testing techniques are not likely to be sufficient by themselves to detect most of the faults present in the code, and they need to be complemented with other testing methods.

The above approaches use only one kind of behavioral UML model for test generation, either sequence diagrams or state machines. The approach in this dissertation is novel in that it combines the information from component level UML sequence diagrams and statecharts to derive a graph-based test model for the purposes of test input generation.

Wu et al. [87] investigated faults that can be identified at the integration of components. They presented a test model that depicts a generic infrastructure of component based systems and identified key test elements. A *Component Interaction Graph* is generated from the implementation, in which the interactions and the dependence relationships among components are illustrated. Test adequacy criteria were developed to cover context dependence relationship and content dependence relationship.

While Wu's test elements and test criteria are useful to test component-based software, their work is in the stage of approach development. This paper does not discuss and give practical ways on how to use their approach to generate actual test cases for component-based testing. Their test model mainly illustrates the context/content-dependence relationships defined in the paper. Additional work is required to effectively drive test generation from the test model.

In addition, the authors made several assumptions in their work, including: (i) assuming that each individual component has been adequately tested by the component providers when testing component-based software; (ii) assuming that each interface only includes one operation, and the references to the interfaces and to the operation are identical. These assumptions imply that their work considers only some simplified situations, which could have limitations in applying their approach to actual component-based testing practice.

From the above survey, we note that different kinds of UML diagrams have been used for software testing from different perspectives. UML state charts have been widely used to test the state-based behavior of software. Similarly, UML interaction diagrams have been used for integration testing. However, existing approaches do not focus on exercising the composition behavior of interacting components. More specifically, none of the above papers discuss testing by integrating UML interaction and statechart diagrams to uncover component interaction faults.

The approach in this dissertation uses UML statecharts and sequence diagrams from software architecture and design models to generate an intermediate model, the Component-based Real-time Embedded Architecture-based Test Model (CREATEG), and applies different coverage criteria based on the CREATEG graph representation. We use sequence diagrams to determine the order of messages between components.

There has been a great deal of work on using dataflow-based testing approaches to test interactions among system components [86] [56] [45] [68]. However, none of this work has addressed problems in testing embedded systems or studied the use of such algorithms on these systems. While we employ analyses similar to those used in these papers, we direct them at specific interactions within embedded systems.

Integration testing based on scenarios using finite state machines is discussed by Li et al. [55]. The work was furthered extened by Bouaziz and Berrada to test real time component-based systems [18] .

In an embedded system, extra-functional requirements are as important as functional requirements. A real-time embedded system needs to achieve its functionality under the constraints caused by its extra-functional properties. Therefore, they should be considered while testing embedded software behavior. Due to this specialty, the adoption of the component-based testing approach to the embedded system has encountered difficulties. For example, an important part of extra-functional characteristics of many embedded systems is timing requirements. It is difficult to describe timing requirements in models traditionally used in component-based design. Also, the tight integration between hardware and software makes it hard to model and implement software separately from hardware. Component integration testing in an embedded system cannot be solely achieved by examining components' behaviors through their interfaces. When modeling real-time systems, timing aspects and constraints become essential. Testing component-based real-time embedded systems is even more challenging than testing untimed reactive systems.

Few component-based testing models incorporate component extra-functional behavioral aspects in their frameworks.

Bouaziz and Berrada [19] proposed an aproach to model and test component-based real-time systems. To avoid constructing an entire system, they seperate the individual behavior of components from their interactions. They use a particular component called the assembly controller to model intra-component interactions, and only test relevant behaviors related to intra-component synchronizations. An assembly controller is a particular type of Timed Inout/Output Automata (TIOA) used to restrict the overall behavior of the composite system to ensure a correct interaction between components.

The most popular approaches for specifying real-time systems are based on Timed Automata [60]. It is a graph containing a finite set of nodes and a finite set of labelled edges extended with clocks. UPPAAL is a commonly used automatic verification tool for timed automata [46]. It is very good at modelling by providing a simulator and a well-developed

GUI, but the tool provides only binary synchronization between processes and can only verify reachability properties. Larsen et al. [53] created a number of small academic specifications and implementations to evaluate an online testing tool named UPPAAL-TRON for real-time systems [52]. An embedded system industrial case has been evaluated by Mikucionis et al. [62]. They found that real-time online testing is an effective way to detect discrepancies between the model and the implementation in practice. However, large and very non-deterministic models can run into state explosions, making it problematic to update the state-set in real-time. This may limit the granularity of time constraints that can be checked in real-time. All the above approaches are formal verification methods to prove conformance with a predefined specification. The goal is to check whether an extracted model satisfies a certain specification. My test method, in contrast, defines input data to the object program and observes the reactions of the program. The goal of my testing is to find cases where the software reactions do not meet its expected results.

There has also been research on component-based software engineering for embedded systems such as [26], which focused on embedded software. There has been work on using informal specifications to test embedded systems focusing on the application layer. Tsai et al. [75] presented an approach to test embedded applications using class diagrams and state machines. Cunning and Rozenblit [29] generated test cases from finite state machine models built from specifications. Sung et al. [72] tested interfaces to the kernel via the kernel API and global variables that are visible in the application layer. All of these papers focused on the application layer, while my approach applies the technique on the components across all the layers: application layer, hardware adaption layer, and operating system infrastructure layer.

To consider extra-functional requirements impact into the component integration level testing, I introduce additional notations in my testing model to integrate timing requirements. The generated test cases are annotated with real-time constraints. In contrast with the above models, my approach tests both functional and non-functional behavior.

Even though the empirical study of testing techniques has made some progress in recent years, studies of embedded systems have not. Many of the papers cited in this section on testing embedded systems include no empirical evaluation at all. My research work described in this dissertation goes beyond most of this prior work by examining fault detection capabilities relative to the use of techniques in an industrial real-time embedded system.

In UML, the notion of time is not clearly defined for the design of RTE systems. The UML profile for modeling and analysis of real-time and embedded systems (MARTE) has been proposed and was adopted as an Object Management Group (OMG) specification [40] [15] [88]. In comparison, MARTE presents time in a more precise and clear manner. A new clock constraint language (CCSL) is used in MARTE to specify chronometric and logical time constraints. A chronometric clock implicitly refers to physical time and a logical clock mainly addresses concrete instant ordering [73] [31] [16]. Although MARTE is capable of modelling logical and chronometric time, it does not specify the dynamic behavior of integration systems [58].

Kanstren [49] presented a study on design for testability in component-based embedded software based on two large-scale companies in the European telecom industry. He discussed Design for Testability (DFT) solutions to support test automation from two European telecommunications companies, working on similar large scale component-based embedded systems. Their techniques to support effective test automation were discussed. A common communication protocol provides support for implementing reusable test components. Especially in the case of embedded systems, a good host test environment enables efficient software testing. When this environment matches the target system as much as possible, efficient host testing is possible. One way to support testing is to use an operating system that is supported on both the target hardware and in a host-testing environment, as simulated on a desktop. Including support for test automation as a first-class feature allows more effective analysis of the system, including analysis of long running tests and deployed systems, and enables efficient field-testing. Effectively implementing this requires possibilities for dynamic configuration of test functionality during execution. Abstracting test cases from the implementation minimizes the effects of internal system changes to the test cases. This mostly applies at the system testing level, as in earlier testing phases it is often necessary to observe more detailed properties of the system.

Kanstren's methods addressed test automation and the different techniques to make this more effective at the architectural level. But they were still limited to regular functional testing to fulfill system requirements instead of designing a formalized and abstract structured testing model.

Chapter 4: A Component-based Real-time Embedded Software Architecture and Design Based Testing Technique

The Object Management Group (OMG) has proposed the Model Driven Architecture (MDA) approach, which allows developers to create systems entirely with models. It provides a set of guidelines to structure specifications expressed as models [81]. MDA itself is not a new OMG specification but rather an approach to software development that is enabled by existing OMG specifications such as the Unified Modeling Language (UML), the Meta Object Facility (MOF), and the Common Warehouse Metamodel (CWM). The Unified Modeling Language (UML 2.0) [6] [38] allows designers to describe component-based embedded systems. It provides constructs to deal with varying levels of modeling abstraction to visualize and specify both the static and dynamic aspects of systems. And with the mechanisms of stereotypes, tagged values, and constraints, the semantics of model elements can be customized and extended. Therefore, it is useful and significant to combine the MDA approach and UML models to develop software testing technique for component-based embedded systems.

Component-based modeling supports the representation of substantial aspects of the software architecture views as structural models, which consist of components and their relationships. It represents the static configuration of a system through the dependencies and connections between components. It includes details of the interface, such as data structures, services, and physical characteristics, on different abstraction levels. Behavioral models are used to describe the dynamic aspects of the components, component interaction, and resource constraints. It can be divided into component-interaction parts, which show the messages (behavior name and message parameters) sent between components, and state transition parts, which present the state transitions inside each component or the interactions between the components. It also shows dynamic aspects such as tasks, services, and operating conditions to provide scheduling information.

Functional models specify functionality of a component such as data flows, control flows and functional relations. The run-time interaction of a component-based real-time embedded system is modeled by well-defined sequences of messages passed among component level sequence diagrams. Component level sequence diagrams are interaction diagrams consisting of a set of components and their relationships, including the messages that may be dispatched among them, as well as interactions with external system. Sequence diagrams address the dynamic view of a system by emphasizing the time-ordering of messages.

In many cases, the states of the components sending and receiving a message at the time of message passing strongly influence their behavior:

- A component receiving a message can provide different functionalities from different states
- Certain functionalities may not be available if the receiving component is not in the right state
- The functionalities provided by a component may depend on the states of other components including the sending component of a message

UML state diagrams are useful modeling tools in certain domains, such as embedded software development. State diagrams are the de-facto accepted industry standard for modeling the behavior of distributed reactive systems [30]. The UML state machine diagram is used to model discrete behavior of a component through finite state transition. A system that is described by a UML state machine contains states at a particular time. A state machine is a behavior that specifies the sequence of states an object visits during its lifetime in response to events, together with its responses to those events. A state is a condition that satisfies some condition, performs some activity, or waits for some external event. They are connected by transitions. System states can be changed if the system receives a trigger associated with the current states. An event is the occurrence of a stimulus that can trigger a state transition. The trigger will activate a transition that is adjacent to the current states. If the trigger fires a transition, the current state will move to the next state. A transition is an edge that connects states. It is a relationship between two states indicating that an object in the current state will, when a specified set of events and conditions are satisfied, perform certain actions and enter the next state. Each transition has four components: transition name, trigger, guard condition, and action expression [6] [38]. Figure 4.1 presents a simple state diagram.

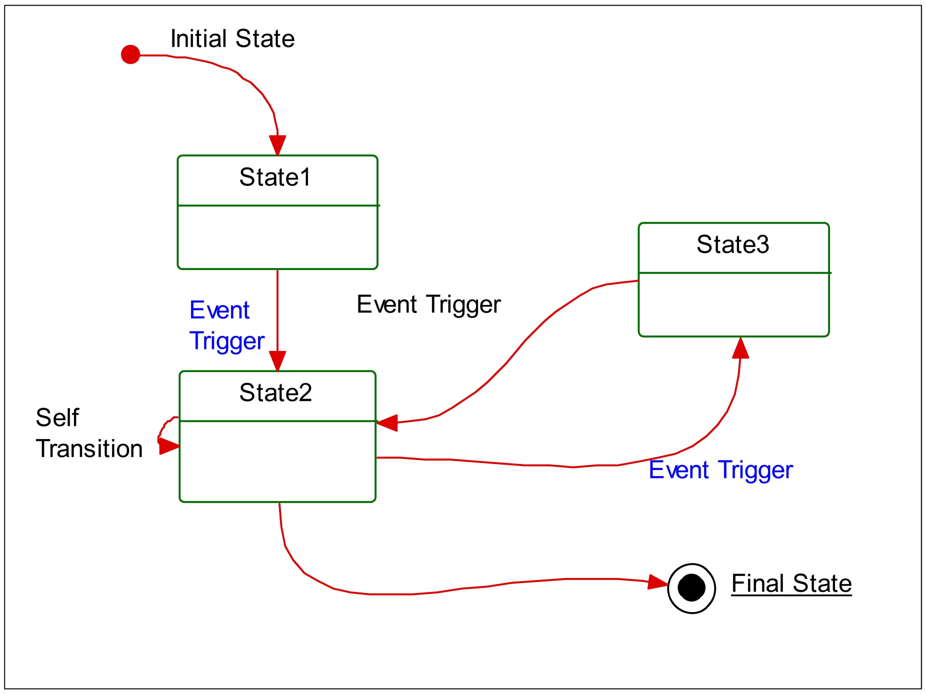


Figure 4.1: A simple state diagram

This research proposes an integration testing technique that is based on the idea that the interactions between components should ideally be exercised for all possible states of the components involved. This is of particular importance for component-based real-time embedded software as many components exhibit state-dependent behavior. This testing research objective is achieved by generating a graph-based test model called Component-based Real-time Embedded Architecture-based TEst Graph (CREATEG) and by covering paths in the model.

The proposed technique can be applied during the integration test phase, after the completion of component testing. It consists of the following four steps:

1. CREATEG Generation: The intermediate test model CREATEG is constructed from a component interaction sequence diagram and its corresponding state diagrams.

2. Test Paths Generation: Test paths are generated from the CREATEG based on several possible alternative coverage criteria.

3. Test Execution: All selected test paths are executed by using manually-generated test data and an execution log is created, which records component states before and after execution of each message in a test path.

4. Result Evaluation: The object states in the execution log are compared with the expected object states in the test paths generated from CREATEG. If a test path does not generates the required resultant state, then the corresponding test case is considered to have failed.

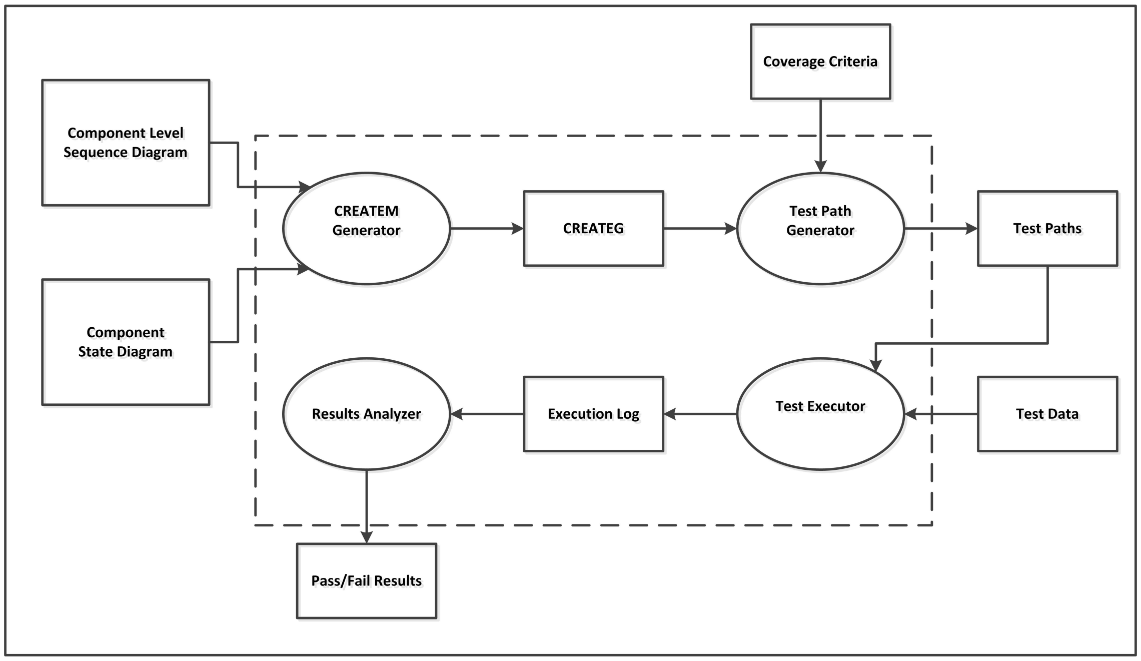


Figure 4.2: Flowchart for the proposed testing technique

Figure 4.2 presents a flowchart for the proposed technique, which shows the above four phases and their relationships with various artifacts. The following sub-sections describe the proposed testing technique in greater detail with the help of a simple example.

4.1 The CREATEG Model

The CREATEG is a test model used to automatically generate test specifications for component integration testing. In this sub-section, we describe how this test model is constructed from a given UML sequence diagram and its corresponding state diagrams. We have, however, made some assumptions about these UML models:

- Sequence diagrams are available that show critical scenarios of system operation.
- State diagrams are available that specify states and transitions for each component.
- Sequence diagrams may contain synchronous call messages or asynchronous signal messages.

- States in the component that are relevant to the selected sequence diagram are specified.
- All guards and conditions are evaluated together with events for generating state transitions.

In a component-based embedded system, applications are built as a collection of interconnected components, which are transformed into executable units such as tasks that can be managed by the underlying real-time operating system. The execution time of a component does not only depend on the component behavior but also on the time-constraint and platform characteristics. For example, each task is assigned with a priority. Higher priorities are assigned to tasks that have real-time deadlines. The impact of the extra-functional characteristics to a component integration test can be illustrated in the following example:

Sequence diagram 4.3 shows four components that interact through messages. Upon receiving a message, each component reacts with a sequence of events, changing states and sending messages to other components. Each component maps to a task and gets its computing resource based on its task's priority.

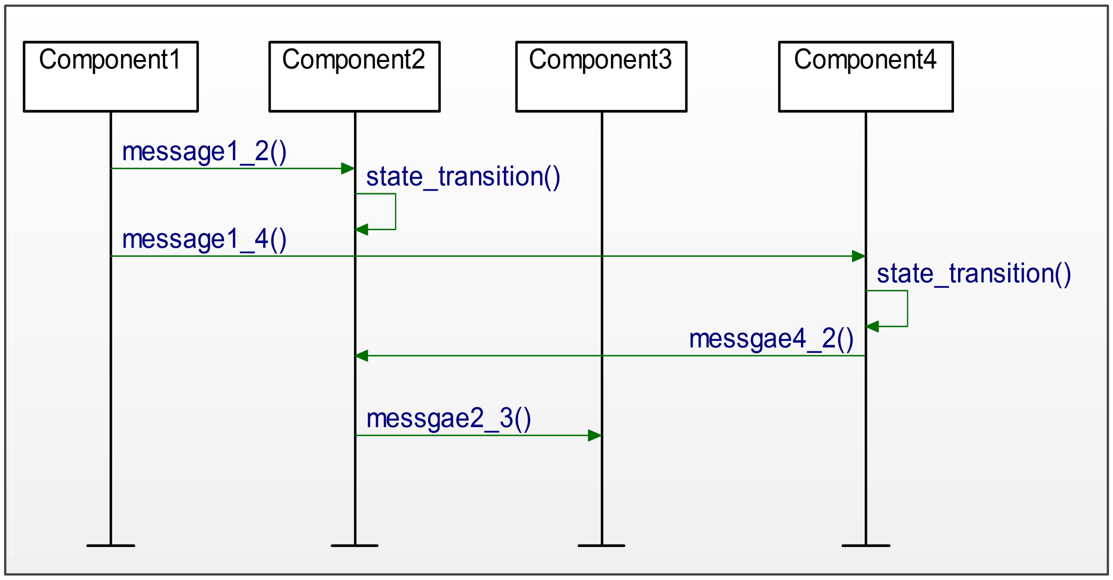


Figure 4.3: A component level sequence diagram

Figures 4.4, 4.5, and 4.6 demonstrate three different senarios. In all three figures, Component1's task is assigned the highest priority while Component3's task is assigned the lowest priority. In Figure 4.4, when Component4's task's priority is higher than Component2's, Component2 may not complete its state transition before Component4 finishes its state transition and calls Message4_2(). On the other hand, in Figure 4.5, when Component4's task's priority is lower than Component2's, Component2 will complete its state transition before Component4 starts its state transition and calls Message4_2(). In Figure 4.6, a timed delay is added between Message1_2() and Message1_4(), even through Component2's task has lower priority than Component4's task. Thus, Component2 will complete its state transition before Component4 starts its task.

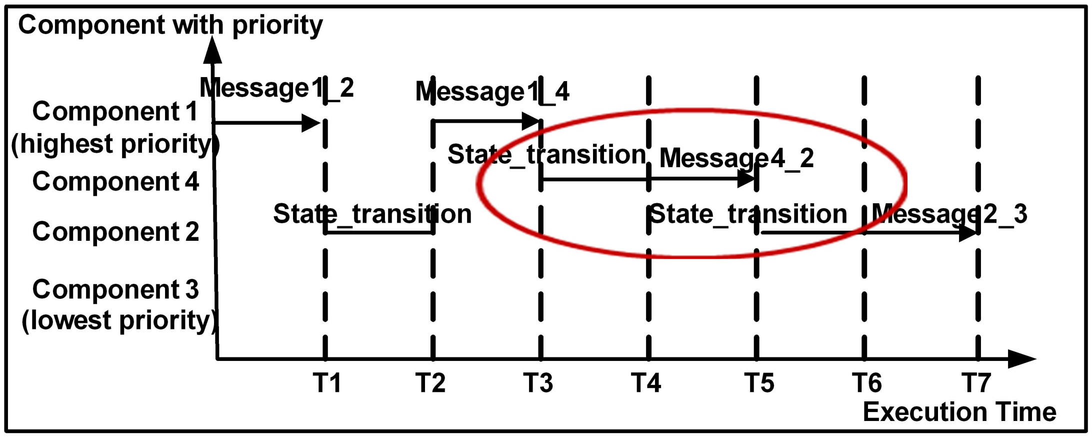


Figure 4.4: Component interaction scenario 1

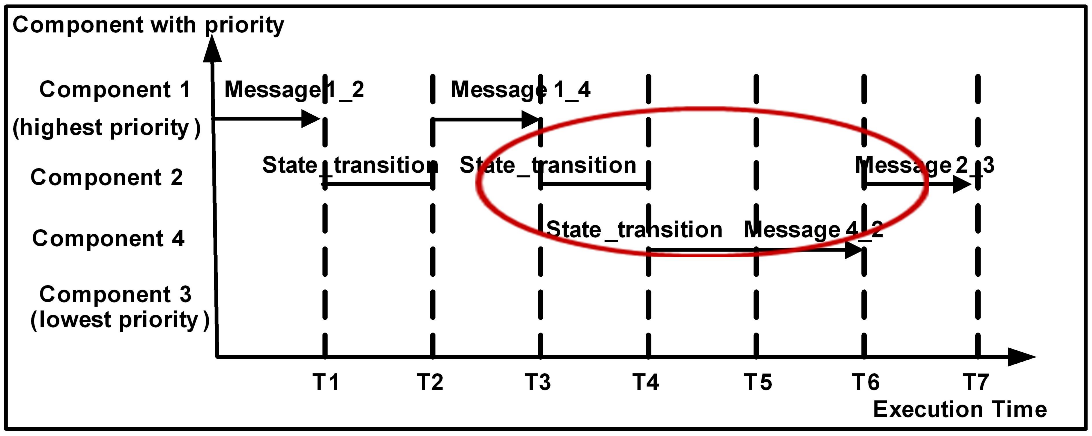


Figure 4.5: Component interaction scenario 2

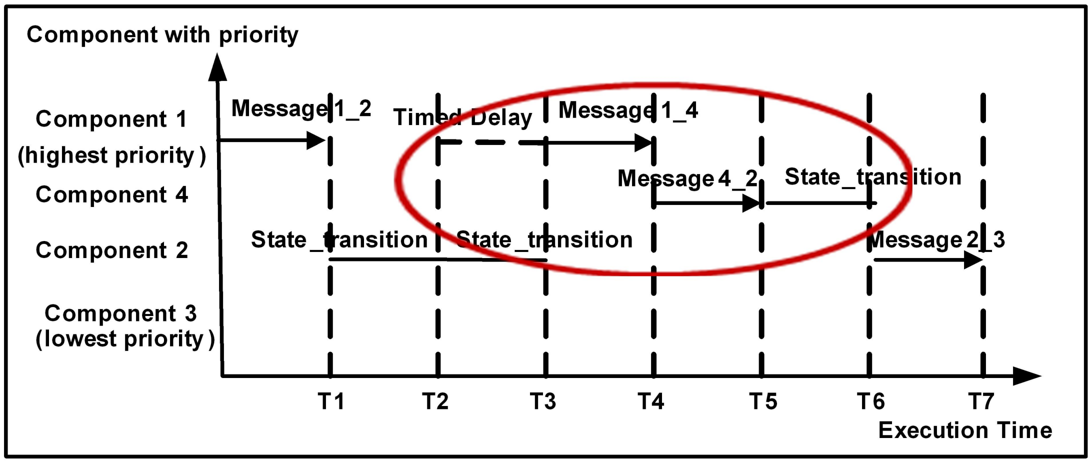


Figure 4.6: Component interaction scenario 3

To consider how extra-functional requirements impact component integration level testing, this test model introduces additional timing notations. They are placed onto the graphs that are defined in the next section.

The CREATEG is a multi-graph structure composed of the Component Interface Interaction Graph (CIIG), the Component State-based Interaction Behavior Graph (CSIBG), and the Component State-based Event-driven Interaction Behavior Graph (CSEDIBG), each of which is described below.

4.1.1 Component Interface Interaction Graph (CIIG)

Here I first introduce a graphical representation of the interactions between real-time embedded system components. Components in this graph not only include application layer components but also hardware adaption layer and infrastructure layer components. Interactions involve message exchanges occur at specific times.

A CIIG represents the time-dependent connectivity relationships between these components as well as time-dependent relations inside a component and a component interface. A

CIIG is composed of a set of components (visually as rectangular boxes), component interfaces (small rectangular boxes on the edge of the component boxes), connections between components (solid arrows), connections inside components (dash-line arrows), and times when connections happen.

Definition 1 (**Correlated Components**). *If two components are connected through an interface message, these two components are correlated. The component that initiates the interface message is the sending component, while the component that accepts the interface message is the receiving component. A component has to provide at least one interface to specify the dependencies between the services provided by the component and the services required to fulfill its task. Two components can interact during execution if their provided and required interfaces are associated by exchanging data. Component interfaces can be categorized into two groups: provider interfaces and user interfaces.*

In real-time embedded software, a layered architecture of basic software modules comprises communication modules, an operating system, and modules that access microcontroller peripheral devices as well as a component based infrastructure for application components.

Definition 2 (**Provider/Callee Interfaces**). *A provider interface in a component provides a function service for the other components to call and to transfer information to this component. A provider interface specifies the same types of elements as a user interface that it provides to the systems.*

Definition 3 (**User/Caller Interfaces**). *A user interface specifies variables, messages, services, calibration parameters, and other software elements required for the component to execute. A user interface in a component connects the other components by calling their provider services.*

Definition 4 (**Message Edges**). *A message edge represents a connection between two interfaces. The message edges in the CIIG are of two types: internal message edges and external message edges.*

1. An **external message edge** represents a call action between two components. It is a connection from the sending component's user interface to the receiving component's provider interface.

2. An **internal message edge** represents a direct or indirect interface relationship within a component. It is a connection from the receiving component's provider interface to its own user interface. Component internal flow refers to a message sequence within a software component. This information helps to identify the dependencies of an output of a component on its input values.

Definition 5 (**Time Instants**). *A time instant is an observation point when a specific event happens. It is an instant of a physical time or logical clock. A physical clock is a chronometric clock. A logical clock is a finite set of instants, which are ordered for discrete time clocks and indexed by natual numbers. For example, a logical clock can be represented as the number of frames passed since the start up of the systems. Time duration represents distances between time points.*

Definition 6 (**Message Sequences**). *A message sequence is a sequence of message edges representing a system operation. External message edges and internal message edges together comprise a message sequence.*

In a message sequence, message edges can represent synchronous (blocking) or asynchronous (non-blocking) method calls.

During a synchronous method call the thread of execution changes from the component with the required interface to the component with the provided interface until the latter has finished its task. The synchronous method call completes and the thread of execution returns to the first calling component.

When a component sends an asynchronous message, it can continue processing and does not have to wait for a response. Asynchronous calls are commonly seen in multithreaded applications and in message-oriented middleware. Asynchrony gives better responsiveness and reduces the temporal coupling but is harder to design and test.

Definition 7 (**Component Interface Interaction Graph (CIIG)**). *Given a software architecture defined by a specific UML sequence diagram, the Component Interface Interaction Graph (CIIG) is defined as:*

$CIIG = (C,\ C\_P\_Interf,\ C\_U\_Interf,\ C\_In\_edge,\ C\_Ex\_edge,\ T\_Ex\_edge, TC)$

- $C = \{C_1, C_2,\ \ldots,\ C_m\}$*, a finite set of components*
- $C\_P\_Interf = \{C_i.P_1, C_i.P_2,\ \ldots, C_i.P_n\}$*, a finite set of provider interfaces in component Ci,* $i \in \{1, 2, ..., m\}$
- $C\_U\_Interf = \{C_i.U_1, C_i.U_2,\ \ldots, C_i.U_n\}$*, a finite set of user interfaces in component Ci,* $i \in \{1, 2, ..., m\}$
- $C\_In\_edge = \{C_i.P \rightarrow C_i.U\}$*, a finite set of internal component message edges* $i \in \{1, 2, ..., m\}$
- $C\_Ex\_edge = \{C_i.U \rightarrow C_{i+1}.P\}$*, a finite set of external component message edges* $i \in \{1, 2, ..., m\}$
- $T\_Ex\_edge = \{t(_i \rightarrow_j)\}$*, a finite set of time stamps at external component message edges* $i \in \{1, 2, ..., m\}$*,* $j \in \{1, 2, ..., m\}$
- $TC = \{tc_i\}$ *is a finite set of time constraints on components internal messages* $i \in \{1, 2, ..., n\}$*,* $j \in \{1, 2, ..., m\}$*,* $tc_i = (tst_i \vee tpm_i \vee tum_i) - (tst_j \vee tpm_j \vee tum_j)\ rop\ c, c \in Integer,\ rop\ \in \{<, \leq, =, >, \geq\}$

As an example, consider a software system that consists of nine components.

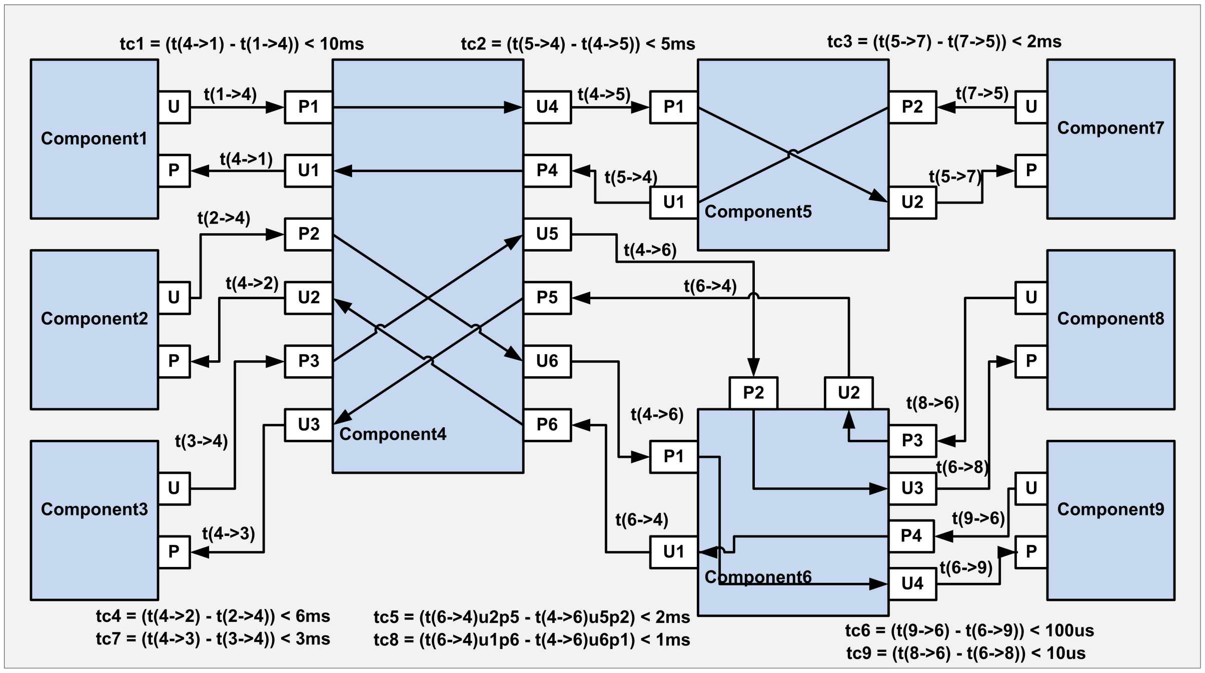

tc1 = (t(4->1) - t(1->4)) < 10ms
tc2 = (t(5->4) - t(4->5)) < 5ms
tc3 = (t(5->7) - t(7->5)) < 2ms
Component1
Component2
Component3
Component4
Component5
Component6
Component7
Component8
Component9
t(1->4)
t(4->1)
t(2->4)
t(4->2)
t(3->4)
t(4->3)
t(4->5)
t(5->4)
t(7->5)
t(5->7)
t(4->6)
t(6->4)
t(8->6)
t(6->8)
t(9->6)
t(6->9)
tc4 = (t(4->2) - t(2->4)) < 6ms
tc7 = (t(4->3) - t(3->4)) < 3ms
tc5 = (t(6->4)u2p5 - t(4->6)u5p2) < 2ms
tc8 = (t(6->4)u1p6 - t(4->6)u6p1) < 1ms
tc6 = (t(9->6) - t(6->9)) < 100us
tc9 = (t(8->6) - t(6->8)) < 10us

Figure 4.7: A software system that consists of multiple componentse

Figure 4.7 has nine components connected by eighteen message edges. Details are given as follows:

$C = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9\}$

$C\_P\_Interf = \{C_1.P, C_2.P, C_4.P, C_4.P_1, C_4.P_2, C_4.P_3, C_4.P_4, C_4.P_5, C_4.P_6,$
$C_5.P_1, C_5.P_2, C_6.P_1, C_6.P_2, C_6.P_3, C_6.P_4, C_7.P, C_8.P, C_9.P\}$

$C\_U\_Interf = \{C_1.U, C_2.U, C_4.U, C_4.U_1, C_4.U_2, C_4.U_3, C_4.U_4, C_4.U_5, C_4.U_6,$
$C_5.U_1, C_5.U_2, C_6.U_1, C_6.U_2, C_6.U_3, C_6.U_4, C_7.U, C_8.U, C_9.U\}$

$C\_In\_edge = \{C_4.P_1 \rightarrow C_4.U_4, C_4.P_4 \rightarrow C_4.U_1, C_4.P_2 \rightarrow C_4.U_6, C_4.P_3 \rightarrow C_4.U_5, C_4.P_5 \rightarrow$
$C_4.U_3, C_4.P_6 \rightarrow C_4.U_2, C_5.P_1 \rightarrow C_5.U_2, C_5.P_2 \rightarrow C_5.U_1, C_6.P_1 \rightarrow C_6.U_4, C_6.P_2 \rightarrow C_6.U_3, C_6.P_3 \rightarrow$
$C_6.U_2, C_6.P_4 \rightarrow C_6.U_1\}$

$C\_Ex\_edge = \{C_1.U \rightarrow C_4.P_1, C_2.U \rightarrow C_4.P_2, C_4.U \rightarrow C_4.P_3, C_4.U_4 \rightarrow C_5.P_1, C_4.U_5 \rightarrow$
$C_6.P_2, C_4.U_6 \rightarrow C_5.P_1, C_5.U_2 \rightarrow C_7.P, C_6.U_3 \rightarrow C_8.P, C_6.U_4 \rightarrow C_9.P, C_9.U \rightarrow C_6.P_4, C_8.U \rightarrow$
$C_6.P_3, C_7.U \rightarrow C_5.P_2, C_6.U_2 \rightarrow C_4.P_5, C_6.U_1 \rightarrow C_4.P_6, C_5.U_1 \rightarrow C_4.P_4, C_4.U_3 \rightarrow C_4.P, C_4.U_2 \rightarrow$

$C_2.P, C_4.U_1 \rightarrow C_1.P\}$

The time instant at which a component interaction occurs is modeled as a temporal property on the edge of an external message interface. The form of the time instant can be both chronometric and logical. The units can be milliseconds or number of time frames.
$T\_Ex\_edge = \{t(_1 \rightarrow_4), t(_4 \rightarrow_5), t(_5 \rightarrow_7), t(_7 \rightarrow_5), t(_5 \rightarrow_4), t(_4 \rightarrow_1), t(_2 \rightarrow_4), t(_4 \rightarrow_6),$
$t(_6 \rightarrow_8), t(_6 \rightarrow_9), t(_8 \rightarrow_6), t(_9 \rightarrow_6), t(_6 \rightarrow_4), t(_3 \rightarrow_4), t(_4 \rightarrow_3)\}$

Time constraints define timing relationships between component interactions. In a message sequence, the time duration between the beginning of an input message and the end of the message sequence is calculated and compared to a predefined value. For example, tc_1 and tc_4 depict the time constraints for time durations of the above message sequences.
$TC = \{tc_1, tc_2, tc_3, tc_4, tc_5, tc_6, tc_7, tc_8, tc_9\}$

$tc_1 = (t(_4 \rightarrow_1) - t(_1 \rightarrow_4)) < 10ms$

$tc_2 = (t(_4 \rightarrow_5) - t(_5 \rightarrow_4)) < 5ms$

$tc_3 = (t(_5 \rightarrow_7) - t(_7 \rightarrow_5)) < 2ms$

$tc_4 = (t(_4 \rightarrow_2) - t(_2 \rightarrow_4)) < 6ms$

$tc_5 = (t(_6 \rightarrow_4) - t(_4 \rightarrow_6)) < 2ms$

$tc_6 = (t(_9 \rightarrow_6) - t(_6 \rightarrow_9)) < 100us$

$tc_7 = (t(_4 \rightarrow_3) - t(_3 \rightarrow_4)) < 3ms$

$tc_8 = (t(_6 \rightarrow_4) - t(_4 \rightarrow_6)) < 1ms$

$tc_9 = (t(_8 \rightarrow_6) - t(_6 \rightarrow_8)) < 10us$

A timed **message sequence** is depicted by C_Ex_edge, C_In_edge and T_Ex_edge together. The following example traces along the top of Figure 4.7.

$C_1.U \xrightarrow{t(_1 \rightarrow_4)} C_4.P_1 \rightarrow C_4.U_4 \xrightarrow{t(_4 \rightarrow_5)} C_5.P_1 \rightarrow C_5.U_2 \xrightarrow{t(_5 \rightarrow_7)} C_7.P \rightarrow C_7.U \xrightarrow{t(_7 \rightarrow_5)} C_5.P_2 \rightarrow$
$C_5.U_1 \xrightarrow{t(_5 \rightarrow_4)} C_4.P_4 \rightarrow C_4.U_1 \xrightarrow{t(_4 \rightarrow_1)} C_1.P$

Another message sequence is shown below:

$C_2.U \xrightarrow{t(_2 \rightarrow_4)} C_4.P_2 \rightarrow C_4.U_6 \xrightarrow{t(_4 \rightarrow_6)} C_6.P_1 \rightarrow C_6.U_4 \xrightarrow{t(_4 \rightarrow_9)} C_9.P \rightarrow C_9.U \xrightarrow{t(_9 \rightarrow_6)} C_6.P_4 \rightarrow$

$C_6.U_1 \xrightarrow{t(_1 \rightarrow _6)} C_4.P_6 \rightarrow C_4.U_2 \xrightarrow{t(_4 \rightarrow _2)} C_2.P$

A CIIG contains all the components and their interfaces, and shows connections between components and interfaces. A CIIG also captures the time when an interaction between two components occurs. But a CIIG does not reflect the internal behavior of a component upon reception of any incoming external message, or before triggering any outgoing external message. As shown in diagram 4.4, 4.5, and 4.6, in a typical real-time embedded system, complexity arises when components interact with each other and multiple threads are running independently. Component behavior largely depends on the timeline of the interation messages.

The next time-dependent model is used to achieve the description of components' interactions and behavior.

4.1.2 Component State-based Interaction Behavior Graph (CSIBG)

For real-time embedded systems, a component is commonly modeled by state diagrams to describe component behavior. A component can receive a message in more than one state and exhibit distinct behavior for the same message in different states. To capture this characteristic and reflect this type of information in testing, we introduce a new type of graphical representation, the *Component State-based Interaction Behavior Graph* (CSIBG), to represent the information about the component behavior.

In many cases, the message sequence through a system depends on the different states of operation. This means that the sequence of messages may take different paths depending on the state in which the system is operating. As an example, Figure 4.8 shows that the message through component C1 is different depending on whether the component is in state S1 or state S2.

State dependent messages enable better understanding of the system and more precise analysis. They can help manage the complexity of system flow visualization by highlighting execution paths relevant to a particular state. Moreover, since the path of execution through a component is state dependent, the output of a component also depends upon the state in which the component is executing. Thus, specifying state dependent system flow helps to thoroughly test the overall software.

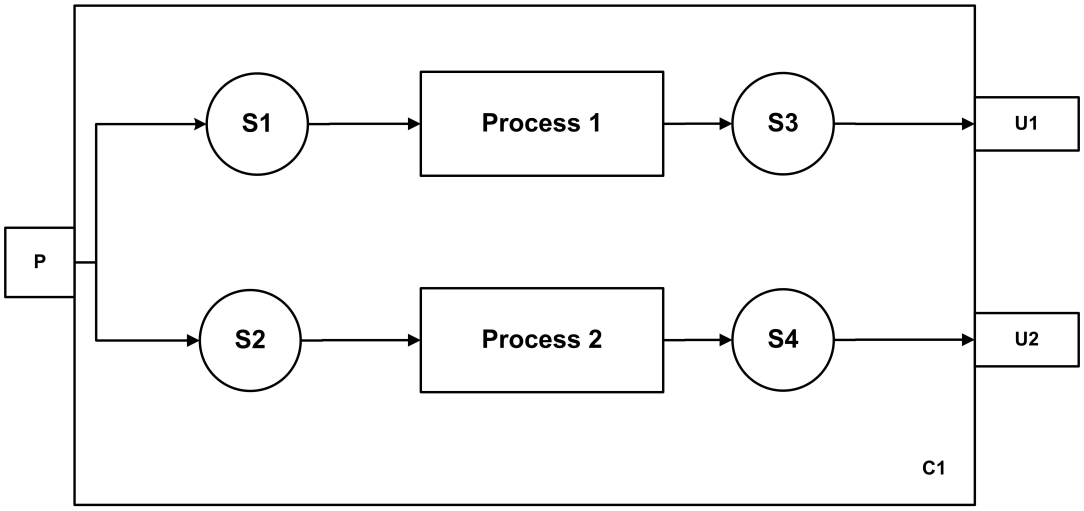


Figure 4.8: State dependent message paths

A Component State-based Interaction Behavior Graph (CSIBG) is a diagram that shows the behavior and the relationships of multiple components in a message sequence. A CSIBG is composed of a set of component subnets (visually as rectangular boxes), where each represents the behavior of one component, component interfaces (small rectangular boxes on the edge of the component subnet boxes), connections between component subnets (solid arrows), states in component subnets (circles inside the component subnet boxes), connections between component interface and states (solid arrows), connections between states inside component subnets (solid arrows), and times when connections happen.

A CSIBG consists of multiple component subnets. Each component subnet contains at least one state. A *Finite State Machine* (FSM) is a model of behavior composed of a finite number of states, transitions between those states, and actions. FSMs contain four main elements:

- States – Define behavior and may produce actions

- State Transitions – Switching from one state to another
- Conditions – Set of rules that must be met to allow a state transition
- Input Events – Triggers that are either externally or internally generated, which may possibly invoke conditions and upon fulfilling the conditions lead to state transitions.

Every FSM has one *initial state*, which is the starting point. The input events act as triggers, which cause an evaluation of the conditions imposed. On fulfilling those, the current state of the system switches to some other state (a state transition). State transitions often happen with associated actions. The actions can happen before entering a state, when exiting the state, or while in the state itself.

The *source* (incoming message) is the origin of information and is an input to a component. A source is used for a computation or control, or both. A *sink* represents consumption of information and specifies a particular output of a component. The incoming message plays a role on generating an external event in the component. An event capture is responsible for capturing the events, doing pre-processing, and identifying the event type. In each state, an event dispatcher maps the events to a handler and calls the handler. An event handler implements the activities, checks conditions, and decides state transitions. The event dispatcher and event handler for one event in different states are normally different. As an example, Figure 4.9 shows that the message through component C1 triggers different event handlers depending on whether the component is in state S1 or state S2.

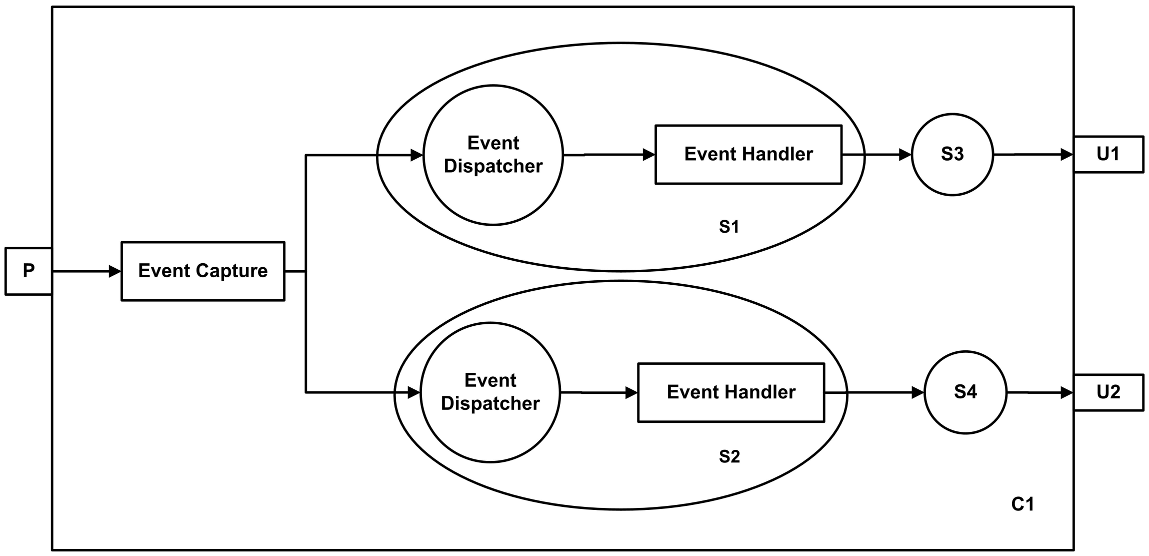


Figure 4.9: State dependent transition paths

My model defines the following elements to compose a message sequence in the component model. A message sequence for a component consists of one or more sources, one or more sinks, and dependencies between the sources and the sinks.

Definition 8 (**Correlated States**). *If two states are connected through a transition, these two states are correlated. The state that initiates the transition is the start state, while the state that is transitioned to is the target state.*

States are connected by transitions. States can be changed if the system receives a trigger associated with the current states. The trigger will activate the transition that is adjacent to the current state. If the trigger fires a transition, the CSIBG will move to the next state.

The edges in the CSIBG are of two types: message edges and transition edges. Message edges can be input message edges or output message edges.

Definition 9 (**Input Message Edges**). *An input message edge is a connection from the receiving component's provider interface to its current state, which is defined as an entry state. There may be more than one input message edge from one provider interface for the* *same external message edge, because the message may arrive in a different state.*

Definition 10 (**Output Message Edges**). *An output message edge is a connection from an exit state of the receiving component to its user interface, which generates an outgoing external message to the next receiving component. There can be multiple output message edges to one user interface. Input message edges are modeled in the CSIBG by attributes of a message including exit state and user interface.*

Both input message edges and output message edges belong to the internal component message edges defined as internal message edges (C_In_edge) in a CIIG.

Besides external events, *internal events* invoke conditions leading to state transitions. A typical internal event is a *timer event*, where a one-time event or a periodic event may be activated by a timer and internal generated events. The timer event is created due to the requirement of timely and predictable behavior from real-time systems. Internal events also include internally generated events from another internal event inside the component. The outcome of these events is to execute corresponding methods, compute results, and perform state transitions. In a CSIBG, state transitions are represented by transition edges.

Definition 11 (**Transition Edges**). *A transition edge is a connection between two states. The start state transitions to the target state.*

A component subnet also contains two types of external interfaces: provider interface and user interface, which are already defined in the CIIG.

In a sequence diagram, messages are processed sequentially, each message causing the execution of a run-to-completion (RTC) step. The next external event is processed after the current RTC step has completed and the state machine has reached a stable state. In a real-time system, a transition often occurs at a time that depends on the occurrence time of another transition. But it does not describe all the possible scenarios because some messages coming to a component can happen at any state inside the component. In a CSIBG, transfer relations are described by placing a transition from the receiving component's provider interface to all the states where the incoming message may happen.

In the component subnet, upon receiving an external message from sending component, a series of state transitions happen following an entry state.

Transition edges are modeled by the attributes of a transition, including the accepting and the sending state.

Formally, a CSIBG is defined as follows:

Definition 12 (**Component State-based Interaction Behavior Graph (CSIBG)**).
$CSIBG = (Comp1(Sn_1, Pn_1, Un_1, Tn_1, In_1, On_1, TSTn_1, TPMn_1, TUMn_1),$
$Comp2(Sn_2, Pn_2, Un_2, Tn_2, In_2, On_2, TSTn_2, TPMn_2, TUMn_2), \ \ldots,$
$Compm(Pn_k, Un_k, Sn_k, Tn_k, In_k, On_k, TSTn_k, TPMn_k, TUMn_k), Msgn, TC)$, *where*

Comp1 is the graph that describes component subnet C1.

- $Sn_1 = \{S_1, S_2, \ \ldots, S_n\}$, a finite set of states in C1 .

- $Pn_1 = \{P1_1, P2_1, \ \ldots, Pn_1\}$, is the set of all the provider interfaces in C_1. The naming format of a provider interface is *"P"* + *"the number of the correlated component"* + *"the number of this component"*.

- $Un_1 = \{U1_1, U1_2, \ \ldots, U1_n\}$, is the set of all the user interfaces in C_1. The naming format of a user interface is *"U"* + *"the number of this component"* + *"the number of the correlated component"*.

- $Tn_1 = \{S_1\_S_2, S_2\_S_3, \ \ldots, S_n - 1\_S_n\}$, is the set of all transition edges representing transitions between states in C1. The naming format of a transition edge is *"sourceState"*+ *"_"* + *"targetState"*.

- $In_1 = \{Pn_1\_S_1, Pn_1\_S_2, \ \ldots, Pn_1\_S_n\}$, is the set of all the input message edges in C1. The naming format of an input message edges is *"providerInterface"*+ *"_"* + *"entrytState"*.

- $On_1 = \{S_1\_On_1, S_2\_On_1, \ \ldots, S_n\_On_1\}$, is the set of all the output message edges in C1. The naming format of an output message edges is *"destinationtState"* + *"_"* + *"userInterface"*.

- $TSTn_1 = \{tst(_i \rightarrow_j)\}$, a finite set of time stamps on transition edges $i \in \{1, 2, ..., m\}$, $j \in \{1, 2, ..., m\}$.
- $TPMn_1 = \{tpm(_i \rightarrow_j)\}$, a finite set of time stamps on input message edges $i \in \{1, 2, ..., m\}$, $j \in \{1, 2, ..., m\}$.
- $TUMn_1 = \{tum(_i \rightarrow_j)\}$, a finite set of time stamps on output message edges $i \in \{1, 2, ..., m\}$, $j \in \{1, 2, ..., m\}$.
- $TC = \{tc_i\}$ is a finite set of time constraints on components internal messages $i \in \{1, 2, ..., n\}$, $j \in \{1, 2, ..., m\}$, $tc_i = (tst_i \vee tpm_i \vee tum_i) - (tst_j \vee tpm_j \vee tum_j)\ rop\ c, c \in Integer,\ rop\ \in \{<, \leq, =, >, \geq\}$

The other component subnet graphs, $Comp_2, \ldots, Comp_k$, are defined similiarly. $Msgn = \{Msg_1, Msg_2, \ldots, Msg_n\}$ is the set of external component message edges defined as C_Ex_edge in CIIG. The starting point of an Msg is a user interface of a correlated component while the ending point of the Msg is a provider interface of the current component. Each message includes a sequence number. The format of an external component message edges is *"Msg"* + *"sequence_number"*.

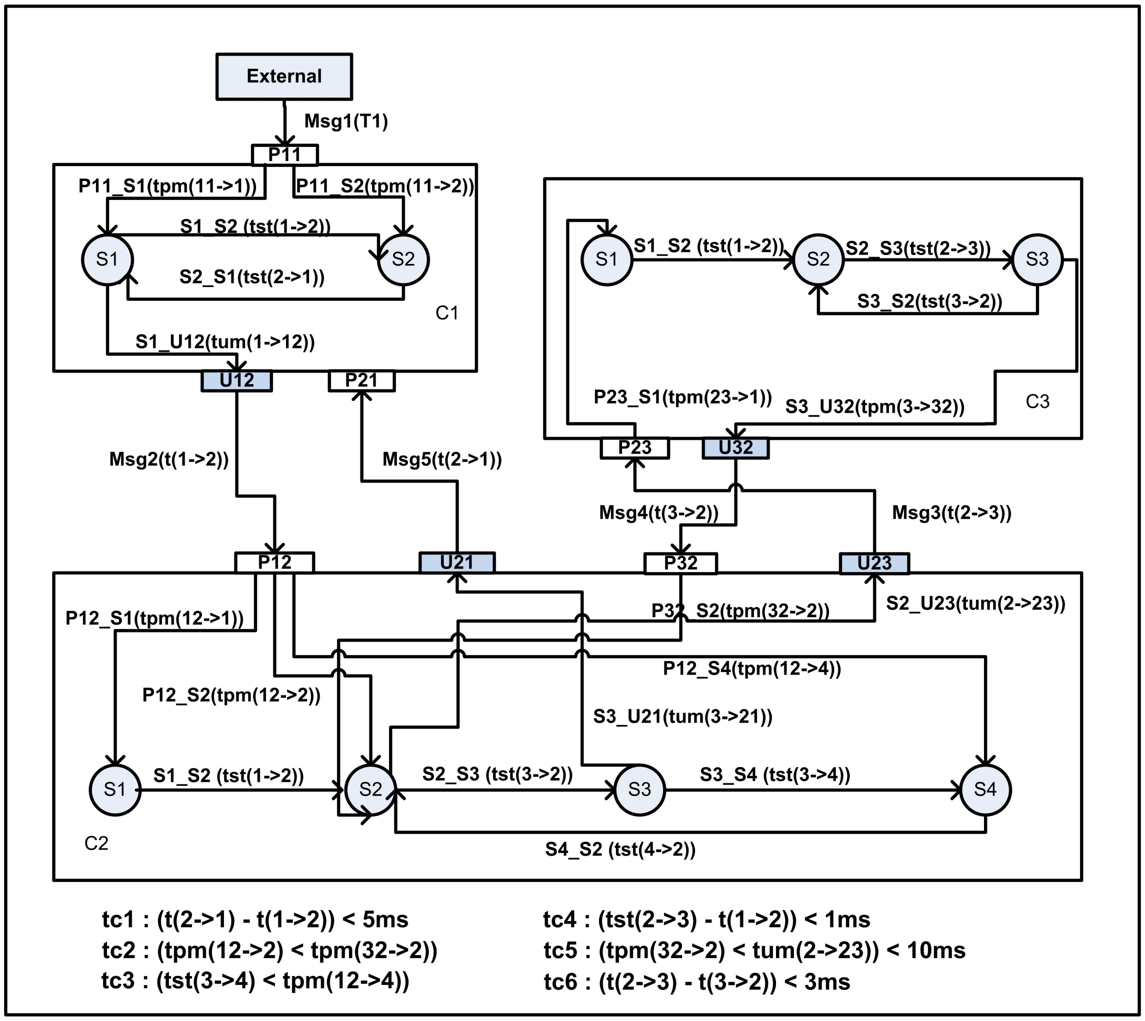


Figure 4.10: A CSIBG example

Figure 4.10 shows the CSIBG of a three component system. This example has three components. Each component contains multiple states. Details are given as follows:

$CSIBG = (Comp_1(Pn_1, Un_1, Sn_1, Tn_1, In_1, On_1), Comp_2(Pn_2, Un_2, Sn_2, Tn_2, In_2, On_2), Comp_3(Pn_3, Un_3, Sn_3, Tn_3, In_3, On_3), Msgn)$

- $Msgn = \{Msg_1, Msg_2, Msg_3, Msg_4, Msg_5\}$

- $Comp_1(Pn_1, Un_1, Sn_1, Tn_1, In_1, On_1)$ includes:

  $Pn_1 = \{C_1.P1_1, C_1.P2_1\}$, where $C_1.P1_1$ is a provider interface for the external entity; $C_1.P2_1$ is a provider interface for C_2, and it is also an acceptance point for Msg_5.

$Un_1 = \{C_1.U1_2\}$, where $C_1.U1_2$ is a user interface to C_2, and it is also a starting point for Msg_2.

$Sn_1 = \{S_1, S_2\}$.

$Tn_1 = \{S_1\_S_2, S_2\_S_1\}$, where S_1 may transition to S_2 and S_2 may transition to S_1 according to associated events.

$In_1 = \{P1_1\_S_1, P1_1\_S_2\}$, where $P1_1$ is the provider interface of Msg_1, when Msg_1 enters C_1, the message can be accepted in either S_1 or S_2.

$On_1 = \{S_1\_U1_2\}$, where $U1_2$ is a user interface, as well as the starting point of Msg_2. The message is generated in S_1.

$TSTn_1 = \{tst(_1 \rightarrow_2), tst(_2 \rightarrow_1)\}$, where $tst(_1 \rightarrow_2)$ is the time of the transition from S_1 to S_2, and $tst(_2 \rightarrow_1)$ is the time of the transition from S_2 to S_1.

$TPMn_1 = \{tpm(_{11} \rightarrow_1), tpm(_{11} \rightarrow_2)\}$, where $tpm(_{11} \rightarrow_1)$ is the time of the provider message from P_{11} to S_1, and $tpm(_{11} \rightarrow_2)$ is the time of the transition from P_{11} to S_2.

$TUMn_1 = \{tum(_1 \rightarrow_{12})\}$, where $tum(_1 \rightarrow_{12})$ is the time of the user message from S_1 to U_{12}.

- $Comp_2(Pn_2, Un_2, Sn_2, Tn_2, In_2, On_2)$ includes:

$Pn_2 = \{C_2.P1_2, C_2.P3_2\}$, where $C_1.P1_2$ is a provider interface for C_1, and it is also an acceptance point for Msg_2; $C_1.P3_2$ is a provider interface for C_3, and it is also an acceptance point for Msg_4.

$Un_2 = \{C_2.U2_1, C_2.U2_3\}$, where $C_2.U2_1$ is a user interface to C_1, and it is also a starting point for Msg_5; $C_2.U2_3$ is a user interface to C_3, and it is also a starting point for Msg_3.

$Sn_2 = \{S_1, S_2, S_3, S_4\}$

$Tn_2 = \{S_1\_S_2, S_2\_S_3, S_3\_S_4, S_4\_S_3\}$, where S_1 may transition to S_2, S_2 may transition to S_3, S_3 may transition to S_4 and S_4 may transition to S_3 according to associated events.

$In_2 = \{P1_2\_S_1, P3_2\_S_2\}$, where $P1_2$ is the provider interface of Msg_2, when Msg_2 enters C_2, the message can be accepted in S_1; $P3_2$ is the provider interface of Msg_4, when Msg_4 enters C_2, the message can be accepted in S_2.

$On_2 = \{S_2\_U2_3, S_3\_U2_1\}$, where $U2_3$ is a user interface, as well as the starting point of Msg_3, the message is generated in S_2; $U2_1$ is a user interface, as well as the starting point of Msg_5, the message is generated in S_4.

$TSTn_2 = \{tst(_1 \rightarrow_2), tst(_2 \rightarrow_3), tst(_3 \rightarrow_4), tst(_4 \rightarrow_2)\}$, where $tst(_1 \rightarrow_2)$ is the time of the transition from S_1 to S_2, $tst(_2 \rightarrow_3)$ is the time of the transition from S_2 to S_3, $tst(_3 \rightarrow_4)$ is the time of the transition from S_3 to S_4, and $tst(_4 \rightarrow_2)$ is the time of the transition from S_4 to S_2.

$TPMn_2 = \{tpm(_{12} \rightarrow_1), tpm(_{12} \rightarrow_4), tpm(_{32} \rightarrow_2)\}$, where $tpm(_{12} \rightarrow_1)$ is the time of the provider message from P_{12} to S_1, $tpm(_{12} \rightarrow_4)$ is the time of the provider message from P_{12} to S_4, $tpm(_{32} \rightarrow_2)$ is the time of the provider message from P_{32} to S_2,and $tpm(_{31} \rightarrow_2)$ is the time of the transition from P_{32} to S_2.

$TUMn_2 = \{tum(_3 \rightarrow_{21}), tum(_3 \rightarrow_{23})\}$, where $tum(_3 \rightarrow_{21})$ is the time of the user message from S_3 to U_{21} and $tum(_3 \rightarrow_{23})$ is the time of the user message from S_3 to U_{23}.

- $Comp_3(Pn_3, Un_3, Sn_3, Tn_3, In_3, On_3)$ includes:

  $Pn_3 = \{C_4.P2_3\}$, where $C_4.P2_3$ is a provider interface for C_3, and it is also an acceptance point for Msg_3.

  $Un_3 = \{\text{C4}.U3_2\}$, where $C_4.U3_2$ is a user interface to C_2, and it is also a starting point for Msg_4.

  $Sn_3 = \{S_1, S_2, S_3\}$

$Tn_3 = \{S_1\_S_2, S_2\_S_3, S_3\_S_2\}$, where S_1 may transition to S_2, S_2 may transition to S_3 and S_3 may transition to S_2 according to associated events.

$In_3 = \{P2_3\_S_1\}$, where $P2_3$ is the provider interface of Msg_3. When Msg_3 enters C_3, the message can be accepted in S_1.

$On_3 = \{S_2\_U3_2\}$, where $U3_2$ is a user interface, as well as the starting point of Msg_4. The message is generated in S_2.

$TSTn_3 = \{tst(_1 \rightarrow_2), tst(_2 \rightarrow_3), tst(_3 \rightarrow_2)\}$, where $tst(_1 \rightarrow_2)$ is the time of the transition from S_1 to S_2, $tst_2 \rightarrow_3)$ is the time of the transition from S_2 to S_3, $tst(_3 \rightarrow_2)$ is the time of the transition from S_3 to S_2.

$TPMn_3 = \{tpm(_{23} \rightarrow_1)\}$, where $tpm(_{23} \rightarrow_1)$ is the time of the provider message from P_{23} to S_1.

$TUMn_3 = \{tpm(_3 \rightarrow_{23})\}$, where $tum(_3 \rightarrow_{23})$ is the time of the user message from S_3 to U_{23}.

- $TC = \{tc_1, tc_2, tc_3, tc_4, tc_5\}$

  $tc_1 = (t(_2 \rightarrow_1) - t(_1 \rightarrow_2)) < 5ms$

  $tc_2 = (tpm(_{12} \rightarrow_2)) < (tpm(_{32} \rightarrow_2))$

  $tc_3 = (tst(_3 \rightarrow_4)) < (tpm(_(12) \rightarrow_4))$

  $tc_4 = (tst(_2 \rightarrow_3) - t(_1 \rightarrow_2)) < 1ms$

  $tc_5 = (tpm(_{32} \rightarrow_2) - tum(_2 \rightarrow_{23})) < 10ms$

  $tc_6 = (t(_2 \rightarrow_3) - t(_3 \rightarrow_2)) < 3ms$

A **CSIBG** describes some characteristics of the component-based real-time embedded system, which is that a component can receive a message in more than one state and exhibit distinct behavior for the same message in different states with the times of state transitions, provider message recieved and user message sent . In component C_1, it shows that there are three different internal message paths between the same external incoming message Msg_1 and external outgoing message Msg_2:

$\mathbf{Msg_1} \xrightarrow{T_1} P_{11} \xrightarrow{tpm(_{11 \to 1})} S_1 \xrightarrow{tum(_{1 \to 12})} U_{12} \xrightarrow{t(_{1 \to 2})} \mathbf{Msg_2}$

$\mathbf{Msg_1} \xrightarrow{T_1} P_{11} \xrightarrow{tpm(_{11 \to 1})} S_1 \xrightarrow{tst(_{1 \to 2})} S_2 \xrightarrow{tst(_{2 \to 1})} S_1 \xrightarrow{tum(_{1 \to 12})} U_{12} \xrightarrow{t(_{1 \to 2})} \mathbf{Msg_2}$

$\mathbf{Msg_1} \xrightarrow{T_1} P_{11} \xrightarrow{tpm(_{11 \to 2})} S_2 \xrightarrow{tst(_{2 \to 1})} S_1 \xrightarrow{tum(_{1 \to 12})} U_{12} \xrightarrow{t(_{1 \to 2})} \mathbf{Msg_2}$

Even when they are all in the same external message path, the signatures of the output message can be different.

Similarly, component C_2 has three different internal message paths between Msg_2 and Msg_3:

$\mathbf{Msg_2} \xrightarrow{t(_{1 \to 2})} P_{12} \xrightarrow{tpm(_{12 \to 1})} S_1 \to S_1 \xrightarrow{tst(_{1 \to 2})} S_2 \xrightarrow{tum(_{2 \to 23})} U_{23} \xrightarrow{t(_{2 \to 3})} \mathbf{Msg_3}$

$\mathbf{Msg_2} \xrightarrow{t(_{1 \to 2})} P_{12} \xrightarrow{tpm(_{12 \to 2})} S_2 \xrightarrow{tum(_{2 \to 23})} U_{23} \xrightarrow{t(_{2 \to 3})} \mathbf{Msg_3}$

$\mathbf{Msg_2} \xrightarrow{t(_{1 \to 2})} P_{12} \xrightarrow{tpm(_{12 \to 4})} S_4 \to S_1 \xrightarrow{tst(_{4 \to 2})} S_2 \xrightarrow{tum(_{2 \to 23})} U_{23} \xrightarrow{t(_{2 \to 3})} \mathbf{Msg_3}$

With the above information, it is possible to specify an end-to-end sequence in terms of message exchanges between components and component internal flow. It is a sequence that originates at an external system, propagates through various software components and terminates at an external system. The total number of message paths between Msg_1 and Msg_3 can be calculated by taking the cross product of all message paths of components involved in an interaction. In this particular example, there are 3 (from C_1) * 3 (from C_2) = 9 test paths. The overall number of message paths from the first message of a system operation to the last message of a system operation can be calculated by taking the cross product of all message paths of components involved in an interaction.

Time constraints define timing relationships between component external messages and internal messages. In a message sequence, the time window between message exchanges and state transitions is calculated and compared to a predefined value. For example, tc_2 and tc_4 depicted the time constraints for incoming messages to state 2 of component 2 from component 1 and component 3.

4.1.3 Component State-based Event-driven Interaction Behavior Graph (CSIEDBG)

The CIIG and CSIBG clearly described the timed sequences of method calls and paths of state transitions, but they do not deal with concurrency. As in other concurrent programs, multitasking is one of the most important aspects in real-time embedded system. Complexity arises when multiple threads are running independently. The order of the message generated from each thread is not deterministic. As a result, when multiple messages from different components arrive at one state in a component, the system can have non-deterministic behavior, which complicates the determination of expected outputs for a given input. I therefore extended the CSIBG in several ways to include features required to visualize concurrent events. The new graphical representation is named the *Component State-based Event-driven Interaction Behavior Graph* (CSIEDBG).

Definition 13 (**Event Capture**). *An event capture (EC) is at the end of an input message edge. It processes an incoming message carried by a provider service and identifies an event in a specific state, with a time stamp $tpm \in TPM$ as specified in Definition 11 in Chapter 4.1.2. Based on the component whose user serive initiated the call to this provider service, the event generated can be different types – external event, timer event, semaphore and message queue. Each produced event is mapped to an event handler.*

Definition 14 (**Event Handler**). *An event handler (EH) is at the beginning of an output message edge or a transition edge. It takes all the events generated from different threads, checks conditions, decides state transitions, and triggers interactions to other components with a time stamp $tum \in TUM$ as specified in Definition 11.*

Figure 4.11 shows the CSIEDBG of the three component system mentioned above.

1. Each component contains multiple provider services. For each provider service in a component, there is one event capture to process the incoming external message.

2. Each component contains multiple states. Each state contains one event handler.

3. Each event capture generates one event.

4. All events will be handled by one event handler.

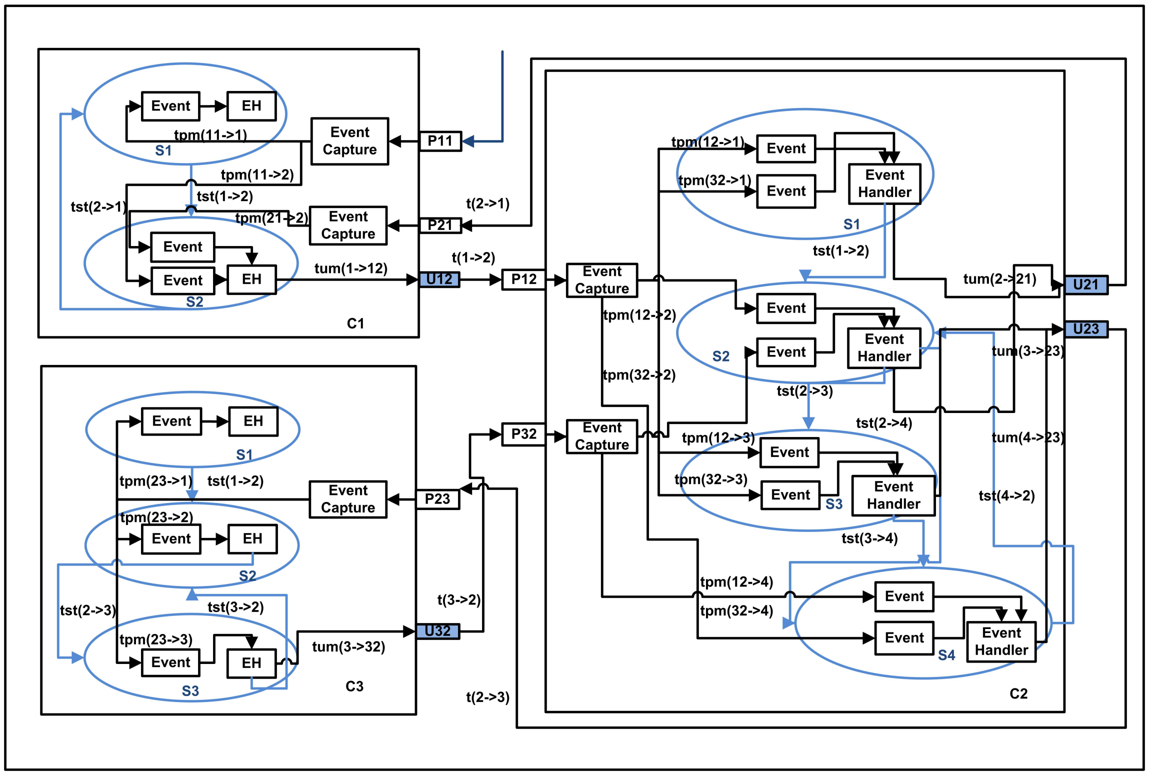


Figure 4.11: A CSIEDBG example

4.2 Constructing the CREATEG

This section presents a running example of CREATEG construction based on algorithms presented to generate CIIGs, CSIBGs, and CSIEDBGs. In subsequent sections, we will discuss the process of test path generation (section 4.3) and how to handle a potentially large number of test paths (section 4.4).

The example is an Ethernet communication system in a software defined radio system. It consists of four components: an IP Core, an IP Interface Adapter, an SCA Ethernet

Device, and an Ethernet Device Adapter. Each component contains multiple states.

A component level sequence diagram is used to address the dynamic view of a system emphasizing the time-ordering of messages. It consists of these four components and their relationships, including the messages that may be dispatched among them, as well as interactions with an external system.

Upon connection to the IP core, the Ethernet Device issues *enableRTSCTS* and *enableFlowResumeSignal* commands to the SCA Device Adapter. When data is available, the EthernetDevice pushes it to the IP Core using the *pushPacket* method. If the Device Adapter cannot hold at least one more packet of *maxPayloadSize*, the Device Adapter halts *pushPackets* so the *pushPacket* returns FALSE. Once the Device Adapter is ready to receive more packets, it issues a *signalFlowResume* to the EthernetDevice. Then the EthernetDevice resumes pushing data to the Device Adapter using the *pushPacket* method. When the IP Interface adapter receives packets from a SCA Ethernet device adapter, it forwards the packets to the IP Core.

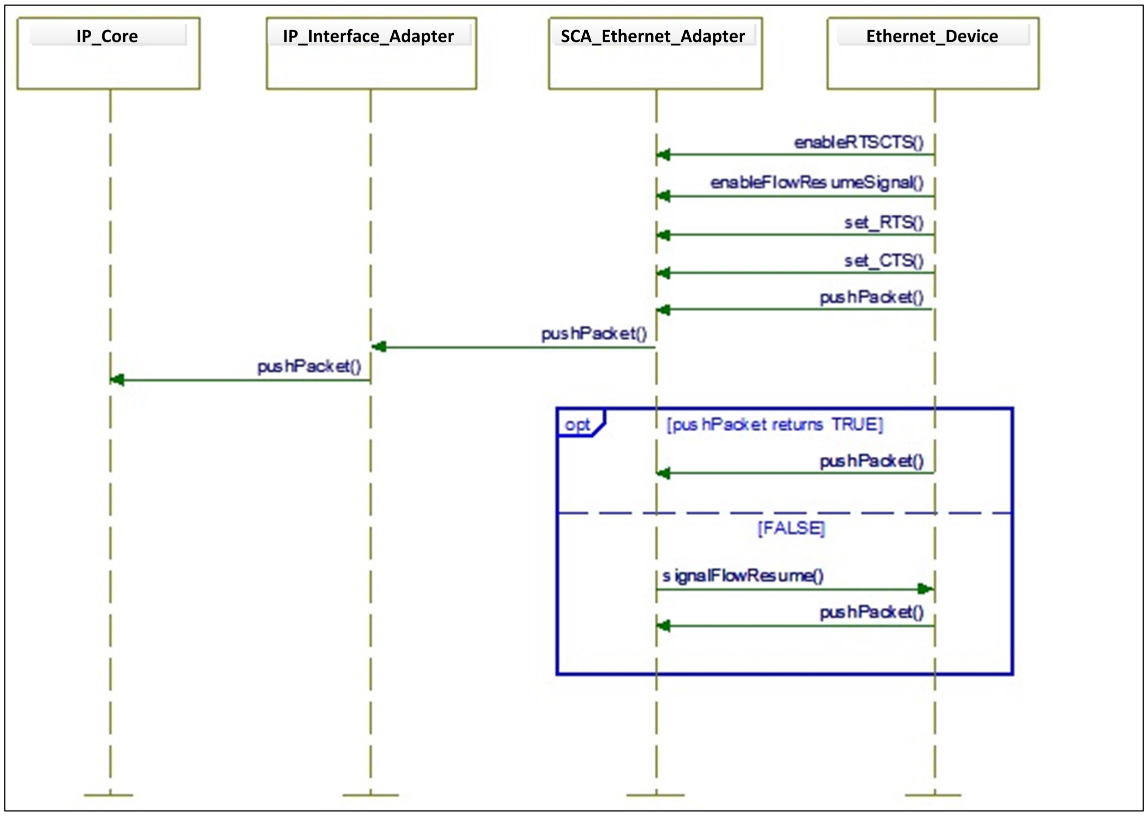


Figure 4.12: Sequence diagram for receiving data from ethernet device

Figure 4.12 shows a component level sequence diagram for a system-level operation. A brief sequence of *receivePackets()* is described below:

- Upon connection, the Ethernet Device issues *enableRTSCTS*.
- The Ethernet Device issues *enableFlowResumeSignal*.
- When data is available, the Ethernet Device pushes it to the Device Adapter using the *pushPacket* method.
- The Ethernet Device Adapter pushes it to the IP Interface Adapter using the *push-Packet* method.

- When the IP Interface adapter receives packets from a SCA Ethernet device adapter, it forwards the packets to the IP Core.

Two alternative sequences are also shown according to different condition decisions:

1. If the Device Adapter cannot hold at least one more packet of *maxPayloadSize*, the Device Adapter halts *pushPackets* so the *pushPacket* returns FALSE.
    - Once the Device Adapter is ready to receive more packets, it issues a *signalFlowResume* to the Device.
    - The Device resumes pushing data to the Device Adapter using the *pushPacket* method.
2. Else if the Device Adapter returns TRUE
    - The Device continuess pushing data to the Device Adapter using the *pushPacket* method.

Sequence diagrams usually model the execution of a use case, triggered by one or more system-level operations. The recipient of a system-level message is generally a provider interface that forwards the message to one or more user interface through a sequence of actions.

To construct the CREATEG model for *receivePackets()*, we start from the sequence diagram in Figure 4.12. The CIIG is created based on the objects and messages in a sequence diagram.

For each component in the sequence diagram:

- One component subnet is created.

For each message in the sequence diagram:

- One component provider interface is created.
- One component user interface is created.

- One external component message edge is created.

Between two consecutive messages in the sequence diagram:

- One internal component message edge is created.

For clarity, external message edges are shown in the CIIG with solid lines and are labeled with the message sequence numbers as in the sequence diagram. We assume that each sequence number corresponds to a full message signature, condition, and iteration. Internal message edges are shown with dotted lines, and correspond to an enabling condition.

The algorithm to generate a CIIG is shown in Figure 4.13.

Algorithm BuildCIIG(SQ): CIIG

Inputs SQ: *Component level sequence diagram corresponding to a message*

Output CIIG: *CIIG model generated by the algorithm*

Declare:

```
    $C = \{C_1, C_2, \ldots, C_k\}$:
        A finite set of components
    $C\_P\_Interf = (C_1.P\_interf_1, \ldots, C_k.P\_interf_t)$:
        A finite set of component provider interfaces
    $C\_U\_Interf = (C_1.U\_interf_1, \ldots, C_k.U\_interf_t)$:
        A finite set of component user interfaces
    $C\_In\_edge = (C_i\_P_Interf, \ldots, C_i\_U\_Interf)$:
        A finite set of internal component message edges
    $C\_Ex\_edge = (C_i\_U\_Interf, \ldots, C_ii\_P\_Interf)$:
        A finite set of external component message edges
    $T\_Ex\_edge = (T_i \rightarrow_j, \ldots, T_j \rightarrow_i)$:
        A finite set of time stamps at external component message edges
    addComponentNode(C, CIIG):
        A function that adds a node C to the CIIG
    addProviderInterface(C_P_Interf, CIIG):
        A function that adds a provider interface C_P_Interf to the CIIG
    addUserInterface(C_U_Interf, CIIG):
        A function that adds a user interface C_U_Interf to the CIIG
    addInternalMessageEdge (C_In_edge, CIIG):
        A function that adds a provider interface C_In_edge to the CIIG
    addExternalMessageEdge (C_Ex_edge, CIIG):
        A function that adds a user interface C_Ex_edge to the CIIG
    addExternalMessageEdgeTimeStamp (T_Ex_edge, CIIG):
        A function that adds a time stamp T_Ex_edge to the CIIG
```

Steps:

```
1:      begin
2:      C = null
3:      for all components $\in SQ$ do
4:          addComponentNode(C,CIIG)
5:          if (an incoming message)
6:              addProviderInterface(C_P_Interf, CIIG)
7:          if (an outgoing message)
8:              addUserInterface(C_U_Interf, CIIG)
9:          if (an incoming message and an outgoing message are consecutive)
10:             addInternalMessageEdge(C_In_edge, CIIG)
11:     for all messages $\in SQ$ do
12:         addExternalMessageEdge(C_Ex_edge,CIIG)
13:         addExternalMessageEdgeTimeStamp(T_Ex_edge,CIIG)
```

Figure 4.13: An algorithm to generate the CIIG

Explanation of the algorithm

The algorithm takes a component level sequence diagram as an input and returns a CIIG model as an output.

Lines 3 and 4 identify all the components involved in a sequence diagram. Using the sequence diagram shown in Figure 4.12 as an example, components include Ethernet Device, SCA Ethernet Adapter, IP Interface Adapter, and IP Core.

Lines 5 through 8 identify all the interactions between components. Lines 9 through 13 connect all the components with external message edges as well as determining all the internal message edges in a component.

A CIIG is generated by applying the above algorithm as shown in Figure 4.14.

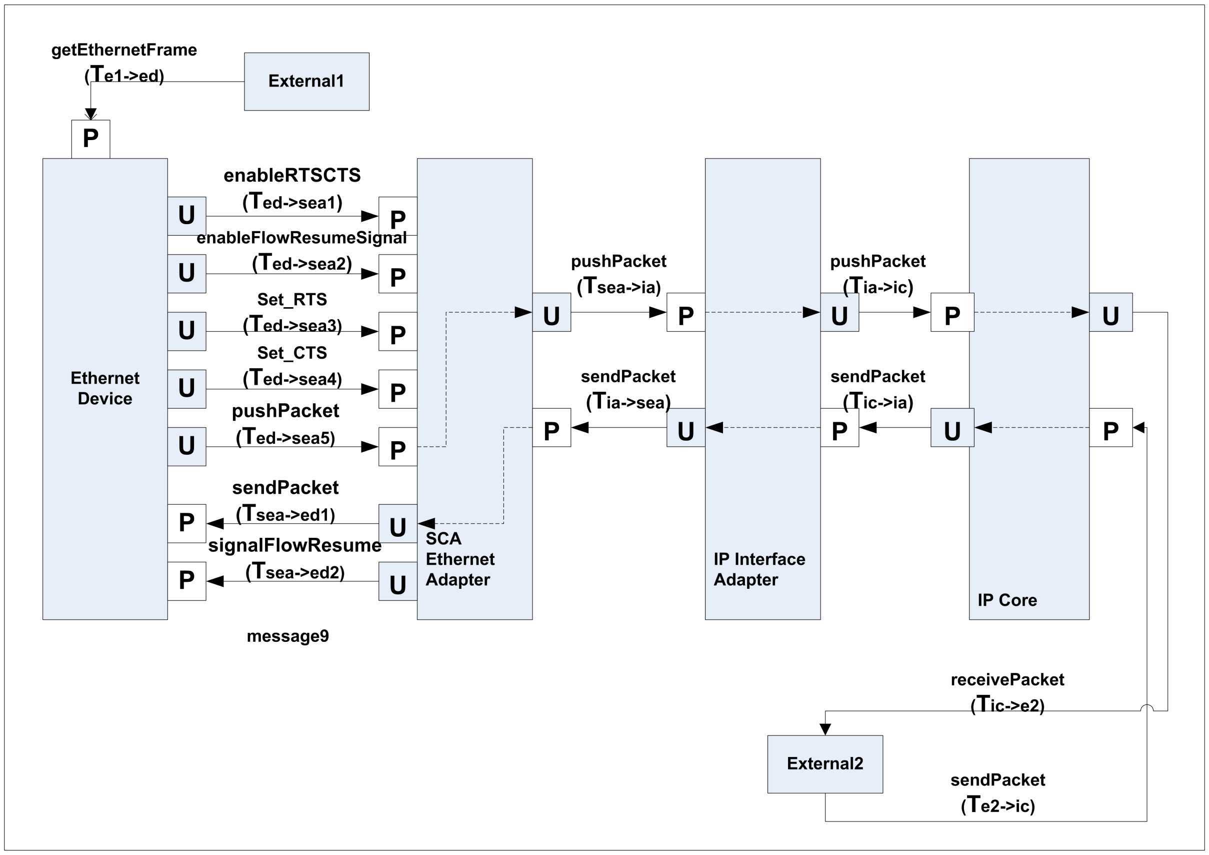


Figure 4.14: The CIIG of the system

Each message in the sequence diagram has a well-defined sequence number, source, and destination component names. On the other hand, the state diagram of a component defines its states and the transition messages it can receive in those states. The CREATEG annotates the chain of messages defined in the sequence diagrams with the state information by generating a CSIBG diagram. A CSIBG is created from a CIIG and state transitions in state diagrams for each corresponding component.

Figures 4.15, 4.16, 4.17, and 4.18 show state diagrams for each component. Events are also shown in each state diagrams.

An Ethernet Device component has five states. The Ethernet Device states are illustrated in Figure 4.15. The Ethernet Device states ensure that received operations are only executed when the Ethernet Device is in the proper state. The five states of the Ethernet Device are as follow:

- CONSTRUCTED – The state transitioned to upon successful creation.
- INITIALIZED – The state transitioned to upon successful initialization.
- ENABLED – The state transitioned to upon successful start.
- DISABLED – The state transitioned to upon successful stop.
- RELEASED – The state transitioned to upon successful release.

The Ethernet Device transitions between states in response to the *initialize*, *start*, *stop* and *releaseObject* operations.

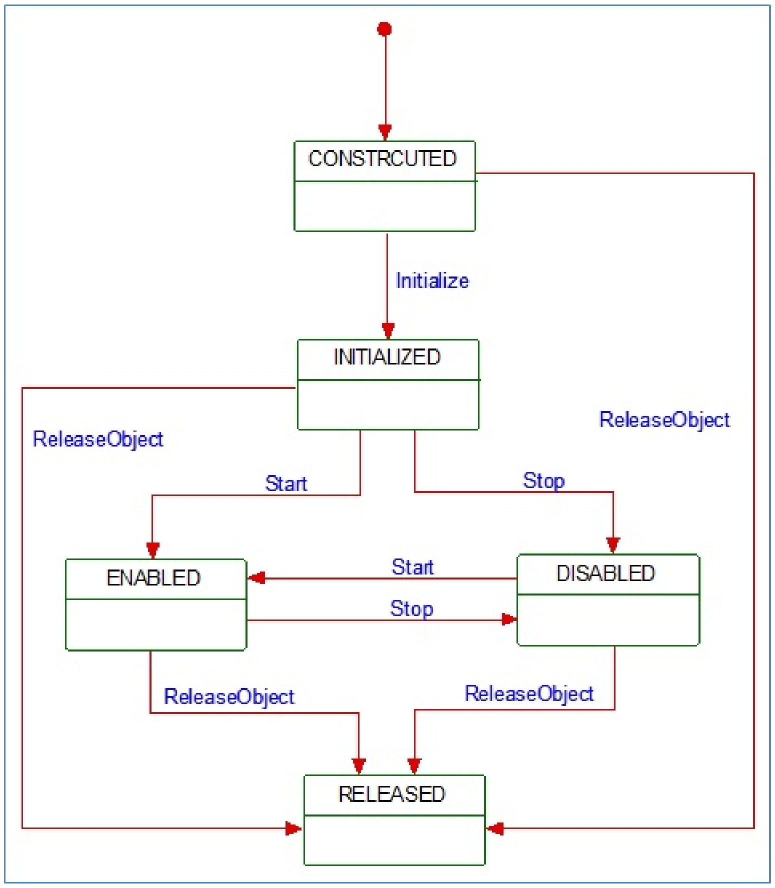


Figure 4.15: State diagram for ethernet device

An Ethernet Device Adapter component has two states. The Ethernet Device Adapter states are illustrated in Figure 4.16. A transition from start to the ENABLED operational state occurs whenever the Ethernet Device Adapter is instantiated, initialized and configured with an initial configuration. While in this state, the Ethernet Device Adapter is able to receive and transmit packets. The Ethernet Device Adapter transitions to the DISABLED operational state from ENABLED operational state upon invocation of the stop or reset operations, or upon detection of an unrecoverable error. The operational state indicates whether or not the Ethernet Device Adapter is functioning.

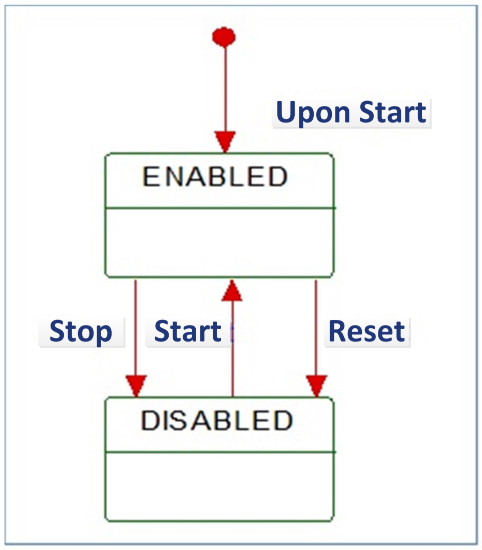


Figure 4.16: State diagram for SCA ethernet adapter

An IP Interface Adapter component has four states. The IP Interface Adapter states are illustrated in Figure 4.17. The IP Interface Adapter stores a list of interface IDs along with the type of interface object for each. This information is obtained from the configuration file shared with the IP Core applications. To communicate with the multiple channels from the radio application, the list will be created dynamically by the IP Core configuration manager for each interface. When the IP Interface Adapter receives a packet from an SCA Ethernet interface, it goes to the Processing state to look up the interface table to find the appropriate IP Core interface handle for the Ethernet frame. The IP Interface Adapter removes the Ethernet header, puts the packets in the queue and moves to the Storing state. When the IP Core is ready to accept the packet, the IP Interface Adapter moves to the Forwarding state and sends the packet to IP Core.

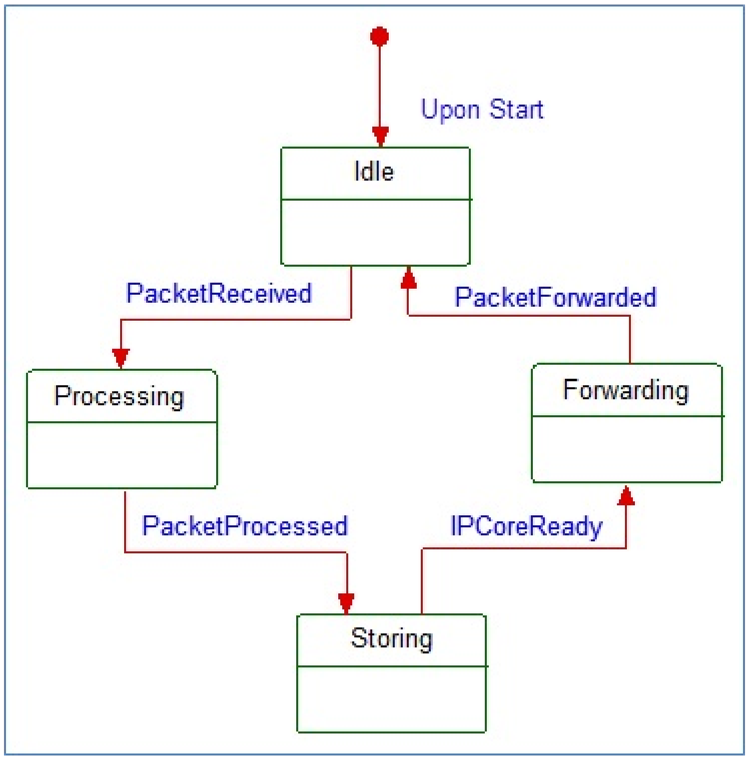


Figure 4.17: State diagram for IP interface adapter

An IP Core component has four states. The IP Core states are illustrated in Figure 4.18. It receives packets in the Receiving state and transmits packets in the Sending state.

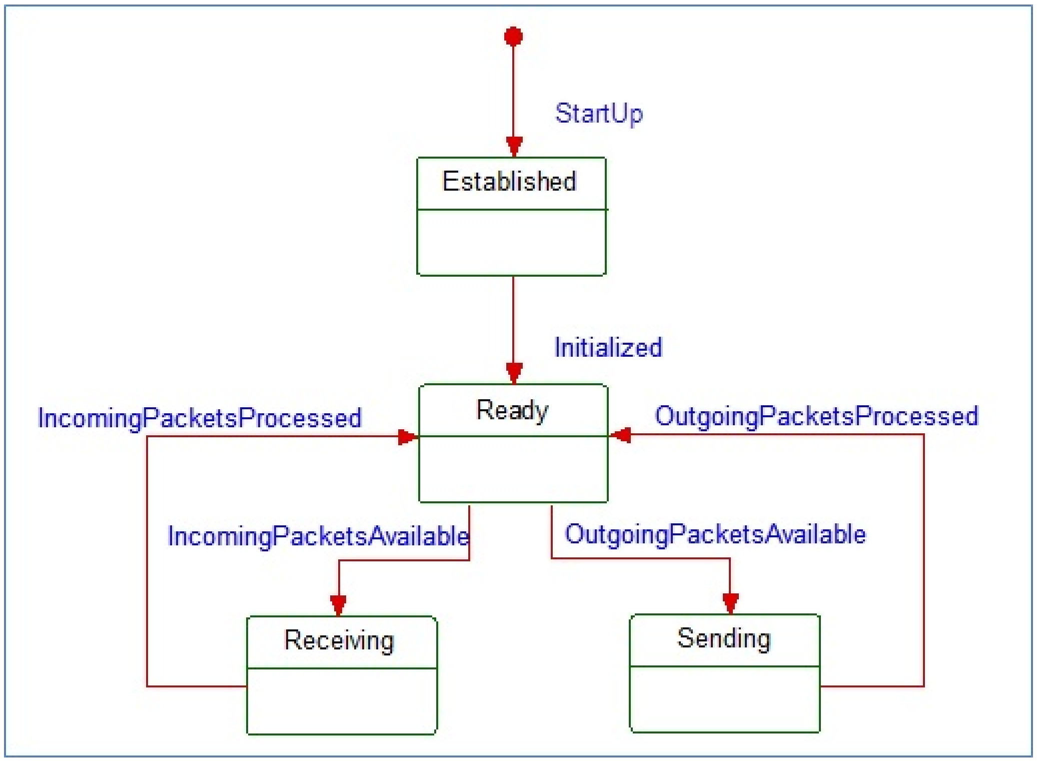


Figure 4.18: State diagram for IP core

The CSIBG is created based on state transitions in each state diagram on top of the CIIG generated earlier:

- For each component in the CIIG, one component subnet is created.
- For each provider interface in the CIIG, one component provider interface is created in its component subnet.
- For each user interface in the CIIG, one component user interface is created in its component subnet.
- For each external component message edge in the CIIG, one external component message edge is created.
- For each state in a state diagram, one state node is created in its component subnet.

- For each transition in the state diagram, one transition edge is created.
- For each provider interface in the CSIBG, one or more input message edge is created corresponding to the possible entry state.
- For each user interface in the CSIBG, one output message edge is created to connect from the exit state.

The CSIBG for the above system is shown in Figure 4.19.

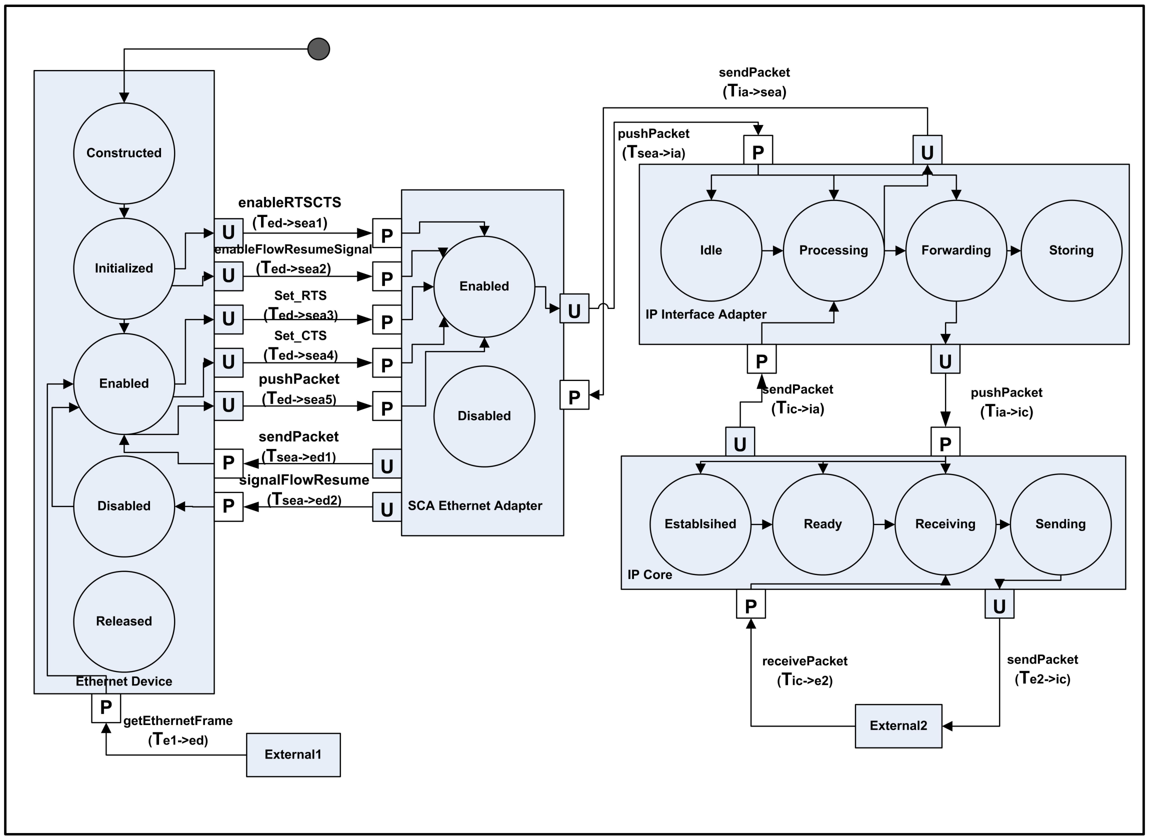


Figure 4.19: The CSIBG of the system

The algorithm to generate a CSIBG is shown in Figure 4.20.

Algorithm BuildCSIBG(SQ and state diagrams): CSIBG

Inputs SQ: *Component level sequence diagram corresponding to a message*

S: *Set of state diagrams required for components involved in a sequence diagram*

Output CSIBG: *CSIBG model generated by the algorithm*

Declare:

```
$C_n = \{Cn_1, Cn_2, \ldots, Cn_k\}$:
    A finite set of component subnets
$S_i = (S_1, S_2, \ldots, S_k)$:
    A finite set of states in Component Ci
$P_i = (P_1, P_2, \ldots, P_k)$:
    A finite set of provider interfaces in Component C
$U_i = (U_1, U_2, \ldots, U_k)$:
    A finite set of user interfaces in Component C
$T_i = (T_1, T_2, \ldots, T_k)$:
    A finite set of transition edges in Component Ci
$In_i = (In_1, In_2, \ldots, In_k)$:
    A finite set of input message edges in Component Ci
$O_i = (O_1, O_2, \ldots, O_k)$:
    A finite set of output message edges in Component Ci
$Msg_n = (Msg_1, Msg_2, \ldots, Msg_k)$:
    A finite set of interaction messages between component subnets
$Tst_n = (Tst_1, Tst_2, \ldots, Tst_k)$:
    A finite set of time stamps on transition edges
$Tpm_n = (Tpm_1, Tpm_2, \ldots, Tpm_k)$:
    A finite set of time stamps on input message edges
$Tum_n = (Tum_1, Tum_2, \ldots, Tum_k)$:
    A finite set of time stamps on output message edges
$T\_Ex\_edge = (T_i \rightarrow_j, \ldots, T_j \rightarrow_i)$:
    A finite set of time stamps at external component message edges
addComponentSubnet(Cn, CSIBG):
    A function that adds a node Cn to the CSIBG
addStateNode(S, CSIBG):
    A function that adds a node S to a component subnet Cn to the CSIBG
addProviderInterface(P, CSIBG):
    A function that adds a provider interface P to the CSIBG
addUserInterface (U, CSIBG):
    A function that adds a user interface U to the CSIBG
addInputMessageEdge (providerInterface , state, CSIBG):
    A function that adds a message edge from a provider interface to a state in the CSIBG
addInputMessageEdgeTimeStamp (tpm, providerInterface , state, CSIBG):
    A function that adds a time stamp with the message edge from a provider interface
    to a state in the CSIBG
```

(Continued)

addOutputMessageEdge *(state, userInterface, CSIBG):*
A function that adds a message edge from a state to a user interface in the CSIBG
addOutputMessageEdgeTimeStamp *(tum, state, userInterface, CSIBG):*
A function that adds a time stamp with the message edge from a state to a user interface in the CSIBG
addTransitionEdge *(sourceState, targetState, CSIBG):*
A function that adds a transition edge from a source state to a target state including guard if any exists in the CSIBG
addTransitionEdgeTimeStamp *(tst, sourceState, targetState, CSIBG):*
A function that adds a time stamp with the transition edge from a source state to a target state
addMessageEdge *(Msgn, CSIBG):*
A function that adds a message Msgn to the CSIBG
addExternalMessageEdgeTimeStamp *(T_Ex_edge, CSIBG):*
A function that adds a time stamp T_Ex_edge to the CSIBG

Steps:

```
1:   begin
2:   Cn = null
3:   for all C ∈ CIIG do
4:       addComponentSubnet(Cn,BG)
5:       for all C_P_Interf ∈ C do
6:           addProviderInterface(P, CSIBG)
7:       for all C_U_Interf ∈ C do
8:           addUserInterface(U, CSIBG)
9:   for all C_Ex_edge ∈ CIIG do
10:      addMessageEdge(Msgn, CSIBG)
11:          addMessageEdgeTimeStamp(T, Msgn, CSIBG)
12:  for all Cn ∈ CSIBG do
13:      for all states ∈ State Diagram do
14:          addStateNode(S, CSIBG)
15:          if (an incoming message)
16:              addInputMessageEdge(providerInterface, state, CSIBG)
17:              addInputMessageEdgeTimeStamp(tpm, providerInterface, state, CSIBG)
18:          if (an outgoing message)
19:              addIOutputMessageEdge(state, userInterface, CSIBG)
20:              addOutputMessageEdgeTimeStamp(tum, state, userInterface, CSIBG)
21:          if (a transition from sourceState to targetState)
22:              addTransitionEdge(sourceState, targetState, CSIBG)
23:              addTransitionEdgeTimeStamp(tst, sourceState, targetState, CSIBG)
```

Figure 4.20: An algorithm to generate the CSIBG

Explanation of the algorithm

The algorithm takes a component level sequence diagram and a set of state diagrams of each component involved in a sequence diagramas an input and returns a CSIBG model as an output.

Lines 3 through 11 identify all the components and interactions between components.

Lines 12 and 13 identify all the states involved in a state diagram.

Lines 14 through 23 identify all the connections between states. Lines 15 through 20 connect all the states with external message edges as well as determining all the transitions between states in a component.

A CSIBG is automatically generated by applying the algorithm shown in Figure 4.21.

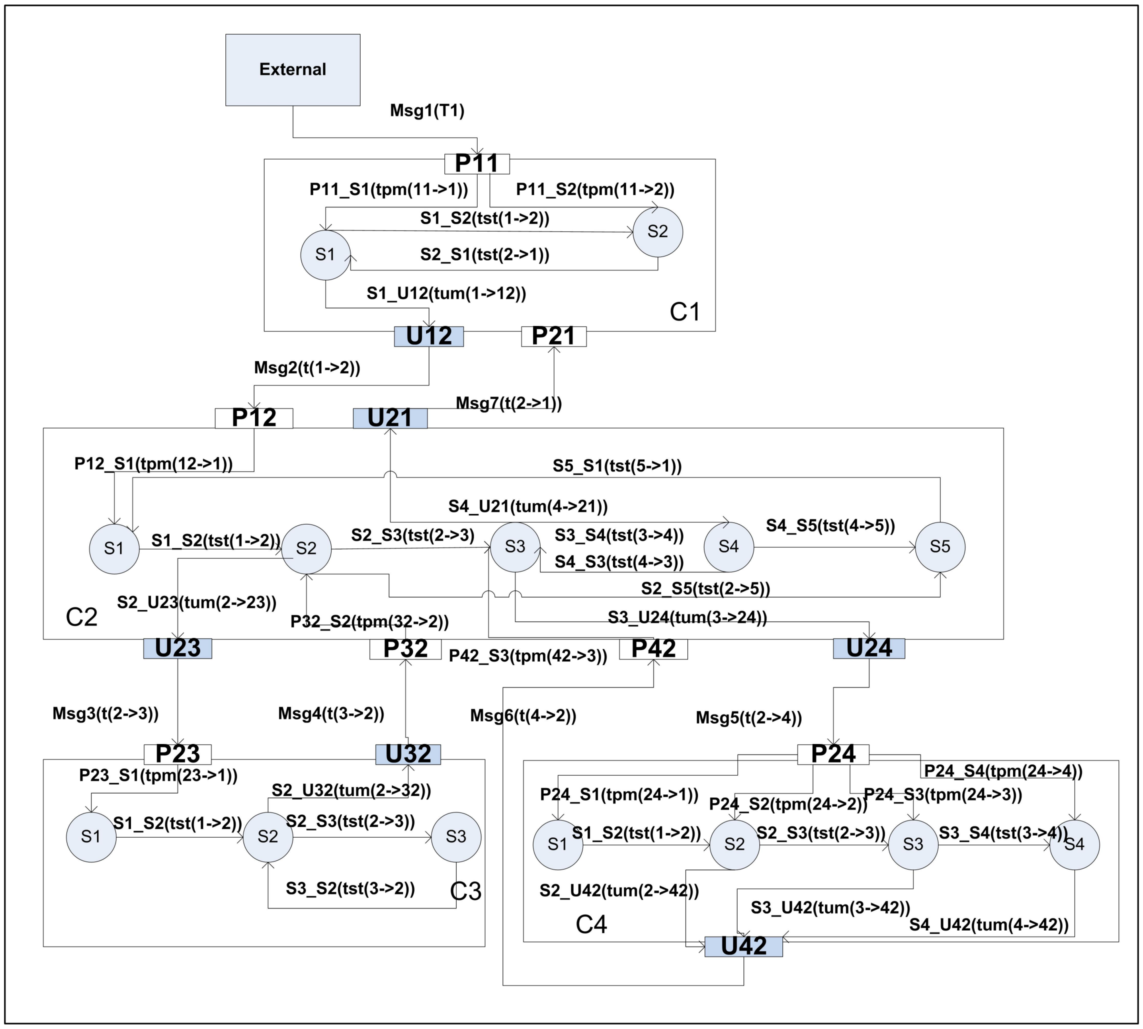


Figure 4.21: A complete CSIBG

This CSIBG will be used as a basis for generating test paths in the next subsection. The algorithm to generate a CSEDIBG is shown in Figure 4.22.

Algorithm BuildCSEDIBG(SQ and state diagrams): CSEDIBG

Inputs SQ: *Component level sequence diagram corresponding to a message*

S: *Set of state diagrams required for components involved in a sequence diagram*

Output CSEDIBG: *CSEDIBG model generated by the algorithm*

Declare:

```
    $C_n = \{Cn_1, Cn_2, \ldots, Cn_k\}$:
        A finite set of component subnets
    $S_i = (S_1, S_2, \ldots, S_k)$:
        A finite set of states in Component Ci
    $EC_i = (EC_1, EC_2, \ldots, EC_k)$:
        A finite set of event captures in State Si
    $E_i = (E_1, E_2, \ldots, E_k)$:
        A finite set of events in State Si
    $EH_i = (EH_1, EH_2, \ldots, EH_k)$:
        A finite set of event handlers in State Si
    $P_i = (P_1, P_2, \ldots, P_k)$:
        A finite set of provider interfaces in Component C
    $U_i = (U_1, U_2, \ldots, U_k)$:
        A finite set of user interfaces in Component C
    $T_i = (T_1, T_2, \ldots, T_k)$:
        A finite set of transition edges in Component Ci
    $In_i = (In_1, In_2, \ldots, In_k)$:
        A finite set of input message edges in Component Ci
    $O_i = (O_1, O_2, \ldots, O_k)$:
        A finite set of output message edges in Component Ci
    $Msg_n = (Msg_1, Msg_2, \ldots, Msg_k)$:
        A finite set of interaction messages between component subnets
    $Tst_n = (Tst_1, Tst_2, \ldots, Tst_k)$:
        A finite set of time stamps on transition edges
    $Tpm_n = (Tpm_1, Tpm_2, \ldots, Tpm_k)$:
        A finite set of time stamps on input message edges
    $Tum_n = (Tum_1, Tum_2, \ldots, Tum_k)$:
        A finite set of time stamps on output message edges
    $T\_Ex\_edge = (T_i \rightarrow_j, \ldots, T_j \rightarrow_i)$:
        A finite set of time stamps at external component message edges
    addComponentSubnet(Cn, CSEDIBG):
        A function that adds a node Cn to the CSEDIBG
    addStateNode(S, CSEDIBG):
        A function that adds a node S to a component subnet Cn to the CSEDIBG
    addProviderInterface(P, CSEDIBG):
        A function that adds a provider interface P to the CSEDIBG
    addUserInterface (U, CSEDIBG):
        A function that adds a user interface U to the CSEDIBG
    addInputMessageEdge (providerInterface , state, CSEDIBG):
        A function that adds a message edge from a provider interface to a state in the CSEDIBG
```

(Continued)

addInputMessageEdgeTimeStamp *(tpm, providerInterface, state, CSEDIBG):*
A function that adds a time stamp with the message edge from a provider interface to a state in the CSEDIBG

addECEdge *(providerInterface, EC, CSEDIBG):*
A function that adds a message edge from a provider interface to an event capture in the CSEDIBG

addEEdge *(EC , E, CSEDIBG):*
A function that adds a message edge from an event capture to an event of a state in the CSEDIBG

addEHEdge *(E , EH, CSEDIBG):*
A function that adds a message edge from an event to an event handler of a state in the CSEDIBG

addIOutputMessageEdge *(state, userInterface, CSEDIBG):*
A function that adds a message edge from a state to a user interface in the CSEDIBG

addOutputMessageEdgeTimeStamp *(tum, state, userInterface, CSEDIBG):*
A function that adds a time stamp with the message edge from a state to a user interface in the CSEDIBG

addTransitionEdge *(sourceState, targetState, CSEDIBG):*
A function that adds a transition edge from a source state to a target state including a guard if any exists in the CSEDIBG

addTransitionEdgeTimeStamp *(tst, sourceState, targetState, CSEDIBG):*
A function that adds a time stamp with the transition edge from a source state to a target state

addMessageEdge *(Msgn, CSEDIBG):*
A function that adds a message Msgn to the CSEDIBG

addMessageEdgeTimeStamp *(T_Ex_edge, CSEDIBG):*
A function that adds a time stamp T_Ex_edge to the CSEDIBG

Steps:

```
1:   begin
2:   $Cn = null$
3:   for all C $\in$ CIIG do
4:       addComponentSubnet(Cn,CSEDIBG)
5:       for all C_P_Interf $\in$ C do
6:           addProviderInterface(P, CSEDIBG)
7:       for all C_U_Interf $\in$ C do
8:           addUserInterface(U, CSEDIBG)
9:   for all C_Ex_edge $\in$ CIIG do
10:      addMessageEdge(Msgn, CSEDIBG)
11:          addMessageEdgeTimeStamp(T, Msgn, CSEDIBG)
12:  for all Cn $\in$ CSEDIBG do
13:      for all states $\in$ State Diagram do
14:          addStateNode(S, CSEDIBG)
15:          if (an incoming message)
16:              addInputMessageEdge(providerInterface, state, CSEDIBG)
17:              addInputMessageEdgeTimeStamp
18:              (tpm, providerInterface, state, CSEDIBG)
19:             for each incoming message $\in$ CSEDIBG do
20:              addECEdge(providerInterface , ec, CSEDIBG)
21:              addEEdge(ec , event, CSEDIBG)
22:              addEHEdge(event , eh, CSEDIBG)
```

(Continued)

```
23:         if (an outgoing message)
24:             addIOutputMessageEdge(state, userInterface, CSEDIBG)
25:             addOutputMessageEdgeTimeStamp
26:             (tum, state, userInterface, CSEDIBG)
27:         if (a transition from sourceState to targetState)
28:             addTransitionEdge(sourceState , targetState, CSEDIBG)
29:             addTransitionEdgeTimeStamp(tst, sourceState, targetState, CSEDIBG)
```

Figure 4.22: An algorithm to generate the CSEDIBG

Explanation of the algorithm

The algorithm takes a component level sequence diagram and a set of state diagrams of each component involved in a sequence diagramas an input and returns a CSIBG model as an output.

Lines 3 through 16 identify all the components and interactions between components.

Lines 17 through 27 identify all the connections between external message, event captures, event handlers and states.

4.3 Generating Test Paths from the CREATEG

This section discusses the generation of test paths from the CREATEG CSIBG diagram. Each path tests some interactions between components in appropriate states. Traversing all CREATEG paths tests all interactions between components in all possible and valid states of components involved in a particular interaction.

A test path derived from the CREATEG represents a path that starts with the initial (null) node and contains a complete message sequence of a system level operation. The total number of test paths in a CREATEG can be determined by taking the product of the number of transition paths in each CSIBG component subnet, where each transition path is an internal transition of a component from a source state to a target state on receipt of a particular message.

In component-based real-time embedded systems, complexity arises when components interact with each other concurrently. Multiple threads are running independently so we allow the model to associate a set of events with a transition. A transition may be triggered by the concurrent occurrence of a set of events.

In the specification of a real-time system, a transition often occurs at a time that depends on the time of another transition. It is therefore necessary to reference a given transition occurrence time. UML does not provide a way to identify transitions in state diagrams, although it is possible in sequence diagrams. We removed this limitation by allowing the modeler to assign labels to transitions, and to make reference to these labels in guards.

Recall from the earlier discussion on the CREATEG construction in section 4.2 that we select only those transitions from the state diagram of a component that are valid for a particular message of the interaction. However, when they are guard conditions, not all paths generated by traversing the CREATEG are necessarily feasible. Infeasible paths must therefore be detected manually by inspecting all paths containing guard conditions.

The total number of test paths can be calculated by taking the cross product of all transition paths of components involved in an interaction. It gives a complete coverage corresponding to a called method in an interaction.

Figure 4.23 presents an algorithm for test path generation from a CREATEG instance.

Algorithm AllPathCoverage(T): TPSSeq

Input T: *CREATEG Test Model*

Output TPSSeq: *A sequence of test paths*

Declare:

TPSSeq: A sequence of test paths
MESSeq: A sequence of message edges in T
medg: An interaction message edge in T in the form of "Msgn" + "→"
+ " the provider interface Pi" with a time stamp in the form of "Tn"
or in the form of " the user interface Uij" + "→" + "Msgn"
with a time stamp in the form of "Tn"
Imedg: An input message edge in T in the form of "Pi" incoming Msgn
from the provider interface Pij) + "→" + "entryState in a Cn"
with a time stamp in the form of "tpm_ij"+ "→" + "n"
Omedg: An output message edge in T in the form of "destinationState in a Cn"
+ "Uij" (outgoing Msgn to the user interface Uij)
with a time stamp in the form of "tum_n"+ "→" + "ij"
tedg: A transition edge in T, in the form of "sourceState" + "→" + "targetState"
with a time stamp in the form of "tst_m"+ "→" + "tst_n"
TMESeqn: A sequence of test sub paths in a Cn
from an Imedge (corresponding to an incoming interaction message Msgn)
to an Omedge (corresponding to an outgoing interaction message Msg(n+1))
tmepath: A complete test sub path in TMESeqn

Steps:

```
1:   begin
2:   TPSSeq ← {}
3:   MESSeq ← T.edge.message
4:   for each medg ∈ MESSeq do
5:       for each provider component subnet where the medg comes to
6:           TMESeqn = createPath(Msgn, Msg(n+1))
7:           for each reachable entry state
8:               if there exists a path from the entryState to a destinationState to Msg(n+1)
9:                   addPath(Imedg, TMESeqn)
10:                   addPath(tedg, TMESeqn)
11:               while (a tedg exists on the path)
12:                   addPath(tedg, TMESeqn)
13:               addPath(Omedg, TMESeqn)
14:   for each tmepath ∈ TMESeqn do
15:       TPSSeq = CreatePath(Msgn, Msgm)
16:       addPath(tmepath, TPSSeq)
17:       while (a TMESeqn exists on the path)
18:           for each tmepath ∈ TMESeq(n+1) do
19:               addPath(tmepath, TPSSeq)
```

Figure 4.23: An algorithm to generate test paths for the All-path coverage criterion

Explanation of the algorithm

The algorithm takes a CREATEG test model as an input and returns a set of test paths as an output.

Line 4 identifies all the message edges in a CREATEG test model. Using the CREATEG shown in Figure 4.21 as an example, message edges include Msg1, Msg2, Msg3, Msg4, Msg5, Msg6, and Msg7.

Lines 5, 6 and 13 identify all the sequence of test paths between an input message edge and an output message edge. Sequences from Figure 4.21 include:

TMESeq1{Msg1 $\rightarrow Msg_2$}

TMESeq2{Msg2 $\rightarrow Msg_3$}

TMESeq3{Msg3 $\rightarrow Msg_4$}

TMESeq4{Msg4 $\rightarrow Msg_5$}

TMESeq5{Msg5 $\rightarrow Msg_6$}

TMESeq6{Msg6 $\rightarrow Msg_7$}

Lines 7 to 12 generate a test path including all the transition edges in a sequence. The complete sequence are shown in Figure 4.24.

Lines 14 to 19 generate a complete test path.

$$\textit{TMESeq1}\{\mathbf{Msg1}\xrightarrow{T_1}P_{11}\xrightarrow{tpm_{11\to1}}S_1\xrightarrow{tum_{1\to12}}U_{12}\xrightarrow{T_2}\mathbf{Msg_2};$$
$$\mathbf{Msg1}\xrightarrow{T_1}P_{11}\xrightarrow{tpm_{11\to1}}S_1\xrightarrow{tst_{1\to2}}S_2\xrightarrow{tst_{2\to1}}S_1\xrightarrow{tum_{1\to12}}U_{12}\xrightarrow{T_2}\mathbf{Msg_2}\}$$

$$\textit{TMESeq2}\{\mathbf{Msg2}\xrightarrow{T_2}P_{12}\xrightarrow{tpm_{12\to1}}S_1\xrightarrow{tst_{1\to2}}S_2\xrightarrow{tum_{2\to23}}U_{23}\xrightarrow{T_3}\mathbf{Msg_3};$$
$$\mathbf{Msg2}\xrightarrow{T_2}P_{12}\xrightarrow{tpm_{12\to1}}S_1\xrightarrow{tst_{1\to2}}S_2\xrightarrow{tst_{2\to5}}S_5\xrightarrow{tst_{5\to1}}S_1\xrightarrow{tst_{1\to2}}S_2$$
$$\xrightarrow{tum_{2\to23}}U_{23}\xrightarrow{T_3}\mathbf{Msg_3};$$
$$\mathbf{Msg2}\xrightarrow{T_2}P_{12}\xrightarrow{tpm_{12\to1}}S_1\xrightarrow{tst_{1\to2}}S_2\xrightarrow{tst_{2\to3}}S_3\xrightarrow{tst_{3\to4}}S_4\xrightarrow{tst_{4\to5}}S_5$$
$$\xrightarrow{tst_{5\to1}}S_1\xrightarrow{tst_{1\to2}}S_2\xrightarrow{tum_{2\to23}}U_{23}\xrightarrow{T_3}\mathbf{Msg_3}\}$$

$$\textit{TMESeq3}\{\mathbf{Msg3}\xrightarrow{T_3}P_{23}\xrightarrow{tpm_{23\to1}}S_1\xrightarrow{tst_{1\to2}}S_2\xrightarrow{tum_{2\to32}}U_{32}\xrightarrow{T_4}\mathbf{Msg_4};$$
$$\mathbf{Msg3}\xrightarrow{T_3}P_{23}\xrightarrow{tpm_{23\to1}}S_1\xrightarrow{tst_{1\to2}}S_2\xrightarrow{tst_{2\to3}}S_3\xrightarrow{tst_{3\to2}}S_2\xrightarrow{tum_{2\to32}}U_{32}\xrightarrow{T_4}\mathbf{Msg_4}\}$$

$$\textit{TMESeq4}\{\mathbf{Msg4}\xrightarrow{T_4}P_{32}\xrightarrow{tpm_{32\to2}}S_2\xrightarrow{tst_{2\to3}}S_3\xrightarrow{tum_{3\to24}}U_{24}\xrightarrow{T_5}\mathbf{Msg_5};$$
$$\mathbf{Msg4}\xrightarrow{T_4}P_{32}\xrightarrow{tpm_{32\to2}}S_2\xrightarrow{tst_{2\to5}}S_5\xrightarrow{tst_{5\to1}}S_1\xrightarrow{tst_{1\to2}}S_2\xrightarrow{tst_{2\to3}}S_3$$
$$\xrightarrow{tum_{3\to24}}U_{24}\xrightarrow{T_5}\mathbf{Msg_5}\}$$

$$\textit{TMESeq5}\{\mathbf{Msg5}\xrightarrow{T_5}P_{24}\xrightarrow{tpm_{24\to1}}S_1\xrightarrow{tst_{1\to2}}S_2\xrightarrow{tum_{2\to42}}U_{42}\xrightarrow{T_6}\mathbf{Msg_6};$$
$$\mathbf{Msg5}\xrightarrow{T_5}P_{24}\xrightarrow{tpm_{24\to1}}S_1\xrightarrow{tst_{1\to2}}S_2\xrightarrow{tst_{2\to3}}S_3\xrightarrow{tum_{2\to42}}U_{42}\xrightarrow{T_6}\mathbf{Msg_6};$$
$$\mathbf{Msg5}\xrightarrow{T_5}P_{24}\xrightarrow{tpm_{24\to1}}S_1\xrightarrow{tst_{1\to2}}S_2\xrightarrow{tst_{2\to3}}S_3\xrightarrow{tst_{3\to4}}S_4\xrightarrow{tum_{2\to42}}U_{42}\xrightarrow{T_6}\mathbf{Msg_6};$$
$$\mathbf{Msg5}\xrightarrow{T_5}P_{24}\xrightarrow{tpm_{24\to2}}S_2\xrightarrow{tum_{2\to42}}U_{42}\xrightarrow{T_6}\mathbf{Msg_6};$$
$$\mathbf{Msg5}\xrightarrow{T_5}P_{24}\xrightarrow{tpm_{24\to3}}S_3\xrightarrow{tum_{3\to42}}U_{42}\xrightarrow{T_6}\mathbf{Msg_6};$$
$$\mathbf{Msg5}\xrightarrow{T_5}P_{24}\xrightarrow{tpm_{24\to2}}S_4\xrightarrow{tum_{4\to42}}U_{42}\xrightarrow{T_6}\mathbf{Msg_6}\}$$

$$\textit{TMESeq6}\{\mathbf{Msg6}\xrightarrow{T_6}P_{42}\xrightarrow{tpm_{42\to3}}S_3\xrightarrow{tum_{3\to21}}U_{21}\xrightarrow{T_7}\mathbf{Msg_7};$$
$$\mathbf{Msg6}\xrightarrow{T_6}P_{42}\xrightarrow{tpm_{42\to3}}S_3\xrightarrow{tst_{3\to4}}S_4\xrightarrow{tst_{4\to5}}S_5\xrightarrow{tst_{5\to1}}S_1\xrightarrow{tum_{1\to21}}U_{21}\xrightarrow{T_7}\mathbf{Msg_7};$$
$$\mathbf{Msg6}\xrightarrow{T_6}P_{42}\xrightarrow{tpm_{42\to3}}S_3\xrightarrow{tst_{3\to4}}S_4\xrightarrow{tst_{4\to3}}S_3\xrightarrow{tum_{1\to21}}U_{21}\xrightarrow{T_7}\mathbf{Msg_7}\}$$

Figure 4.24: Sequences of test path for all the component subnets

Algorithm *AllPathCoverage* in Figure 4.23 takes a CREATEG Test Model as its input and returns a set of all possible test paths shown in Figure 4.24 for the CREATEG. Whenever a component is encountered, the current number of test paths is multiplied by the number of transition paths within the encountered component.

Each generated test path is in the form of a string representing the sequence of messages on objects in particular states starting from the system-level operation call message that triggers the use case execution modeled by the sequence diagram.

In test paths, messages are identified by their names and sequence numbers. Test paths consist of sequences of message expressions describing a complete message sequence for a system-level operation call. The general form of a message expression is shown in Figure 4.25.

$$
\begin{aligned}
&\text{“}Msg\text{”} + Sequence\_NO \rightarrow InputMessageEdge \rightarrow entry\_state \rightarrow \\
&\textstyle\sum(start\_state \rightarrow resultant\_state) \rightarrow exit\_state \rightarrow OutputMessageEdge \rightarrow \\
&\text{“}Msg\text{”} + (Sequence\_NO + 1) \rightarrow InputMessageEdge \rightarrow entry\_state \rightarrow \\
&\textstyle\sum(start\_state \rightarrow resultant\_state) \rightarrow exit\_state \rightarrow OutputMessageEdge \rightarrow \\
&\text{“}Msg\text{”} + (Sequence\_NO + 2) \rightarrow \ \ldots) \\
&\rightarrow InputMessageEdge \rightarrow entry\_state \rightarrow \\
&\textstyle\sum(start\_state \rightarrow resultant\_state) \rightarrow exit\_state \rightarrow OutputMessageEdge \rightarrow \\
&\text{“}Msg\text{”} + (Sequence_N O + numberOfTotalMessage - 1)
\end{aligned}
$$

Figure 4.25: Message expression for a message sequence

The sequence number on the CREATEG is identical to the corresponding sequence diagram sequence number. In this specific case, the total number of test paths is calculated by taking the cross product of all transition paths of components involved in an interaction $= 2 * 3 * 2 * 2 * 6 * 3 = 432$. It gives a complete coverage corresponding to a called method in an interaction. The test generator is responsible for generating these test paths.

4.4 Coverage Criteria for Test Paths

Section 4.3 presented an algorithm to generate all test paths from CREATEG and illustrated it using an example. However, for more complex systems, the number of components involved in a collaboration and the number of states in components can be large. This typically results in exponential growth of the number of test paths that can be generated. It may be impractical, or even impossible, to test all the paths due to the cost involved. To allow for an acceptable level of testing while keeping the cost reasonable, we define several coverage criteria based on CREATEG. These coverage criteria are adapted from graph coverage criteria and logic criteria defined in Ammann and Offutt's book [9].

An appropriate path coverage criterion can be chosen based on the level of testing required, and the test budget available.

4.4.1 All-Interface coverage

This criterion ensures that each interface between components is tested once. It is adapted from *edge coverage*, which tours each reachable subpath of length less than or equal to one in a CIIG [9]. However, this can be used to check if the interactions between components are taking place correctly, regardless of the component states. A test path generation algorithm for All-Interface Coverage is given in Figure 4.26.

Algorithm SinglePathCoverage(T): TPSSeq

Input T: *CREATEG Test Model*

Output TPSSeq: *A sequence of test paths*

Declare:

TPSSeq: A sequence of test paths
MESSeq: A sequence of message edges in T
medg: An interaction message edge in T in the form of "Msgn" + "→"
+ " the provider interface Pi" with a time stamp in the form of "Tn"
or in the form of " the user interface Uij" + "→" + "Msgn"
with a time stamp in the form of "Tn"
Imedg: An input message edge in T in the form of "Pi" (incoming Msgn
from the provider interface Pij) + "→" + "entryState in a Cn"
with a time stamp in the form of "tpm_ij"+ "→" + "n"
Omedg: An output message edge in T in the form of "destinationState in a Cn"
+ "Uij" (outgoing Msgn to the user interface Uij)
with a time stamp in the form of "tum_n"+ "→" + "ij"
tedg: A transition edge in T, in the form of "sourceState" + "→" + "targetState"
with a time stamp in the form of "tst_m"+ "→" + "tst_n"
TMESeqn: A sequence of test sub paths in a Cn
from an Imedge (corresponding to an incoming interaction message Msgn)
to an Omedge (corresponding to an outgoing interaction message Msg(n+1))
tmepath: A complete test sub path in TMESeqn

Steps:

```
1:     begin
2:     TPSSeq ← {}
3:     MESSeq ← T.edge.message
4:     for each medg ∈ MESSeq do
5:         for each provider component subnet where the medg comes to
6:             TMESeqn = createPath(Msgn, Msg(n+1))
7:             for each reachable entry state
8:                 if there exists a path from the entryState to a destinationState to Msg(n+1)
9:                     addPath(Imedg, TMESeqn)
10:                     addPath(tedg, TMESeqn)
11:                   while (a tedg exists on the path)
12:                     addPath(tedg, TMESeqn)
13:                 addPath(Omedg, TMESeqn)
14:      for each tmepath ∈ TMESeqn do
15:          TPSSeq = CreatePath(Msgn, Msgm)
16:          tmepath = random(TMESeq(n))
17:          addPath(tmepath, TPSSeq)
```

Figure 4.26: An algorithm to generate test paths for the All-Interface Coverage criterion

Explanation of the algorithm

The algorithm takes a CREATEG test model as an input and returns a set of test paths as an output.

Line 4 identifies all the message edges in a CREATEG test model.

Lines 5, 6 and 13 identify all the sequences of test paths between an input message edge and an output message edge.

Lines 7 through 12 generate a test path including all the interface edges in a sequence.

Lines 14 through 19 generate a complete test path.

4.4.2 All-Interface-Transition Coverage

This criterion ensures that each internal state transition path in a component is followed at least once. It is adapted from *edge coverage*, which tours each reachable subpath of length less than or equal to one in a CSIBG [9]. The All-Interface-Transition Coverage subsumes *All-Interface Coverage*.

The number of state transition paths in the component with the maximum number of sub-paths determines the number of paths generated by this criterion. For example, in the CREATEG model, the maximum number of transitions in components C1, C2, C3, and C4 is six, therefore the number of test paths generated by this criterion is six. It reveals some of the faults occurring due to invalid transitions within a component. However, this mainly focuses on how to check if the transitions between states are taking place correctly. It does not consider the impact from the events triggered by component interactions. A test path generation algorithm for All-Interface-Transition coverage is given in Figure 4.27.

Algorithm AllTransitionCoverage(T): TPSSeq

Input T: *CREATEG Test Model*

Output TPSSeq: *A sequence of test paths*

Declare:

TPSSeq: A sequence of test paths
MESSeq: A sequence of message edges in T
medg: An interaction message edge in T in the form of "Msgn" + "→"
+ " the provider interface Pi" with a time stamp in the form of "Tn"
or in the form of " the user interface Uij" + "→" + "Msgn"
with a time stamp in the form of "Tn"
Imedg: An input message edge in T in the form of "Pi" (incoming Msgn
from the provider interface Pij) + "→" + "entryState in a Cn"
with a time stamp in the form of "tpm_ij"+ "→" + "n"
Omedg: An output message edge in T in the form of "destinationState in a Cn"
+ "Uij" (outgoing Msgn to the user interface Uij)
with a time stamp in the form of "tum_n"+ "→" + "ij"
tedg: A transition edge in T, in the form of "sourceState" + "→" + "targetState"
with a time stamp in the form of "tst_m"+ "→" + "tst_n"
TMESeqn: A sequence of test sub paths in a Cn
from an Imedge (corresponding to an incoming interaction message Msgn)
to an Omedge (corresponding to an outgoing interaction message Msg(n+1))
tmepath: A complete test sub path in TMESeqn
countOfpath: The number of test sub paths in the TMESeq of a component
maxPath: The number of maximum test sub paths in among TMESeqn
TMEseq_key: A sequence of test sub paths in a Component which contains the maxPath

Steps:

```
1:   begin
2:   TPSSeq ← {}
3:   MESSeq ← T.edge.message
4:   for each medg ∈ MESSeq do
5:       for each provider component subnet where the medg comes to
6:           TMESeqn = createPath(Msgn, Msg(n+1))
7:           for each reachable entry state
8:               if there exists a path from the entryState to a destinationState to Msg(n+1)
9:                   addPath(Imedg, TMESeqn)
10:                   addPath(tedg, TMESeqn)
11:                while (a tedg exists on the path)
12:                   addPath(tedg, TMESeqn)
13:              addPath(Omedg, TMESeqn)
```

```
14:    for each tmepath ∈ TMESeqn do
15:        countOfpath = getCountOfPaths(TMESeq)
16:        if (countOfpath ≥ maxPath)
17:            maxPath = countOfpath
18:            TMEseq_key = TMESeq
19:    for (i=1 to maxPath )
20:        TPSSeq = createPath(Msgn, Msgm)
21:        for each tmepath ∈ TMEseq_key do
22:            addPath(tmepath, TPSSeq)
23:        while (a TMESeqns exists on the path)
24:            tmepath = random(TMESeq(n))
25:            addPath(tmepath, TPSSeq)
```

Figure 4.27: An algorithm to generate test paths for the All-Interface-Transition coverage criterion

Explanation of the algorithm

The algorithm takes a CREATEG test model as an input and returns a set of test paths as an output.

Line 4 identifies all the message edges in a CREATEG test model.

Lines 5, 6 and 13 identify all the sequence of test paths between an input message edge and an output message edge.

Lines 7 through 12 generate a test path including all the transition edges in a sequence.

Lines 14 through 25 generate a complete test path.

To test the other real-time embedded software characteristics, I define one more coverage criterion.

4.4.3 All-Interface-Event Coverage

A component interface passes messages from a user component to a provider component. In the provider component, the received message is processed by *event capture* to further generate *event* in the current component state. All the generated events are sent to *event handler* for making a decision in this state, to either transition to the *next state* or send an outgoing message to the next component by using a user to provider service. The decision made in *event handler* is modeled as a predicate expression.

The clauses in a predicate appear in the CSEDIBG model as events. To provide coverage for the event handler, an incoming message may need to be tested several times with different test data. For full event decision coverage, test data need to be generated such that all combinations of truth values of clauses in the predicate are formed. For instance, if a predicate expression contains n clauses, its complete coverage would require each path to be tested $2n$ times with different data values. This results in an enormous increase in the total number of paths to be tested. To reduce this exponential growth in the number of test paths, while still maintaining an acceptable level of coverage, several coverage criteria have been proposed in the literature. Simple approaches such as All True, All False, and All Primes were proposed by Binder [17].

Active Clause Criteria, Inactive Clause Criteria, and their variants were defined by Ammann and Offutt [9]. Guan, Offutt, and Ammann [41] presented results from an industrial case study of logic-based testing applied to safety-critical embedded software, and compared the logic-based tests with tests created by testers at the company using manual functional testing. This study gives us confidence that logic-based test criteria can successfully increase the reliability of industry software.

In my approach, I use Correlated Active Clause Coverage (CACC) [8] to test predicates in the test paths. Its goal is to test individual clauses within logical expressions, yielding tests that are identical to tests developed to satisfy the masking form of MCDC [24]. This coverage criterion ensures that each clause in the predicate determines, in at least one test case, the truth value of the predicate such that it is true and false. CACC ensures we exercise each clause in a predicate for both true and false values, without testing all possible combinations of clause truth values and without masking effects for faulty clauses. To illustrate this point, an example is presented in Figure 4.28. Consider a component C that has two states S1 and S2 as shown in Figure 4.28. The transition from state S1 to state S2 can occur if the guard predicate $p = (a \wedge (b \vee c))$ is true.

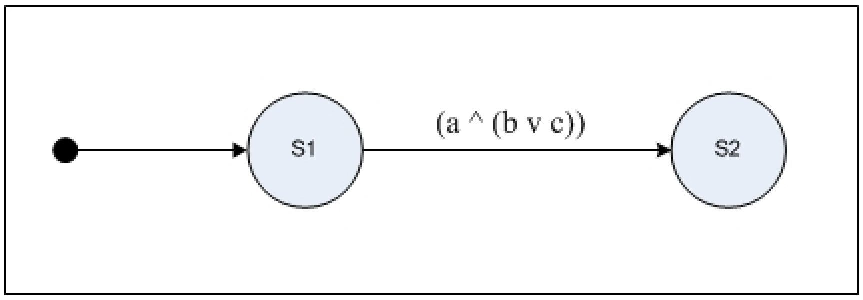


Figure 4.28: State diagram for component C

The guard predicate consists of three clauses:

$a = (currentTime > startTime)$

$b = (hardwareStatusisOK)$

$c = (systemModeisNORMAL)$

For clause a to determine the value of p, the expression $(b \vee c)$ must be true, which can be done in one of three ways: b true and c false, b false and c true, or both b and c true. So, it would be possible to satisfy Correlated Active Clause Coverage with respect to clause a with the two test requirements: f(a = true; b = true; c = false); (a = false; b = false; c = true). There are other possible sets of test requirements with respect to a, as enumerated in the following partial truth table in Table 4.1. There are nine possible truth assignments that will satisfy CACC for a, by choosing one test requirement from rows 1, 2 and 3, and another from rows 5, 6 and 7.

Table 4.1: Partial truth table for a CACC example

	a	**b**	**c**	$(a \wedge (b \vee c))$
1	True	True	True	True
2	True	True	False	True
3	True	False	True	True
5	False	True	True	False
6	False	True	False	False
7	False	False	True	False

All-Interface-Event Coverage can be defined as: Each event within each component must be used at least once and each interface message must be paired with every every in the provider component.

This coverage criterion not only checks the transitions between states and the interface between components, but also considers impacts from the events triggered by component interactions. A test path generation algorithm for the All-Interface-Event Coverage is given in Figure 4.29.

All-Interface-Event Coverage(T): TPSSeq

Input T: *CREATEG Test Model*

Output TPSSeq: *A sequence of test paths*

Declare:

TPSSeq: A sequence of test paths
MESSeq: A sequence of message edges in T
medg: An interaction message edge in T in the form of "Msgn" + "→"
+ " the provider interface Pi" with a time stamp in the form of "Tn"
or in the form of " the user interface Uij" + "→" + "Msgn"
with a time stamp in the form of "Tn"
Imedg: An input message edge in T in the form of "Pi" (incoming Msgn from
the provider interface Pij) + "→" + "entryState in a Cn"
with a time stamp in the form of "tpm_ij"+ "→" + "n"
Omedg: An output message edge in T in the form of "destinationState in a Cn"
+ "Uij" (outgoing Msgn to the user interface Uij)
with a time stamp in the form of "tum_n"+ "→" + "ij"
tedg: A transition edge in T, in the form of "sourceState" + "→" + "targetState"
with a time stamp in the form of "tst_m"+ "→" + "tst_n"
TMESeqn: A sequence of test sub paths in a Cn
from an Imedge (corresponding to an incoming interaction message Msgn)
to an Omedge (corresponding to an outgoing interaction message Msg(n+1))
tmepath: A complete test sub path in TMESeqn
ispcTMEpath: A chosen test sub path in TMESeqn to form a message sequence path

Steps:

```
1:   begin
2:   TPSSeq ← {}
3:   MESSeq ← T.edge.message
4:   for each medg ∈ MESSeq do
5:       for each provider component subnet where the medg comes to
6:           TMESeqn = createPath(Msgn, Msg(n+1))
7:           for each reachable entry state
8:               if there exists a path from the entryState to a destinationState to Msg(n+1)
9:                   addPath(Imedg, TMESeqn)
10:                  addPath(tedg, TMESeqn)
11:              while (a tedg exists on the path)
12:                  addPath(tedg, TMESeqn)
13:              addPath(Omedg, TMESeqn)
14:    for each Omedge ∈ TMESeqn do
15:        while (Omedgei != Omedge j)
16:        ispcTMEpath = tmepath
```

(Continued)

```
17:     for each $ispcTMEpath \in TMESeqn$ do
18:         $TPSSeq$ = createPath($Msgn, Msgm$)
19:         addPath($ispcTMEpath, TPSSeq$)
20:         while (a TMESeqns exists on the path)
21:             for each $ispcTMEpath \in TMESeq(n+1)$ do
22:                 addPath($ispcTMEpath, TPSSeq$)
```

Figure 4.29: An algorithm to generate test paths for the All-Interface-Event coverage criterion

This algorithm identifies optimal path subsets such that these subsets reduce the redundant paths in All-Path Coverage and keeps the necessary paths for detecting faults.

Explanation of the algorithm

The algorithm takes a CREATEG test model as an input and returns a set of test paths as an output.

Line 4 identifies all the message edges in a CREATEG test model.

Lines 5, 6, and 13 identify all the sequence of test paths between an input message edge and an output message edge.

Lines 7 through 12 generate a test path including all the interface and transition edges in a sequence.

Lines 14 through 22 generate a complete test path.

4.5 Executing the Test Cases

The algorithms in Sections 4.4.3 and 4.4.2 generate a set of test paths. Each test path is parsed to identify the objects and their initial states. The sequence numbers in a test path determine the sequence of sending the messages.

The execution of each test path requires test data, which is generated manually. Once an operation call message has been triggered, the rest of the message chain is executed automatically. The states of all objects involved in a message sequence must be set before the execution of a test case begins.

The test data includes a timing property to express when the data is input to the system.

As described in 1.1, observability requires appropriate instrumentation, which is a challenging problem for embedded systems where multiple tasks can have nondeterministic output. I designed the instrumentation to record various aspects of execution behavior so that the observed behavior can be compared with expected results. The instrumentation includes task id, operation, time instant, and execution state. For every interface between components, the interacting message is logged with a timing property. For every state transition within a component, the transition is logged with a timing property. All the forms of time instants are recorded in the instrumentation. With this capability, the timing relationship between each component can be evaluated as defined in time constraints TC.

Chapter 5: Experimental Tool

An experimental tool named AM2TM (Architecture Model to Test Model) was developed to demonstrate the effectiveness of the software architecture model-based testing technique. This chapter describes the design and implementation of AM2TM. AM2TM automatically constructs a CREATEG model from an XML Metadata Interchange (XMI) representation of a UML sequence diagram and associated state chart diagrams. It uses the CREATEG to generate test paths according to the specified coverage criterion. Test data are manually generated to execute the test paths, expected results are defined to compare with the execution results, and logs are created to manually check the results. The tool is composed of four major modules, CREATEG Constructor, Test Path Generator, Test Executor, and Results Evaluator, as highlighted in Figure 5.1.

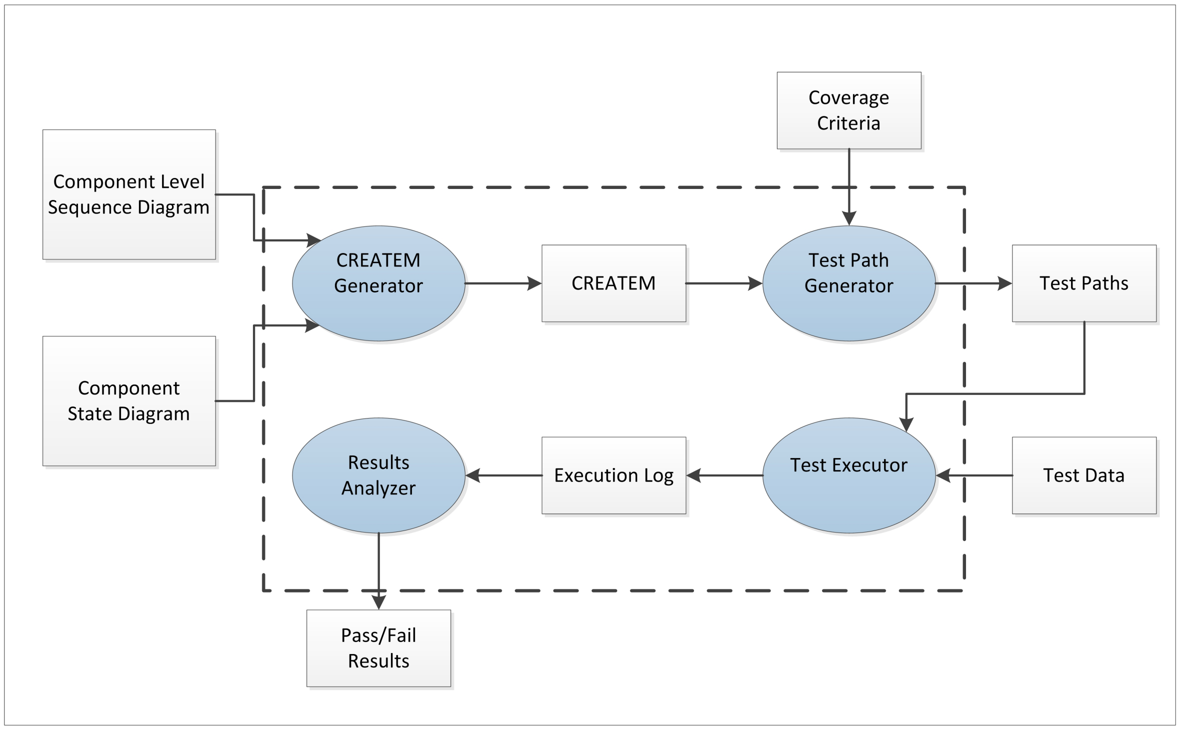


Figure 5.1: Structure of AM2TM experimental tool

This section describes the function of each of the four modules.

5.1 The CREATEG Constructor

The CREATEG constructor is responsible for constructing the test model CREATEG from the UML sequence diagram and UML state diagrams. These UML models are provided to the system in XMI format [39] [77]. The XMI format file representing the models can be generated by a number of UML development tools such as IBM Rational Rhapsody [5]. Figure 5.2 shows the feature in IBM Rational Rhapsody to generate XMI format file forUML diagrams.

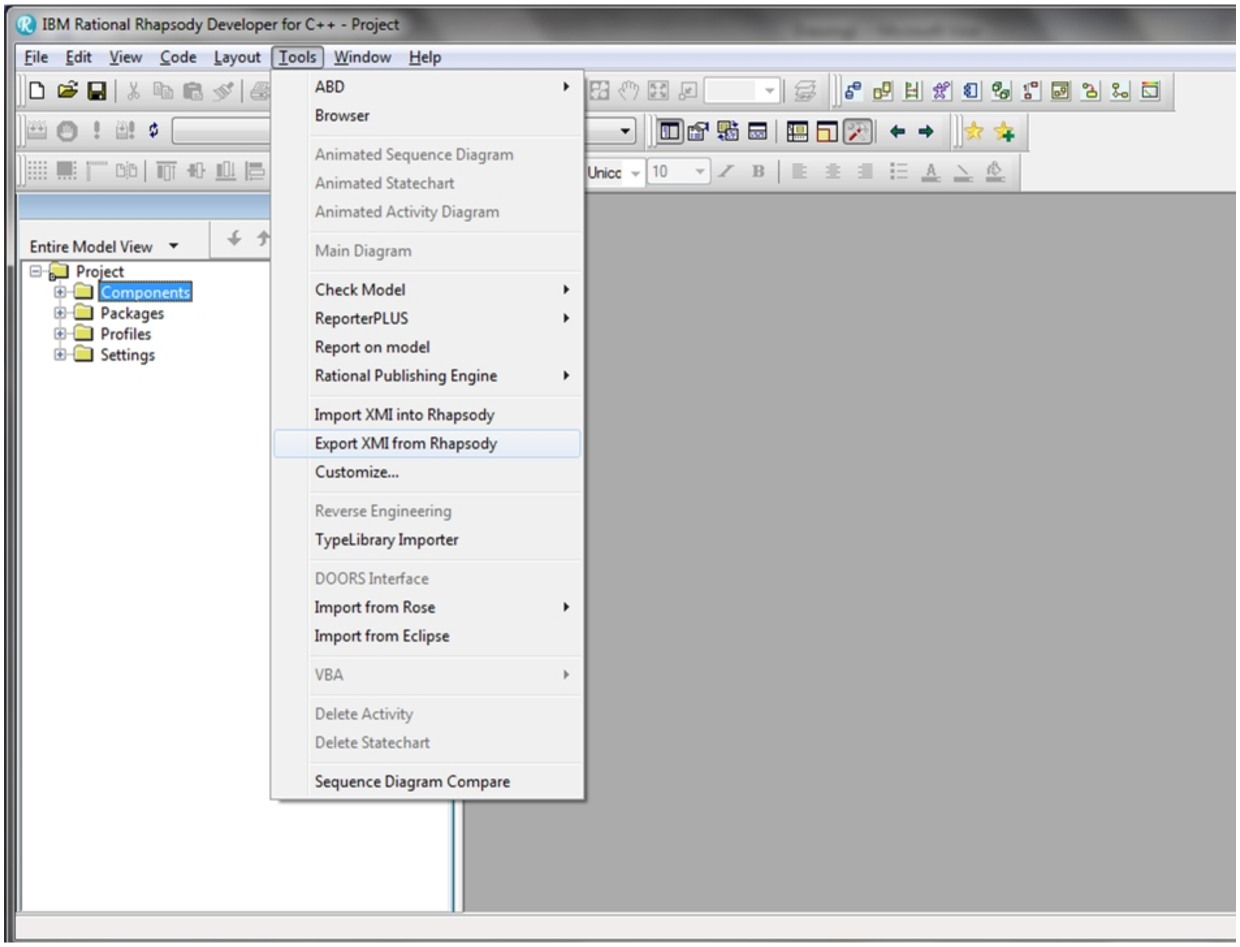
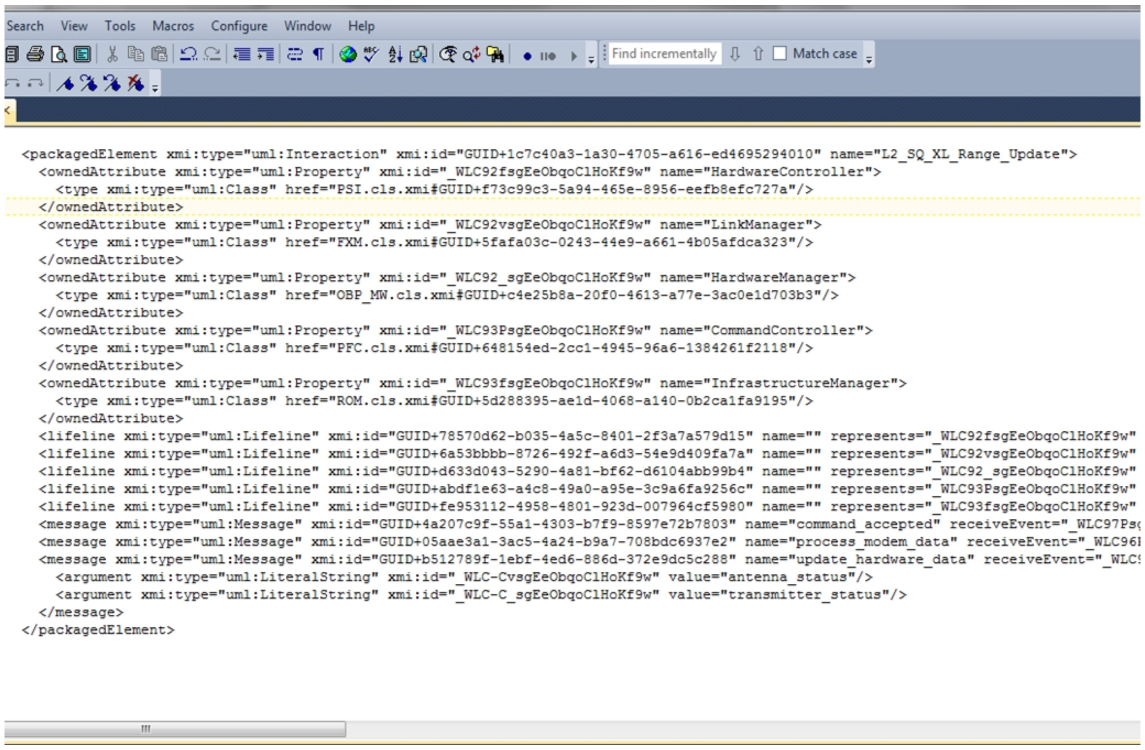


Figure 5.2: XMI generation in Rational Rhapsody

Figure 5.3 shows an example of a generated XMI format file from a UML sequence diagram.

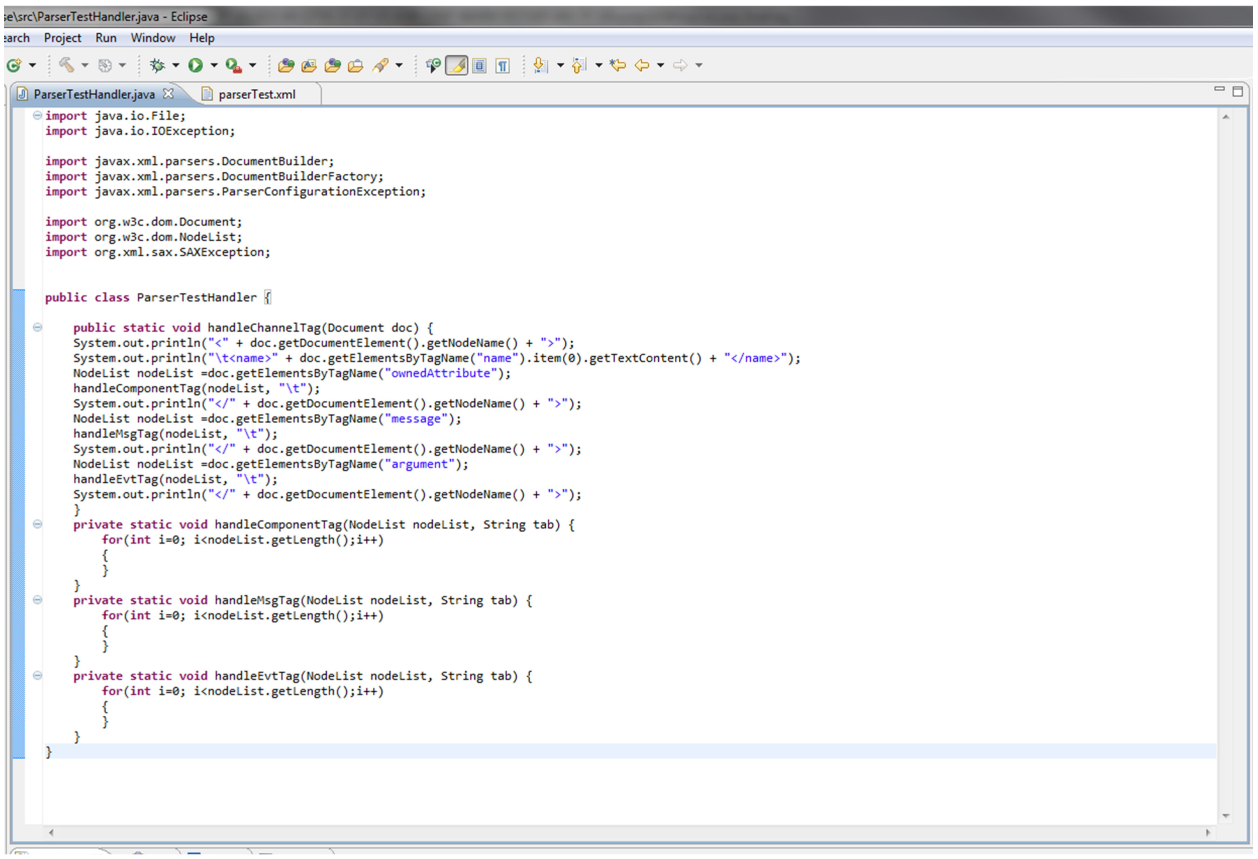


```
<packagedElement xmi:type="uml:Interaction" xmi:id="GUID+1c7c40a3-1a30-4705-a616-ed4695294010" name="L2_SQ_XL_Range_Update">
  <ownedAttribute xmi:type="uml:Property" xmi:id="_WLC92fsgEeObqoClHoKf9w" name="HardwareController">
    <type xmi:type="uml:Class" href="PSI.cls.xmi#GUID+f73c99c3-5a94-465e-8956-eefb8efc727a"/>
  </ownedAttribute>
  <ownedAttribute xmi:type="uml:Property" xmi:id="_WLC92vsgEeObqoClHoKf9w" name="LinkManager">
    <type xmi:type="uml:Class" href="FXM.cls.xmi#GUID+5fafa03c-0243-44e9-a661-4b05afdca323"/>
  </ownedAttribute>
  <ownedAttribute xmi:type="uml:Property" xmi:id="_WLC92_sgEeObqoClHoKf9w" name="HardwareManager">
    <type xmi:type="uml:Class" href="OBP_MW.cls.xmi#GUID+c4e25b8a-20f0-4613-a77e-3ac0e1d703b3"/>
  </ownedAttribute>
  <ownedAttribute xmi:type="uml:Property" xmi:id="_WLC93PsgEeObqoClHoKf9w" name="CommandController">
    <type xmi:type="uml:Class" href="PFC.cls.xmi#GUID+648154ed-2cc1-4945-96a6-1384261f2118"/>
  </ownedAttribute>
  <ownedAttribute xmi:type="uml:Property" xmi:id="_WLC93fsgEeObqoClHoKf9w" name="InfrastructureManager">
    <type xmi:type="uml:Class" href="ROM.cls.xmi#GUID+5d288395-ae1d-4068-a140-0b2ca1fa9195"/>
  </ownedAttribute>
  <lifeline xmi:type="uml:Lifeline" xmi:id="GUID+78570d62-b035-4a5c-8401-2f3a7a579d15" name="" represents="_WLC92fsgEeObqoClHoKf9w"
  <lifeline xmi:type="uml:Lifeline" xmi:id="GUID+6a53bbbb-8726-492f-a6d3-54e9d409fa7a" name="" represents="_WLC92vsgEeObqoClHoKf9w"
  <lifeline xmi:type="uml:Lifeline" xmi:id="GUID+d633d043-5290-4a81-bf62-d6104abb99b4" name="" represents="_WLC92_sgEeObqoClHoKf9w"
  <lifeline xmi:type="uml:Lifeline" xmi:id="GUID+abdf1e63-a4c8-49a0-a95e-3c9a6fa9256c" name="" represents="_WLC93PsgEeObqoClHoKf9w"
  <lifeline xmi:type="uml:Lifeline" xmi:id="GUID+fe953112-4958-4801-923d-007964cf5980" name="" represents="_WLC93fsgEeObqoClHoKf9w"
  <message xmi:type="uml:Message" xmi:id="GUID+4a207c9f-55a1-4303-b7f9-8597e72b7803" name="command_accepted" receiveEvent="_WLC97Ps
  <message xmi:type="uml:Message" xmi:id="GUID+05aae3a1-3ac5-4a24-b9a7-708bdc6937e2" name="process_modem_data" receiveEvent="_WLC96
  <message xmi:type="uml:Message" xmi:id="GUID+b512789f-1ebf-4ed6-886d-372e9dc5c288" name="update_hardware_data" receiveEvent="_WLC
    <argument xmi:type="uml:LiteralString" xmi:id="_WLC-CvsgEeObqoClHoKf9w" value="antenna_status"/>
    <argument xmi:type="uml:LiteralString" xmi:id="_WLC-C_sgEeObqoClHoKf9w" value="transmitter_status"/>
  </message>
</packagedElement>
```

Figure 5.3: XMI file generated from a Rational Rhapsody UML sequence diagram

An XMI parser written in Java extracts UML collaboration diagrams and state diagrams using the XMI file. It also generates the CREATEG from these models using the algorithms presented in Chapter 4. The CREATEG constructor contains four classes and approxiately 500 lines of code.

Figure 5.4 shows a Java XML parser via DOM interface.



se\src\ParserTestHandler.java - Eclipse
earch Project Run Window Help
ParserTestHandler.java
parserTest.xml

```
import java.io.File;
import java.io.IOException;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.xml.sax.SAXException;


public class ParserTestHandler {

    public static void handleChannelTag(Document doc) {
    System.out.println("<" + doc.getDocumentElement().getNodeName() + ">");
    System.out.println("\t<name>" + doc.getElementsByTagName("name").item(0).getTextContent() + "</name>");
    NodeList nodeList =doc.getElementsByTagName("ownedAttribute");
    handleComponentTag(nodeList, "\t");
    System.out.println("</" + doc.getDocumentElement().getNodeName() + ">");
    NodeList nodeList =doc.getElementsByTagName("message");
    handleMsgTag(nodeList, "\t");
    System.out.println("</" + doc.getDocumentElement().getNodeName() + ">");
    NodeList nodeList =doc.getElementsByTagName("argument");
    handleEvtTag(nodeList, "\t");
    System.out.println("</" + doc.getDocumentElement().getNodeName() + ">");
    }
    private static void handleComponentTag(NodeList nodeList, String tab) {
        for(int i=0; i<nodeList.getLength();i++)
        {
        }
    }
    private static void handleMsgTag(NodeList nodeList, String tab) {
        for(int i=0; i<nodeList.getLength();i++)
        {
        }
    }
    private static void handleEvtTag(NodeList nodeList, String tab) {
        for(int i=0; i<nodeList.getLength();i++)
        {
        }
    }
}
```

Figure 5.4: A Java XML parser via DOM interface

The procedure BuildCIIG() in the algorithm takes a sequence diagram SQ as input. The procedure BuildCSIBG() and BuildCSEDIBG take a sequence diagram SQ and a set of state diagrams as input for all the components participating in an operation. The output is a CREATEG test model for these UML artifacts. The procedure iteratively retrieves each message from the sequence diagram following the message sequence numbering, and builds the CREATEG.

As specified in the algorithm to generate the CIIG (shown in Figure 4.13), the algorithm to generate the CSIBG (shown in Figure 4.20) and the algorithm to generate the CSEDIBG (shown in Figure 4.22) from Chapter 4, the procedure iteratively builds the CREATEG by adding a message to it within each iteration. The algorithm stores all possible states of a component in which a message can be received. The current state of this component set must be a subset of the set of all the states capable of receiving the message. A violation of this condition implies a design inconsistency because the previous operation(s) left the component in an inconsistent state for the subsequent messages. Thus, the process of building CREATEG from UML artifacts can also be used to detect inconsistencies in the design of the system.

Once these consistency checks have been performed, all the nodes in the source component are connected to each node for the states in the target component and message edges are added on each edge. The CREATEG Constructor algorithm to generate the CIIG in Figure 4.13, the algorithm to generate the CSIBG in Figure 4.20, and the algorithm to generate the CSEDIBG in Figure 4.22 are used to construct CTEATEM models in experimental software presented in chapter 6.

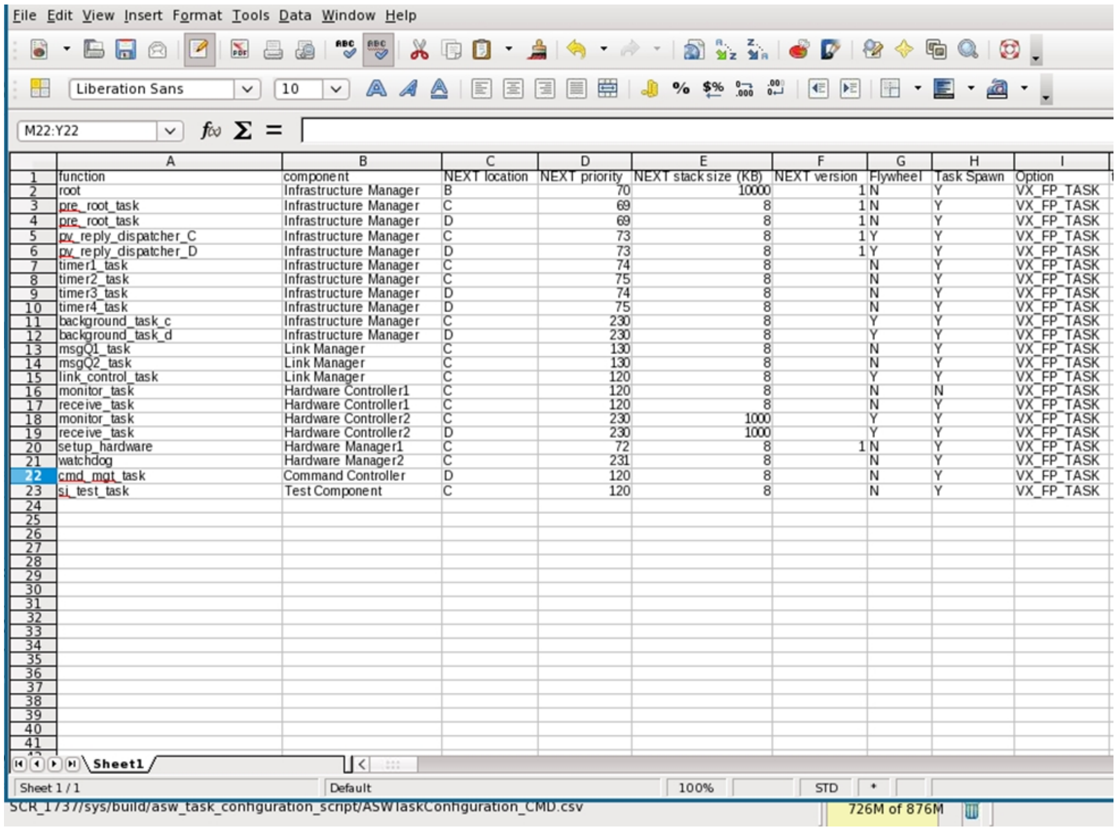
5.2 Test Path Generator

Test Path Generator tool is implemented in Java. It generates test paths from the test model developed by the CREATEG constructor. The inputs to this module are the CREATEG model and one of the four path coverage criteria defined in Section 4.4: All-Interface Coverage, All-Interface-Transition Coverage, All-Interface-Event Coverage, or All-Path Coverage. The output of the Test Path Generator is a set of test paths that meet the specified coverage criteria. Test Path Generator contains five classes and approxiately 800 lines of code.

5.3 Test Executor

Test inputs need be generated to execute test paths. Test input data is generated manually for the test paths. The test inputs include the values of parameters in initial message calls, event trigger calls and values of variables to set states of components involved in sequence diagrams. The user manually generates test values for each test path. Test cases are implemented with test input data. One test case covers one test path.

Test Executor is placed in a separate real-time task and the task is spawned by the operating system as all the other application user tasks. Figure 5.5 shows an interface to set up and spawn a test task in the operating system.

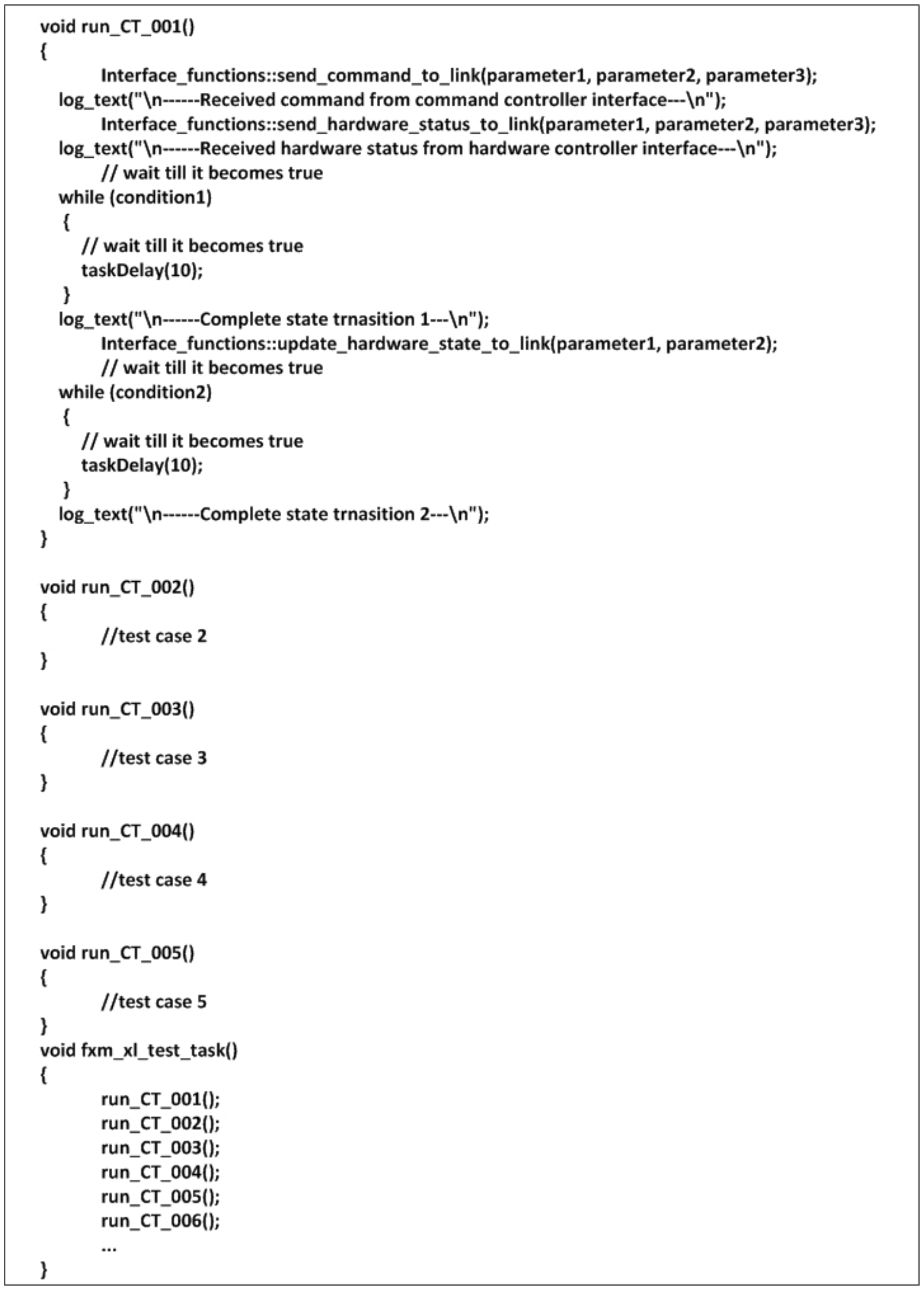


function	component	NEXT location	NEXT priority	NEXT stack size (KB)	NEXT version	Flywheel	Task Spawn	Option
root	Infrastructure Manager	B	70	10000	1	N	Y	VX_FP_TASK
pre_root_task	Infrastructure Manager	C	69	8	1	N	Y	VX_FP_TASK
pre_root_task	Infrastructure Manager	D	69	8	1	N	Y	VX_FP_TASK
pv_reply_dispatcher_C	Infrastructure Manager	C	73	8	1	Y	Y	VX_FP_TASK
pv_reply_dispatcher_D	Infrastructure Manager	D	73	8	1	Y	Y	VX_FP_TASK
timer1_task	Infrastructure Manager	C	74	8		N	Y	VX_FP_TASK
timer2_task	Infrastructure Manager	C	75	8		N	Y	VX_FP_TASK
timer3_task	Infrastructure Manager	D	74	8		N	Y	VX_FP_TASK
timer4_task	Infrastructure Manager	D	75	8		N	Y	VX_FP_TASK
background_task_c	Infrastructure Manager	C	230	8		Y	Y	VX_FP_TASK
background_task_d	Infrastructure Manager	D	230	8		Y	Y	VX_FP_TASK
msgQ1_task	Link Manager	C	130	8		N	Y	VX_FP_TASK
msgQ2_task	Link Manager	C	130	8		N	Y	VX_FP_TASK
link_control_task	Link Manager	C	120	8		Y	Y	VX_FP_TASK
monitor_task	Hardware Controller1	C	120	8		N	N	VX_FP_TASK
receive_task	Hardware Controller1	C	120	8		N	Y	VX_FP_TASK
monitor_task	Hardware Controller2	C	230	1000		Y	Y	VX_FP_TASK
receive_task	Hardware Controller2	D	230	1000		Y	Y	VX_FP_TASK
setup_hardware	Hardware Manager1	C	72	8	1	N	Y	VX_FP_TASK
watchdog	Hardware Manager2	C	231	8		N	Y	VX_FP_TASK
cmd_mgt_task	Command Controller	D	120	8		N	Y	VX_FP_TASK
si_test_task	Test Component	C	120	8		N	Y	VX_FP_TASK

SCR_1737/sys/build/asw_task_configuration_script/ASWTaskConfiguration_CMD.csv

Figure 5.5: Spawning a test task

Test Executor constructs concrete test cases by filling in the test data to the input calls in test paths. Each test case is then executed when the test task is activated. The execution results are logged in a file. The execution log for a test path consists of all the messages exchanged between components and all the state transitions within the component for each message sequence in a test path. The component states are determined from the instrumentation provided in the source code for getting component states. Figure 5.5 shows how to start a test task. Figure 5.6 shows what the test code looks like. The entry point in the test code is the test task name spawned in Figure 5.5.



```
void run_CT_001()
{
        Interface_functions::send_command_to_link(parameter1, parameter2, parameter3);
   log_text("\n------Received command from command controller interface---\n");
        Interface_functions::send_hardware_status_to_link(parameter1, parameter2, parameter3);
   log_text("\n------Received hardware status from hardware controller interface---\n");
        // wait till it becomes true
   while (condition1)
    {
      // wait till it becomes true
      taskDelay(10);
    }
   log_text("\n------Complete state trnasition 1---\n");
        Interface_functions::update_hardware_state_to_link(parameter1, parameter2);
        // wait till it becomes true
   while (condition2)
    {
      // wait till it becomes true
      taskDelay(10);
    }
   log_text("\n------Complete state trnasition 2---\n");
}

void run_CT_002()
{
        //test case 2
}

void run_CT_003()
{
        //test case 3
}

void run_CT_004()
{
        //test case 4
}

void run_CT_005()
{
        //test case 5
}
void fxm_xl_test_task()
{
        run_CT_001();
        run_CT_002();
        run_CT_003();
        run_CT_004();
        run_CT_005();
        run_CT_006();
        ...
}
```

Figure 5.6: Test code example

5.4 Results Evaluator

The Results Evaluator is responsible for comparing the results of a test run with the expected results. The expected results are derived from the generated test paths. An expected result for a test path consists of the component state transitions and time constraints for each message in the test path. Results Evaluator compares the component states in the execution log with the expected result. A test path is considered to pass if all the interface interactions occu rred in the correct sequence, all the component states transitions occured in the right order and all the time constraints occurred within the correct range per the incoming messages, otherwise it is considered to fail. The Pass/Fail results are logged in a file. For failed test cases, Results Evaluator also logs the test path number, the failed message whose expected state does not match the resultant state, and component states. Results Evaluator generates the overall results as a report to the users. The report contains the following information:

Step number

  Evaluation:

  Status: PASS/FAIL

  Reason:

  Evidence:

    Logfile::

    Logtext:

Chapter 6: Validation on a Large Scale Industrial System

This research was validated by applying the novel testing method to an industrial software system. The goal of the validation is to determine whether the testing method can detect faults effectively. To facilitate this experiment, the prototype tool was developed to evaluate the proposed test criteria. This chapter presents an industrial post-hoc observational field study [85] and discusses the results. Validation for the research was carried out by developing and executing tests on faulty versions of an industrial software system. An actual architecture design of a software system is described in several sequence diagrams and state diagrams. Test cases were generated and used to find faults seeded in the software system.

6.1 Experimental Design

The experimental design is described in terms of the subject program, the criteria (from Chapter 4), the seeded faults, the measurement procedure, and the experimental procedure.

Subject Program: An industrial application system was used as the subject program. The program was written in C and C++. It has six major components at the architecture level. The major functionality is satellite communication link initiation, control, and management. It runs on the VXWorks real-time operating system. The system contains approximately 75,000 lines of code. We tested a part of the system that contains about 10,000 lines of code.

Test Adequacy Criteria: Three methods for designing tests were compared:

1. Traditional manual specification testing based on experience and requirements specification

2. The all-interface-transition coverage testing technique defined in Chapter 4

3. The all-interface-event coverage testing technique defined in Chapter 4

Test Data: A set of test data was generated for each testing method applied on the subject program. The generation of each test data set is specific to each test method applied.

Fault Set: Faults were hand seeded according to mutation rules. The effectiveness of the software architecture-based testing criteria were validated by detecting faults seeded in the program's source code. The selection of seeded faults was based on the characteristics of the code. More details are in subsection 6.1.5.

Measurement: The fault detecting effectiveness of a given test adequacy criterion c for a given architecture a with respect to a specific fault set f is defined as the ratio of the number of faults detected to the number of faults seeded. This measurement was made for each pair of subject architecture program and test adequacy criterion. In this experiment, we have fifteen sets of test cases, five for each of the three testing techniques.

Experimental Procedure: The conduct of the experiment consisted of several steps. Let A be an architecture, C be the set of test adequacy criteria, and T be the set of test data generated for each test adequacy criterion.

For each $(a \in A)$ and $(c \in C)$:

Step 1. Generate c-adequate test data set $T(a, c)$.

Step 2. Define fault set $F(a)$ for a.

Step 3. For each $f \in F(a)$ define the fault seeded architecture $A(f)$ by seeding with faults, yielding a fault-seeded architecture $a(f)$ where each $a(f) \in A(f)$.

Step 4. For each $t \in T(a, c)$, if it detects at least one fault, increase the number of faults $Num(a, c)$ detected by test data set $T(a, c)$.

Step 5. Determine the fault detection rate $R(a, c)$ for test adequacy criterion c with respect to architecture a as:

$R(a, c) = Num(a, c)/|F(a, c)|$

Step 6. Determine the fault detection effectiveness $E(a, c)$, for test adequacy criterion c with respect to an achitecture a as:

$E(a, c) = Num(a, c)/|T(a, c)|$

6.1.1 Experimental Procedure

The experiment procedure is summarized in Figure 6.1. First, we chose errors (detailed in Section 6.1.5) that are applicable to this subject program (detailed in Section 6.1.2). Then we implemented faults and seeded them into the subject program (detailed in Section 6.1.5). Test sets are generated using the novel model-based testing techniques defined in this dissertation. Each test case was executed against all faulty versions of the subject program. After each execution, failures (if any) were checked and corresponding faults were identified by hand. This process was repeated on each test case until no more failures occurred. The number of faults detected was recorded and used in the analysis.

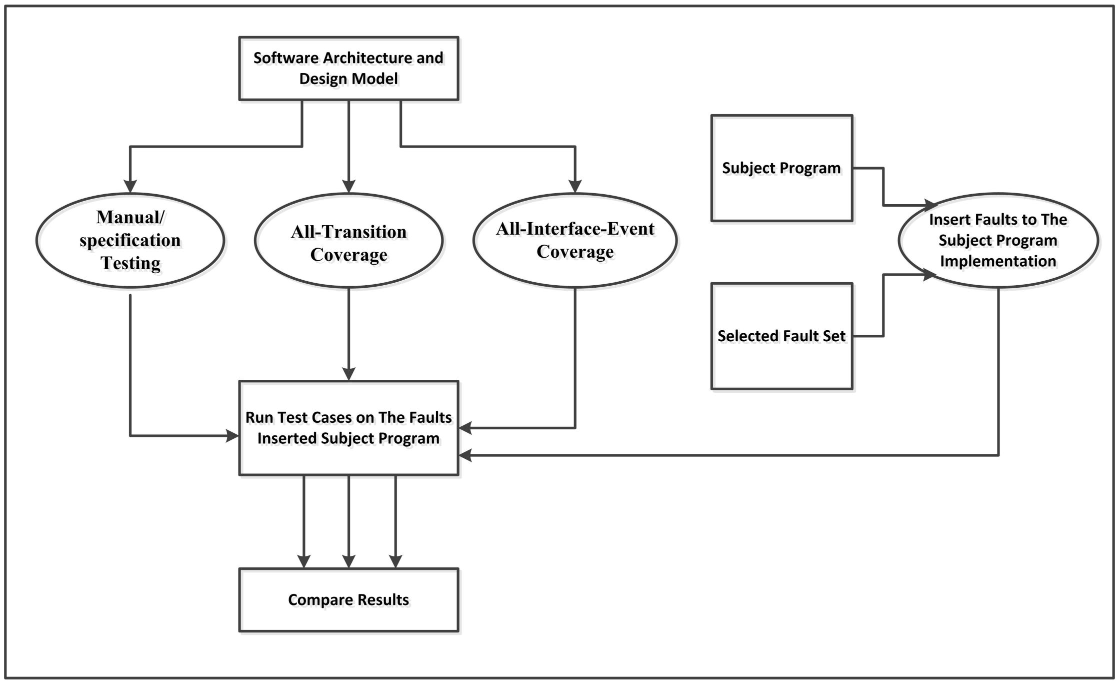

Software Architecture and Design Model
Manual/ specification Testing
All-Transition Coverage
All-Interface-Event Coverage
Subject Program
Selected Fault Set
Insert Faults to The Subject Program Implementation
Run Test Cases on The Faults Inserted Subject Program
Compare Results

Figure 6.1: Experiment procedure

6.1.2 The Subject Program

The subject program is an industrial software system build by Lockheed Martin (LM). The program was written in C and C++. The system contains more than 100 classes and approximately 75,000 lines of code. Because of its proprietary nature, this dessertation only shows a high level abstract of the program structure, but not the code.

This software system receives real-time command and data from a ground user interactive system, processes command and data, and performs a series of actions and controls multiple types of hardware. These hardware systems also send hardware metrics and status to the software. Users can request telemetry data, causing the system to send data that meets the customer's criteria to destination points (external user systems). An overview of the system is shown in Figure 6.2.

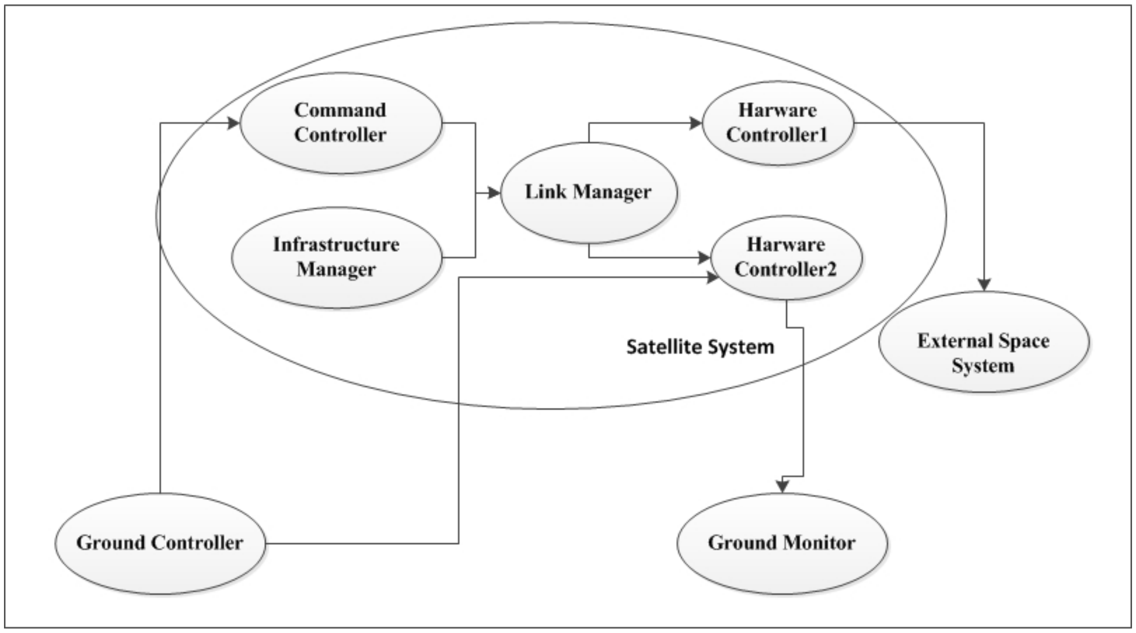


Figure 6.2: The subject program

Command Controller receives the data from the ground system and passes it to Communication Link Manager where the command gets processed and takes effect right away or at a later time based on information in the command. Communication Link Manager controls the link hardware (antenna, modem and transmitter/receiver) through Hardware Controller 1 and Hardware Controller 2. It monitors performance metrics and hardware status during link activation, maintenance, and deactivation. Communication Link Manager maintains overall system state. Infrastructure Manager provides all the interfaces to the real-time operating system, including semaphore control, message queue, and timer management.

This subject program has three benefits:

1. This program uses all the architecture relations described in the testing technique

2. The subject program is a typical component-based real-time embedded system that has many communications and connections between components

3. The subject program runs on a real-time hardware environment

6.1.3 Test Adequacy Criteria

We use three types of test adequacy criteria in this experiment application.

1. In the manual specification method, test requirements were generated based on the specification of the subject program. A brief system specification of the subject system was available, where high level data flow and control flow were presented. Test requirements and test cases were generated based on the data flow, control flow, as well as the text description

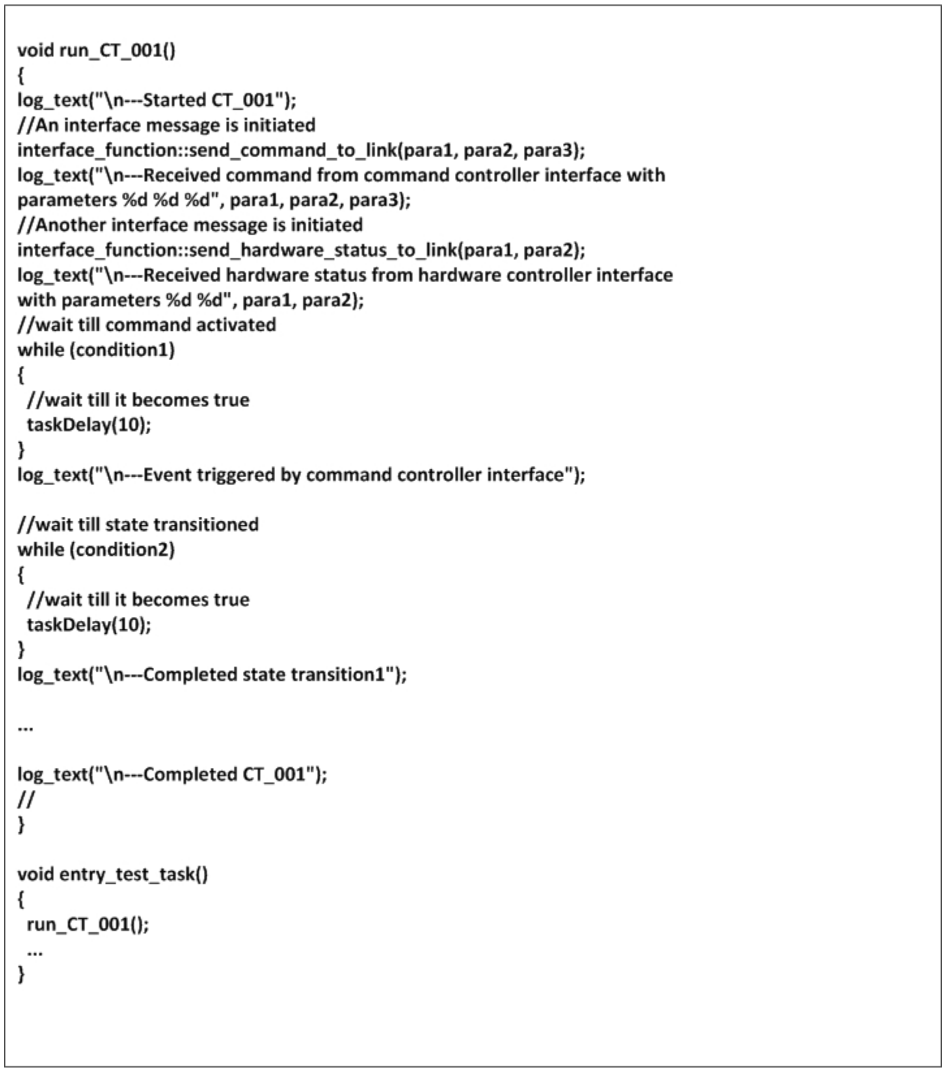
2. The all-interface-event coverage testing technique defined in Chapter 4

3. The all-transition coverage testing technique defined in Chapter 4

6.1.4 Test Data

Test data were manualy generated by the author to fulfill test requirements in CREATEG model. The process was specific to each test method. For the manual specification method, test data were generated to meet each functional requirement. For the all-transition coverage criterion and the all-transition-event coverag, test data were generated to cover each test path. Test data were taken as inputs to a proprietary test automation tool that was developed by the company using the Perl programming language. The test tool reads the input test procedure files, launches the software, waits for the test to complete, retrieves output log files, and analyzes the results to determine whether the test was successful.

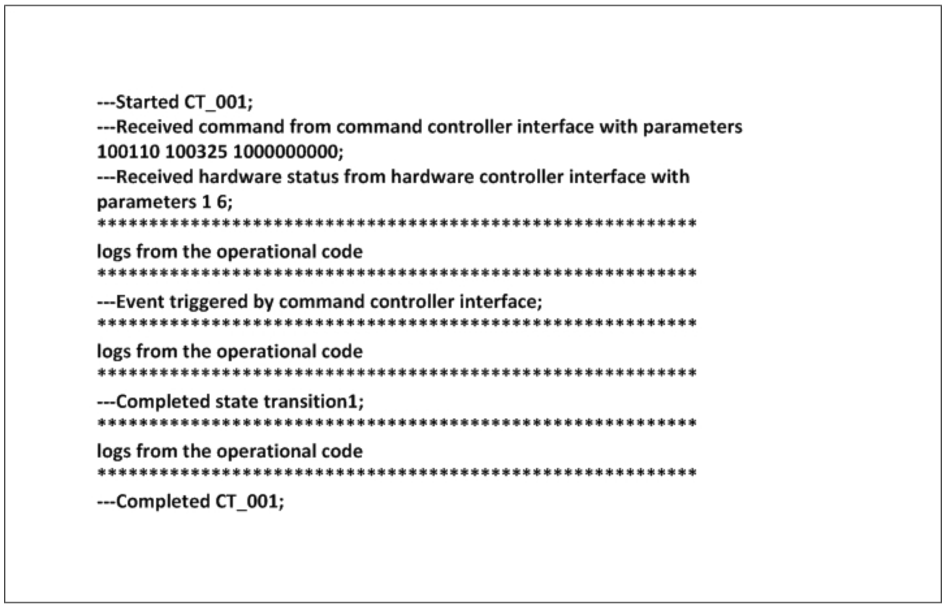
Figure 6.3 shows a test case example. Test data are input from interface functions as parameters. A function called log_text() is used to output program behavior to a log file.



```
void run_CT_001()
{
log_text("\n---Started CT_001");
//An interface message is initiated
interface_function::send_command_to_link(para1, para2, para3);
log_text("\n---Received command from command controller interface with
parameters %d %d %d", para1, para2, para3);
//Another interface message is initiated
interface_function::send_hardware_status_to_link(para1, para2);
log_text("\n---Received hardware status from hardware controller interface
with parameters %d %d", para1, para2);
//wait till command activated
while (condition1)
{
  //wait till it becomes true
  taskDelay(10);
}
log_text("\n---Event triggered by command controller interface");

//wait till state transitioned
while (condition2)
{
  //wait till it becomes true
  taskDelay(10);
}
log_text("\n---Completed state transition1");

...

log_text("\n---Completed CT_001");
//
}

void entry_test_task()
{
  run_CT_001();
  ...
}
```

Figure 6.3: Test case example

Figure 6.4 shows a log file example. The log file contains both logs from test cases and logs from operational software.



```
---Started CT_001;
---Received command from command controller interface with parameters
100110 100325 1000000000;
---Received hardware status from hardware controller interface with
parameters 1 6;
************************************************************
logs from the operational code
************************************************************
---Event triggered by command controller interface;
************************************************************
logs from the operational code
************************************************************
---Completed state transition1;
************************************************************
logs from the operational code
************************************************************
---Completed CT_001;
```

Figure 6.4: Test log example

6.1.5 Fault Sets

The research was validated by determining the effectiveness of the software architecture-based testing criteria at detecting faults that are associated with the connections of the state oriented components. Faults were seeded by hand using 21 mutation operators. A mutation operator defines syntactic changes to a program, creating variants of the original program that are called *mutants*. If a test causes a mutant to generate a different result from the original program, this mutant is *killed*. In this researach, mutation analysis was applied to architecture and design models. For each mutation operator, faults were selected based on specific artifacts in the model. For example, one of the mutation operators removes the condition of a conditional message in the code. We first check the model to see how many conditional messages were defined in the design model and implement faults for those conditional messages. The faults were implemented and seeded by hand by modifying the corresponding source code. Mutation testing was done by selecting a set of mutation operators and then applying them to the source program one at a time for each applicable piece of the source code.

For example, consider the following C++ code fragment:

```
if (a && b) {
    c = 1;
} else {
    c = 0;
}
```

The condition mutation operator would replace && with ||, producing the following mutant:

```
if (a || b) {
    c = 1;
} else {
    c = 0;
}
```

Three integration operators were defined by Delamaro et al. [33], five specification operators were defined by Souza et al. [32], two activity model operators were defined by Swain et al. [74], and seven mutation operators were defined by Nilsson et al. [65]. To introduce more variety in the type of seeded faults, I defined four additional operator categories. The selection of seeded faults was based on the characteristics of the software for the industrial software, and the ability of the operators to introduce interaction faults, i.e., faults that can only be revealed by messages sent from one component to the other.

The following list defines the 21 mutation operators used in the study. After each definition the source for the operator is given.

1. Initial State Exchanged (ISE): This operator changes the initial state of a component before it receives a message. The initial state is replaced by an invalid state in which the component should not receive a particular message. (Souza et al. [32])

2. Replace Return Statement (RetStaDel): This operator replaces each return statement in a method with each other return statement in the method, one at a time. (Delamaro et al. [33])
3. Condition Missing (CM): This operator removes the condition of a conditional message in the code. (Souza et al. [32])
4. Transition Deletion (TD): This operator removes a transition in the code. (Souza et al. [32])
5. Event Exchange (EE): This operator switches an event with another to trigger a transition to a different state in the code. (Souza et al. [32])
6. Event Missing (EM): This operator removes an event in the code. (Souza et al. [32])
7. Argument Switch/Parameters Exchange (AS) at calling point: This operator changes the order of the parameters (and type) passed in an interface call. (Delamaro et al. [33])
8. Interface Variables Exchange (IVE) at called module: This operator changes the parameter passed in an interface called. The valid value of the parameter is replaced with an invalid value. (Delamaro et al. [33])
9. Alter Condition Operator (ACO): This operator changes the condition in the code corresponding to a path condition in collaboration. (Swain et al. [74])
10. Guard Condition Violated (GCV): This operator negates the guard condition of a transition. (Swain et al. [74])
11. Missing Provider Function (MPF): This operator removes the functions that are called by a component. (Defined in this dissertation)
12. State Exchange (SE): This operator sets the target state of a component as the source state. (Defined in this dissertation)

13. Wrong User State (WUS): This operator sets the state of the calling component to an invalid state. (Defined in this dissertation)

14. Conflicting State (CS): This operator sets the states of two components in states that conflict with each other. (Defined in this dissertation)

15. $\triangle$+ execution time: This mutation operator changes the execution time of one components task from T to T+ $\triangle$. (Nilsson et al. [65])

16. $\triangle$- execution time: This mutation operator changes the execution time of one components task from T to T- $\triangle$. (Nilsson et al. [65])

17. - precedence constraint: This mutation operator removes a precedence constraint relation between two components. (Nilsson et al. [65])

18. + precedence constraint: This mutation operator adds a precedence constraint relation between two components. (Nilsson et al. [65])

19. $\triangle$- inter-arrival time: This mutation operator decreases the inter-arrival time between requests for a task execution by a constant time $\triangle$. (Nilsson et al. [65])

20. $\triangle$+ pattern offset: Changes clock constraint from C to C+ $\triangle$ in guards on a transition from one state to another. (Nilsson et al. [65])

21. $\triangle$- pattern offset: Changes clock constraint from C to C- $\triangle$ in guards on a transition from one state to another. (Nilsson et al. [65])

The faults seeded in each category were based on the design and code of the subject program. For instance, the sequence diagram has three path conditions, so we could seed three faults in each of the CM and ACO categories. Similarly, there are five guard conditions in a state chart diagram, so we could seed up to five faults in the GCV category.

Based on the above 21 operators, I created 60 faults for the subject system. A summary of the seeded faults for the study is given in Table 6.1.

Table 6.1: Number and types of faults seeded

Fault Number	**mutation operator**	**Faults in the Subject Programs**
1	Initial State Exchanged (ISE)	Change the initial state of link manager from inactive to disconnecting
2	Initial State Exchanged (ISE)	Change the initial state of hardware controller 2 from inoperative to operative
3	Replace Return Statement (RetStaDel)	Hardware controller 1 returns a statement that is not based on the result of the hardware module 1 status and the status impacts hardware controller 1s states
4	Replace Return Statement (RetStaDel)	Hardware controller 1 returns a statement that is not based on the result of the hardware module 1 status and the status doesnt impact hardware controller 1's states
5	Replace Return Statement (RetStaDel)	Hardware controller 2 returns a statement that is not based on the result of the hardware module 2's status
6	Replace Return Statement (RetStaDel)	Infrastructure manager returns a statement that is not reflecting the status of the operating system
7	Condition Missing (CM)	A condition generated from an external event triggered by hardware controller 1 request interface to link manager provider interface is deleted for a state transition in link manager

Continued on next page

Table 6.1 – continued from previous page

Fault Number	**mutation operator**	**Faults in the Subject Programs**
8	Condition Missing (CM)	A condition triggered by hardware controller 2 is missing
9	Condition Missing (CM)	A condition comparing with the current time stamp with an designated time stamp is missing
10	Transition Missing (TM)	Transition from state acquired to maintenance is missing in hardware controller 2
11	Transition Missing (TM)	One transition from state connecting to state connected is missing in Link Manager
12	Transition Missing (TM)	One transition from state connected to state unconnected is missing in Link Manager
13	Transition Missing (TM)	One transition from state ready to state notReady is missing in hardware manager
14	Event Exchanged (EE)	Two events in the link manager event handler table for the same state are switched
15	Event Exchanged (EE)	Two events in the link manager event handler table for different states are switched
16	Event Exchanged (EE)	Two events in the link manager event handler table for the same state are switched
17	Event Missing (EM)	One external event triggered by hardware manager request interface to link manager provider interface is missing

Continued on next page

Table 6.1 – continued from previous page

Fault Number	mutation operator	Faults in the Subject Programs
18	Event Missing (EM)	One event triggered by command controller is missing
19	Event Missing (EM)	One internal event is missing
20	Calling point: Parameters exchanged	Parameters in a request interface function between hardware controller 1 and link manager are not in the right order
21	Calling point: Parameters exchanged	Parameters in a request interface function between hardware controller 2 and link manager are not in the right order
22	Calling point: Parameters exchanged	Parameters in a request interface function between Infrastructure manager and link manager are not in the right order
23	Calling point: Parameters exchanged	Parameters in a request interface function between hardware manager and link manager are not in the right order
24	Called Module: Interface variables exchanged	The values of the parameters in a provider interface function between hardware controller 1 and link manager are opposite
25	Called Module: Interface variables exchanged	The parameters in a provider interface function between hardware controller 2 and link manager are opposite
26	Called Module: Interface variables exchanged	The parameters in a provider interface function between Infrastructure manager and link manager are opposite
Continued on next page		

Table 6.1 – continued from previous page

Fault Number	mutation operator	Faults in the Subject Programs
27	Called Module: Interface variables exchanged	The parameters in a provider interface function between hardware manager and link manager are opposite
28	Called Module: Interface variables exchanged	The parameters in a provider interface function between command controller and link manager are opposite
29	Called Module: Interface variables exchanged	The parameters in a provider interface function between hardware controller 1 and link manager are opposite
30	Alter Condition Operator (ACO)	A condition in a link activation sequence for two alternative path is switched
31	Alter Condition Operator (ACO)	A condition in a link deactivation sequence for two alternative path is switched
32	Guard Condition Violated (GCV)	A condition in a transition from state idle to connecting in link manager is invalid
33	Guard Condition Violated (GCV)	A condition in a transition from state connecting to disconnecting in link manager is invalid
34	Guard Condition Violated (GCV)	A condition in a transition from state connecting to connected in link manager is invalidd
35	Guard Condition Violated (GCV)	A condition in a transition from state connected to unconnected in link manager is invalid

Continued on next page

Table 6.1 – continued from previous page

Fault Number	mutation operator	Faults in the Subject Programs
36	Guard Condition Violated (GCV)	A condition in a transition from state disconnecting to idle in link manager is invalid
37	Missing Provider Function (MPF)	In a hardware controller 2 request interface function, the corresponding link manager provided interface function is missing
38	Missing Provider Function (MPF)	In a hardware controller 1 request interface function, the corresponding link manager provided interface function is missing
39	State Exchange (SE)	A target state in link manager is exchanged with its corresponding source state
40	State Exchange (SE)	A target state in hardware controller 2 is exchanged with its corresponding source state
41	Wrong User State (WUS)	A request interface from the link manager is activated in a wrong state
42	Wrong User State (WUS)	A request interface from the hardware controller 1 is activated in a wrong state
43	Wrong User State (WUS)	A request interface from the hardware controller 2 is activated in a wrong state
44	Conflicting State (CS)	The state in a request interface from link manager conflicts with the corresponding state of a provider interface from hardware controller 2

Continued on next page

Table 6.1 – continued from previous page

Fault Number	mutation operator	Faults in the Subject Programs
45	Conflicting State (CS)	The state in a request interface from link manager conflicts with the corresponding state of a provider interface from hardware manager
46	Conflicting State (CS)	The state in a request interface from link manager conflicts with the corresponding state of a provider interface from hardware controller 1
47	$\triangle$ + execution time	A timed delay $\triangle$ was added between sending two outgoing messages from the component link manager to the other two components hardware controller 1 and hardware controller 2.
48	$\triangle$ + execution time	A timed delay $\triangle$ was added in component hardware controller 1's intialization sequence.
49	$\triangle$ - execution time	One action was taken out of component hardware controller 2 so the execution time of the component task was reduced.
50	$\triangle$ - execution time	A timed delay $\triangle$ was reduced from component hardware controller 2's task.
51	- precedence constraint	A precedence constraint relation between component link manager and component hardware controller 1 is removed.

Continued on next page

Table 6.1 – continued from previous page

Fault Number	mutation operator	Faults in the Subject Programs
52	- precedence constraint	A precedence constraint relation between component link manager and component hardware controller 2 is removed.
53	+ precedence constraint	A precedence constraint relation between component link manager and component hardware controller 1 is added.
54	+ precedence constraint	A precedence constraint relation between component link manager and component hardware controller 2 is added.
55	△ - inter-arrival time	A timer based event triggered by the infrastructure manager has a delay in its interval.
56	△ - inter-arrival time	A timer based event triggered by the command comtroller has a delay in its interval.
57	△ + pattern offset	Clock constraint from component command controller was added a time delay. The constraint was applied to a guard in component link manager.
58	△ + pattern offset	Clock constraint from component hardware controller 2 was added a time delay. The constraint was applied to a guard in component link manager.
Continued on next page		

Table 6.1 – continued from previous page

Fault Number	mutation operator	Faults in the Subject Programs
59	△ - pattern offset	Clock constraint from component command controller was reduced by certain time. The constraint was applied to a guard in component link manager.
60	△ - pattern offset	Clock constraint from component hardware controller 2 was reduced by certain time. The constraint was applied to a guard in component link manager.

The faults shown in Table 6.1 were manually inserted into the subject program by the author.

6.2 Experimental Results

Following the experimental procedure described in Figure 6.1, first I generated the CREATEG model of the subject program. The software has six components. Each component contains multiple provider services, user services, message edges, states, transitions, event captures, event handlers and events. Each message edge or transition is marked with the cooresponding time stamp. The duration and execution cost of inter-component and intra-component interactions and state transitions can be calculated. The overall CSIEDBG is shown in Figure 6.5.

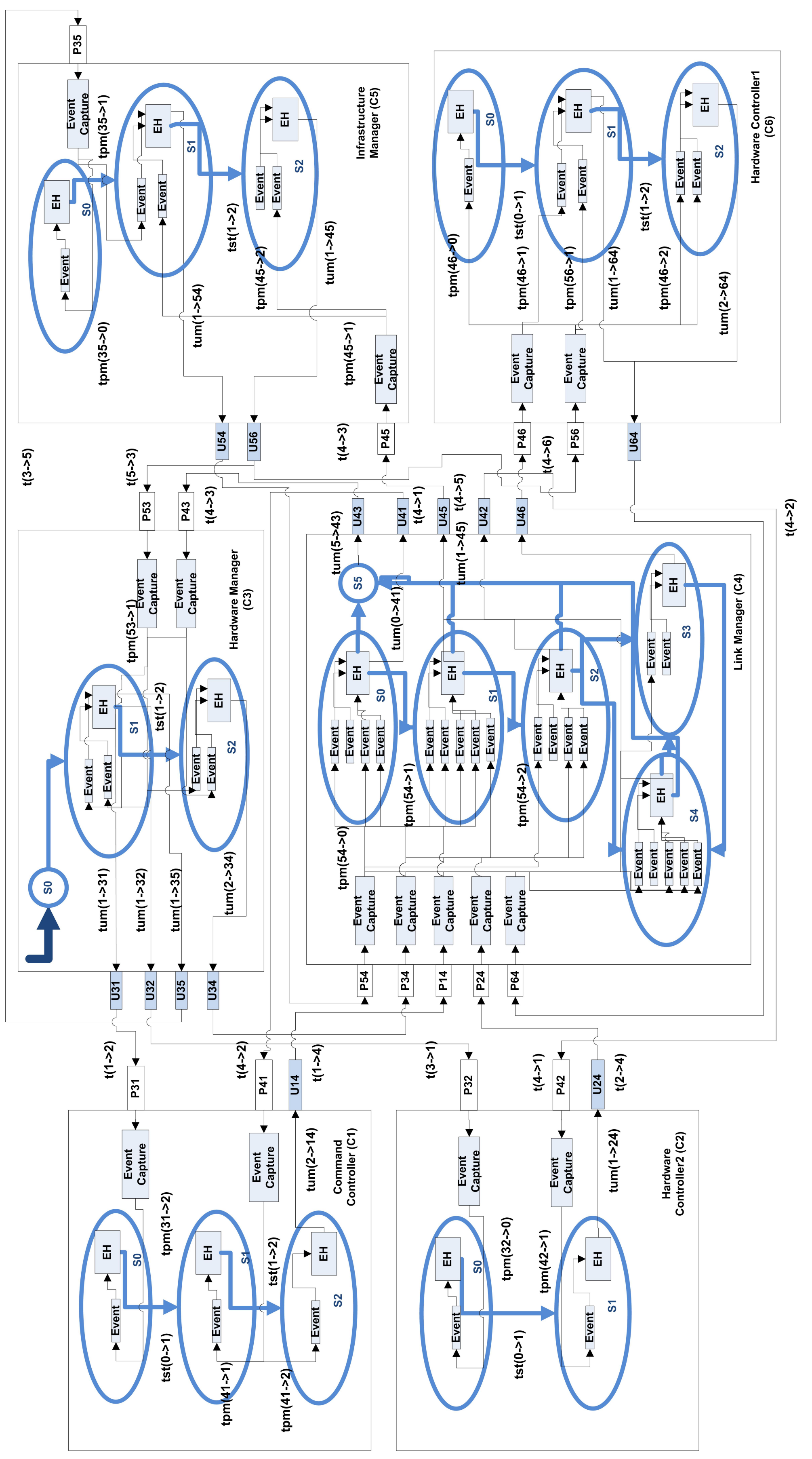


Figure 6.5: The CSIEDBG of the subject program

Test cases are generated from three test criteria. Test input data with time stamps are designed to trigger timed message sequences. Instrumentations are inserted throughout the subject program to provide trace information of program execution. During test execution, real-time system behaviors are recorded by the instrumentation. Execution results are compared with expected results. For the manual specification testing method, results are observed at the output of the integrated system. For the all-interface-transition coverage testing criteria, results are compared at all the provider services, user services, message edges, states, and transitions that are defined in CSIBG model. For the all-interface-event coverage testing technique, results are compared at all the provider services, user services, message edges, states, transitions, event captures, event handlers, and events that are defined in the CSIEDBG. From the latter technique, the observability is increased and the faults that do not propagate to system output in the context of particular test executions could be detected.

Embedded systems employing multiple tasks can have non-deterministic outputs, which complicates the determination of expected outputs for given inputs. To provide a comprehensive evaluation of each test method, I designed five sets of test cases for each coverage criterion, and then analyzed the minimum, average, and maximum numbers of faults detected by each test method. Different test cases were generated by varying test data values and test data input sequence for each test path. The results for all the five test sets are listed in Table 6.2. The fault numbers are taken from Table 6.1.5. Table 6.3 lists the number of faults detected for each test method in each test set. Table 6.5 lists the mutants score for each test method.

Table 6.2: Number of faults detected for each test method in each test set

	Manual Specification					**All-Interface-Transition**					**All-Interface-Event**				
Fault	**S1**	**S2**	**S3**	**S4**	**S5**	**S1**	**S2**	**S3**	**S4**	**S5**	**S1**	**S2**	**S3**	**S4**	**S5**
1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
3	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
4	NF	F	NF	NF	NF	NF	NF	NF	F	NF	NF	F	NF	NF	NF
5	NF	NF	NF	NF	NF	F	F	F	F	F	F	F	F	F	F
6	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
7	F	NF	F	F	NF	F	NF	F	F	NF	F	F	F	F	F
8	NF	NF	F	NF	NF	NF	F	F	NF	F	F	F	F	F	F
9	NF	NF	F	NF	NF	NF	NF	F	NF	NF	F	F	F	F	F
10	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
11	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
12	NF	F	NF	NF	F	F	F	F	F	F	F	F	F	F	F
13	NF	F	NF	NF	F	F	F	F	F	F	F	F	F	F	F
14	NF	NF	NF	NF	NF	NF	NF	NF	NF	NF	F	F	F	F	F
15	F	F	NF	F	F	F	F	F	F	F	F	F	F	F	F
16	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
17	NF	NF	F	NF	NF	NF	NF	NF	F	NF	F	F	F	F	F
18	NF	F	NF	NF	F	F	F	F	F	F	F	F	F	F	F
19	NF	NF	NF	NF	NF	NF	NF	NF	F	NF	NF	NF	NF	F	NF
20	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
21	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
22	F	F	F	F	F	NF	F	F	NF	F	F	F	F	F	F
23	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F

continued on next page

	Manual Specification					All-Interface-Transition					All-Interface-Event				
Fault	**S1**	**S2**	**S3**	**S4**	**S5**	**S1**	**S2**	**S3**	**S4**	**S5**	**S1**	**S2**	**S3**	**S4**	**S5**
24	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
25	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
26	F	F	F	F	F	NF	F	F	NF	F	F	F	F	F	F
27	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
28	F	F	F	F	F	NF	F	NF	NF	F	F	F	F	F	F
29	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
30	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
31	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
32	NF	F	NF	NF	F	F	F	F	F	F	F	F	F	F	F
33	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
34	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
35	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
36	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
37	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
38	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
39	NF	F	NF	NF	NF	F	F	F	F	F	F	F	F	F	F
40	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
41	NF	F	NF	NF	NF	F	F	F	F	F	F	F	F	F	F
42	F	NF	F	F	NF	F	F	F	F	F	F	F	F	F	F
43	F	F	NF	F	F	F	F	F	F	F	F	F	F	F	F
44	NF	F	NF	NF	F	F	F	F	F	F	F	F	F	F	F
45	NF	NF	NF	NF	NF	NF	NF	NF	NF	NF	NF	NF	NF	NF	F
46	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F

continued on next page

	Manual Specification					All-Interface-Transition					All-Interface-Event				
Fault	**S1**	**S2**	**S3**	**S4**	**S5**	**S1**	**S2**	**S3**	**S4**	**S5**	**S1**	**S2**	**S3**	**S4**	**S5**
47	F	NF	F	F	F	F	F	F	F	F	F	F	F	F	F
48	F	NF	F	F	NF	F	NF	NF	F	NF	F	F	NF	F	F
49	NF	NF	F	F	F	F	F	NF	F	F	F	F	F	F	F
50	F	NF	F	F	F	F	F	F	F	NF	F	F	F	F	F
51	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
52	NF	NF	F	F	F	NF	NF	NF	F	F	F	F	F	F	F
53	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
54	NF	NF	F	F	F	F	F	F	NF	NF	F	F	F	F	F
55	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
56	NF	NF	F	F	F	F	F	NF	F	F	F	F	F	F	F
57	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
58	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
59	NF	F	F	F	F	NF	F	F	NF	F	F	F	F	F	F
60	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F

Table 6.3: Number of faults detected for each test method in each test set

	Faults Detected				
Test Method	**Test Set 1**	**Test Set 2**	**Test Set 3**	**Test Set 4**	**Test Set 5**
Manual Specification Tests	40	43	47	46	47
All-Interface-Transition Coverage Tests	48	50	49	53	50
All-Interface-Event Coverage Tests	57	58	56	58	58

Table 6.4: Number of test case for each test method in each test set

	Test Cases				
Test Method	**Test Set 1**	**Test Set 2**	**Test Set 3**	**Test Set 4**	**Test Set 5**
All-Interface Coverage Tests	25	25	27	27	27
All-Interface-Transition Coverage Tests	31	33	31	33	33
All-Interface-Event Coverage Tests	40	41	40	41	41

Table 6.5: Quantitative Analysis of Mutants Score for each test method

	Manual Specification Tests	**All-Interface-Transition Coverage Tests**	**All-Interface-Event Coverage Tests**
Minimum	67% (40)	80% (48)	93% (56)
Average	75% (45)	83% (50)	95% (57)
Maximum	78% (47)	88% (53)	97% (58)

The following graph shows the mutation score against test method:

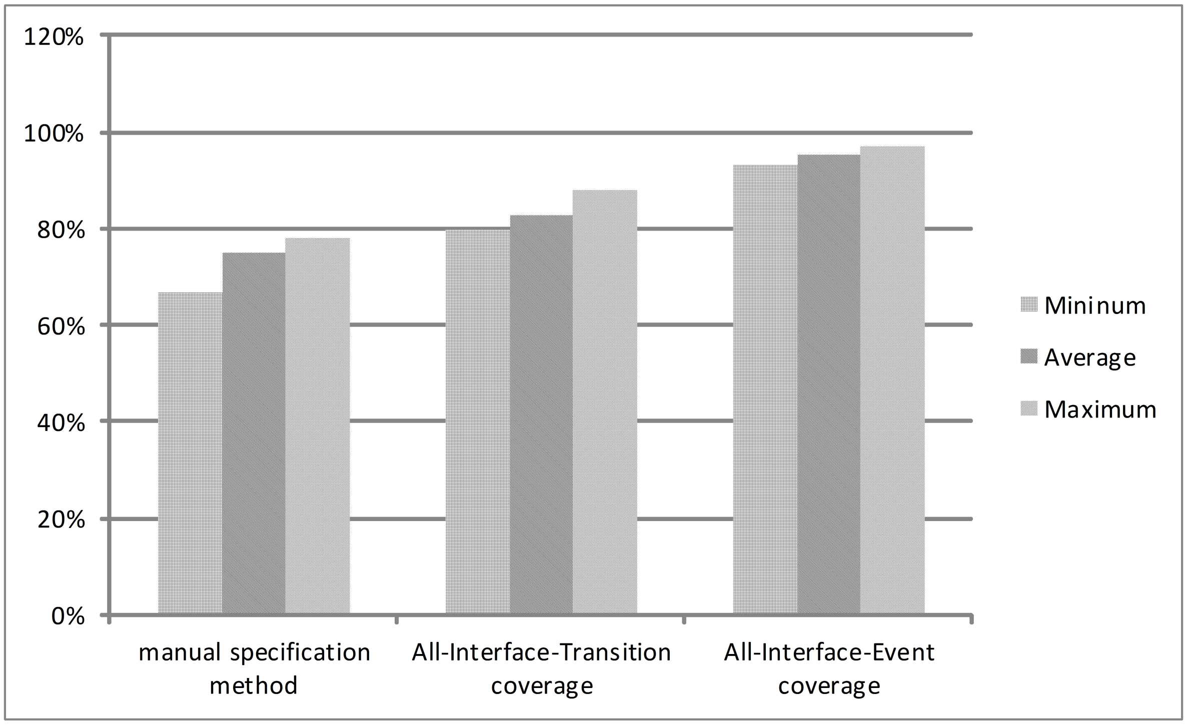


Figure 6.6: The graph of mutant score

Given a distribution of the above five test sets, the level of confidence indicates the probability that the confidence range captures this true population parameter. It does not describe any single sample. This value is represented by a percentage. I set the desired level of confidence as 95%. In order to prove that 95% of the observed confidence intervals will hold the true value of the parameter, I calculated the sample mean, standard deviation, standard variance, and confidence level from the mutantion score of three test methods. As shown in Table 6.6, we can say that, with 95% confidence level, AITC is better than Manual Specification and AIEC is better than AITC.

Table 6.6: Statistic Analysis of Mutant Scores Comparison

	AITC vs. Manual Specification	**AIEC vs. AITC**
mean	9.0	12.3
std dev	4.5	2.5
variance	20.3	6.39
n	5	5
conf. level	0.95	0.95
1/2 of conf. interval	5.59	3.14
upper bound	14.6	15.5
lower bound	3.4	9.2

The goal of this study was twofold. One was to see if architecture and design-based testing could be practically applied. The second was to evaluate the merit of architecture and design-based technique by comparing it with traditional testing. From the experimental results we conclude that the goals were satisfied; the architecture and design-based testing technique was applied and worked fine, and performed better than the manual specification-based techniques. All-Interface-Event Coverage was able to detect more faults than All-Interface-Transition Coverage. All-Interface-Event Coverage detected 95% of the faults.

The faults not detected by at least one of the five test sets from AIEC are related to the hardware design and timing. Fault 4 could be detected if loop transitions were considered when deriving test message sequences in the test case. Fault 19 was hard to detect because the missing internal event was not part of the triggering conditions to any state transition. Fault 45 was not detected because the state conflict did not cause any failure in the hardware manager. Fault 48 was not detected because the added timed delay in hardware controller did not cause any conflicts to other components during its operation.

I also measured the code coverage ratio using the Wind River Workbench development tool [84] with the code coverage feature. The averaged results from the five test sets for all the three test techniques are listed in Table 6.7.

Table 6.7: Code coverage ratio

Coverage Unit	**Manual Specification Tests**	**All-Interface-Transition Coverage Tests**	**All-Interface-Event Coverage Tests**
Function	82.3%	88.1%	95.3%
Block	73.5%	84.6%	93.6%

Function coverage reports whether tests invoked each function or procedure. Block coverage reports whether each executable statement is encountered. Most testers would agree that 73.5% statement coverage is so low that it's almost meaningless. Indeed, randon values thrown at the program will usually achieve almost 60% to 75% [9]. 84.6% is better, but general agreement is that many faults are likely to be missed with such low coverage. Over 90% coverage is considered good covergae. The fault finding results in table 6.2 agree with these general beliefs.

6.3 Threats to Validity

This experiment has several threats to validity. Most obviously, the experiment was performed on a single component-based real-time embedded system. Thus we cannot be sure that the success would be duplicated in other settings. Getting access to an industrial real-time embedded system for reserach purpose is very difficult. The process to get the work approved may take multiple discussions from different groups in the organization. Another potential validity threat is that lots of work has been done by hand. Human testers had to make decisions. It is possible that different testers would have different results. The faults we used in the subject program may not cover all the typical faults at the architectural level. An architectural fault classification is needed for further experiment. Taken together, these threats mean that I cannot conclude that this type of testing will succeed in all settings. Rather, I know that it is possible for this type of testing to improve testing and lead to higher quality software in some settings.

6.4 Conclusion

From this experiment application, we can see that the architecture and design-based testing technique can be practically applied, and the evaluation shows that it can find different faults effectively. This result indicates that this testing approach can benefit testers who are performing architecture/system testing on software.

Chapter 7: Conclusion and Future Work

7.1 Conclusion

This dissertation has presented a new strategy for component-based real-time embedded software integration testing that is based on a test model CREATEG that combines information from architecture design artifacts into a graph.

The motivation is to exercise component interactions in the context of multiple component state combinations to detect potential faults. Therefore, it takes into account the states of all components involved in a collaboration to exercise component interactions occuring at specific times in the context of integration testing. For instance, if the functionality provided by a component depends on the states of other components, then the proposed technique can effectively detect faults due to invalid component states.

We validate the research with an industrial post-hoc observational field study [85]. We built an experimental tool and generated faulty versions of a real-time embedded system under test using carefully selected mutation operators. The empirical results show that the proposed approach effectively detects various kinds of integration faults. In particular, the All-Interface-Event Coverage Criterion successfully detected 93% of the seeded faults and was particularly effective at detecting faults related to the state-behavior of interacting components.

We have also presented various coverage criteria to generate test paths, and algorithms to automate generation of test paths. The most demanding criterion, All-Path Coverage, is very expensive and it is not clear that it can scale up in all situations. It is therefore important that less expensive criteria be carefully investigated in future work as they will likely be more suitable in many situations where test budgets are limited. Now that the potential of our CREATEG approach has been demonstrated, new criteria to select high yield subsets of the CREATEG test paths could be devised.

The method presented in the dissertation has a very clear, structured, process to follow. It was also very convenient to have a range of test criteria, allowing testers to start with simpler criteria and move up to stronger criteria when needed. Table 6.7 in Chapter 6.1 showed that the tests generated from All-Interface-Event coverage criterion accomplished 93.6% coverage of source code, the tests generated from All-Interface-Transition coverage criterion provided 84.6% coverage of the source code, and the tests derived from manaul-specification method produced only 73.5% coverage of the source code. The manual specification method definitely needs some auxiliary tests to achieve high coverage of the source code.

This method also improves the low observability problem. We addressed the observability problem by logging intermediate values on each test path, making it much easier to diagnose the differences in expected and actual results.

A disadvantage of the modeling approach is that it puts a burden on the testers. To create the models, the test design team needs to understand software architecture design to analyze UML diagrams. In addition, the test team needs to have substantial domain knowledge.

This dissertation focused on real-time requirements instead of other extra-functional requirements. Task scheduling, task priorities that change over time and clock-based deadlines were out of the scope of our research, however they remain as part of a future research topic.

7.2 Future Work

There are several future areas to explore.

First, we applied this technique only to one component-based real-time embedded system. To access broad feasibility, we would like to apply this technique to other component-based real-time embedded systems.

Second, we could promote this model with requirement analysts, designers, programmers, and testers in the industry to greatly improve the understanding of the entire process. Having the models available to all the team members will make it very easy to adapt to changes in the requirements, and identify relations or constraints among input attributes to the software.

Third, more research can be carried out in the analyzing of component-based real-time embedded software architectures. As pointed out in Chapter 2, properties such as concurrency, timeliness, and multi-tasking need to be checked as we carry out testing and analysis at the architecture level.

Bibliography

Bibliography

[1] http://www.microsoft.com/net, last access August 2013.

[2] http://www.oracle.com/technetwork/java/javaee/ejb/, last access August 2013.

[3] http://www.omg.org/spec/CCM/, last access August 2013.

[4] http://www.corba.org/, last access December 2013.

[5] IBM rational rhapsody. Online. http://www-03.ibm.com/software/products/en/ratirhapfam, last access May 2014.

[6] Object management group (OMG). http://www.omg.org, last access April 2014.

[7] Software communications architecture specification (v2.2). Online, December 2001. http://www.public.navy.mil/jpeojtrs/sca, last access Janurary 2012.

[8] Paul Allen. *Component-based development for enterprise systems: Applying the SELECT Perspective.* Cambridge University Press, Cambridge, UK, 1998.

[9] Paul Ammann and Jeff Offutt. *Introduction to Software Testing.* Cambridge University Press, Cambridge, UK, 2008. ISBN 0-52188-038-1.

[10] Aritra Bandyopadhyay and Sudipto Ghosh. Test input generation using UML sequence and state machines models. In *29th International Conference on Verification and Validation*, number 9793667, pages 416–429, April 2009.

[11] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Addison Wesley, 2nd edition, 2003.

[12] Benoit Baudry and Yves Le Traon. Measuring design testability of a UML class diagram. *Information and Software Technology*, 47(13):859–879, October 2005.

[13] Stefan Berner, Roland Weber, and Rudolf K. Keller. Observations and lessons learned from automated testing. In *27th International Conference on Software Engineering, ICSE 2005*, pages 571–579, New York, NY, July 2005.

[14] Sami Beydeda and Volker Gruhn. An integrated testing technique for component-based software, computer systems and applications. In *Computer Systems and Applications, ACS/IEEE International Conference on*, pages 328–334, Beirut, Lebanon, June 2001.

[15] I. Bicchierai, G. Bucci, L. Carnevali, and E. Vicario. Combining uml-marte and preemptive time petri nets: An industrial case study. page 18061818, October 2013.

[16] I. Bicchierai, G. Bucci, L. Carnevali, and E. Vicario. Combining uml-marte and preemptive time petri nets: An industrial case study. In *Industrial Informatics, IEEE Transactions on (Volume:9 , Issue: 4)*, page 18061818. IEEE, October 2013.

[17] Robert V. Binder. Design for testability with object-oriented systems. *Communications of the ACM*, 37(9):87–101, September 1994.

[18] R. Bouaziz and I. Berrada. Testing component-based real time systems. In *Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing SNPD '08*, pages 888–894. IEEE, August 2008.

[19] Rachid Bouaziz and Ismail Berrada. Testing component-based real time systems. In *Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, page 888894. IEEE, August 2008.

[20] L.C. Briand, J. Cui, and Y. Labiche. Towards automated support for deriving test data from UML statecharts. In *Proceedings of the ACM/IEEE International Unified Modeling Language conference (UML 2003)*, pages 249–264, July 2003.

[21] Lionel C. Briand and Yvan Labicher. A uml-based approach to system testing. 1(1):10 – 42, September 2002.

[22] Bart Broekman and Edwin Notenboom. *Testing Embedded Software.* Addison Wesley, 2002.

[23] Magiel Bruntink and Arie van Deursen. An empirical study into class testability. *Journal of Systems and Software*, 79(9):1219–1232, September 2006.

[24] J. Chilenski and L. A. Richey. Definition for a masking form of modified condition decision coverage (MCDC). Online, 1997. http://www.boeing.com/nosearch/mcdc/.

[25] Matjavz Colnaric and Domen Verber. *Distributed Embedded Control Systems.* Springer, 1 edition, December 2007.

[26] I. Crnkovic. Component-based software engineering for embedded systems. In *the 27th International Conference on Software engineering ICSE'05*, pages 712–713. ACM, 2005.

[27] Ivica Crnkovic. Component-based approach for embedded systems. In *27th International Conference on Software Engineering, 2005*, pages 712–713, May 2005.

[28] Ivica Crnkovic, Severine Sentilles, Aneta Vulgarakis, and Michel Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 16(8):844–857, September 2010.

[29] S. J. Cunning and J. W. Rozenblit. Automating test generation for discrete event oriented embedded systems. *Intelligent and Robotic Systems*, 41:87–112, 2004.

[30] Nrusingh Prasad Dash, Ranjan Dasguptay, Jayakar Chepadaz, and Arindam Halderx. Event driven programming for embedded systems - a finite state machine based approach. In *The Sixth International Conference on Systems*, The Netherlands Antilles, January 2011.

[31] R. de Medeiros, M.M. Gois, D.L. Rossi, and V. Bonato. Designing embedded systems with marte: A pim to psm converter. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. IEEE, 2012.

[32] S. R. S. de Souza, S. C. P. F. Fabbri, W. L. de Souza, and J. C. Maldonado. Mutation testing applied to Estelle specifications. *Software Quality Journal*, 8(4):285–301, 1999.

[33] M. E. Delamaro, J. C. Maldonado, and A.P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, March 2001.

[34] Trung T. Dinh-Trong, Sudipto Ghosh, and Robert B. France. A systematic approach to generate inputs to test UML design models. In *ISSRE '06. 17th International Symposium on Software Reliability Engineering*, number 1071-9458, pages 95–104, Raleigh, NC, November 2006. IEEE.

[35] J. Fredriksson and R Land. Reusable component analysis for component-based embedded real-time systems. In *29th International Conference on Information Technology Interfaces, 2007*, number 9793667, pages 615– 620, Cavtat, Croatia, June 2007. IEEE.

[36] Leonard Gallagher and Jeff Offutt. Test sequence generation for integration testing of component software. *The Computer Journal*, 52(5):514–529, 2009.

[37] S. Ganesan, V. Alladi, J. Wei, and K. Alladi. Designing embedded real-time systems (ERTS) with model driven architecture (MDA), 2004. SAE Technical Paper.

[38] Object Management Group. OMG unified modeling language specification (v2.1), November 2007. http://www.omg.org.

[39] Object Management Group. OMG MOF 2 XMI mapping specification. Online, 2011. http://www.omg.org/spec/XMI/2.4.1/, last access March 2014.

[40] Object Management Group. UML profile for marte: Modeling and analysis of real-time embedded systems, June 2011. http://www.omg.org/spec/MARTE.

[41] Jing Guan, Jeff Offutt, and Paul Ammann. An industrial case study of structural testing applied to safety-critical embedded software. In *International Symposium on Empirical Software Engineering, ISESE 2006*, Rio de Janeiro, Brazil, September 2006. ISESE 2006.

[42] M. J. Harrold, D. Liang, and S. Sinha. An approach to analyzing and testing component-based systems. In *First International ICSE Workshop on Testing Distributed Component-Based Systems*, 1999.

[43] J. Hartmann, C. Imoberdorf, and M. Meisinger. Uml-based integration testing. In *ACM Transactions on Software Engineering and Methodology*, volume 25, pages 60–70, New York, NY, September 2000. ACM.

[44] Pao-Ann Hsiung, Win-Bin See, Trong-Yen Lee, Jih-Ming Fu, and Sao-Jie Chen. Formal verification of embedded real-time software in component-based application frameworks. In *Software Engineering Conference*, pages 21–28, Eighth Asia-Pacific, 2001. APSEC 2001.

[45] Zhenyi Jin and Jeff Offutt. Coupling-based criteria for integration testing. *The Journal of Software Testing, Verification, and Reliability*, 8(3):133–154, September 1998.

[46] A. Johnsen, K. Lundqvist, P. Pettersson, and O. Jaradat. Automated verification of aadl-specifications using uppaal. In *Proceedings of the 14th IEEE International Symposium on High-Assurance Systems Engineering (HASE)*, page 130138. IEEE, October 2012.

[47] Stefan Jungmayr. Identifying test-critical dependencies. In *Proceedings of IEEE International Conference on Software Maintenance*, Montréal, Canada, October 2002.

[48] Joao Cadamuro Junior and Douglas Renaux. Efficient monitoring of embedded realtime systems. In *Fifth International Conference on Information Technology: New Generations*, pages 651–656, Las Vegas, NV, USA, April 2008. IEEE.

[49] Teemu Kanstren. A study on design for testability in component-based embedded software. In *Sixth International Conference on Software Engineering Research, Management and Applications*, pages 31–38, August 2008.

[50] Saehwa Kim and Jamison Masse. SCA-based component framework for software defined radio. In *IEEE Workshop on Software Technologies for Future Embedded Systems, 2003*, number 7953256, pages 3–6. IEEE, May 2003.

[51] Ronny Kolb and Dirk Muthig. Making testing product lines more efficient by improving the testability of product line architectures. In *Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis*, pages 22–27, Portland, Maine, July 2006.

[52] K. Larsen, M. Mikucionis, B. Nielsen, and A. Skou. Testing real-time embedded software using uppaal-tron: an industrial case study. In *In 5th ACM Intl. Conf. on Embedded Software*, page 299306. ACM, May 2005.

[53] Kim Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal. In *In Jens Grabowski and Brian Nielsen, editors, International workshop on Formal Approaches to Testing of Software*. IEEE, September 2004.

[54] Shaw Ping Lee. Experience report: Rapid model-driven waveform development with UML. In *Software Defined Radio08 Technical Conference*, pages 26–30, October 2008.

[55] K. Li, R. Groz, and M. Shahbaz. Integration testing of components guided by incremental state machine learning. In *Testing: Academic and Industrial Conference - Practice And Research Techniques,2006. TAIC PART 200*, pages 59–70. IEEE, August 2006.

[56] Nan Li, Jeff Offutt, Paul Ammann, and Wuzhi Xu. The model-driven test design process. Online, August 2010. Unpublished manuscript.

[57] Jie Liu and Edward A. Lee. Timed multitasking for real-time embedded software. *IEEE Control Systems*, 23(1):65–75, Feburary 2003.

[58] Ziwei Liu, Jing Liu, Jifeng He, Frdric Mallet, and Miaomiao Zhang. Formal specification of hybrid marte statecharts. In *In 2012 IEEE Sixth International Symposium on Theoretical Aspects of Software Engineering*, page 5966. IEEE, June 2012.

[59] Shourong Lu, Wolfgang A. Halang, and Lichen Zhang. A component-based UML profile to model embedded real-time systems designed by the MDA approach. *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 23(1), August 2005.

[60] G. Macariu and V. Cretu. A timed automata testing model for component-based embedded real-time software. In *Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on*, pages 121–130. IEEE, 2010.

[61] P. D. L. Machado, J. C. A. Figueiredo, E. F. A. Lima, A. E. V. Barbosa, and H. S. Lima. Component-based integration testing from uml interaction diagrams. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pages 2679–2686. IEEE, Octiber 2007.

[62] Marius Mikucionis, Kim G. Larsen, Brian Nielsen, and Arne Skou. Testing realtime embedded software using uppaal-tron an industrial case study. In *In Embedded Software (EMSOFT)*, New Jersey, September 2005. IEEE.

[63] A. Moller, M. Akerholm, J. Froberg, and M. Nolin. Industrial grading of quality requirements for automotive software component technologies. In *International Conference on Software Testing Verification and Validation, Real-Time Systems ICST'09*, Denver, March 2009.

[64] Glenford J. Myers. *The Art of Software Testing.* John Wiley and Sons, Inc., Hoboken, New Jersey.

[65] Robert Nilsson, Jeff Offutt, and Sten F. Andler. Mutation-based testing criteria for timeliness. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC04)*, Hong Kong, China, September 2004.

[66] Jeff Offutt and A. Abdurazik. Generating tests from UML specifications. In *Proceedings of the 2nd International Conference on the UML*, volume 1723, page 416429, Fort Collins, CO, USA, October 1999. LNCS.

[67] Jeff Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *Journal of Software Testing, Verification and Reliability*, 13(1):25–53, 2003.

[68] Hemant D. Pande and William Landi. Interprocedural def-use associations in c programs. In *In Proceedings of the Symposium on Testing, Analysis, and Verification, TAV4*, pages 139–153, 1991.

[69] S. P. Shashank, P. Chakka, and D. V. Kumar. A systematic literature survey of integration testing in component-based software engineering. In *2010 International Conference on Computer and Communication Technology (ICCCT)*, pages 562–568. IEEE, 2010.

[70] Muhammad Jaffar-ur Rehman Hajra Asghar Muhammad Zohaib Z. Iqbal Shaukat Ali, Lionel C. Briand and Aamer Nadeem. a state-based approach to integration testing based on uml models. 49(11-12):10871106, November 2007.

[71] David E. Simon. *An Embedded Software Primer*. Addison-Wesley, 1 edition, August 1999.

[72] Ahyoung Sung, Byoungju Choi, and Seokkyoo Shin. An interface test model for hardware-dependent software and embedded OS API of the embedded system. *Computer Standards and Interfaces*, 29:430–443, May 2007.

[73] J. Suryadevara, C. Seceleanu, F. Mallet, and P. Pettersson. Verifying marte/ccsl mode behaviors using uppaal. In *in 11th International Conference on Software Engineering and Formal Methods (SEFM 2013)*. IEEE, September 2013.

[74] S. K. Swain, D. P. Mohapatra, and R. Mall. Test case generation based on state and activity models. *Journal of Object Technology*, 9(5):1–27, 2010.

[75] Wei-Tek Tsai, Lian Yu, Feng Zhu, and Ray Paul. Rapid embedded system testing using verification patterns. *IEEE Software*, 22:6875, 2005.

[76] George. T. Vadakkumcheril, M. Mythily, and M. L.Valarmathi. A simple implementation of UML sequence diagram to Java code generation through XMI representation. *International Journal of Emerging Technology and Advanced Engineering*, 3(12), December 2013.

[77] George. T. Vadakkumcheril, M. Mythily, and M. L.Valarmathi. A simple implementation of uml sequence diagram to java code generation through xmi representation. *International Journal of Emerging Technology and Advanced Engineering*, 3, December 2013.

[78] Franz Wotawa Valentin Chimisliu. Using dependency relations to improve test case generation from UML statecharts. In *IEEE 37th Annual Computer Software and Applications Conference Workshops (COMPSACW)*, July 2013.

[79] E. Valentini, G. Fliess, and E. Haselwanter. A framework for efficient contract-based testing of software components. In *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, volume 2, pages 219–222. IEEE, July 2005.

[80] Harold P.E. Vranken, Marc F. Witteman, and Ronald C. van Wuijtswinkel. Design for testability in hardware-software systems. *IEEE Design and Test of Computers*, 13(3):79–87, Fall 1996.

[81] A Kleppe J. Warmer and W. Bast. *MDA explained: The model driven architecture: practice and promise*. Addison Wesley, 2003.

[82] E. J Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15:54–59, September/October 1998.

[83] E. J Weyuker. Testing component-based software: a cautionary tale. 15(5):54 – 59, September 1998.

[84] Wind River. Wind River development tools. Online. http://www.windriver.com/products/development-tools/, last access May 2014.

[85] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslen. *Experimentation in software engineering: An introduction.* Kluwer Academic Publishers, 2008. ISBN 0-7923-8682-5.

[86] Ye Wu, Mei-Hwa Chen, and Jeff Offutt. UML-based integration testing for component-based software. In *The 2nd International Conference on COTS-Based Software Systems (ICCBSS)*, pages 251–260, Ottawa, Canada, February 2003.

[87] Ye Wu, Dai Pan, and Mei-Hwa Chen. Techniques for testing component-based software. In *7th IEEE International Conference on Engineering of Complex Computer Systems, 2001*, number 6984991, pages 222–232, Skovde, June 2001. IEEE.

[88] H. Yu, J.-P. Talpin, L. Besnard, T. Gautier, F. Mallet, C. Andre, and R. de Simone. Polychronous analysis of timing constraints in uml marte. In *In IEEE International Workshop on Model-Based Engineering for Real-Time Embedded Systems Design (hosted by ISORC 2010).* IEEE, May 2010.

[89] Weiqun Zheng and G Bundell. Model-based software component testing: A uml-based approach. In *Computer and Information Science, 2007. ICIS 2007. 6th IEEE/ACIS International Conference*, number 9864240, pages 891–899, New York, NY, July 2007. IEEE.

[90] Weiqun Zheng and G. Bundell. Test by contract for uml-based software component testing. In *Computer Science and its Applications, 2008. CSA '08. International Symposium on*, pages 377–382. IEEE, October 2008.

Biography

Jing Guan is a Staff Software Engineer at Lockheed Martin with 15 years of experience in embedded software design and development. She has been working in the area of real-time embedded system on a variety of projects involving Software Defined Radio, Tactical Network Management research programs, JTRS programs, and Iridium Satellite Systems. She has an undergraduate degree in Electrical Engineering from Northwestern Polytechnic University, an M.S. in Electrical engineering from University Maryland at College Park. She continues to learn and develop in her field by completing of a software engineering PhD program at the George Mason University in Fairfax, VA. Her research interests include component based real-time embedded system architecture, design and test. Her adviser is Dr. Jeff Offutt.