Systematic Reliability Analysis of a Class of Application-Specific Embedded Software Frameworks

Sung Kim, Farokh B. Bastani, *Member*, *IEEE*, I-Ling Yen, *Member*, *IEEE*, and Ing-Ray Chen, *Member*, *IEEE*

Abstract—Dramatic advances in computer and communication technologies have made it economically feasible to extend the use of embedded computer systems to more and more critical applications. At the same time, these embedded computer systems are becoming more complex and distributed. As the bulk of the complex application-specific logic of these systems is realized by software, the need for certifying software systems has grown substantially. While relatively mature techniques exist for certifying hardware systems, methods of rigorously certifying software systems are still being actively researched. Possible certification methods for embedded software systems range from formal verification to statistical testing. 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. Even though substantial research has been carried out to reduce the complexity of the software system through decomposition, one major hurdle is the need to certify the overall system on the basis of the aspect properties. One way to address this issue is to focus on architectures in which the aspects are relatively independent of each other. However, complex embedded systems are typically comprised of multiple architectures. In this paper, we present an alternative approach based on the use of application-oriented frameworks for implementing embedded systems. 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 method is illustrated using a distributed multimedia collaboration system.

Index Terms—Distributed embedded systems, software composition, application-oriented frameworks, software reliability assessment.

1 Introduction

Dramatic advances in computer and communication technologies have made it economically feasible to extend the use of embedded computer systems to more and more critical applications, such as process-control systems, manufacturing systems, avionics, railway systems, etc. At the same time, these embedded computer systems are becoming more complex and distributed. 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. For these real-time critical applications, it is necessary to be able to rigorously certify the quality (reliability, safety, performance) of the system.

While relatively mature techniques exist for certifying hardware systems, including the use of "overdesign" techniques for dealing with worst-case situations, methods of rigorously certifying software systems are still being actively researched. In addition, most of the hardware system failures are caused by some random process (i.e., heat, physical wear and tear, etc.) in the normal course of hardware usage whereas software failures are due to design flaws. Therefore, certifying a software system becomes a very difficult task. Furthermore, as the bulk of the complex system application logic is realized by software, one must be able to deal with extremely large and complex programs. Compounding this difficult situation even 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 software systems range from formal verification to statistical testing. Verification is especially effective for certifying logical properties of the system, 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,

- *S. Kim, F.B. Bastani, and I.-L. Yen are with the Computer Science Department, University of Texas at Dallas, MS EC-31, Box 830688, Richardson, TX 75038-0688. E-mail: {sungkim, bastani, ilyen}@utdallas.edu.*
- *I.-R. Chen is with the Department of Computer Science, Virginia Polytechnic Institute and State University, 7054 Haycock Road, Falls Church, VA 22043. E-mail: irchen@cs.vt.edu.*

Manuscript received 2 Sept. 2003; revised 4 Feb. 2004; accepted 4 Feb. 2004. Recommended for acceptance by J. Bechta Dugan.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0132-0903.

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 systems. 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 of the earliest works related to software decomposition is [33] where a requirements specification is decomposed into multiple views, each of which captures some behavior of the system. The concept of multiple views has also been used in StateCharts [18], separation using rely-guarantee assertions [24], behavioral inheritance [2], and Aspect-Oriented Programming [23]. One major hurdle, however, is the need to certify the overall system on the basis of the aspect properties [15], [26], [27], [28], [31]. Even though these decompositions reduce the complexity of the system, two different views are not necessarily independent. This complicates the reliability analysis. For example, if the reliabilities of components f and g are 1.0 and 0.9999, respectively, then the reliability of a system consisting of f and g can range from 1.0 to 0.0 depending on how f and g interact. This uncertainty means that considerable effort has to be expended to reason about global properties of the system, rather than by simple deductions from the component properties, 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 [7]. 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. The framework is classified according to how different classes of applications are composed to make up the framework. The class includes applications that can be decomposed into Independently-Developable End-user Assessable Logical (IDEAL) aspects. Hence, 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 multimedia collaboration.

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. Sections 3, 4, and 5 present the reliability assessments of systems that are composed from different classes of aspects. Section 6 applies the approach to a case study of a distributed multimedia collaboration system. Section 7 reviews the related work while Section 8 summarizes the paper and outlines some future research directions.

2 System Model

We consider a framework consisting of two sets of independent aspects and one composition component, $\{\mathbf{S}, \mathbf{A}, C\}$, where $\mathbf{S}$ denotes the set of fixed (static) framework aspects, $\mathbf{A}$ denotes the set of "plug-and-play" application aspects, and C denotes the composition component that ties everything together. The set $\mathbf{S}$ can contain aspects that address functional as well as nonfunctional requirements. These include user interface aspects, communication aspects, scheduling/security/fault-tolerance aspects, etc. Essentially, the static framework aspects provide "services" that may enhance the framework's computing platform. Let $\mathbf{S} = \{s_1, s_2, \ldots, s_{n_S}\}$, where s_1, s_2, $\ldots$, s_{n_S} 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 $\mathbf{A} = \{a_1, a_2, \ldots, a_{n_A}\}$ 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.

Whether it is a static framework aspect or an application specific plug-in aspect, an aspect can be separated from other aspects depending on how it accesses the shared state space. The notion of "access" here includes read and write operations (i.e., an operation that affects the shared state space). Aspects can affect different regions of the shared state space or may affect at least one common point in the shared state space. For the cases where there is a clear separation in the regions they affect, the separation may occur statically (i.e., deterministically) or dynamically. If the accesses can be known statically (i.e., at compile-time), the separation is denoted as compile-time separation; however, if the accesses are dynamic in nature (i.e., the access point in the state space is determined at runtime), the separation is denoted as runtime separation. In both cases, aspects can affect the shared state space at the same time or at different times. Fig. 1 shows the classification of such separation.

- Compile-Time Spatially Separable.
- Compile-Time Temporally Separable.

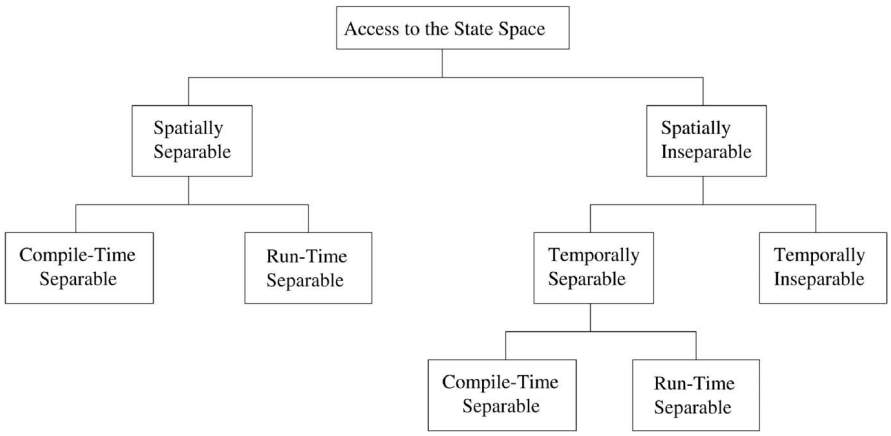

Fig. 1. Classification of accesses to a shared state space.

- Runtime Spatially Separable.
- Runtime Temporally Separable.
- Spatially Inseparable and Temporally Inseparable.

A given framework can incorporate a combination of the above aspects. For example, consider a simple framework for a VoIP communication management system. The framework consists of a slot for data capture on the transmission side and a slot for data presentation on the receiving side. It allows a varying number of components to be added at either end. Potential aspects include error handler, event reporting, time manager, call record keeper, resource checker as well as complementary pairs of components on each side, such as compression/decompression and encryption/decryption components. On the transmission side, there is also a component for data integrity management (i.e., connection management) that assembles packets using the output of the pipeline stage as well as checksum computation, sequence number assignment, 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 similar component (i.e., connection management) 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 inverses of the components that are responsible for decryption and decompression as well as appropriate handler components and, finally, to the presentation component. Fig. 2 shows the UML package diagram of a simple framework for VoIP communication.

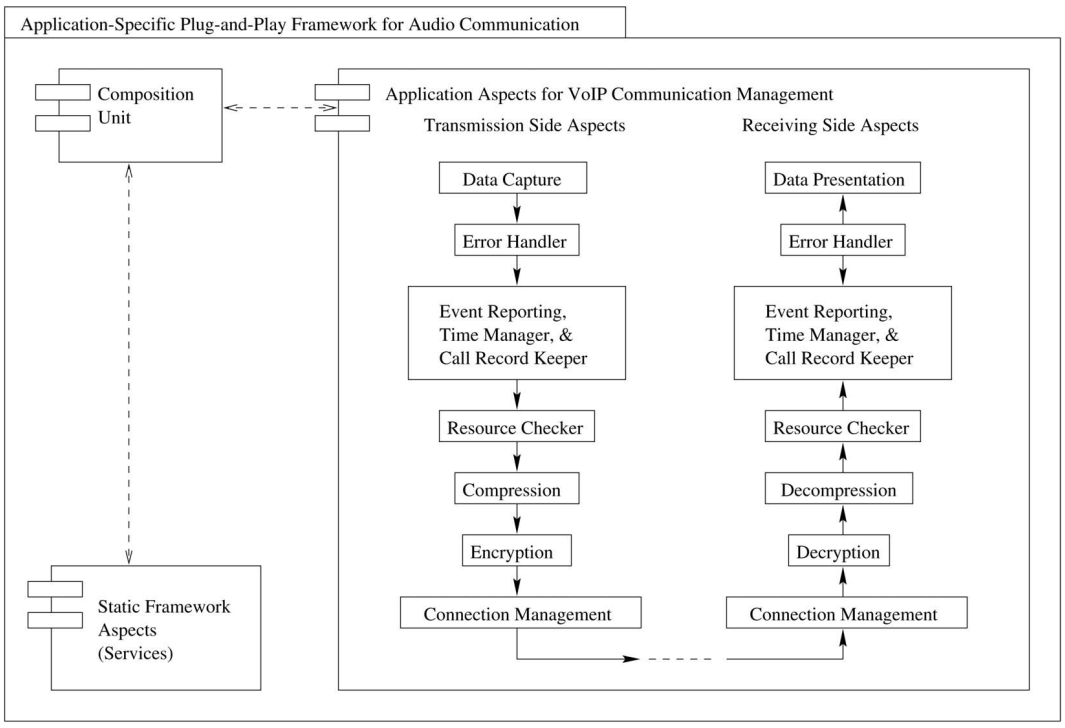

Fig. 2. Specification of application-specific plug-and-play framework using UML package diagram.

The components in the above VoIP communication management framework can be classified into different aspects as mentioned earlier. For example, the error handler, the various input handlers (i.e., Event Reporting, Time Manager, and Call Record Keeper), and the resource checker are compile-time temporally separable aspects as they can be positioned in a pipes-and-filters fashion at compile time. In addition, the paired components are also compile-time temporally separable aspects for the same reason. The only difference is that the former is the property preserving aspects and the latter is the property restoring type. However, the various input handlers are compile-time spatially separable aspects as each looks at different regions of the data passed. Therefore, Event Reporting, Time Manager, and Call Record Keeper can execute in parallel without affecting each other.

Considering the above example of the VoIP communication management system containing the set of aspects data capture, error handler, event reporting, time manager, call record keeper, resource checker, compression, encryption, transmission on the sending side, and receiving component, decryption, decompression, resource checker, call record keeper, time manager, event reporting, error handler, data presentation on the receiving side, the reliability of these aspects can be assessed independently:

1. *Data capture and presentation.* These aspects can be tested and evaluated as one unit. The presentation routine can be analyzed independently by using standard preassembled data, but it is very difficult to test the data capture module independently. Also, these aspects involve quantitative quality assertions, such as fidelity of the data capture or presentation. Hence, data capture and presentation are easier to test and analyze as one unit.
2. *Error handler, event reporting, time manager, call record keeper, and resource checker aspects.* These can be tested independently using standard test patterns or in conjunction with the presentation 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 protection provided by the encryption can be analyzed in general terms.
5. *Communication management.* 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.

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 fixed part of the framework provides the framework services to meet other nonfunctional requirements for the application. For example, hardware failures can be dealt with by using a *fault-tolerance service*, coordination of multiple accesses to the system resources is provided by the *concurrency control service*, finite memory constraints are addressed by the *garbage collection service*, etc.

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 runtime—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,

1. abnormal input error handling and ignored events can be traced to the error handling and event reporting aspects,
2. abnormally large packet size may signal a problem with the compression routine,
3. poor security will require changes to the encryption/decryption routines, and
4. out of order packets implies a connection management problem.

This type of dynamic fault diagnosis is more difficult to achieve in monolithic systems where there can be many interdependencies among the components.

3 Reliability Assessment of Compile-Time Separable Aspects

3.1 Compile-Time Spatially Separable Systems

As mentioned above, the event reporting, time manager, and call record keeper aspects of the VoIP communication management system are compile-time spatially separable aspects (i.e., they affect different regions of the state space). In particular, two aspects, A and B, of the system, R, are compile-time spatially separable if the system postcondition, $PC(S)$, can be decomposed into $P(S_0, S_1) \wedge Q(S_0, S_2)$, where $S_1 \cap S_2 = \phi$, $S_0 \times S_1 \times S_2 = S$, and neither P nor Q modifies S_0. If A implements P and B implements Q, then A and B together implement R. A and B have noninterfering failure aspects if the system is designed to guarantee that it is physically impossible for A to update S_2 and for B to update S_1. It is also necessary in this as well as the other cases to ensure that one aspect does not "walk" over another aspect, i.e., update the code or data of another aspect. For example, the time manager aspect records the call time, and the call record keeper simply notes any significant call related events for future billing purposes; thus, neither of these aspects updates the code or data of the other aspect. Under these conditions, we have the following lemma.

Lemma 1. *For independent aspects A and B—i.e., they access and modify disjoint regions in the state space (i.e., $S_0 = \phi$)—the probability of the correct outcome produced by A and B is the product of the probability that A produces the correct output and the probability that B produces the correct output.*

Proof. For aspects A and B, assume that $S_0 = \phi$ and let $S_1 = \{a_1, a_2, \cdots, a_n\}$ and $S_2 = \{b_1, b_2, \cdots, b_m\}$ denote the state spaces of A and B, respectivley. Let $S_1' = \{a_1', a_2', \cdots, a_j'\}$ and $S_2' = \{b_1', b_2', \cdots, b_k'\}$ be the sets of points for which A and B are correct, respectivley. Then, the probability of the

correct outcome produced by A and $B = (|S_1'| \times |S_2'|)/(n \times m)$ since A and B are both correct for the combination of any a_h' where $1 \leq h \leq j$ and any b_i' where $1 \leq i \leq k$. Then, the probability of the correct outcome produced by A and B equals the probability that A produces the correct output $\times$ the probability that B produces the correct output since the probability that A produces the correct output equal $|S_1'|/n$ and the probability that B produces the correct output equal $|S_2'|/m$. □

Lemma 2. *For compile-time spatially separable aspects A and B of R, Reliability of R equals reliability of $A \times$ Reliability of B if A and B are independent (i.e., $S_0 = \phi$).*

Proof. Since A and B are independent components, i.e., $S_0 = \phi$, therefore, the system reliability, $Reliability_{System}(t) = probability$ (system performs correctly in $[0,t]||$ system is up at time 0) $= probability$ ($P(S_0, S_1) \wedge Q(S_0, S_2)$ holds for every input satisfying $precondition()$ of system in $[0,t]||$ system is up at time 0). Since $probability\,(A \wedge B|C) = probability\,(A|C) \times probability\,(B|C)$ when A and B are independent and using Lemma 1, we then have $R_{System}(t) =$ $probability$ ($P(S_0, S_1)$ holds for every input satisfying $precondition()$ of system in $[0,t]||$ system is up at time 0) $\times$ $probability$ ($Q(S_0, S_2)$ holds for every input satisfying $precondition()$ of system in $[0,t]||$ system is up at time 0) $= R_A(t) \times R_B(t)$. □

Lemma 3. *For compile-time spatially separable aspects A and B of R, the reliability of R is bounded by the following expression:*

$$max\{0, Reliability_A + Reliability_B - 1\} \leq Reliability_{System} \leq min\{Reliability_A, Reliability_B\}$$

if the input spaces of A and B are not disjoint nor decomposable.

Proof. Input space is shared by both A and B and we do not know exactly how the input is shared and accessed. Let n be the total number of discrete points in the input space. Then, if $Reliability_A$ is the reliability of A, $n \cdot Reliability_A$ is the number of inputs for which A is correct, and $(1 - Reliability_A)n$ is the number of inputs for which A fails. Likewise, $n \cdot Reliability_B$ is the number of inputs for which B is correct, and $(1 - Reliability_B)n$ is the number of inputs for which B fails. The maximum reliability of the system is obtained when the system works correctly for the maximum number of inputs. Since the system works correctly if and only if both A and B work correctly, the maximum reliability is generated when there is the largest overlap of the inputs on which both A and B work correctly. With complete overlap, the number of inputs for which the system works correctly is min (maximum number of inputs for which A works correctly, maximum number of inputs for which B works correctly). Therefore, the maximum reliability of the system is $min(n \cdot Reliability_A/n, n \cdot Reliability_B/n)$. On the other hand, the minimum reliability of the system is obtained when there is the smallest overlap of the inputs on which both A and B work correctly. In the case where there is minimal overlap, the number of inputs for which the system works correctly is max(0, total number of inputs—(number of inputs for which A fails + number of inputs for which B fails)). Therefore, the minimum reliability of the system is $max(0, (n - (1 - Reliability_A) n - (1 - Reliability_B)n)/n)$ which is $max(0, Reliability_A + Reliability_B - 1)$. □

As seen in Lemmas 2 and 3, the input space for the aspects of the system may be overlapping or disjoint. Let us consider the same example mentioned above where two aspects A and B of the system access and modify the state spaces (S_0, S_1) and (S_0, S_2), respectively, where $S_1 \cap S_2 = \phi$, $S_0 \times S_1 \times S_2 = S$, state space for the system, and neither A nor B modifies S_0. Disjoint input space (i.e., $S_0 = \phi$) causes no complications; however, in the case where the input space is overlapping (i.e., $S_0 \neq \phi$), we do not know exactly what each aspect would do with a random input (i.e., fail or succeed) since aspects can be developed by different people at different times. Therefore, we can only assume that there is no correlated input failures among aspects for the given input. In addition, it is not possible in practice to know the exact behavior of each aspect since it is very difficult to tabulate the input-output relationships for infinite input space for each aspect. With these assumptions, we can generally say that the reliability of the system is the product of the reliabilities of the aspects when aspects are independently tested with sample points from the input space and we have the average reliabilities of the aspects.

3.2 Montage Composition

In general, the output of each aspect influences the correctness of the overall output in the system that is decomposed into compile-time spatially separable aspects. This is especially true for a serial composition of compile-time spatially separable aspects. However, users may be satisfied with a partial output where the system output is composed of outputs from a subset of the aspects. This is a special case where the compile-time spatially separable aspects can exist simultaneously. In such a situation, the output of individual aspects are composed into the system output *via* the *montage composition*.

In montage composition, 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 (Hypertext Transfer Protocol) and SIP (Session Initiation Protocol) message processing, GUI (Graphical User Interface), and visualization applications. For example, consider a Web page composed of different text, images, links, and mpeg clips. The Web page is a montage of these different components. Each output of these different components of the page is evident in the overall display of the page.

Let $\mathbf{R}(t) = \{r_1(t), r_2(t), \ldots, r_{n_A}(t)\}$ represent the reliability vector for the aspects. The montage composition makes the success or 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 $\mathbf{Q} = \{q_1, q_2, \ldots, q_{n_A}\}$ represent the quality vector for the aspects. Let $C(\mathbf{Q})$ represent the composition expression for the system quality. Then, the average quality of the system output is $C(\mathbf{QR}(t))$, where $\mathbf{QR}(t) = \{q_1 \times r_1(t), q_2 \times r_2(t), \ldots, q_{n_A} \times r_{n_A}(t)\}$.

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 $C(\mathbf{Q}) = \sum_{i=1}^{n_A} q_i$. Hence, the average quality $q(t) = \sum_{i=1}^{n} r_i(t) \times q_i$.

As another example, let us expand our VoIP system a bit further. In addition to the voice, the system can now offer video display. Assume that the video display consists of two video streams (brightness and color frames). Also assume that the color stream adds value to the display in addition to the brightness, but the system works if we have at least the audio stream and the brightness stream. The added color capability is a "nice to have" feature, and users are partially satisfied with just the voice and the brightness stream. Then, the montage composition expression is $C(\mathbf{Q}) = q_b \times q_a \times (1 + q_c)$, where "$b$" denotes brightness, "$a$" denotes audio, and "$c$" denotes color. Hence, the average quality $q(t) = q_b \times r_b(t) \times q_a \times r_a(t) \times (1 + q_c \times r_c(t))$.

3.3 Compile-Time Temporally Separable Systems

The case considered in the previous section is conceptually simple. In every cycle, each aspect looks at its portion of the state space, performs its computation, and updates its portion of the state space. Thus, there is no direct functional dependency between the aspects. The class of compile-time temporally separable systems is a richer class than compile-time spatially separable systems. Since actions can be separated in time, it is now possible to have functional dependency among the aspects. However, the key property of this class is that temporal separation is achieved at compile time which limits the architecture to a subset of the pipes-and-filters architecture. That is, aspect A processes the input that is then forwarded to aspect B for further processing. (We denote this by $A \rightarrow B$.) In the VoIP communication management framework, the error handler, the various input handlers, resource checker, compression, encryption, and connection management aspects are compile-time temporally separable. For example, the output of the compression aspect (i.e., compressed data) is passed down to the encryption aspect for further processing. The error handler may also "filter out" the captured data from any further processing. In the following sections, we identify two subclasses for which the system reliability can be inferred from the aspect reliabilities.

3.3.1 Visible Intermediate Result

In this case, it is assumed that the output of A is visible to the end-user and A is functional; thus, the specifications for A and B can be derived directly from the requirements specification. This is the case with compression and encryption aspects. The output of the compression aspect (i.e., compressed data) is visible to the end-user and the compression aspect is functional. To estimate the reliability of $A \rightarrow B$ from the individual reliabilities of A and B, it is necessary to be able to accurately determine the operational profile of the input to B. This requires the specification of A to define a function (rather than a relation). Under the condition that the specification of A is functional, we have:

Lemma 4. *The reliability of $A \rightarrow B$ is equal to the reliability of $A \times$ the reliability of B if A is functional.*

Proof. Since the operational profile of A, OP_A, is known and A is functional, then OP_B can be generated from OP_A and the function F that is A. That is, $OP_B = F(OP_A)$. Given the state space S, A is correct if $precondition_A \Rightarrow wp(A, postcondition_A)$, and B is correct if $precondition_B \Rightarrow wp(B, postcondition_B)$ and $postcondition_A \Rightarrow precondition_B$. Then, the correctness of A and B can be estimated by sampling OP_A and OP_B and comparing the generated output with $postcondition_A$ and $postcondition_B$, respectively. From this observation, we can conclude that we are dealing with a system composed of two serial and independent aspects (A and B). Therefore, the reliability of the system (i.e., $A \rightarrow B$) = probability(A is correct in [0, t] $\wedge$ B is correct in [0, t]) = probability(A is correct in [0, t]) $\times$ probability(B is correct in [0, t]) = the reliability of $A \times$ the reliability of B. □

The restriction that A should be a functional component can be removed by having filters that establish some specific property and, at the same time, preserve all other properties. We call such filters **Property Preserving Filters**. Suppose $P(S)$ is the precondition of the system and $R(S'')$ is the postcondition. Further, suppose that $R(S'')$ can be decomposed into two parts, namely, $R_1(S'') \wedge R_2(S'')$. Let A establish R_1 while B preserves R_1 and establishes R_2. For verification purposes, assume that the postcondition of A is $Q(S') \wedge R_1(S')$ which is also the precondition of B.

To assess the reliability of $A \rightarrow B$ from the reliabilities of A and B, it is necessary to know the operational profile of B. Since A establishes a partial result, it is not practical to require the specification of A to be functional. Instead, we require the implementation of A to be relational, that is, A generates all possible values that satisfy its requirements. This effectively makes the input to B invariant of the actual implementation of A and, hence, the operational profile of B can be determined from the specification of A. Under this condition, we have:

Lemma 5. *The reliability of $A \rightarrow B$ is equal to the reliability of $A \times$ the reliability of B if A is relational.*

Proof. The proof is similar to the proof of Lemma 4. The difference is that A is relational instead of functional. However, A generates all possible values that satisfy its requirements. This effectively makes A "functional" since, for a fixed input, different correct implementations of A must produce the same output set. Therefore, B's operational profile can be determined from the specification of A since the input to B is invariant of the actual implementation of A. That is, given I_A (the actual input space for A), A generates a set of possible outputs ($A : I_A \rightarrow 2^{I_B}$) that will be passed on to B. The input selector for B, F_B, generates the actual input for B from a set of possible inputs for B ($F_B : 2^{I_B} \rightarrow I_B$). Then, $F_B \circ A : I_A \rightarrow I_B$. Therefore, we can generate B's operational profile from the specification $Spec$ of A and the specification of F_B. Then, given the state space S, A is

correct if $precondition_A \Rightarrow wp(A, postcondition_A)$, and B is correct if $precondition_B \Rightarrow wp(B, postcondition_B)$. Then, the correctness of A and B can be estimated by sampling OP_A and OP_B and comparing the generated output with $postcondition_A$ and $postcondition_B$, respectively. Hence, we can conclude that we are dealing with a system composed of two serial and independent aspects (A and B). Therefore, the reliability of the system (i.e., $A \rightarrow B$) = the reliability of $A \times$ the reliability of B. □

All the aspects in property-preserving filters 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 $(\sum_{i=1}^{n_A} q_i) \times \prod_{i=1}^{n_A} r_i(t)$, where q_i is the average quality of aspect a_i.

Filtering composition, whether it is composed of property-preserving or property restoring filters, occurs in signal processing applications, text processing applications, and also certain types of process control applications. In the VoIP communication management system, the error handler "filters out" abnormal data from any further processing. Aspects in filtering composition are domain-specific and, hence, require testing or domain-specific analysis for certification.

3.3.2 Nonvisible Intermediate Result

In this case, each aspect is implemented by a pair of filters, one (say, A) that satisfies a specific nonfunctional requirement and another (say, A') that restores the information, i.e., $A'(A(I)) = I$. We call such filters **Property Restoring Filters**. The verification of the system is similar to the methods in the previous case for the functional aspect of the requirements. This essentially means verifying that $A'(A(I)) = I$. However, nonfunctional aspects, such as performance and security properties of the system, can be more difficult to verify and are impossible to do so if the properties are achieved only statistically. Statistical reliability assessment works well for this type of system. First, since the inverse of A must exist, the specification of A must be functional. Hence, the operational profile at the output of A can be determined accurately from the operational profile at the input of A and the specification of A. Second, the test oracle is obvious if A and A' are tested as one unit. This substantially reduces the cost of testing. Third, the nonfunctional aspects can be validated statistically as part of the reliability assessment procedure. Hence, we have:

Lemma 6. *The reliability of $A \rightarrow B$ = reliability of $A \times$ reliability of B.*

Proof. (The proof follows the same line as Lemma 4 and Lemma 5.) Since the inverse of A must exist, the specification of A must be functional. Therefore, OP_A and $Spec_A$ can be used to determine OP_B much like in the proof of Lemma 5. Then, given the state space S, A is correct if $precondition_A \Rightarrow wp(A, postcondition_A)$, and B is correct if $precondition_B \Rightarrow wp(B, postcondition_B)$. Then, the correctness of A and B can be estimated by sampling OP_A and OP_B and comparing the generated outputs with $postcondition_A$ and $postcondition_B$, respectively. Hence, we can conclude that we are dealing with a system composed of two serial and independent aspects (A and B). Therefore, the reliability of the system (i.e., $A \rightarrow B$) = the reliability of $A \times$ the reliability of B. □

Since any faulty aspect in property-restoring filters 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 $(\sum_{i=1}^{n_A} q_i) \times \prod_{i=1}^{n_A} r_i(t)$.

Property restoring filters occur in communication systems. For example, as discussed in Section 2, for a VoIP communication management system, aspects pairs can include (capture, presentation), (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 property restoring filters 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 runtime 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 $\sum_{i=1}^{n_A}(q_i \times r_i(t))$.

4 Reliability Assessment of Runtime Separable Aspects

4.1 Runtime Spatially Separable Systems

A typical example of this type of system is a delegation design pattern where a coordinator monitors events or inputs and selects an appropriate handler for each event or portions of the state space. For a two aspect runtime spatially separable system, the coordinator C scans the state space S and decomposes it into orthogonal sets S_0, S_1, and S_2. It then invokes $A(S_0, S_1)$ and $B(S_0, S_2)$ that can operate in parallel. The outputs of A and B are passed to modify S_1 and S_2, respectively. The verification of the system is done by verifying that the coordinator identifies S_0, S_1, S_2 correctly and verifying that A and B are correct. Statistical reliability assessment for these systems is complicated by the fact that the operational profiles induced at A and B must be accurately estimated. This can be done independently of C only if the specification of C is functional; otherwise, the internal logic of the moderator has to be analyzed in order to determine the correct operational profile, which makes the reliability estimates of A and B dependent on the implementation of C. Once the operational profiles have been estimated, the reliability of the system is assessed by first determining the reliabilities of A, B, and C separately. The reliability of the system is equal to the product of the reliabilities of C, A, and B provided that

1. the decomposition specification is derived from the requirements specification,

2. a defect in one aspect cannot modify the code or data of another aspect,
3. A and B only modify their own portions of the state space containing S_1 and S_2, respectively, and
4. A, B, and C operate in parallel.

4.1.1 Selection Composition

Selection composition falls into the runtime spatially separable category. In the selection composition method, at any given instant of time, one of the aspects is selected. 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.

Let $\mathbf{P} = \{p_1, p_2, \ldots, p_{N_A}\}$ be a vector such that p_i is the probability that aspect a_i is selected at any time t. Then, the system reliability is given by $\sum_{i=1}^{n_A} r_i(t) \times p_i$, where $r_i(t)$ is the reliability for the aspect a_i for a random input. The average quality of the system for a random input is $\sum_{i=1}^{n_A} p_i \times r_i(t) \times q_i$, where q_i is the average quality of the output of aspect a_i. As an example, consider the VoIP communication management system using different techniques to optimize its performance. We have seen the compression/decompression aspects, encryption/decryption aspects, and other aspects that may use different techniques to achieve satisfactory solutions. For example, the compression/decompression aspects must consider compression level, compressed audio quality, and compression/decompression speed to select appropriate techniques to meet the user requirements. The reliability of the system increases if a reliable technique is plugged that meets the user requirements into the framework and decreases if such a technique is unavailable, or the chosen technique is unreliable. Since the system is correct as long as one aspect produces the correct output, the probability that the output is correct is $1 - \prod_{i=1}^{n_A}(1 - r_i(t))$. The average quality of the output is $[\sum_{i=1}^{n_A} p_i \times q_i] \times [1 - \prod_{i=1}^{n_A}(1 - r_i(t))]$, where p_i is the probability that the ith aspect is selected.

4.2 Runtime Temporally Separable Systems

One way of preventing two spatially inseparable aspects, A and B, from interfering with each other when they access a shared state space is to coordinate their execution to ensure that their accesses are serializable. Usually, this type of coordination logic is interspersed within the code that implements the functional aspects of the process, but it is also possible to separate out the coordination details from the implementation of functional aspects. Two cases of coordination logic are presented here, namely, *sequence coordinators* and *concurrency control*.

4.2.1 Sequence Coordinators

In the case of *Sequence Coordinators*, the coordinator C implements a schedule of activation patterns for aspects that perform specific tasks. For a system consisting of aspects A and B, the coordinator alternates the activation of A and B. The verification of C consists of proving that it correctly enforces the invocation schedule. The reliability assessment again requires the operational profile to be determined for A and B. Since these aspects do not run in parallel, it is also necessary to deduce the proportion of time for which each one is active. This is impacted by the service times of A, B, and C as well as the coordination policy implemented by C. To make the analysis independent of the implementation of C, the specification of the coordinator must be functional. Let p denote the proportion of time that A is active and let q denote the proportion of time that B is active. ($p + q$ can be less than 1 if there is a delay between the completion of an aspect and the selection of the next aspect.) Then, the reliability of the system is the reliability of $C \times$ ($p \times$ reliability of $A + q \times$ reliability of B).

4.2.2 Concurrency Control

In *Concurrency Control*, a number of independent tasks vie for access to the shared state space. The coordinator processes the current state of the system as well as all events and requests. Based on specified concurrency control policies, it enqueues or activates certain tasks. In addition to policies that impact the functional behavior of the system, the coordinator may need to meet other constraints, such as performance constraints, absence of deadlock, absence of starvation, etc. The system can be verified by verifying each individual task separately. The verification of the coordinator consists of proving that it enforces the specified coordination policies. Model checking can be used to verify other properties, such as absence of deadlocks and absence of starvation. These proofs together imply the correctness of the overall system. Statistical reliability assessment is extremely complex for such systems due to the nondeterminism inherent in the coordination process. It is difficult to determine the operational profile and equally difficult to recreate the operational profile during testing.

5 Reliability Assessment of Spatially Inseparable and Temporally Inseparable Aspects

In this case, it is neither possible to separate the accesses of the aspects in different dimensions of the state space nor is it possible to separate their accesses at different times. Instead, the aspects have to operate in parallel and have to update the state space at the same time. One way of enabling the code for the programs to be developed independently and to then be composed together is to write the programs so that they generate all possible output values corresponding to an input rather than simply one value. The programs can be composed together at the output level by taking the intersection and union of their output sets as needed.

Suppose the postcondition is $P \wedge Q$ and the precondition is T for a 2-aspect system consisting of aspects A and B, respectively. Let the output set of A be $p(A, T)$ and the output set of B be $q(B, T)$. Then, verification consists of showing that $T \Rightarrow wp(A, P)$ and $T \Rightarrow wp(B, Q)$. This implies that $p(A, T)$ satisfies P and $q(B, T)$ satisfies Q. Hence, the intersection of the outputs, if it is nonnull, satisfies $P \wedge Q$. If the intersection is null, then it means that P and Q cannot be satisfied simultaneously for this input.

The reliability of the system can be estimated from the reliabilities of the aspects. Each of the aspects accesses the shared state space directly and, hence, it sees the same operational profile. Testing using random sampling is used to validate each aspect independently. The test oracle must accept a set of values and check whether all the values satisfy the postcondition.

In the following, we discuss **fusion composition**, where 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. In particular, there are three types of fusion composition, namely, *cooperative fusion* [1], *competitive fusion* [21], and relational fusion. In all these cases, 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.

In a cooperative fusion, the output from two or more independent aspects are fused together to achieve an outcome that could not have been achieved by the individual outputs of the aspects. For example, a set of thermometers over a region can produce a gradient of temperature difference within the entire region [1]. In a competitive fusion, the outputs from the individual aspects compete for the overall output of the system. However, this is different from the selection type where one output from a set of individual outputs is selected as the overall system output. In a competitive fusion, the individual outputs "influence" the overall system output rather than determine the outcome [21]. Statistical mean, majority voting, and n-version voting are some of the examples of competitive fusions. The cooperative and competitive fusions are well-known in the sensor network area. In the following section, relational fusion is discussed in detail.

5.1 Relational Fusion

In relational fusion, the top level specification, g, is first decomposed into a **conjunction** of predicates, $g = g_1 \wedge g_2 \wedge \cdots \wedge g_n$. The individual predicates, g_i, are further decomposed into a **disjunction** of predicates, i.e., $g_i = g_{i1} \vee g_{i2} \vee \cdots \vee g_{in_i}$. Each predicate g_{ij} represents one way of achieving g_i. Note that conjunctive and disjunctive decompositions can be applied to the specification iteratively as necessary. Let P_{ij} be a program that achieves g_{ij}. In conventional programs, P_{ij} is viewed as a *function*. There is no obvious mathematical model for merging independently developed P_{ij}s into the overall system program P since the output of the P_{ij}s may be incompatible with each other. In our approach, P_{ij} is viewed as a general relation and, hence, it returns the set of all output values for each input. P can be obtained by simply forming the intersection of the output sets of P_is, $P \equiv P_1 \cap P_2 \cap \cdots \cap P_n$, where P_i is the program for achieving g_i. Similarly, each P_i can be obtained via a systematic *union* operation from its components, i.e., $P_i \equiv P_{i_1} \cup P_{i_2} \cup \cdots \cup P_{in_i}$.

Since each relational program, P_{ij}, returns the set of all possible output values for each input, P_{ij} is correct if and only if there are no extra values or missing values in the output set. Having an extra value in the output set causes the system to work incorrectly when the extra value is chosen as the final selection for the system output. On the other hand, missing values pose a serious problem in the case where the system output set is obtained from an intersection of the output sets of the components since the missing values would be carried to the system output. Hence, the incorrect output sets can be categorized into two types—one that is missing the necessary output values and the other that contains extra values since we do not know how each output set will be used. Note also that any incorrect value in the output set is considered as an extra value. It is also reasonable to assume that the probability of having missing values in the output set and the probability of having extra values in the output set are independent. Let $Prob_m(P_{ij})$ denote the probability that P_{ij} has some missing values and $Prob_e(P_{ij})$ denote the probability that P_{ij} has some extra values. Consequently, let $Prob_{ne}(P_{ij})$ be the probability that P_{ij} has no extra values and $Prob_{nm}(P)$ be the probability that P_{ij} has no missing values in the output set.

Lemma 7. *For a relational program P, the reliability of $P = Prob_{ne}(P) \times Prob_{nm}(P)$.*

Proof. Since a relational program returns a set of all possible output values for a given input set, it is considered correct if and only if there are no extra values nor any missing values in the output set. The reliablity of P, $Reliability_P$, is then $Prob\{P \text{ has no extra value and P has no missing values}\} = (1 - Prob_e(P)) \times (1 - Prob_m(P)) = Prob_{ne}(P) \times Prob_{nm}(P)$. □

The system level reliability then follows from the relational aspect level reliability in a simple way.

Lemma 8. *If P_1 and P_2 are two relational programs, the reliability of the system, $Reliability(P_1 \cap P_2)$, is bounded by the following expression: $\{Prob_{ne}(P_1) \times Prob_{nm}(P_1)\} \times \{Prob_{ne}(P_2) \times Prob_{nm}(P_2)\} \leq Reliability(P_1 \cap P_2)$ provided P_1 and P_2 have independent failure processes.*

Proof. If both P_1 and P_2 work correctly, then the system, $P_1 \cap P_2$, works correctly. As seen in Lemma 7, a relational program works correctly if and only if it contains no extra values nor missing values in its output set. Therefore, if P_1's output set contains no extra values nor missing values AND P_2's output set contains no extra values nor missing values, then the system works correctly. Then, the reliability of the system, $Reliability(P_1 \cap P_2)$, becomes

$$\{Prob_{ne}(P_1) \times Prob_{nm}(P_1)\} \times \{Prob_{ne}(P_2) \times Prob_{nm}(P_2)\}.$$

However, there are some cases where the reliability of the system may be higher than the expression mentioned. This happens when one relational component's extra values do not fall within the range of the other relational component's output set and/or one relational component's missing values do not fall within the range of the other relational component's output set. In such a situation, those extra or missing values get discarded during intersection process. Therefore, the reliability of the system, $Reliability(P_1 \cap P_2)$, is bounded by:

$$\{Prob_{ne}(P_1) \times Prob_{nm}(P_1)\} \times \{Prob_{ne}(P_2) \times Prob_{nm}(P_2)\} \leq Reliability(P_1 \cap P_2).$$

□

Lemma 9. *If P_1 and P_2 are two relational programs, the reliability of the system, $Reliability(P_1 \cup P_2)$, is bounded by the following expression: $\{Prob_{ne}(P_1) \times Prob_{nm}(P_1)\} \times \{Prob_{ne}(P_2) \times Prob_{nm}(P_2)\} \leq Reliability(P_1 \cup P_2)$ provided P_1 and P_2 have independent failure processes.*

Proof. The proof is similar to the proof of Lemma 8 except that the system is correct when one relational component's extra values do fall within the range of the other relational component's correct output set and/or one relational component's missing values do fall within the range of the other relational component's correct output set. Therefore, the reliability of the system, $Reliability(P_1 \cap P_2)$, is bounded by: $\{Prob_{ne}(P_1) \times Prob_{nm}(P_1)\} \times \{Prob_{ne}(P_2) \times Prob_{nm}(P_2)\} \leq Reliability(P_1 \cap P_2)$. □

6 Case Study

In this section, we illustrate the framework-based reliability assessment approach using a multimedia collaboration tool. The tool not only provides voice chat capability using VoIP as shown in Section 2, but also allows users to carry out video conferencing as well as file transfers. A high level description of selected functionalities is as follows: Consider a multimedia collaboration tool that supports up to three users in doing voice and video conferencing. Each participant of a collaboration session can hear and see all participants of the conference. The tool also allows users to choose different audio codec techniques (i.e., G.711, G.723.1, and G.729) for the voice conferencing. Users of the tool can also monitor overall network usages of a collaboration session as well as the average system resource usages. These usage statistics can be used for future billing purposes and resource optimizations.

The multimedia collaboration system enables a user to

1. invite up to two people to participate in a three-person voice/video conference and hear and see all participants,
2. select different audio codec technique that works best for a particular situation,
3. monitor overall network usages in terms of session related packets exchanged among all participants, and
4. select up to three session related resources to monitor average resource usage.

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 different usage readings. These are upgradable 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, 2D, along a curve, etc.), circular gauges, etc. The capability provided to the user enables grouping and placement of different usage displays.

Let $R_F(t)$ denote the overall reliability of the framework excluding the reliabilities of the usage data acquisition aspect, the visual aspect, the chosen audio codec technique, and the video displays. Assume that a user is monitoring three resource usages, RS_1, RS_2, and RS_3, two network usages, N_1 and N_2 (i.e., network usages between two other participants and himself or herself). To enhance the audio quality for a voice chatting session, the user is able to select a codec scheme from three different options, G_1, G_2, and G_3 using the ITU-T recommendations for codecs G.711, G.723.1, and G.729, respectively. In addition, there are three video output windows, V_1, V_2, and V_3 to see all three participants including himself or herself.

Assume that

1. the resource usage is combined together using component C_1 and their average is displayed as a moving bar in display region D_1,
2. the network usage is combined together using component C_2 and the aggregate value is displayed as a color in display region D_2,
3. the audio codec technique selection is displayed in region D_3(the user can only select one scheme at a time), and
4. the outputs of the Web-cams of the three participants are displayed in regions D_4, D_5, and D_6.

The overall display is a montage of six 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 RS_1, RS_2, and RS_3,

$$R_{D_1}(t) = \left(\prod_{i=1}^{3} R_{dataAcquisition-RS_i}(t)\right) \times R_{composition-C_1}(t) \times R_{visualization-D_1}(t),$$

where $R_{dataAcquisition-RS_i}(t)$ is the reliability of the plug-in resource usage monitoring aspect RS_i, $1 \leq i \leq 3$,

$$R_{composition-C_1}(t)$$

is the reliability of the composition routine that computes the average of RS_1, RS_2, and RS_3, and $R_{visualization-D_1}(t)$ is the reliability of the visual aspect for display region D_1.

Similarly, the composition for region D_2 is of the fusion type, so

$$R_{D_2}(t) = \left(\prod_{i=1}^{2} R_{dataAcquisition-N_i}(t)\right) \times R_{composition-C_2}(t) \times R_{visualization-D_2}(t),$$

where $R_{dataAcquisition-N_i}(t)$ is the reliability of the plug-in network usage monitoring aspect N_i, $1 \leq i \leq 2$, $R_{composition-C_2}(t)$ is the reliability of the composition routine that computes the effective network usage from $N1$ and N_2, and $R_{visualization-D_2}(t)$ is the reliability of the visual aspect for display region D_2. Since the composition for region D_3 is of the selection type,

$$R_{D_3}(t) = \left(\sum_{i=1}^{3} P_{select-G_i} \times R_{dataAcquisition-G_i}(t)\right) \times R_{visualization-D_3}(t)),$$

where $P_{select-G_i}$ is the probability that the ith codec technique is selected, $1 \leq i \leq 3$. The video camera displays conform to the selection composition. Hence, for $4 \leq i \leq 6$,

$$R_{D_i}(t) = R_{dataAcquisition-V_{i-3}}(t)) \times R_{visualization-D_i}(t).$$

Hence, the overall reliability of the system is

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

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 since the system reliability is mathematically inferred from the aspect reliabilities. For example, consider a system of two compile-time temporally separable aspects A and B. Since the system reliability, $Reliability_{system}$, is $Reliability_A \times Reliability_B$, if the aspect A is modified to or replaced with A', the system reliability can easily be recomputed using the reliability of A' (i.e., $Reliability_{system}$ simply becomes $Reliability_{A'} \times Reliability_B$).

7 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 [33] 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 [17], Objectcharts [11], and other related methods. It has also been applied to existing languages, e.g., Z [20]. 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 [25] 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 [24], behavioral inheritance [2], IDEAL components [7], and Aspect-Oriented Programming [23]. 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 [22] 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 [2]. The approach proposed in [2] 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 [7], 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 [23] 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 [19] and operational profile testing [29]. Also, each aspect can be made fault-tolerant more cost-effectively using design diversity [14], 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 [3]. Nonmonotonic logic [9], especially paraconsistent nonmonotonic logic [10], 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 [16]. This difficulty is compounded when different specification methods are used to specify different views [34]. 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 some composition methods, for example, set 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 reliabilities to a high degree of confidence.

8 Summary

In this paper, we have presented and evaluated a framework-based approach for decomposing and implementing complex software systems that 1) enable the system reliability to be rigorously inferred from the aspect reliabilities based on the composition method, 2) allow the aspects to be upgraded or removed, and 3) 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 software 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 and testing depending on the characteristics of each aspect in order to maximize the confidence in the correctness of the system.

In addition to the above advantages, the approach also allows aspect-level hardening for redundancy, customization, etc. With end-user observability of aspects, each aspect can be tested, validated, and verified by the end-user (not the developers). Furthermore, interactions among aspects are straight-forward, and users do not have to worry about new glue code synthesis when changing aspects. The approach also supports multiple architectures in implementation of different parts of the system.

The approach currently has some limitations, the main one being that the application must be decomposable into a set of IDEAL aspects. The need for relational composition for some of the aspects implies that the approach is most suitable for applications where the output sets can be represented concisely, such as continuous process-control systems, or where the composition can be done statically. The approach has been successfully applied to process-control systems [8], a high-assurance measurement repository system [6], and telecommunications systems. The current version of the approach is not optimized, and methods of improving the performance of the framework needs more investigation. The framework currently encompasses two types of architectures, namely, the pipes-and-filters architecture and the shared repository architecture. A future research direction is to investigate the applicability of the approach to other types of software architectures. It is also interesting to explore the possibility of other composition methods that support plug-in application aspects. Other research issues include incorporating quality of service specifications by associating a quality of service measure with each output, decomposition of a given specification into finer aspects, and the automated identification of such decompositions.

Acknowledgments

This research was supported in part by the US National Science Foundation under grant numbers CCR-9900922 and EIA-0103709 and by the Texas ATP under grant number 009741-0143-1999.

References

[1] J. Agre and L. Clare, "An Integrated Architecture for Cooperative Sensing Networks," *Computer,* vol. 33, no. 5, pp. 106-108, May 2000.
[2] C. Atkinson, *Object-Oriented Reuse, Concurrency and Distribution.* New York: Addison-Wesley & ACM Press, 1991.
[3] R. Balzer, "Tolerating Inconsistency," *Proc. 13th Int'l Conf. Software Eng.,* pp. 158-165, May 1991.
[4] F.B. Bastani, B. Cukic, V. Hilford, and A. Jamoussi, "Toward Dependable Safety-Critical Software," *Proc. Second Workshop Object-Oriented Real-Time Dependable Systems,* Feb. 1996.
[5] F.B. Bastani, "Relational Programs: Architecture for Robust Process-Control Programs," *Annals,* 1999.
[6] F.B. Bastani, S. Ntafos, I.-L. Yen, D.E. Harris, R.R. Morrow, and R. Paul, "High-Assurance Measurement Repository System," *Proc. Fifth IEEE Int'l Symp. High Assurance Systems Eng.,* Nov. 2000.
[7] F.B. Bastani, I.-L. Yen, S. Kim, J. Linn, and K. Rao, "Reliability of Systems of Independently Developable End-User Assessable Logical (IDEAL) Programs," *Proc. IEEE Int'l Symp. Software Reliability Eng.,* Nov. 2001.
[8] F.B. Bastani, I.-L. Yen, and S. Kim, "Highly Reliable Relational Control Programs for Robust Rapid Transit Systems," *Proc. Sixth IEEE Int'l Symp. High Assurance Systems Eng.,* Oct. 2001.
[9] J. Bell, "Non-Monotonic Reasoning, Non-Monotonic Logics, and Reasoning about Change," *Artificial Intelligence Rev.,* vol. 4, pp. 79-108, 1990.
[10] H. Blair and V.S. Subrahmanian, "Paraconsistent Logic Programming," *Theoretical Computer Science,* vol. 68, pp. 135-154, 1989.
[11] D. Coleman, F. Hayes, and S. Bear, "Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design," *IEEE Trans. Software Eng.,* vol. 18, no. 1, pp. 9-18, Jan. 1992.
[12] B. Cukic and F.B. Bastani, "On Reducing the Sensitivity of Software Reliability to Variations in the Operational Profile," *Proc. IEEE Int'l Symp. Software Reliability Eng.,* Oct. 1996.
[13] J.B. Dugan and R. Van Buren, "Reliability Evaluation of Fly-by-Wire Computer Systems," *J. Systems and Safety,* June 1993.
[14] J.B. Dugan and M.R. Lyu, "System Reliability Analysis of an N-Version Programming Application," *IEEE Trans. Reliability,* pp. 513-519, Dec. 1994.
[15] W.W. Everett, "Software Reliability Component Analysis," *Proc. Ninth Ann. Software Reliability Engineering Workshop,* July 1998.
[16] A.C.W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency Handling in Multiperspective Specifications," *IEEE Trans. Software Eng.,* vol. 20, no. 8, pp. 569-578, Aug. 1994.
[17] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming,* vol. 8, pp. 231-274, 1987.
[18] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "Statemate: A Working Environment for the Development of Complex Reactive Systems," *IEEE Trans. Software Eng.,* vol. 16, no. 4, pp. 403-414, Apr. 1990.
[19] C.L. Heitmeyer, B.L. Labaw, and D. Kiskis, "Consistency Checking of SCR-Style Requirements Specifications," *Proc. Second IEEE Int'l Symp. Reliability Eng.,* pp. 56-63, Mar. 1995.
[20] D. Jackson, "Structuring Z Specifications with Views," *ACM Trans. Software Eng. and Methodology,* vol. 4, no. 4, pp. 365-389, Oct. 1995.
[21] D.N. Jayasimha, S.S. Iyengar, and R.L. Kashyap, "Information Integration and Synchronization in Distributed Sensor Networks," *IEEE Trans. Systems, Man, and Cybernetics,* vol. 21, no. 5, pp. 1032-1043, Sept./Oct. 1991.
[22] C.B. Jones, "Tentative Steps Towards a Development Method for Interfering Programs," *ACM Trans. Programming Languages and Systems,* vol. 5, no. 4, pp. 596-619 Oct. 1983.
[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loigtier, and J. Irwin, "Aspect-Oriented Programming," *Proc. European Conf. Object-Oriented Programming,* June 1997.
[24] S.S. Lam and A.U. Shankar, "A Theory of Interfaces and Modules: I—Composition Theorem," *IEEE Trans. Software Eng.,* vol. 20, no. 1, pp. 55-71, Jan. 1994.

[25] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese, "Requirements Specification for Process-Control Systems," *IEEE Trans. Software Eng.*, vol. 20, no. 9, pp. 684-707, Sept. 1994.
[26] *Handbook of Software Reliability Eng.*, M. Lyu, ed., McGraw-Hill and IEEE Press, 1996.
[27] D. Mason and D. Woit, "Software System Reliability from Component Reliability," *Proc. Ninth Ann. Software Reliability Eng. Workshop,* July 1998.
[28] J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Applications,* professional ed., McGraw-Hill, pp. 178-180, 1990.
[29] J.D. Musa, "Operational Profiles in Software Reliability Engineering," *IEEE Software,* pp. 14-32, Mar. 1993.
[30] D.A. Russo, Private Communication, May 2000.
[31] C. Smidts, D. Sova, and G.K. Mandela, "An Architectural Model for Software Reliability Quantification," *Proc. Eighth Int'l Symp. Software Reliability Eng.*, pp. 324-335, Nov. 1997.
[32] V.L. Winter, Private Comm., July 1998.
[33] P. Zave, "A Distributed Alternative to Finite-State-Machine Specifications," *ACM Trans. Programming Languages and Systems,* vol. 7, no. 1, pp. 10-36, Jan. 1985.
[34] P. Zave and M. Jackson, "Where Do Operations Come From? A Multiparadigm Specification Technique," *IEEE Trans. Software Eng.*, vol. 22, no. 7, pp. 508-528, July 1996.

Sung Kim received the BA degree from the University of Texas at Austin and the MS degree in computer science from the University of Texas at Dallas. He is currently working on the PhD degree in computer science at the University of Texas at Dallas, while working as a member of Scientific Staff at Nortel Networks, Inc. His research interests are in the areas of software decomposition, high-assurance software systems engineering, software reliability assessment, component-based software security, and certifications.

Farokh B. Bastani received the BTech degree in electrical engineering from the Indian Institute of Technology, Bombay, and the MS and PhD degrees in computer science from the University of California, Berkeley. He is currently a professor of computer science at the University of Texas at Dallas. His research interests are in the areas of relational programs, high-assurance hardware/software systems engineering, hardware/software reliability assessment, self-stabilizing systems, inherent fault tolerance, and high-performance modular parallel programs. Dr. Bastani was the Editor-in-Chief of the *IEEE Transactions on Knowledge and Data Engineering* and is on the editorial boards of the *International Journal of Artificial Intelligence Tools*, the *Journal of Knowledge and Information Systems (KAIS)*, and the Springer-Verlag book series on *Knowledge and Information Management (KAIM)*. He was the program cochair of the 1997 Symposium on Reliable Distributed Systems and the 1998 International Symposium on Software Reliability Engineering, the program vice-chair of the 1999 Symposium on Autonomous Decentralized Systems, and the program chair for the 1995 International Conference on Tools with Artificial Intelligence. He has been on the program committees of numerous conferences and workshops and on the steering committee of the IEEE Symposium on High-Assurance Systems Engineering, the IEEE Symposium on Application-Specific Software Engineering and Technology, and the IEEE Knowledge and Data Engineering Workshop. He was on the editorial board of the *IEEE Transactions on Software Engineering* and the *High Integrity Systems Journal*. He was a guest editor of the April 1992 special issue of *IEEE Transactions on Knowledge and Data Engineering* devoted to self-organizing systems and the December 1985, January 1986, and November 1993 special issues of *IEEE Transactions on Software Engineering* devoted to software reliability. He is a guest editor of a special issue of the *International Journal of Software Engineering and Knowledge Engineering (JSEKE)* devoted to embedded software systems. He is a member of ACM and the IEEE.

I-Ling Yen received the BS degree from Tsing-Hua University, Taiwan, and the MS and PhD degrees in computer science from the University of Houston. She is currently an associate professor of computer science at the University of Texas at Dallas. Her research interests include fault-tolerant computing, security systems and algorithms, distributed systems, self-stabilizing systems, and Internet technologies. She has developed many efficient fault tolerance methods, including inherent fault tolerance, multilevel robust data structures, and scalable, adaptive replication protocols. Dr. Yen and her students have developed a mobile agent system, PeAgent, that supports enhanced security and fine-grained access control. The system is used in a Weblet infrastructure to achieve better security and performance for Web services. In addition, she has developed a novel secure computation algorithm which is designed for commercial database systems. Dr. Yen has served as a program committee member for many conferences and program chair or cochair for the IEEE Symposium on Application-Specific Software and System Engineering and Technology, the IEEE High Assurance Systems Engineering Symposium, the IEEE International Computer Software and Applications Conference, and the IEEE International Symposium on Autonomous Decentralized Systems. Dr. Yen is a member of the IEEE and the IEEE Computer Society.

Ing-Ray Chen received the BS degree from the National Taiwan University, Taipei, Taiwan, and the MS and PhD degrees in computer science from the University of Houston, Texas. He is currently an associate professor in the Department of Computer Science at Virginia Tech. His research interests include mobile computing, pervasive computing, multimedia, distributed systems, real-time intelligent systems, and reliability and performance analysis. Dr. Chen has served on the program committees of numerous conferences, including being the program chair of the 14th IEEE International Conference on Tools with Artificial Intelligence in 2002, and the Third IEEE Symposium on Application-Specific Systems and Software Engineering Technology in 2000. Dr. Chen currently serves as an associate editor for the *IEEE Transactions on Knowledge and Data Engineering*, the *Computer Journal*, and the *International Journal on Artificial Intelligence Tools*. He is a member of the IEEE, the IEEE Computer Society, and the ACM.