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

Jing Guan* and Jeff Offutt*
*Software Engineering, George Mason University, Fairfax VA, USA, jguan2@gmu.edu, offutt@gmu.edu

***Abstract*—The growing complexity of modern real-time embedded systems is leading to increased use of component-based software engineering (CBSE) technology. Although many ideas have been proposed for building component-based real-time embedded software, techniques for testing component-based real-time systems are scarce. The challenges not only come from the integration of different types of 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. Therefore, they should be considered while testing embedded software behavior. This paper addresses problems that emerge during the integration of component-based real-time embedded software. It 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 that describe functional interactions and their dependence relationships, as well as the time-dependent interaction among components. By using a graph-based test model, this paper presents a noval family of test adequacy criteria that help generate effective test cases and new algorithms to facilitate automate generation of the test cases.**

Keywords*—Software testing; model-based testing; component-based software engineering; real-time embedded systems

I. INTRODUCTION

The growing complexity of 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, more companies are using component-based modeling for embedded systems. This is expected to bring a number of advantages to embedded systems, such as rapid development times, the ability to reuse existing components, and the ability to compose sophisticated software [3].

Component-based technology has been extensively used for many years to develop software systems in desktop environments, office applications, and web-based distributed application [1]. 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. Embedded systems have constraints not found in desktop systems such as real-time requirements and resource efficiency. So adapting component-based software engineering (CBSE) to real-time embedded systems is harder than desktop systems [9].

Software components may be written in different programming languages, executed in various operational platforms, and 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). 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 testing. As a result, complex behaviors are observed when related components are integrated and many faults may not be visible until integration. Thus testing each component independently does not eliminate the need for system testing. A large number of possible interactions between components may need to be tested to ensure the correct functionality of the system [10].

As the size and complexity of software systems increase, problems stemming from the design and specification of overall 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 [12].

Testability for embedded software is difficult due to the low observability and controllability of embedded systems [21]. 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. The characteristics of component-based software introduce additional properties that increase the complexity, which affects the observability and controllability.

Also, embedded platforms are complex integrated systems where multiple real-time tasks execute in multi-tasking environments. In a component-based real-time embedded system, each individual component may run several tasks, so the number of messages between tasks across components is dramatically increased [15]. When integrating these components, 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.

This paper describes research to develop a new systematic software testing technique to thoroughly examine component interaction behavior during system testing activities. The technique is based on models created from software architectures and design specifications that specify the primary components, interfaces, dependences and behaviors of software systems. This method is based on analyzing models of the component interfaces, dependencies, and implementations. Finally, we present results from an industrial post-hoc observational field study [23].

This paper makes three contributions: (1) a new architec-

tural and design-based model for component-based real-time embedded software; (2) criteria and a process for generating tests from the new model; and (3) an industrial post-hoc observational field study applying the model and test generation process to an industrial embedded software system.

The remainder of this paper is organized as follows. A brief description of relevant concepts in real-time component-based engineering is presented in Section II. Related work is discussed in Section V. Section III provides a test model for real-time component-based software, and future work is described in section VI.

II. Background

This paper presents new test criteria to test component-based real-time embedded software. Thus, this section provides brief overviews of embedded systems and test criteria.

A. Component-based Real-Time Embedded Software

An *embedded system* is a special-purpose computer system built into a larger device [17]. They usually have no disk drive, keyboard or screen. Broekman and Notenboom [5] define embedded systems as a generic term for a broad range of systems covering cellular phones, railway signal systems, hearing aids, and missile tracking systems, among others. They specify that all embedded systems have a common feature in that they interact with the real physical world, controlling hardware.

These features greatly affect software testability and measurability in embedded systems. In embedded software systems, two main viewpoints of testability are considered from the architectural viewpoint: controllability and observability [4]. To test a component, we must be able to control its input and behavior. To see the results of testing, we must be able to observe the component's output and behavior. Finally, the system control mechanisms and observed data must be combined to form meaningful test cases.

A typical structure of embedded systems includes four layers: the application layer, the real-time operating system layer, the hardware adaptation layer, and the hardware layer. An application layer contains software that uses services from underlying layers, including a Real-Time Operating System (RTOS) and a Hardware Adaptation Layer (HAL). An RTOS performs task management, interrupt handling, inter-task communication, and memory management [17], allowing developers to create embedded system applications that meet functional requirements and deadlines using provided libraries and APIs. The HAL 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, transmit data created in one layer to other layers for processing. Faults in such interactions often result in execution failures.

Multitasking is an important aspect of embedded system design, and is managed by the RTOS. Due to thread interleaving, embedded systems that use multiple tasks can have non-deterministic output, which complicates the determination of expected outputs for test inputs.

Embedded systems often function under real-time constraints, which means the request must be completed within a required time interval after the triggering event [6]. In real-time systems, the temporal behavior [13] is as important as the functional behavior. Embedded systems can be classified into *hard real-time* and *soft real-time*. 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 leads to failure, possibly catastrophic. In soft real-time embedded systems, missed deadlines lead to performance degradation.

B. Criteria-based Testing

Criteria-based testing creates abstract models and then manipulates these models to design high quality tests. Ammann and Offutt's book [2] define criteria as being based on input domains, graphs, logical expressions, or grammars. Our research combines graph criteria with logic criteria in a novel way. Test criteria on graphs define test requirements to visit or tour specific nodes, edges, or sub-paths. The test requirements are expanded to *test paths*, which are complete traversals through the graph that are then used to generate test cases. This research adapts graph criteria to a new graph model of software architecture, specifically using edge coverage, which requires that each edge in a graph be covered. We combine edge coverage with the logic coverage criterion of Correlated Active Clause Coverage (CACC) [11] to deeply evaluate predicates that determine which edge is taken. CACC ensures that each clause in the predicate is tested with both *true* and *false* values in a situation where that clause determines the value of the predicate. A clause *determines* the value of a predicate if the other clauses have values such that if the clause under test is changed, the value of the entire predicate also changes.

III. Methodology

This paper presents a technique to perform integration testing of components by accounting for all possible states and events of collaborating components in an interaction. This is important as interactions may trigger correct behavior for certain states and not for others. To achieve this, we build a test model called the Component-based Real-time Embedded Model-based Test Graph (CREMTEG) model from component level sequence diagrams and component state diagrams involved in the component interactions. Test criteria for generating integration tests are developed from the CREMTEG models.

A. A Test Model for Component-Based Real-Time Embedded System

The CREMTEG is used to automatically generate test specifications for component integration testing. Components interact with each other through interfaces, which can be invoked by external or internal events. External events can be user actions, interrupts, hardware inputs, or message packets

from another part of the system. A typical internal event is a timer event, where a one-time event or a periodic event may be activated by a timer. Internal events also include internally generated events from another internal event inside the component. An interaction between components can be illustrated through a sequence of events and messages.

Component-based embedded systems 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 depends on the component behavior as well as the time-constraint and platform characteristics. For example, each task is assigned with a priority. The higher priorities are assigned to the tasks that have real-time deadlines. The impact of the extra-functional characteristics to a component integration test is illustrated with the following example.

The sequence diagram shown in Figure 1 has four components. They interact with each other by exchanging 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.

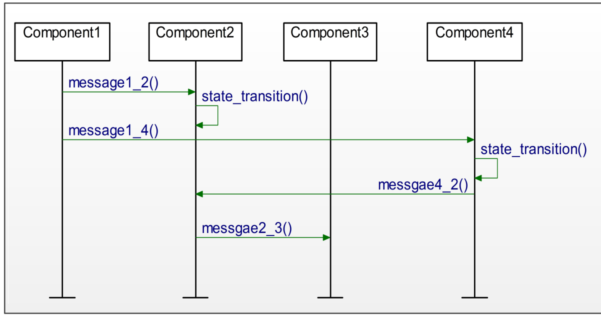

Fig. 1. A component level sequence diagram

Figures 2, 3, and 4 demonstrate three different senarios. In all three figures, Component1's task is assigned with the highest priority while Component3's task is assigned with the lowest priority. In Figure 2, 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 3, 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, 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.

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.

B. Novel Graph Representations for Designing Tests

A major contribution of this paper is the introduction of three novel graphs that model integration aspects of

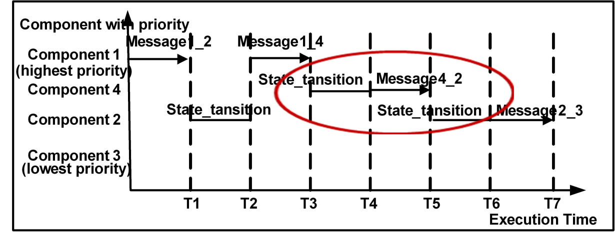

Fig. 2. Component interaction senario 1 per task priority

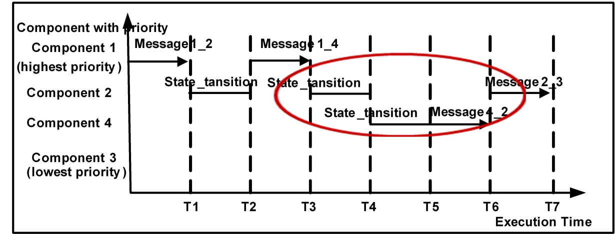

Fig. 3. Component interaction senario 2 per task priority

component-based real-time embedded software. The CIIG models the overall system configuration and connectivity by representing connectivity relationships between software components. The CSIBG represents dependency relationships among interface and states in a component. The CSIEDBG extends the CSIBG to model concurrent events. These three novel graphs are used to define test criteria.

1) The Component Interface Interaction Graph (CIIG): Components in this graph include application layer components, hardware adaption layer components, and infrastructure layer components. Interactions involve message exchanges that 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. As shown in Figure 5, 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. The Component Interface Interaction Graph (CIIG) is defined as:

$CIIG = (C,\ C_P_Interf,\ C_U_Interf,\ C_In_edge,$

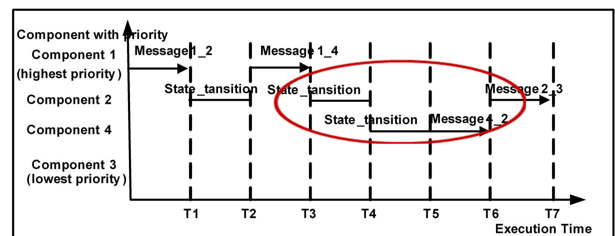

Fig. 4. Component interaction senario 3 per task priority

$C_Ex_edge,\ T_Ex_edge, TC$), where

- $C = \{C_1, C_2,\ \ldots,\ C_m\}$, 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 $i \in \{1,2,...,m\}$
- $C_In_edge = \{C_i.P \to C_i.U\}$, a finite set of internal component message edges $i \in \{1,2,...,m\}$
- $C_Ex_edge = \{C_i.U \to C_{i+1}.P\}$, a finite set of external component message edges $i \in \{1,2,...,m\}$
- $T_Ex_edge = \{t_i \to_j\}$, a finite set of time stamps at external component message edges $i \in \{1,2,...,m\}$, $j \in \{1,2,...,m\}$
- $TC = \{tc_i\}$, a finite set of time constraints on components interaction messages $i \in \{1,2,...,n\}$, $j \in \{1,2,...,m\}$, $tc_i = t_i - t_j\ rop\ c, c \in Integer,\ rop \in \{<, \leq, =, >, \geq\}$

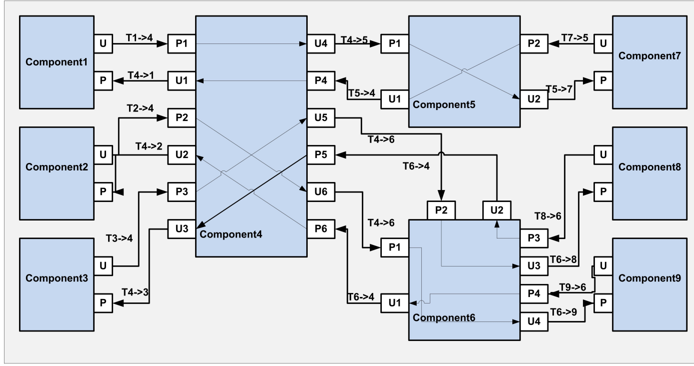

Fig. 5. An example Component Interface Interaction Graph (CIIG)

Figure 5 shows a CIIG of a system with nine components connected by eighteen message edges. This CIIG is formally defined as: $C = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8, C_9\}$, where

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

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

$C_In_edge = \{C_4.P_1 \to C_4.U_4, C_4.P_4 \to C_4.U_1, C_4.P_2 \to C_4.U_6, C_4.P_3 \to C_4.U_5, C_4.P_5 \to C_4.U_3, C_4.P_6 \to C_4.U_2, C_5.P_1 \to C_5.U_2, C_5.P_2 \to C_5.U_1, C_6.P_1 \to C_6.U_4, C_6.P_2 \to C_6.U_3, C_6.P_3 \to C_6.U_2, C_6.P_4 \to C_6.U_1\}$

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

$T_Ex_edge = \{t_1 \to_4, t_4 \to_5, t_5 \to_7, t_7 \to_5, t_5 \to_4, t_4 \to_1, t_2 \to_4, t_4 \to_6, t_6 \to_8, t_6 \to_9, t_8 \to_6, t_9 \to_6, t_6 \to_4, t_3 \to_4, t_4 \to_3\}$

$TC = \{tc_1, tc_2, tc_3, tc_4, tc_5, tc_6, tc_7, tc_8, tc_9\}$

$tc_1 = (t_4 \to_1 - t_1 \to_4) < 10ms$
$tc_2 = (t_4 \to_5 - t_5 \to_4) < 5ms$
$tc_3 = (t_5 \to_7 - t_7 \to_5) < 2ms$
$tc_4 = (t_4 \to_2 - t_2 \to_4) < 6ms$
$tc_5 = (t_6 \to_4 - t_4 \to_6) < 2ms$
$tc_6 = (t_9 \to_6 - t_6 \to_9) < 0.1ms$
$tc_7 = (t_4 \to_3 - t_3 \to_4) < 3ms$
$tc_8 = (t_6 \to_4 - t_4 \to_6) < 1ms$
$tc_9 = (t_8 \to_6 - t_6 \to_8) < 0.01ms$

A *timed message sequence* is depicted by C_Ex_edge, C_In_edge, and T_Ex_edge. The following example traces along the top of Figure 5.

$$C_1.U \xrightarrow{t_1 \to_4} C_4.P_1 \to C_4.U_4 \xrightarrow{t_4 \to_5} C_5.P_1 \to C_5.U_2 \xrightarrow{t_5 \to_7} C_7.P \to C_7.U \xrightarrow{t_7 \to_5} C_5.P_2 \to C_5.U_1 \xrightarrow{t_5 \to_4} C_4.P_4 \to C_4.U_1 \xrightarrow{t_4 \to_1} C_1.P$$

Another message sequence is shown below:

$$C_2.U \xrightarrow{t_2 \to_4} C_4.P_2 \to C_4.U_6 \xrightarrow{t_4 \to_6} C_6.P_1 \to C_6.U_4 \xrightarrow{t_4 \to_9} C_9.P \to C_9.U \xrightarrow{t_9 \to_6} C_6.P_4 \to C_6.U_1 \xrightarrow{t_1 \to_6} C_4.P_6 \to C_4.U_2 \xrightarrow{t_4 \to_2} C_2.P$$

Parallel messages can also be derived from the CIIG to describe the parallel behavior of the embeded system.

Time constraints define timing relationships between component interactions. In a message sequence, the time window 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.

A CIIG contains all the components and their interfaces, and shows connections between components and interfaces. A CIIG also captures the time stamp when an interaction between two components occurs. But a CIIG does not reflect the internal behavior of a component when receiving an incoming external message, or before triggering an outgoing external message. As shown in diagrams 2, 3 and 4, 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 graph (CSIBG) describes components' interactions and behavior.

2) The 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.

A Component State-based Interaction Behavior Graph (CSIBG) shows the behaviors of and the relations among multiple components in a possible message sequence. A CSIBG is composed of a set of component subnets (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 should happen.

In a CSIBG, transfer relations are described by placing a transition link from the receiving component's provider interface to all states where the incoming message may arrive. In the component subnet, upon receiving an external message, a sequence of state transitions occurs from an entry state. Transition edges are modeled by the attributes of a transition, including the accepting and sending states. A CSIBG is defined formally as follows:

$CSIBG = (Comp1(Sn_1, Pn_1, Un_1, Tn_1, In_1, On_1, TSTn_1, TPMn_1, TUMn_1), Comp2(Sn_2, Pn_2, Un_2, Tn_2, In_2, On_2, TSTn_2, TPMn_2, TUMn_2),\ \ldots, Compm(Pn_k, Un_k, Sn_k, Tn_k, In_k, On_k, TSTn_k, TPMn_k, TUMn_k), Msgn, TC)$, where

- Comp1 describes component subnet C1.
- $Sn_1 = \{S_1, S_2,\ \ldots, S_n\}$ is 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"*
- $TSTn_1 = \{tst_i{\to}_j\}$ is a finite set of time stamps at transition edges $i \in \{1, 2, ..., m\}$, $j \in \{1, 2, ..., m\}$
- $TPMn_1 = \{tpm_i{\to}_j\}$ is a finite set of time stamps at input message edges $i \in \{1, 2, ..., m\}$, $j \in \{1, 2, ..., m\}$
- $TUMn_1 = \{tum_i{\to}_j\}$ is a finite set of time stamps at 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 the CIIG. The starting point of an Msg is a user interface of a correlated component and 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"*.

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

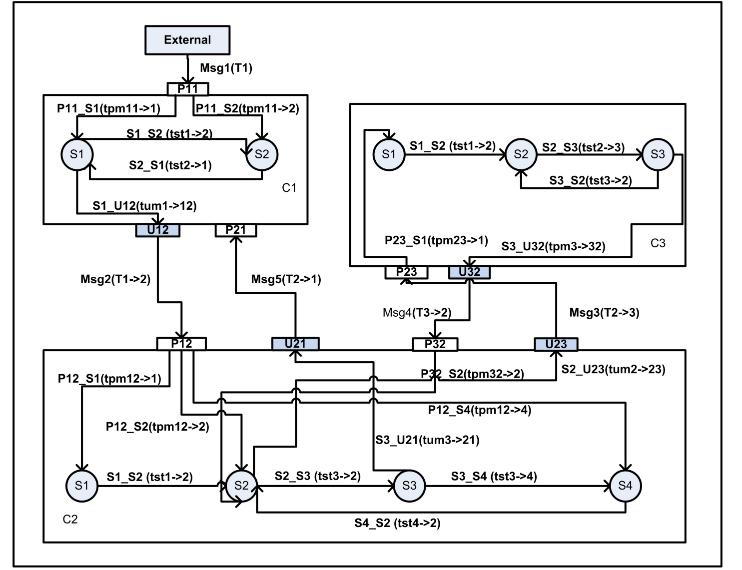

Fig. 6. An example Component State-based Interaction Behavior Graph (CSIBG)

$$\mathbf{Msg_1}\xrightarrow{T_1}P_{11}\xrightarrow{tpm_{11}\to 1}S_1\xrightarrow{tum_1\to 12}U_{12}\xrightarrow{T_1\to 2}\mathbf{Msg_2}$$

$$\mathbf{Msg_1}\xrightarrow{T_1}P_{11}\xrightarrow{tpm_{11}\to 1}S_1\xrightarrow{tst_1\to 2}S_2\xrightarrow{tst_2\to 1}S_1\xrightarrow{tum_1\to 12}U_{12}\xrightarrow{T_1\to 2}\mathbf{Msg_2}$$

$$\mathbf{Msg_1}\xrightarrow{T_1}P_{11}\xrightarrow{tpm_{11}\to 2}S_2\xrightarrow{tst_2\to 1}S_1\xrightarrow{tum_1\to 12}U_{12}\xrightarrow{T_1\to 2}\mathbf{Msg_2}$$

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:

$$\mathbf{Msg_2}\xrightarrow{T_1\to 2}P_{12}\xrightarrow{tpm_{12}\to 1}S_1\to S_1\xrightarrow{tst_1\to 2}S_2\xrightarrow{tum_2\to 23}U_{23}\xrightarrow{T_2\to 3}\mathbf{Msg_3}$$

$$\mathbf{Msg_2}\xrightarrow{T_1\to 2}P_{12}\xrightarrow{tpm_{12}\to 2}S_2\xrightarrow{tum_2\to 23}U_{23}\xrightarrow{T_2\to 3}\mathbf{Msg_3}$$

$$\mathbf{Msg_2}\xrightarrow{T_1\to 2}P_{12}\xrightarrow{tpm_{12}\to 4}S_4\to S_1\xrightarrow{tst_4\to 2}S_2\xrightarrow{tum_2\to 23}U_{23}\xrightarrow{T_2\to 3}\mathbf{Msg_3}$$

This information allows end-to-end sequences to be specified in terms of message exchanges between components and component internal flow. The sequences originate from external systems, propagate through various software components, and terminate at external systems.

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

3) The Component State-based Event-driven Interaction Behavior Graph (CSIEDBG): Multitasking is one of the most important aspects in real-time embedded systems. The CIIG

and CSIBG clearly described the timed sequences of method calls and paths of state transitions, but they do not deal with concurrency. 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. We therefore extend the CSIBG in several ways to include features required to visualize concurrent events. This graph is called the *Component State-based Event-driven Interaction Behavior Graph* (CSIEDBG).

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. Different types of events can be generated–external events, timer events, semaphores, and message queues. Each event is mapped to an 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 sends appropriate messages to other components. Figure 7 shows the CSIEDBG for the system in Figure 6.

1) Each component contains multiple provider services. Each provider service in a component has one event capture to process the incoming external message.
2) Each component contains multiple states and each state contains one event handler.
3) Each event capture generates one event in one state.
4) All events are handled by one event handler.

C. Test Model Generation

The previous subsection presented a method for testing component-based software. This model can be derived from the architectural framework, so can be available before the code is available. Multiple elements from an architecture framework can be used to derive the CIIG, and CSIBG graphs can be derived from several different model elements, including context diagrams, sequence diagrams, or state diagrams.

1) Constructing the CREMTEG: Sequence diagrams model interactions among objects in a functional thread. Interactions are ordered by time. Sequence diagrams may have different levels based on the objects, including the package level, subsystem level, component level, subcomponent level, and class level. Component level sequence diagrams are interaction diagrams consisting of a set of components and their relationships, including messages that may be dispatched among them, as well as interactions with external system. The C_In_edges and C_Ex_edges in CIIG graphs, and provider/user interfaces and message edges in a CSIBG, can be derived from component level sequence diagrams.

One component subnet is created for each component in the sequence diagram. For each message in the sequence diagram, one component provider interface, one component user interface, and one external component message edge is created. Additionally, one internal component message edge is created between two consecutive messages.

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

Each message in the sequence diagram has a sequence number, source, and destination component name. On the other hand, a component's state diagram defines its states and the transition messages it can receive. The CREMTEG 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.

2) Generating Test Paths from the CREMTEG: Test paths are generated from the CREMTEG CSIBG diagram. A *test path* derived from the CREMTEG starts with the beginning of an input event and contains a complete message sequence of a system level operation. Thus, each path tests a sequence of component interactions.

The total number of test paths in a CREMTEG 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 one state to another state after receiving a particular message. Multiple threads run independently, so the model associates a set of events to a transition. A transition may be triggered by the concurrent occurrence of a set of events.

Recall from the CREMTEG construction in section III-B 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 CREMTEG are necessarily feasible. Infeasible paths must therefore be detected manually by inspecting paths that contain guard conditions.

D. Test Criteria

Complex systems may have many components and states, which causes an exponential growth in the number of test paths. Thus we usually cannot test all paths. We adapt graph and logic coverage criteria [2] to define three criteria on the CREMTEG. During testing, a criterion can be chosen based on the level of testing needed.

1) All-Interface Coverage Criterion (AIC): This criterion ensures that each interface is tested once. It is adapted from *edge coverage*, which tours each edge in a graph [2]. AIC checks that interactions between components are taking place correctly, regardless of the component states. This criterion can be manually derived from software requirements without an architecture and design-based test model. It is practical and easy to implement so it is useful when a company has a tight budget and schedule to perform component integration testing.

2) All-Interface-Transition Coverage Criterion (AITC): This criterion ensures that each interface between components is tested once and each internal state transition path in a component is toured at least once. It is also adapted from *edge coverage*. AITC subsumes AIC.

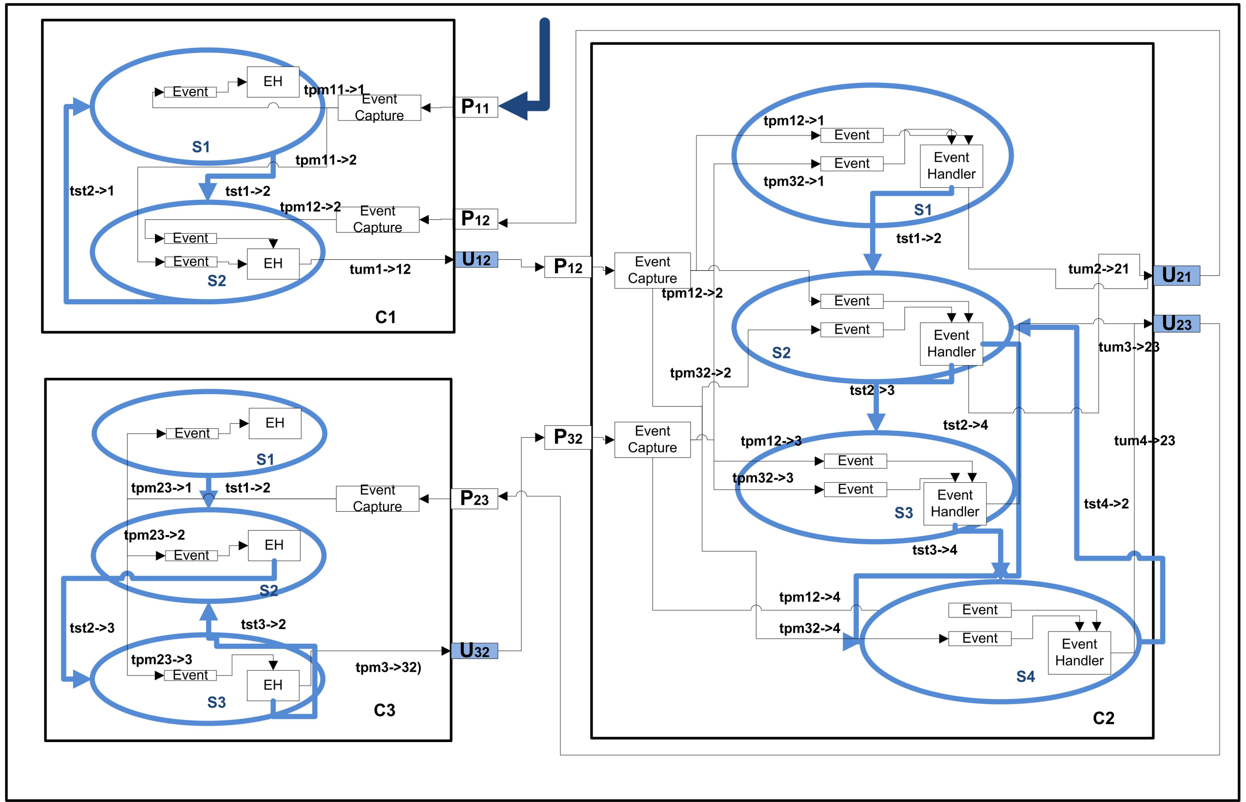

Fig. 7. An example Component State-based Event-driven Interaction Behavior Graph (CSIEDBG)

3) All-Interface-Event Coverage (AIEC): AIEC combines edge coverage with logic coverage (CACC). A component interface passes messages from a user component to a provider component. The provider component processes the message by *event capture* to generate *events* in the current component state. All generated events are sent to *event handler* to decide whether to transition to the *next state* or send an outgoing message to the next component.

Many of the resulting test paths contain decisions that control the branches taken during execution. Decisions in the *event handler* are modeled as predicate expressions, where each clause is an event. Rather than test each test path with only one value for the predicate, we apply CACC to create several test paths, each one with a different truth assignment for the predicate. If a predicate has four clauses ($n = 4$), say, $(a \,||\, b)$ && $(c \,||\, d)$, then we have $2^n = 16$ possible truth values. CACC requires $2 * (n - 1) = 6$ tests ({[t t t f], [t t f t], [t t f f], [t f t t], [f t t t], [f f t t]}).

IV. Empirical Study

To validate this model, we applied the CREMTEG model to an industrial software system to determine whether the test method can detect faults effectively. Wohlin et al. [23] calls this type of study an "industrial post-hoc observational field study." Our study is based on industrial software, as opposed to laboratory or open source software. The evaluation was performed after the software was completed and deployed ("post-hoc"), and observations were made on the field version. An experimental tool was developed to create the graphs and apply the criteria to develop the test paths as test requirements. Tests were run on versions of an industrial software system seeded with mutation-based faults. The system was designed at the architectural level with sequence and state diagrams.

A. Experimental Design

The experimental design is described in terms of the experimental procedure, the subject program, the test criteria and generation, and the seeded faults.

1) Experimental Procedure: First, we chose faults (described below) that are applicable to this subject program (also described below). Then we seeded the faults into the subject. Test sets were generated using the criteria in section III. Each test case was executed against each faulty version 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.

2) The Subject Program: We studied an industrial software system to be used in satellite communications domain. The program was written in C and C++ to run on VXWorks. The system contains more than 100 classes and approximately 75,000 lines of code. We tested a part of the system that contains about 10,000 lines of code. Because of its proprietary nature, we can only show a high level abstraction of the program structure.

This satellite control software receives real-time command and data from a ground user interactive system, processes the commands and data, and controls multiple types of hardware.

These hardware systems also send hardware metrics and status updates to the software. Users can request telemetry data, causing the system to send data to destination points (external user systems).

This subject program has several benefits. First, it is a significant industrial software system that is deployed and in use. It is rare to be able to test software of this nature in a research setting. Second, it uses all the architecture relations described in section III. Third, it is a typical component-based real-time embedded system that has many communications and connections between components. Fourth, it runs on a real-time hardware environment.

3) Test Adequacy Criteria: We used the three test criteria defined in section III: all-interface coverage (AIC), all-interface-transition coverage testing (AITC), and all-interface-event coverage (AIEC). Test data were hand generated by the first author to satisfy the CREMTEG test requirements. Test data were taken as inputs to a proprietary test automation tool that was developed by the industrial 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 test was successful. A test log was generated and saved after each test. A final test report was generated after all tests were run.

4) Fault Sets: Twenty-one mutation operators were designed to emulate faults in this type of software. The operators were defined at the architecture and design level. For example, one mutation operator removes the condition of a conditional message in the code. To seed faults, we first checked the model to see how many conditional messages were defined in the design model, then created faults for all the conditional messages. Faults were then seeded by hand.

For this work, we introduce four new mutation operators that model specific kinds of faults that are unique to this type of software. We adapted seventeen other mutation operators from several sources. Three integration operators are from Delamaro et al. [8], five specification operators from Souza et al. [7], two from Swain et al. [19], and seven timeliness operators are from Nilsson et al. [16]. The following list defines the 21 mutation operators used in the study. The source for the each operator is also given.

1) Initial State Exchanged (ISE): Change 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 the message. ([7])
2) Replace Return Statement (RetStaDel): Replace each return statement in a method with each other return statement in the method, one at a time. ([8])
3) Condition Missing (CM): Remove the condition of a conditional message. ([7])
4) Transition Deletion (TD): Remove a transition. ([7])
5) Event Exchange (EE): Switch an event with another to trigger a transition to a different state. ([7])
6) Event Missing (EM): Remove an event. ([7])
7) Argument Switch/Parameters Exchange (AS) at calling point: Change the order of the parameters (and type) passed in an interface call. ([8])
8) Interface Variables Exchange (IVE) at called module: Change the parameter passed in an interface call. The valid value of the parameter is replaced with an invalid value. ([8])
9) Alter Condition Operator (ACO): Change the condition that corresponds to a path condition in collaboration. ([19])
10) Guard Condition Violated (GCV): Negate the guard condition of a transition. ([19])
11) $\triangle$+ execution time: Changes the execution time of one component's task from T to T+ $\triangle$. ([16])
12) $\triangle$- execution time: This mutation operator changes the execution time of one components task from T to T- $\triangle$. ([16])
13) - precedence constraint: This mutation operator removes a precedence constraint relation between two components. ([16])
14) + precedence constraint: This mutation operator adds a precedence constraint relation between two components. ([16])
15) $\triangle$- inter-arrival time: This mutation operator decreases the inter-arrival time between requests for a task execution by a constant time $\triangle$. ([16])
16) $\triangle$+ pattern offset: Changes clock constraint from C to C+ $\triangle$ in guards on a transition from one state to another. ([16])
17) $\triangle$- pattern offset: Changes clock constraint from C to C- $\triangle$ in guards on a transition from one state to another. ([16])
18) Missing Provider Function (MPF): Remove the functions that are called by a component. (New to this paper.)
19) State Exchange (SE): Set the target state of a component to be the source state. (New to this paper.)
20) Wrong User State (WUS): Set the state of the calling component to an invalid state. (New to this paper.)
21) Conflicting State (CS): Set the states of two components in states that conflict with each other. (New to this paper.)

The fault seeding was 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. These 21 operators created 60 faults for the subject system, which were manually inserted into the subject program.

B. Test Results

To provide a comprehensive evaluation of each test method, we designed five sets of test cases for each coverage criterion, 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 sequences for each test path. Table I gives the number of faults detected for each test method in each test set, and Table II gives the mutation score for each test method.

This study had two goals. The first was to see if architecture and design-based testing could be practically applied. The

TABLE I. NUMBER OF FAULTS DETECTED BY EACH TEST METHOD IN EACH TEST SET

Test Method	Faults detected by test set 1	Faults detected by test set 2	Faults detected by test set 3	Faults detected by test set 4	Faults detected by test set 5
Manual Specification Tests (AIC)	40	43	47	46	47
All-Interface-TransitionCoverage Tests (AITC)	48	50	49	53	50
All-Interface-Event Coverage Tests (AIEC)	57	58	56	58	58

TABLE II. QUANTITATIVE ANALYSIS OF MUTANTS SCORE FOR EACH TEST METHOD

	AIC	AITC	AIEC
Minimum	67% (40)	80% (48)	93% (56)
Average	75% (45)	83% (50)	95% (57)
Maximum	78% (47)	88% (53)	97% (58)

second was to evaluate the merits of architecture and design-based testing by comparing it with other testing methods. The experimental results indicate that the goals were satisfied; the architecture and design-based testing technique, as implemented by the criteria AITC and AIEC, were applied and worked fine, and performed better than the non-architecture and design-based testing technique, AIC.

We also measured code coverage of the test sets using the Wind River Workbench development tool [22] with the "code coverage" feature. Results are shown in Table III, averaged over the five sets of tests.

TABLE III. CODE COVERAGE OF THE ALL-INTERFACE COVERAGE, ALL-INTERFACE-TRANSITION COVERAGE, AND ALL-INTERFACE-EVENT COVERAGE TESTS.

Coverage Unit	AIC	AITC	AIEC
Function	81.6%	87.3%	95.3%
Block	72.6%	83.3%	93.6%

Function coverage reports whether tests invoked each function or procedure, and block coverage reports whether each executable statement was executed. Most testers would agree that 72.6% statement coverage is so low that the testing is almost worthless ... over a quarter of the code is never even touched during testing! Indeed, random values thrown at the program will usually achieve 60% to 75% coverage [2]. 83.3% is better, but general agreement is that many faults are likely to be missed with such low coverage. Over 90% coverage is considered good coverage. These results concur with the fault finding results.

C. 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, which limits our ability to generalize the results. Getting access to an industrial real-time embedded system for research purpose is very difficult. The process to get this approved took multiple discussions from different groups and only succeeded because the first author was on the development team for the system. Another potential validity threat is the amount of work done by hand, including test value generation and fault insertion. In addition, 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.

V. RELATED WORK

Many research papers have studied the problem of effectively testing embedded software.

Several papers used informal specifications to test embedded systems focusing on the application layer. Tsai et al. [20] presented an approach to test embedded applications using class diagrams and state machines. Sung et al. [18] tested interfaces to the kernel via the kernel API and global variables that are visible in the application layer. These papers focused on the application layer, whereas this research tests component interactions across layers.

Several papers have used dataflow-based testing approaches to test interactions among system components [24], [10], [12]. Wu et al. [24] analyzed test elements that are critical to testing component-based software and proposed a group of UML-based test elements and test adequacy criteria to test component-based software. The test method in Gallagher and Offutt's paper [10] operated directly on object oriented software specifications, yielding a data flow graph and executable test cases that adequately cover the graph according to classical graph coverage criteria. However, none of the above papers addressed problems in testing embedded systems, as our work does. While the analysis techniques in this paper are similar, we use them to test interactions within embedded systems.

Although many empirical studies using testing techniques have been published, few have been applied to embedded systems, and none used real industrial software. Many papers on testing embedded systems include no empirical evaluation at all. This research examines fault detection capabilities on an industrial real-time embedded system.

Kanstren [14] presented a study on design for testability in component-based real-time embedded software based on two large-scale companies in the European telecom industry. Kanstren found that 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. 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.

VI. CONCLUSIONS AND FUTURE WORK

This paper presents a novel integration test strategy for component-based real-time embedded software. The strategy is based on a test model, CREMTEG, which combines information from architecture design artifacts into graphs. The motivation is to exercise component interactions in the context of multiple component state combinations. Therefore, the technique examines the states of all components involved in a collaboration to exercise component interactions in the context of integration testing. For example, if the functionality provided by one component depends on the states of other

components, then this new testing technique can effectively detect faults due to invalid component states.

We validated the technique with an industrial post-hoc observational field study [23]. We built an experimental tool and generated faulty versions of an industrial real-time embedded system. The faults were based on mutation operators, some existing and some new to this research. The empirical results show that the proposed approach effectively detects various kinds of integration faults. In particular, the all-interface-event criterion successfully detected 93% of the seeded faults and was particularly effective at detecting faults related to the state behavior of interacting components. Both AIEC and AITC performed much better than the previous manual method (AIC).

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

This modeling approach puts several knowledge burdens on the testers. To create the models, the test design team needs to understand software architecture design well enough to analyze UML diagrams. The test team also needs to have substantial domain knowledge.

To access broad feasibility, in the future we would like to apply this technique to other component-based real-time embedded systems. The challenge, of course, is that it is very difficult to obtain such systems from software companies.

Acknowledgments

Jing Guan's research is supported by Lockheed-Martin. Offutt is partially funded by The Knowledge Foundation (KKS) through the project 20130085, Testing of Critical System Characteristics (TOCSYC).

References

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

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

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

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

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

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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.

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

[23] 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.

[24] 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.