Reliability Assessment of Framework-Based Distributed Embedded Software Systems

Farokh B. Bastani, Sung Kim, I-Ling Yen
Computer Science Dept., MS EC-31
Univ. of Texas at Dallas
Richardson, TX 75083-0688
{bastani,ilyen,sungkim}@utdallas.edu

Ing-Ray Chen
Department of Computer Science
Virginia Tech (VPI&SU)
7054 Haycock Road
Falls Church, VA 22043

Abstract

Distributed embedded software systems, such as sensor networks and command and control systems, are complex systems with stringent performance, reliability, security, and safety constraints. These are also long-lived systems that must be continually upgraded and evolved to incorporate enhanced functionality. One approach for achieving high quality and evolvability for these systems is to organize them in the form of application-oriented frameworks that allow the system to be composed from orthogonal aspects that can be independently developed, evolved, and certified.

In this paper, we define a general framework that allows a distributed embedded system to have relatively independent aspects, including “plug-and-play” capability. We present conditions under which the reliability of the system can be inferred from the reliability of the individual aspects. The approach is illustrated for a framework-based distributed sensor network.

Keywords: Distributed embedded systems, Software composition, Application-oriented frameworks, Software reliability assessment.

1 Introduction

A distributed embedded system consists of an array of sensors, actuators, displays, and control logic. The sensors acquire information regarding the state of the system and the environment and send these to the control logic components and display (monitoring) stations. The control components, typically realized in software, embody all the intelligence in the system. They perform control-related computations driven by the specified control goals and then send commands to the actuators to cause desired transitions in the state space. Distributed embedded systems are typically used in early warning systems, distributed power management systems, traffic monitoring and control systems, defense command-and-control systems, and other emerging applications. These are all real-time embedded systems that have stringent reliability, safety, security, and performance requirements [10].

While relatively mature techniques exist for certifying hardware systems, including the use of “over-design” techniques for dealing with worst case situations, methods of rigorously certifying software systems are still being actively researched. A major problem for distributed embedded systems is that the bulk of the application logic is implemented in software, leading to extremely large and complex programs. Compounding this difficult situation further is the fact that these systems are typically long-lived systems that must be continually updated in the field to enhance their functionality. This further exacerbates the complexity of these systems and requires frequent expensive recertification.

Possible certification methods for embedded systems range from formal verification to statistical testing. Verification is especially effective for certifying logical properties, i.e., properties that either hold or do not hold. Logical properties are generally application-specific but also include more general properties such as the absence of deadlocks, absence of race conditions, assurance that exceptions will not be generated, etc. Verification is less effective for quantitative properties that require domain-specific analysis, such as the average performance, probability of security violation, degree of fault coverage, etc. For these properties, statistical testing and quantitative modeling and analysis techniques are more practical.

It is clear that neither formal verification nor quantitative analysis and testing are by themselves adequate for certifying complex embedded software sys-

---

*This research was supported in part by the National Science Foundation under Grant numbers CCR-9900922 and EIA-0103709 and by the Texas ATP under Grant No. 009741-0143-1999.
tems. These methods have different strengths and weaknesses and can be used to complement each other. One potentially useful approach is to decompose the specification into distinct aspects that can be independently certified using the method that is most effective for it. This approach is especially attractive if each aspect corresponds to a disjoint portion of the program since then a large verification or testing problem is reduced to a set of substantially simpler verification or testing problems. It also enables the use of verification or testing on a per aspect basis rather than using the same method across the entire program. One major hurdle, however, is the need to certify the overall system on the basis of the aspect properties [12, 21, 22, 23, 26]. This is not always possible, especially when there are hierarchical or circular dependencies among the aspects. In such cases, system certification still requires the assessment of the reliability of one complex monolithic program.

One way around this problem is to focus on architectures in which the aspects are relatively independent of each other, as in the shared repository architecture, pipes-and-filters architecture, and their variations [5]. However, complex embedded systems are typically comprised of multiple architectures. Hence, each system needs detailed individual analysis which has to be repeated after each upgrade to the system.

In this paper, we explore an alternative approach based on the use of application-oriented frameworks for implementing embedded systems. Each framework is designed to have a stable portion that provides scheduling and transport functionalities and mechanisms for coping with security threats and component failures. The framework supports "plug-in" aspects that can be added, upgraded, or removed dynamically without having to stop the system. Each plug-in aspect must be designed to be certifiable independently of other aspects or the framework. We show that it is possible to design such frameworks for embedded applications and derive expressions for determining the system reliability from the reliabilities of the framework and the aspects. The approach is illustrated using a detailed example involving distributed sensor networks.

The rest of this paper is organized as follows. Section 2 presents a model of a class of application-oriented frameworks that enable aspects to be dynamically plugged into the framework. Section 3 presents expressions for assessing the reliability of the system. Section 4 applies the approach to a distributed embedded sensor network system. Section 5 reviews the related work while Section 6 summarizes the paper and outlines some future research.

2 Model of "Plug-and-Play" Application-Oriented Frameworks

We consider a framework consisting of two sets of independent aspects and one composition component, \( \{ S, A, C \} \), where \( S \) denotes the set of fixed (static) framework aspects, \( A \) denotes the set of "plug-and-play" application aspects, and \( C \) denotes the composition component that ties everything together. The set \( S \) can contain aspects that address functional as well as nonfunctional requirements. These include user interface aspects, communication aspects, scheduling/security/fault-tolerance aspects, etc. Let \( S = \{ s_1, s_2, ..., s_n \} \), where \( s_1, s_2, ..., s_n \) represent the static framework aspects. A requirement imposed on these aspects is that each one should be independent of the other aspects. That is, aspect \( s_i \) does not invoke or depend on any other aspect \( s_j \) during its execution and its correctness can be certified independently of other aspects. However, an aspect may coordinate the execution of other aspects and its output can be visible to the composition component.

The set \( A = \{ a_1, a_2, ..., a_m \} \) contains application-specific plug-in aspects that can be dynamically added to or removed from the framework. Each aspect must be designed to operate independently given its inputs and it must also be possible to certify it independently of the framework or other aspects. Generally, these aspects implement specific features or functional aspects of the application.

Component \( C \) denotes the composition component. It receives the outputs of the aspects in \( A \) as well as some aspects in \( S \) and assembles the final output of the system. The composition can take several forms, some of which are listed in the following:

1. **Montage.** Here, the output of each component is evident in the final system output which forms a specified montage of the outputs of all the aspects. The aspects in this model may process fully overlapping, partially overlapping, or fully disjoint parts of the input set. Examples include HTTP and SIP (Session Initiation Protocol) message processing, GUI and visualization applications, and coordination systems.

2. **Selection.** Here, all the aspects process the same set of inputs and the output of one of the aspects is selected to be the output of the system. Various criteria can be used to select the appropriate output, such as the earliest response, the response
having the smallest or largest value, etc. Examples include pattern matching in image recognition and bidding systems in e-commerce applications.

3. Fusion. In this case, the composition component processes all the outputs and generates the final system output. The processing can involve set intersection and union operations, arithmetic and boolean operations, and information fusion techniques. Examples include deciding which actuator(s) to activate depending on parallel processing of different goals, composing features in a telephone switching system, determining which packet to transmit next based on fault tolerance and performance constraints, etc.

4. Nesting. This occurs in pipeline processing where an aspect has an inverse aspect, such as coder and decoder aspects, encryption and decryption routines, compression and decompression routines, etc. Examples include multimedia network communication applications.

5. Filtering. This also occurs in pipeline processing where each aspect processes the output of the previous stage and removes or adds some information while preserving other information. Examples in signal processing include noise filtering components, acoustic and line echo cancellation components, signal shaping, etc.

A given framework can incorporate a combination of the above composition methods. For example, consider a simple framework for audio communication. The framework consists of a slot for signal capture on the transmission side and a slot for signal playback on the receiving side. It allows a varying number of filtering components to be added at either end. Potential components include acoustic echo cancellation, line echo cancellation, noise filtering, and signal amplification components. Then there are complementary pairs of nesting components, such as codecs, compression/decompression components, encryption/decryption components. On the transmission side there is also a montage composition routine that assembles packets using the output of the pipeline stage as well as checksum computation (for fault tolerance), sequence number assignment (for sequencing), to/from address headers, and data descriptors (e.g., length and format). The sequence number and data to be transmitted are determined based on the fusion of several aspects, including the transmission window size, elapsed time, pending acknowledgments, etc. On the receiving side, a coordination component is used to parse the message and divide it into the checksum, sequence number, to/from address, and data portions. The checksum component checks whether any failures have occurred. The sequencing component checks whether the packet was already received. The acknowledgment component prepares and sends an acknowledgment if needed. If the packet passes the checks, it is then passed to the inverse of the nesting components, the filtering components, and, finally, to the playback component.

The fixed part of the framework schedules the components in order to meet the data rate requirements. It also provides the buffering and preparation of inputs and outputs for the plug-in components.

Consider a specific instance of the system containing (signal capture, noise filtering, echo cancellation, compression, encryption, transmission) on the sending side and (receiving component, decryption, decompression, amplitude enhancement, and playback) on the receiving side. The reliability of these aspects can be assessed independently:

1. Signal capture and playback. These can be tested and evaluated as one unit. The playback routine can be analyzed independently by using standard pre-recorded signal data but it is very difficult to test the signal capture module independently. Also, these aspects involve quantitative quality assertions, such as fidelity of the signal capture or playback, and, hence, are easier to test and analyze rather than to verify.

2. Noise filtering, echo cancellation, and amplitude enhancement aspects. These can be tested independently using standard test patterns or in conjunction with the playback component. Again, these components involve quantitative quality assertions that make them more amenable to testing and analysis.

3. Compression and decompression aspects. Assuming lossless compression, these components can be tested easily as one unit. A quality measure is the degree of compression achieved, which can be checked via testing and analysis.

4. Encryption and decryption aspects. These components can also be tested as one unit. A key quality factor is how difficult it is for a third party to decipher the data. This is difficult to test, since it is impossible to enumerate all possible strategies that a third party might use. However, the
5. Communication subsystem. This is a part of the framework but can be tested and evaluated independently. Issues, such as congestion and packet loss, need to be analyzed using standard performance analysis techniques.

Each of these aspects is simpler and more focused than the overall framework and its code can be evaluated independently. The user plug-in aspects can be added, removed, or updated dynamically at run-time — the framework must include coordination code to ensure that correct matching components are used on both the sender and receiver sides in spite of these dynamic updates. Another advantage is that by observing the behavior of the system, it is possible to trace defects to the individual components. For example, (a) excessive noise or echo can be traced to the noise reduction or echo suppression aspects, (b) excessive packet size may signal a problem with the compression routine, (c) poor security will require changes to the encryption/decryption routines, and (d) out of order packets implies a communication problem. This type of dynamic fault diagnosis is more difficult to achieve in monolithic systems where there are lots of inter-dependencies among the components.

3 Evaluation of Verification and Statistical Reliability Assessment

The reliability analysis will yield an expression containing a fixed part that models the impact of the reliability of the static aspects of the system, namely, the framework and the composition aspects, on the overall reliability of the system. The rest is a varying expression that models the impact of the reliability of application aspects on the system reliability. In this section we derive the latter expressions for the montage, selection, fusion, nesting, and filtering composition structures. The overall analysis is illustrated in the next section for a distributed sensor network.

3.1 Montage Composition

Let \( R(t) = \{ r_1(t), r_2(t), \ldots, r_n(t) \} \) represent the reliability vector for the application aspects. The montage composition makes the success of failure of each aspect directly visible to the user. Further, the success or failure of aspect \( a_i \) has no impact on the success or failure of other aspects \( a_j, j \neq i \). From the user’s perspective, the system does not necessarily fail the moment one aspect fails. Instead, the system may degrade in the sense that the quality of the output becomes lower and lower as more and more aspects fail. (Here, the “quality” of an aspect refers to the value provided to the user by the correct operation of that aspect. It is similar to a reward function. By definition, the quality of an aspect that has failed is 0.) Let \( Q = \{ q_1, q_2, \ldots, q_n \} \) represent the quality vector for the application aspects. Let \( C(Q) \) represent the composition expression for the system quality. Then, the average quality of the system output is \( C(QR(t)) \) where \( QR(t) = \{ q_1 \times r_1(t), q_2 \times r_2(t), \ldots, q_n \times r_n(t) \} \).

As an example, consider a system that controls the button/light display panel of an elevator. If each button/light is controlled by a separate aspect, then the quality and reliability of each button/light is distinct from those of the other buttons/lights. Hence, we have a montage composition with \( C(Q) = \sum_{i=1}^{n} q_i \). Hence, the average quality \( q(t) = \sum_{i=1}^{n} r_i(t) \times q_i \).

As another example, consider a multimedia transmission system that transmits two audio streams (left and right microphones) and two video streams (brightness and color frames). Assume that the color stream adds value to the display but the system works if we have at least one of the audio streams and the brightness stream. Then the montage composition expression is \( C(Q) = q_b \times (q_a + q_r \times (1 + q_c)) \), where “b” denotes brightness, “a” denotes left audio, “r” denotes right audio, and “c” denotes color. Hence, the average quality \( q(t) = q_b \times r(t) \times (q_a \times \eta_a(t) + q_r \times \eta_r(t)) \times (1 + q_c \times r.c(t)) \).

3.2 Selection Composition

In the selection composition method, at any given instant of time, one of the application aspects is selected. Let \( P = \{ p_1, p_2, \ldots, p_n \} \) be a vector such that \( p_i \) is the probability that aspect \( a_i \) is selected at any time \( t \). Then, the probability that the output of the system is correct for a random input, \( c_i \), is given by \( \sum_{i=1}^{n} c_i \times p_i \), where \( c_i \) is the probability that aspect \( a_i \) correct is a random input. The average quality of the system for a random input is \( \sum_{i=1}^{n} p_i \times c_i \times q_i \), where \( q_i \) is the average quality of the output of aspect \( a_i \).

As an example, consider a system that plans a trip for a user. There can be a wide range of options, including different methods of travel (airplane, automobile, railway, bus), lodging, travel dates, etc. Different aspects may target different transportation modes and query different service providers to achieve satisfactory solutions. The composition routine uses the cost of the trip as a key factor in selecting a solution. The quality of a solution is proportional to its cost. Different aspects can be plugged into the framework. The
reliability of the system increases if a reliable low-cost aspect is plugged into the framework and decreases if a low-cost aspect is unreliable.

The reliability of a selection composition can be improved substantially if the composition routine incorporates last moment (numerical) checks to determine the correctness of the output of each aspect. It then selects the final output from the set of aspects that have produced correct outputs. Since the system is correct as long as one aspect produces the correct output, the probability that the output is correct is

\[
1 - \prod_{i=1}^{n_A} (1 - r_i).
\]

The average quality of the output is

\[
[\sum_{i=1}^{n_A} p_i \times q_i] \times [1 - \prod_{i=1}^{n_A} (1 - c_i)].
\]

### 3.3 Fusion Composition

In fusion composition, the outputs of the different aspects are merged together to obtain the output of the system. Hence, even though the aspects are independent, a failure in one aspect can cause a failure of the whole system. Thus, the reliability of the system is given by

\[
\prod_{i=1}^{N_A} r_i(t),
\]

where \(r_i(t)\) is the reliability of aspect \(a_i\).

A natural question is why would one design a framework where plugged in more aspects will result in deterioration of the system reliability. The reason is that the overall quality of the system may improve even though its reliability may be lower. Assume that \(C(Q)\) is an expression showing the resulting quality of the output given the aspect quality vector \(Q = \{q_1, q_2, ..., q_{N_A}\}\). Then, the average quality of the system is

\[
C(Q) \times \prod_{i=1}^{n_A} r_i.
\]

As an example, consider a spelling correction system that uses multiple aspects to correct misspelled words. Possible aspects can include distance to nearest words in the dictionary, likely keyboard errors (e.g., juxtaposing two consecutive characters or typing in a character surrounding the correct character on the keyboard, vocalization errors (e.g., replacing “oo” by “u” or “i” by “y”), context checker that attempts to find a word that fits within the given context, etc. Assume that each aspect generates a list of words and associated confidence factors. The composition routine takes the outputs of the different aspects and selects the word that maximizes the confidence factor. A possible expression for \(C(Q)\) is

\[
\sum_{i=1}^{N_A} q_i \times \text{confidence factor that aspect } a_i \text{ has for the selected word (0 if the word is not in the list of aspect } a_i\).
\]

In some fusion composition frameworks, the aspects are prioritized such that the final output must satisfy aspects \(a_1\) through \(a_i\) before satisfying aspect \(a_{i+1}\), \(1 \leq i < n_A\). In these cases, simple comparators can be incorporated into the composition routine to ensure that fusion with aspect \(a_{i+1}\) will be performed only if it does not violate aspects \(a_1\) through \(a_i\). Then the probability that the final output satisfies aspect \(a_i\) is given by

\[
\prod_{j=1}^{i} c_j \text{ where } c_j \text{ is the probability that aspect } a_i \text{ works correctly for a random input. The quality of the output for a random input is } \sum_{i=1}^{n_A} q_i \prod_{j=1}^{n_A} c_j.
\]

### 3.4 Nesting Composition

This is a restricted version of the pipes-and-filters architecture where each aspect \(a_i\) consists of a pair of aspects \(a_{i1}\) and \(a_{i2}\) such that \(a_{i2}(a_{i1}(x)) = x\), i.e., \(a_{i2}\) is the inverse of \(a_{i1}\). Since any faulty aspect can corrupt the output, the reliability of this composition is

\[
\prod_{i=1}^{n_A} r_i(t).
\]

Each aspect can be viewed as adding to the quality of the system. Hence, the overall system quality is

\[
(\sum_{i=1}^{n_A} q_i) \times \prod_{i=1}^{n_A} r_i.
\]

Nesting composition occurs in communication systems. For example, as discussed in Section 2, for a streaming multimedia transmission system, aspects pairs can include (capture, playback), (compression, decompression), (encryption, decryption), and (disassemble, reassemble). Each aspect must work correctly for the output of the system to be correct. However, each aspect achieves specific objectives, such as reducing the bandwidth requirement, increasing the security, enhancing the fault tolerance, and so on.

Aspects in nesting composition are amenable to rigorous formal verification with respect to their compositional correctness, i.e., verification that \(\forall x, a_{i2}(a_{i1}(x)) = x\). Once this assertion has been verified for each aspect, the reliability of the system is 1.0 for all time \(t\). Formal verification can be replaced by run-time checking at some extra cost. For example, let \(x\) be the input to aspect \(a_{i1}\). The code for aspect \(a_{i1}\) can be augmented to apply aspect \(a_{i2}\) and to pass \(x\) to the next stage in case \(a_{i2}(a_{i1}(x)) \neq x\); otherwise, it passes \(a_{i1}(x)\) to the next stage. Though the reliability (correctness) is 1, the quality (such as bandwidth required, security achieved, etc.) is affected by the behaviors of the aspects; with checking, the average quality is given by

\[
\sum_{i=1}^{n_A} (q_i \times r_i).
\]

### 3.5 Filtering Composition

This is also a special case of the pipes-and-filters architecture where each stage performs some operation on the input stream and forwards the output to the next stage. All the aspects must work correctly for the system output to be correct. Hence, the reliability of the system is

\[
\prod_{i=1}^{n_A} r_i(t).
\]

The quality of the final output is additive if the composition preserves the effect of each aspect on the data stream. In this case, the average quality is given by

\[
(\sum_{i=1}^{n_A} q_i) \times \prod_{i=1}^{n_A} r_i \text{ where } q_i \text{ is the average quality of aspect } a_i.
\]
Filtering composition occurs in signal processing applications, text processing applications, and also certain types of process control applications. An example from signal processing is a telephone system containing aspects that cancel acoustic echo, cancel line echo, filter noise, and adjust the amplitude of the audio signal.

Aspects in filtering composition are domain-specific and, hence, require testing or domain-specific analysis for certification.

4 Analysis of a Framework for Distributed Sensor System

In this section, we illustrate the framework-based reliability assessment approach using a distributed embedded sensor network. A high level description of the system is as follows: Consider a system (this could be an engineered system or an environment) containing a set of sensors distributed across several nodes of a computer network. Each sensor monitors some attribute of the system, such as temperature, speed, voltage, etc., and sends the sensor reading to clients that have registered with it. The sensor values can be of several different types, including boolean (e.g., a sensor that detects motion), enumeration (e.g., the direction of wind: N, S, E, W, NE, NW, SE, SW), integer (e.g., number of people that have crossed a turnstile), and real (e.g., temperature, speed, etc.). The sampling rate of a sensor, namely the frequency with which it measures the environment and sends the sensor value to a client), can be controlled by sending a “set frequency” message to the sensor. A client registers with a sensor by sending a “register” message to the sensor and unregisters by sending an “unregister” message to it. Sensors can support different sampling frequencies for different concurrent clients.

The system enables a user to (a) identify a set of sensors of interest to him or her, (b) locate and connect to those sensors, and (c) customize the way the sensors readings are graphically displayed on the user’s terminal. The system consists of a framework that enables sensors to be dynamically plugged into the network. It installs the new sensor, makes it known to the name server in the network, and then listens for messages from clients that wish to register with the sensor. When such a request arrives, the client is added to a list. From then onward, the framework sends the sensor readings to the clients registered with that sensor. Thus, the plug-in sensor aspect only consists of the code that acquires data from the hardware sensor, formats it, and writes it to a shared repository object on the sensor machine. When an “unregister” message arrives, the framework removes the client from the list and will no longer send the sensor readings to it.

The user who wishes to view sensors, describes the sensor types that he or she is interested in and the framework displays a list of the sensors matching the description. Once the user has selected a set of sensors, the framework sends registration messages to the corresponding sensors. Also, the framework presents the user with a set of tools and menus that will enable the user to create a customized way of viewing the readings of the sensors. These are upgradeable and extensible “visual aspects” and include time plots, use of different colors, use of different geometric shapes or icons, use of sound, moving bars or other objects along different scales (vertical, horizontal, 2-D, along a curve, etc.), circular gauges, etc. The capability provided to the user enables grouping and placement of different sensor displays. Each “visual aspect” consists of two parts. The first part assists the user in correctly setting up a display object of the type implemented by this aspect. The second part graphically renders the sensor data received by the client in its region of the display.

Let \( R_F(t) \) denote the overall reliability of the framework excluding the sensor data acquisition aspect, the visual object customization aspect, and the visual object rendering aspect. Assume that a user is monitoring three temperature sensors, \( T_1, T_2, \) and \( T_3, \) two motion detector sensors, \( M_1 \) and \( M_2, \) two microphones, \( \mu_1 \) and \( \mu_2, \) and three video cameras, \( V_1, V_2, \) and \( V_3. \) Assume that (1) the temperature sensors are combined together using component \( C_1 \) and their average is displayed as a moving bar in display region \( D_1, \) (2) the motion detector sensors are combined together using component \( C_2 \) and the aggregate value is displayed as a color in display region \( D_2, \) (3) the microphones are monitored using the speakers along with a channel selection button in display region \( D_3 \) (the user can only listen to one microphone at a time), and (4) the outputs of the video cameras are monitored in display regions \( D_4, D_5, \) and \( D_6. \)

The overall display is a montage of 6 display objects. Hence, the system quality is

\[
\left( \sum_{i=1}^{6} q_i \times R_{D_i}(t) \right) \times R_F(t).
\]

Since \( D_1 \) is a fusion of \( T_1, T_2, \) and \( T_3, \)

\[
R_{D_1}(t) = P_{customization-D_1} \times \prod_{i=1}^{3} R_{acquire-T_i}(t) \times R_{composition-C_1}(t) \times R_{render-D_1}(t),
\]

Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE’02) 
1071-9458/02 $17.00 © 2002 IEEE
where \( P_{\text{customization} - D_1} \) is the probability that the customization aspect was correct for display region \( D_1 \), \( R_{\text{acquire} - T_i}(t) \) is the reliability of the plug-in aspect for temperature sensor \( T_i \), \( 1 \leq i \leq 3 \), \( R_{\text{composition} - C_i}(t) \) is the reliability of the composition routine that computes the average of \( T_1, T_2, \) and \( T_3 \), and \( R_{\text{render} - D_2}(t) \) is the reliability of the rendering aspect for display region \( D_1 \).

Similarly, the composition for region \( D_3 \) is of the fusion type, so

\[
R_{D_3}(t) = P_{\text{customization} - D_3} \times \left( \prod_{i=1}^{2} R_{\text{acquire} - M_i}(t) \times R_{\text{composition} - C_i}(t) \times R_{\text{render} - D_2}(t) \right).
\]

Since the composition for region \( D_3 \) is of the selection type,

\[
a R_{D_3}(t) = P_{\text{customization} - D_3} \times \left( \sum_{i=1}^{2} P_{\text{select} - \mu_i} \times R_{\text{acquire} - \mu_i}(t) \times R_{\text{render} - D_2}(t) \right),
\]

where \( P_{\text{select} - \mu_i} \) is the probability that the \( i \)-th microphone is selected. The video camera displays conform to the selection composition. Hence, for \( 4 \leq i \leq 6 \),

\[
R_{D_i}(t) = P_{\text{customization} - D_i} \times R_{\text{acquire} - V_{-3}}(t) \times R_{\text{render} - D_2}(t).
\]

The main benefit of the framework-based assessment approach is that the reliability of the system can be easily recomputed as aspects are modified or as new aspects are added to the system.

5 Related Work

Designing a software system as a framework that can support plug-in aspects is an effective way of simplifying the system and assuring high quality by making the specification of each aspect as well as the composition component more amenable to rigorous analysis. It also enables the framework to deal with generic application-independent issues, such as scheduling, buffering, communication, security fault tolerance, naming services, etc. The basis for plug-in aspects can be traced to extensive work in the area of requirements decomposition. One of the earliest works is reported in [28] where a requirements specification is decomposed into multiple views, each of which captures some behavior of the system. Each view is represented by a sequence diagram. This decomposition reduces the complexity of the system. However, two different views are not necessarily independent, e.g. they can interact via aliases in order to react in a compatible way to a given input. The concept of multiple views has also been used in Statecharts [14], Objectcharts [8], and other related methods. It has also been applied to existing languages, e.g. Z [16]. The primary motivation for these views (achieved by grouping multiple states into one super state) is to reduce the complexity of the underlying Finite State Machine specification of the system. Again, interactions between machines (e.g., via synchronous events) can introduce dependencies between different machines. RSML [20] is a significant extension to Statecharts with the goal of achieving more easily understandable and reviewable specifications. It also has a more intuitive step semantics, but the objective is to assure analyzability of complex specifications rather than to identify plug-in software aspects.

Decomposition methods that persist over the life-cycle include separation using rely-guarantee assertions [19], behavioral inheritance [1], Independently-Developable End-user Assessable (IDEAL) components [5], and Aspect-Oriented Programming [18]. These methods result in distinct pieces of code that can be analyzed separately and are then formally composed together to form the system. The rely-guarantee based approach achieves separation between different components by using a common interface language between two components with a precise specification of rely and guarantee conditions [17] for each separate component. However, components are not required to be observable by the end-user who may not even be aware of some interfaces, especially interfaces with inner components. Behavioral inheritance is an elegant way of separating out synchronization concerns from functional concerns in object-oriented languages [1]. The approach proposed in [1] uses multiple inheritance, by inheriting one functional component and one behavioral component. It satisfies independent assessability but does not guarantee an implementation-invariant state space (so the system properties cannot be inferred from the component properties). In the IDEAL systems approach [5], separate aspects are used to ensure the invariance of the components. In addition, this approach enables the decomposition of a behavioral component into more than one component (e.g., by having separate components for assuring mutual exclusion, priority, FIFO access, enforcement of precedence, etc.). Aspect-Oriented Programming (AOP) is a more recent technique [18] that strives for separation of concerns in implementing object-oriented programs. Features that can be used for
more than one object, such as error detection, exception handling, and synchronization code, are separated from the main functionality of the objects. The code for these features are written once along with identification of the objects that will need the code and the positions/situations that will activate the code. Then, a preprocessor is used to "weave" the code for the features with the code for the objects. There is substantial overlap between the philosophy of Aspect Oriented Programming and framework plug-in aspects. However, there are also some important differences. For example, framework plug-in aspects can be executed as separate processes while aspects in AOP have to be statically "woven" together to form the program. Dynamic composition allows each process to be evaluated separately using model checking [15] and operational profile testing [24]. Also, each aspect can be made fault-tolerant more cost-effectively using design diversity [11], exception handling, and other methods.

In summary, a lot of work has been done in decomposing specifications into multiple views. The key feature of the approach discussed in this paper is that the framework and composition method can be used to infer the system reliability from the aspect reliabilities.

A problem with decomposition of a specification into simpler components is how to compose the components to obtain a system with assessable properties. One difficult problem is how to assure the consistency of the different views [2]. Nonmonotonic logic [6], especially paraconsistent nonmonotonic logic [7], provides some support, but it cannot handle all types of inconsistencies. Formal techniques have been developed to tag inconsistent specifications and remove them either manually or by using rule-based methods [13]. This difficulty is compounded when different specification methods are used to specify different views [29]. Such multiparadigm methods can result in simpler specifications by allowing the use of notations that enable easy expression of specific aspects, e.g., Z specification for abstract data types and Statecharts for reactive components. Inconsistencies are resolved by automatically translating all the specifications into a uniform framework, typically a first order predicate calculus specification. These approaches do not address execution-time concerns, such as striving for end-user assessable components or ensuring that the reliability of the system can be inferred from the reliability of its components.

In the plug-in approach, the system requirements specification is decomposed based on conjunctive and disjunctive connectives and directly mapped to simple composition operations, including fusion, nesting, selection, etc. Each aspect can be implemented and evaluated independently and the different aspects can be composed dynamically by the framework. Each aspect is directly assessable at the system level and can be traced back to the requirements specification. This property facilitates fault-confinement and isolation. Also, inconsistencies can be detected during composition methods, for example, set in intersection in fusion composition, and can be resolved by assigning priorities to components. Another nice feature is that aspect reliability estimates can be statistically combined to obtain the system reliability. Further, each aspect has relatively few states and transitions, which makes it feasible to develop highly reliable components and assess their reliability to a high degree of confidence.

6 Summary

In this paper, we have presented and evaluated a framework-based approach for decomposing and implementing distributed embedded systems that (a) enable the system reliability to be rigorously inferred from the aspect reliabilities based on the composition method, (b) allow the aspects to be upgraded or removed, and (c) allow new aspects to be added dynamically. In both the latter two cases, the reliability of the system can be inferred without having to test the entire system again.

The approach uses orthogonal decomposition methods to partition a complex embedded system into a fixed part, consisting of the framework and composition components, and slots in which application aspects can be added dynamically as needed. Since each aspect is independent of other aspects and is substantially simpler than the whole system, the verification and analysis of each aspect is much simpler than attempting to verify the entire system as one single monolithic entity. Further, it is possible to selectively use verification or analysis testing depending on the characteristics of each aspect in order to maximize the confidence in the correctness of the system.

The paper illustrated, at an abstract level, the framework-based approach for several classes of applications. It also applied the method to a relatively complex distributed embedded sensor system. There are several interesting future research directions for this type of framework-based embedded system design. We have identified five distinct types of composition methods that are of practical use for plug-in aspects, namely, montage, selection, fusion, nesting, and filtering compositions. It is interesting to investigate
the possibility of other composition methods that support plug-in application aspects. Other research issues are related to the decomposability of a given specification into finer aspects and the automated identification of such decompositions.

7 Acknowledgment

The authors wish to thank the three anonymous reviewers for their detailed constructive comments that have greatly improved the quality of the paper.

References


