Assessment of High Integrity Software Components for Completeness, Consistency, Fault-Tolerance, and Reliability

Hye Yeon Kim[1], Kshamta Jerath[2], and Frederick Sheldon[3]

[1] Bluetooth Research Group, Samsung Electro-Mechanics, HQ
314, Meatan-3Dong, Paldal-Gu, Suwon, Kyounggi-Do, South Korea, 442-743
hyekim@ieee.org

[2] School of Electrical Engineering and Computer Science
Washington State University, Pullman, WA 99164, USA
kjerath@eecs.wsu.edu

[3] Oak Ridge National Laboratory (ORNL), PO Box 2008, MS 6363
1 Bethel Valley Road, Oak Ridge, TN 37831-6359, USA
sheldon@acm.org

Abstract. The use of formal model based (FMB) methods to evaluate the quality of components is an important research area. Except for a growing number of exceptions, FMB methods are still not really used in practice. This chapter presents two case studies that illustrate the value of FMB approaches for developing and evaluating component-based software. In the first study, Zed (or Z) and Statecharts are used to evaluate (a priori) the software requirement specification of a Guidance Control System for completeness, consistency and fault-tolerance. The second study evaluates (post-priori) the reliability of a complex vehicle system using Stochastic Activity Networks (SANs). The FMB approach presented here provides further evidence that such methods can indeed be useful by showing how these two different industrial strength systems were assessed and the results. Clearly, future investigations of this nature will help to convince software system developers using component based approaches that such FMB methods should be considered as a valuable tool toward improving the software product lifecycle (quality, schedule and cost).

1 Introduction

To manage increasing complexity and maximize code reuse, the software engineering community has, in recent years, put considerable effort into the design and development of component-based software development systems and methodologies [4]. The concept of building software from existing components arose by analogy with the way that hardware is now designed and built, using cheap, reliable standard "off-the-shelf" modules. The success of component based software technology is dependent on the fact that the *effort* needed to build component based software systems can be significantly decreased compared to traditional custom software development. Consequently, component producers have to ensure that their commercial components possess trusted *quality* [39]. To achieve a predictable, repeatable process for engineering high-quality component based software systems, it is clear that quality must be introduced and evaluated at the earliest phases of the life cycle.

Developing Component-Based Software (CBS) systems is facilitated by component reusability. The development process for CBS is very similar to the conventional software development process. In CBS development, however, the requirements specification is examined for possible composition from existing components rather than direct construction. The components can be functional units, a service provider (i.e., application programs, Web-based agent or enterprise system [13]), or components of an application ranging in size from a subsystem to a single object[1]. To ensure the quality of the final product, assessment of such components is obligatory. Some form of component qualification at the earliest possible phase of system development is therefore necessary to avoid problems in the latter phases and reduce life-cycle costs.

Evaluation of the software system must take into consideration how the components behave, communicate, interact and coordinate with each other [2]. *Reliability*, a vital attribute of the broader quality concept, is defined as the degree to which a software system both satisfies its requirements and delivers usable services [12]. Quality software, in addition to being reliable, is also robust (and fault tolerant), complete, consistent, efficient, maintainable, extensible, portable, and understandable.

In this chapter, we discuss how one can evaluate the quality of the components using formal model based (FMB) methods (e.g., Z, Statecharts, and Stochastic Activity Networks). We present an FMB framework for assessing component properties like completeness and consistency of requirement specifications using Z and Statecharts; and approaches for verifying properties like reliability using stochastic modelling formalisms. Two case studies are discussed in this context based on both a mission critical (guidance control) software requirements specification and a vehicular system with various interacting components (possibly) provided by different vendors. The assessment of quality (i.e., reliability) for elements such as anti-lock brakes, steer-by-wire and traction control are considered based on empirical data. Naturally, a single example showing the complete process would be ideal. However, our group had two different projects (one with NASA and the second with a road vehicle manufacturer). Although different applications dealing with slightly different artifacts, there are convenient similarities (i.e., comparable properties) in their application domain: embedded real-time command and control responsive systems. These different but similar systems understandably interrelate and it is hoped that the reader can *bridge* the difference.

[1] A software component is a unit of composition with contractually specified interface and explicit context dependencies only. It can be deployed independently and is subject to composition by third parties. The most important characteristic is the separation of the component interface from its implementation.

2 Background

Component-Based Software Development (CBSD) approaches are based on developing software systems by selecting appropriate off-the-shelf components and then assembling them using a well-defined software architecture[2]. CBSD *can* significantly reduce development cost and time-to-market, and improve maintainability, reliability and overall quality of software systems. However, quality assurance technologies for CBS must address two inseparable questions: 1) How to *certify quality* of a component? 2) How to *certify quality* of software systems based on components? (Our studies focus on this aspect.) To answer these questions, models should be developed to define the overall quality control of components and systems; metrics should be found to measure the size, complexity, reusability and reliability of components and systems; and tools should be selected to test the existing components and resulting system(s). Component requirements analysis is the process of discovering, understanding, documenting, validating and managing the requirements for a component.

Hamlet et. al., address the first question for quality assurance technologies listed above: Namely, how to certify the quality of a component? They present a theory of software system reliability based on components. The theory describes how component developers can design and test their components to produce measurements that are later used by system designers to calculate composite system reliability (i.e., without having to implement and test the system being developed). Their work describes how to make component measurements independent of operational profiles, and how to incorporate the overall system-level operational profile into the system reliability calculations. In principle, their theory resolves the central problem of assessing a component. Essentially, a component developer cannot know how the component will be used and so cannot certify it for an arbitrary use; but if the component buyer must certify each component before using it, component-based development loses much of its appeal. This dilemma is resolved if the component developer does the certification and provides the results in such a way that the component buyer can factor in the usage information later, without having to repeat the certification [14].

Another natural reason for CBSD is the drive to shorten the SD lifecycle, which motivates the integration of commercial off-the-shelf (COTS) components for rapid software development. To ensure high reliability using software components as their building blocks, dependable components must be deployed to meet the reliability requirements. The process involves assembling components together, determining the interactions among the integrated components, and taking the software architecture into consideration. Black-box based approaches may not be appropriate for estimating the reliability of such systems, as it may be necessary to investigate the system architecture, the testing strategies, as well as the separate component reliabilities. In [27], the authors assume com-

[2] The software architecture of a program or computing system is the structure(s) of the system that comprise the software components, the externally visible properties of those components and the relationship(s) among them.

ponents are independent and can be viewed as composed of logically individual parts that can be implemented and tested independently. In addition, transfer of control among software components follows a Markov process[3]. Sherif et al. [36], propose a similar analysis technique for distributed software systems. The technique is based on scenarios that are modelled as sequence diagrams. Using scenarios, the authors construct Component-Dependency Graphs (CDG) for reliability analysis of component-based systems.

The growing reliance on COTS components for developing *large-scale projects* comes with a price. Large-scale component reuse leads to savings in development resources, but not without having to deal with integration difficulties, performance constraints, and incompatibility of components from multiple vendors. Relying on COTS components also increases the system's vulnerability to risks arising from third-party development, which can adversely affect the quality of the system, as well as cause expenses not incurred in traditional software development. The authors of [35] introduce metrics to accurately quantify factors contributing to the overall quality of a component-based system, guiding quality and risk management by identifying and eliminating sources of risk.

An artifact or component is fit-for-purpose if it manifests the required behavior(s) in the intended context(s), while the same is true for the composed system. The whole system therefore may be fit-for-purpose and, consists of some number of artifacts in some context. Furthermore, we need to know the quality of the whole system. It doesn't make any sense to talk about the quality of a single artifact as a stand-alone entity, independent of any particular context. There is no absolute (context-free) measure of quality. However (see [37]), under some special circumstances, it is possible to carry out a completely definitive test to demonstrate that a given artifact completely satisfies a given (formal) specification. Still, this does not prove that the artifact actually meets the users stated or implied needs. A requirements statement describes what an object must satisfy when used for a given purpose, in a given context (i.e., the *actual* requirements). When developing an object for reuse, however, the developer usually does not have access to the complete set of concrete requirements. Instead, the developer attempts to build reusable objects by working against a generalized statement of requirements that hopefully covers a reasonable range of *actual* requirements.

2.1 Assessing Requirement Specifications Using Z and Statecharts

As is well known, CBS development begins by specifying the requirements like any other software development effort. The Software Requirements Specification (SRS) describes what the software must do. Naturally, the SRS takes the core role as the descriptive documentation at every phase of the life-cycle. Therefore, it is necessary to ensure the SRS contains correct and complete information for the system. For that reason, employing a verification technique is necessary for the specification to provide some support of prototyping, correctness proofs, elaboration of test data, and failure detection. To avoid problems in the later

[3] The next transfer of control to be executed is independent of the past history and depends only on the present component.

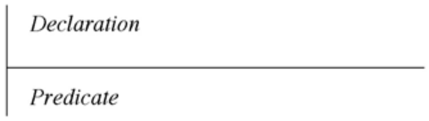


Fig. 1. Form of axiomatic definition

development phases and reduce the life-cycle costs, it is crucial to ensure that the specification be complete and consistent.

The completeness of a specification is defined as a lack of ambiguity in terms of creating an implementation that can satisfy the specified requirements. Thus, the specification is incomplete if the system behavior is not precisely stated because the required behavior for some events or conditions is omitted or is subject to multiple interpretations [24]. On the other hand, consistency means the specification is free from conflicting requirements and undesired non-determinism [7]. The lack of information or precision presents incompleteness, while conflicting information presents inconsistency. Hypothetically, if we distill the specification down to a simple finite state system, we may discover missing states (i.e, incompleteness) or state transitions from one state to possibly different states on the same input (i.e., conflicting non-determinism, which gives inconsistency).

Typically, fault-tolerance is considered as an implementation property that provides for (1) explicit or implicit error detection for all fault conditions, and (2) backup routines to guarantee continued service of critical functions in case errors arise during operation of the primary software [33]. For the SRS, it can be defined as (1) existence of specified requirements to detect errors for all fault conditions, and (2) presence of specified requirements that support the system robustness, software diversity, and temporal redundancy for continuing service of critical system functions in the case of failure.

Most problems can be traced to the requirements specification typically due to ambiguity [10]. Formal methods unambiguously define the requirements of software with respect to its specification. They are the primary way to have a rigorous definition of correctness of the system requirements. The decision to use formal specifications mainly depends on the criticality of the component, in term of severity of fault consequences and of the complexity of its requirements or of its development [33].

Z is classified as a model-based specification language equipped with an underlying theory that enables non-determinism to be removed mechanically from abstract formulations that result in *concrete* specifications. In combination with natural language, it can be used to produce a formal specification [41]. Let's just review some of the basic elements that make Z useful and compose part of our FMB framework strategy.

Axiom is one way to define a global object in Z. It consists of two parts: declaration and predicate (see Fig. 1). The predicate constrains the objects introduced in the declaration.

Schemas are the main structuring mechanism used to create patterns and objects. The schema notation is used to model system states and operations.

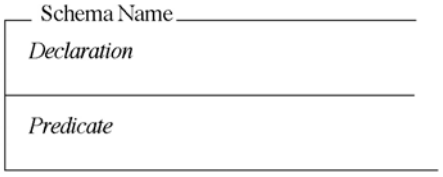


Fig. 2. Form of a schema

A *schema* consists of two parts: a declaration of variables and a predicate constraining their values (see Fig. 2). The name of a schema is optional, however, it is more convenient to give a name because it can be referenced within other schemas.

The *free type* is used to define new types similar to the enumerated types provided by many programming languages [20]. The free type in Fig. 3 introduces a collection of constants, one for each element of the set *source*. Constructor is an injective function whose target is the set *Free_type_name*. Consistency of free type can only be validated when each of the constructions (i.e., the set *source*) is involved with Cartesian products, finite power sets, finite functions, and finite sequences [41]. Axioms and abbreviations are used to define global constants and functions. The abbreviation T_n==seq $\mathcal{N}$ states that T_n is another name for sequence of natural numbers.

Free_type_name ::= constants | constructor ⟪source⟫

Fig. 3. Free type notation

The state of the system and the relationship between the states of various components can be explained using the aforementioned Z formalism. The production of such a specification helps one to understand requirements, clarify intentions to identify assumptions and explain correctness. These facilities are useful and essential in clarifying ambiguities and solidifying one's understanding of the requirements.

Statecharts, a state-based formal diagrammatic language, are a visual formalism for describing states and transitions in a modular fashion, enabling cluster orthogonality (i.e., concurrency) and refinement, and supporting the capability for moving between levels of abstraction. The kernel of the approach is the extension of conventional state diagrams by AND/OR decomposition of states together with inter-level transitions, and a broadcast mechanism for communication between concurrent components. The two essential ideas enabling this extension are the provision for depth (level) of abstraction and the notion of orthogonality. In other words, Statecharts = State-diagrams + depth + orthogonality + broadcast-communication [15].

Statecharts provide a way to specify complex reactive systems both in terms of how objects communicate and collaborate and in terms of how they behave internally[4]. Together, Activity-charts and Statecharts are used to describe the system functional building blocks, activities, and the data that flows between them. These languages are highly diagrammatic in nature, constituting full-fledged visual formalisms, complete with rigorous semantics that provide an intuitive and concrete representation for inspecting and (mechanically) checking for conflicts [16]. The Activity-charts and Statecharts are used to specify conceptual system models for symbolic simulation. Using the simulation method, assumptions are verified, faults may be injected, and hidden errors are identified that represent inconsistencies or incompleteness in the specification.

Ambiguous statements in the SRS are revealed during the construction of Z schemas. When a misinterpreted specification in Z is uncovered during the execution of the Statecharts model, the Z specification can be refined using the test results.

2.2 Predicting Reliability Using Stochastic Formalisms

As with hardware systems, CBS systems can be modelled early on during the system lifecycle. A mathematical model is used to predict (estimate in the case that empirical data is available) the value of some quality attribute. For example, the reliability of the software system is based on parameters that are previously known or evaluated during integration and test of the software components [12]. Modelling and subsequent sensitivity analysis of these models can provide measurements regarding overall software-system reliability and suitability of a particular component for being used as part of the whole system context.

Stochastic Petri Nets (SPNs) and Stochastic Activity Networks (SANs) are formalisms that can be used to create concise representations/models of real-time, concurrent, asynchronous, distributed, parallel and/or non-deterministic systems. Tools exist to automatically generate and solve the underlying Markov chains from these representations.

Structurally, SANs consist of four primitive objects: *places*, *activities*, *input gates* and *output gates* [3]. Places represent the state of the modelled system. They are represented graphically as circles. Each place contains a certain number of tokens which represents the marking of the place. The set of all place markings represents the marking of the network. Activities represent actions in the modelled system that take some specified amount of time to complete. There are of two types: *timed* and *instantaneous*. Timed activities have durations that impact the performance of the modelled system, and are represented as hollow ovals. Instantaneous activities represent actions that complete in a negligible amount of time compared to the other activities in the system. *Case probabilities*, represented graphically as circles on the right side of an activity, model uncertainty associated with the completion of an activity.

Input gates control the enabling of activities and define the marking changes that will occur when an activity completes. They are represented graphically as triangles with their point connected to the activity they control. Like input

[4] Statecharts are utilized in this respect by way of the Statemate Magnum tool.

gates, output gates define the marking changes that will occur when activities complete. The only difference is that output gates are associated with a single case. They are represented graphically as triangles with their flat side connected to an activity or a case.

We discuss reliability modelling of component-based software systems (using SANs) emphasizing failure severity levels and coincident errors among components to predict the overall system reliability. The reliability of a CBS system is a function of the reliabilities of the individual components that compose the complete system. If the components were all independent of each other, the overall reliability would simply be the product of the reliabilities of all the individual components. However, in practice, this is hardly the case. Components interact with each other, depending on other components for control information or data. Any representation claiming to realistically model the system must take this interaction into consideration. Coincident errors have been considered and modelled for predicting system reliability in [1, 8, 9, 21, 25, 29, 34].

Further, errors and/or defects occurring in the system have varying levels of severity and pose different levels of threat to the overall system operation. A system having considerable number of high-severity defects is certainly less reliable than a system having more low-severity defects. Predicting the reliability/availability based on these characteristics of the system provides more objective and concrete information that can be used in assessing the risk tradeoffs and integrity levels. Severity is an important candidate to weight the data used in reliability calculations and must be incorporated into the model to determine the probability that the system survives, including efficient or acceptable levels of degraded operation. Severity of failures has been considered in the context of gracefully degrading systems in [11] and modelled using Markov Reward Models in [17].

Modelling and prediction of system reliability on the basis of these three characteristics is explained in the next section. Practical issues that stand in the way of developing such models include: (1) obtaining component reliability data, (2) a simple yet effective model being able to capture only limited (but significant) interactions among components, (3) the need to estimate fault correlation between components, and (4) reliability depends on how the system is used, making usage information an important part of the evaluation [26].

Further, two distinct problems that arise while using Markov processes are largeness and stiffness [32]. The size of a Markov model used for the evaluation of a system grows exponentially with the number of components in the system. If there are n components, the Markov model may have up to 2^n states. This causes the analysis to take a great deal of time. Stiffness is due to the different orders of magnitude (sometimes 10^6 different) between the rates of occurrence of performance-related events and the rates of rare, failure-related events. Stiffness leads to convergence difficulty in solving the model (i.e., numerical instability). Any attempt at modelling using Markov models must address these two problems. A case study is presented in Section 4 to illustrate the use of our technique on a real-world problem and how the challenges can be overcome.

3 A Framework for Evaluating Quality

We present two different studies that combine three formal approaches (i.e., logical analysis using Z, visualization, simulation and testing using Statecharts, and stochastic analysis using SANs) into a general FMB framework for the development of CBS systems. A CBS system is made up of numerous components that may be derived from different sources, including COTS or other proprietary components. It is important to first identify the appropriate components for the system being built, by carefully analyzing the system requirements [6]. Fig. 4 shows a process for selecting the appropriate components.

– *Identify usable components.* To investigate all possible components that may be useful in the system, a vast number of possible candidates must be available as well as tools for finding them.
– *Select components that meet system requirements.* Often the requirements cannot be fulfilled completely. A trade-off analysis is needed to adjust the system architecture and to reformulate requirements when selected, existing components do not completely cover stated requirements. This analysis will determine whether existing components may be used.
– *As necessary, create proprietary components for use in the system.* In the CBSD process this procedure is less attractive because it involves more effort and lead-time. On the other hand, components that include core-functionality of the product are likely to be developed internally as they will provide the competitive advantage of the product.
– *Adapt the selected components to the existing component model or requirement specification.* Some components may be directly integrated into the system while others will be modified through a modification and refinement process (e.g., using wrapping code for adaptation, etc.).
– *Compose and deploy the components using an appropriate framework.* Component models themselves would provide the framework needed.
– *Replace earlier versions with updated component versions.* This corresponds with system maintenance (both perfective and corrective).

This process enables the selection of suitable components for building the CBS system. The lower part of Fig. 4 illustrates the development stages of a CBS system needed to ensure quality (complete, consistent and dependable). The process starts with the *specification stage*, in which there exist only abstract notions of different components. The components are identified and requirement verification and validation of the software requirement specification can be carried out using Z and Statecharts (or other suitable formal analysis method and tools). It is important to uncover bugs and ambiguities in the requirements earlier in the lifecycle than later, to avoid having to take (more) costly corrective actions at later stages in the process.

After verifying the requirement specification, the CBS system is designed and prototyped using mathematical models (e.g., stochastic or analytic techniques) to evaluate and predict the quality and reliability of the proposed system. Reliability assessment can be carried out using stochastic modelling methods if

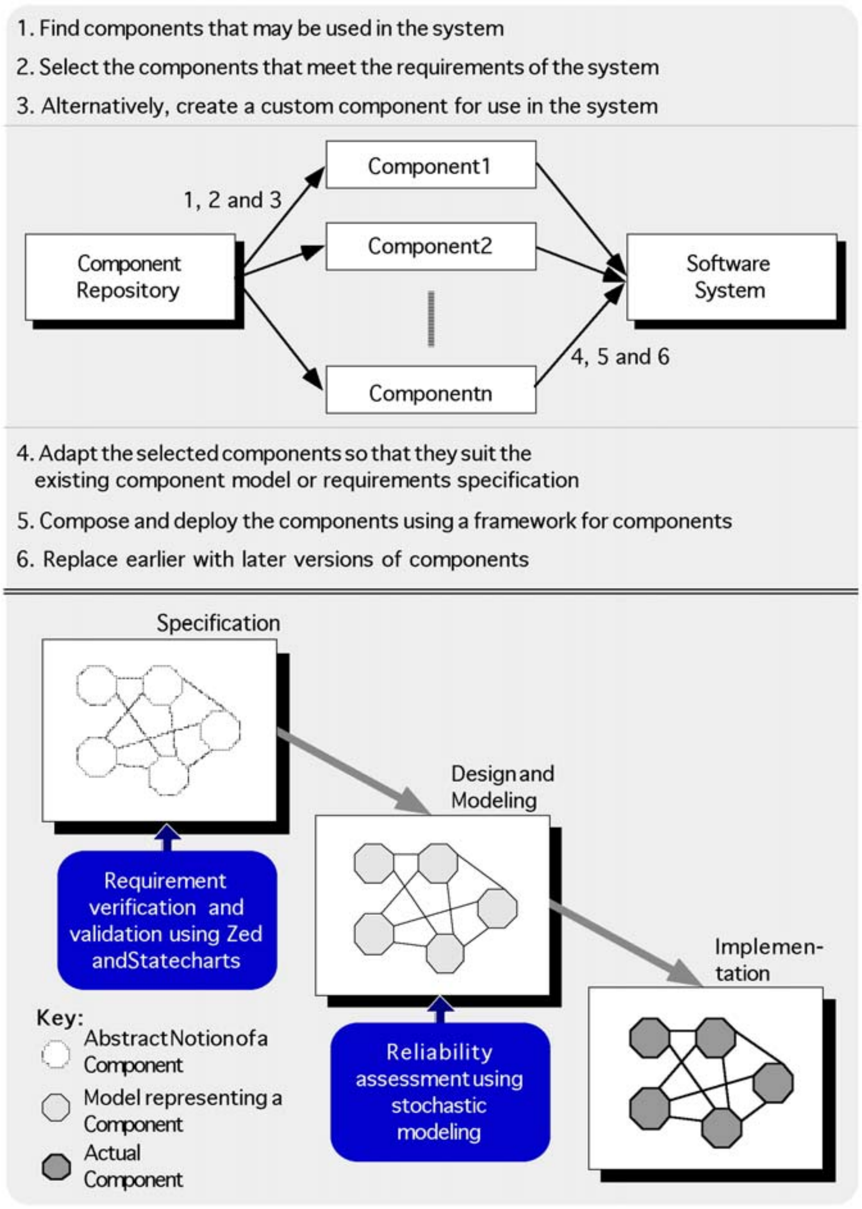

1. Find components that may be used in the system
2. Select the components that meet the requirements of the system
3. Alternatively, create a custom component for use in the system
Component Repository
1, 2 and 3
Component1
Component2
Componentn
4, 5 and 6
Software System
4. Adapt the selected components so that they suit the existing component model or requirements specification
5. Compose and deploy the components using a framework for components
6. Replace earlier with later versions of components
Specification
Requirement verification and validation using Zed andStatecharts
Design and Modeling
Reliability assessment using stochastic modeling
Implementation
Key:
Abstract Notion of a Component
Model representing a Component
Actual Component

Fig. 4. FMB development framework for CBS system

the reliability data for the individual components and possible correlation between components is available. Without such data, the analysis can also be conducted using hypothetical values for the purpose of determining the system sensitivities. As shown in Fig. 5, the use of these formal methods (Z, Statecharts

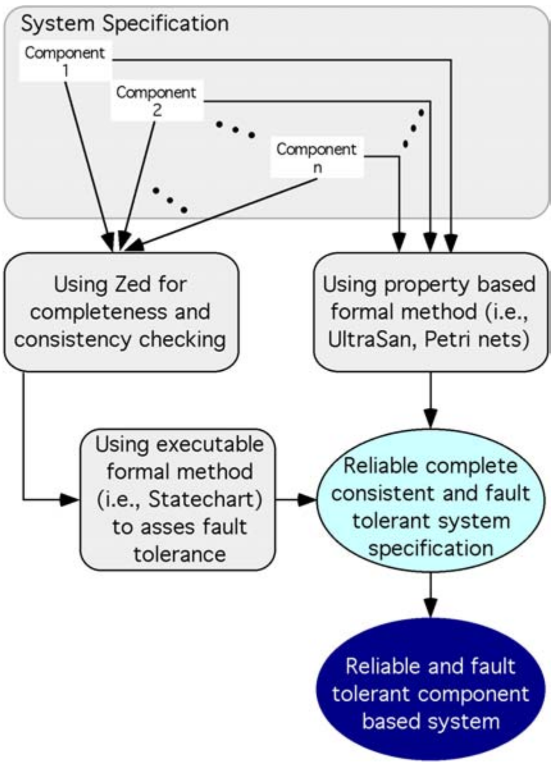

System Specification
Component 1
Component 2
Component n
Using Zed for completeness and consistency checking
Using property based formal method (i.e., UltraSan, Petri nets)
Using executable formal method (i.e., Statechart) to asses fault tolerance
Reliable complete consistent and fault tolerant system specification
Reliable and fault tolerant component based system

Fig. 5. Applying formal methods to CBSD

and SANs) at different stages in the CBSD lifecycle may result in the development of a dependable CBS system (assuming the model is transformed into the implementation—a significant assumption).

4 Case Studies

This section applies the concepts and framework presented above using two different case studies. The first study presents the use of both Z and Statecharts for verification and validation of software requirements of a Guidance Control Software (GCS) System for the Viking Mars Lander. The second study models and analyzes the reliability of an Anti-lock Braking System of a passenger vehicle.

4.1 Assessment of GCS System Requirements

The GCS principally provides vehicle control during the Lander's terminal descent phase[5]. After initialization, the GCS starts sensing vehicle altitude. When

[5] The Lander has three accelerometers, one Doppler radar with four beams, one altimeter radar, two temperature sensors, three gyroscopes, three pairs of roll engines, three axial thrust engines, one parachute release actuator, and a touch down sensor.

a predefined engine ignition altitude is sensed, the GCS begins guidance and control. The software maintains the vehicle attitude along a predetermined velocity-altitude contour. Descent continues along this contour until a predefined engine shut off altitude is reached or touchdown is sensed.

In this first study we qualified a subset (i.e., four components) of the GCS requirements in a two-step process for completeness, consistency, and fault-tolerance. Z was applied first using abstraction to detect and remove ambiguity from the Natural Language based (NL-based) GCS SRS. Next, Statecharts and Activity-charts were constructed from the Z description to enable visualization and symbolic simulation (i.e., inputs, processing and outputs). The system behavior was assessed under normal and abnormal conditions. Faults were seeded into the model (i.e., executable specification) to simulate abnormal conditions. In this way, the integrity of the SRS was assessed which identified both missing and inconsistent requirements.

Using our approach, the NL-based requirements are first re-written into the Z notation. The schema is the principle structuring mechanism (using refinement based predicate and propositional logic and set theory). Eighty percent of the SRS was completely translated into schemas thereby clarifying and concretizing the selected requirement subset. The schemas were subsequently (and iteratively) translated into Statecharts (and Activity-charts), which provided a new (executable) perspective. Simulations were performed to verify that no non-deterministic state and activity transitions exist. Some improperly defined function and data items were found in the schemas. For example, we found that correctly specified (when compared to the SRS) function and data items in both the Z and Statechart models elicited unexpected outputs during simulation. We refined the schemas, as a consequence, to avoid erroneous simulation output.

Furthermore, during the simulations, faults were injected into State and Activity-charts by changing state variable values at various breakpoints (chosen randomly). The outputs were then compared to the expected output (determined by the formula given in the SRS). This procedure enabled us to evaluate the system's ability to cope with unexpected system failures. Fig. 6 shows an example using the FRAME_COUNTER input variable that illustrates the complete translation cycle. The top box in Fig. 6 represents the NL-based SRS. The middle box of Fig. 6 represents the Z Specification while the bottom box shows the Statechart model of the ARSP (Altimeter Radar Sensor Processing) sub-module. In the SRS, the FRAME_COUNTER is defined as an integer with range $[1-(2^{31}-1)]$. In Z, the FRAME_COUNTER is declared as a set of natural numbers in the signature part, and the range of the variable is defined within the schema's predicate. The Statechart representation of the FRAME_COUNTER variable is presented with the direction of data transfer from EXTERNAL to the ARSP Module. Its type and value range are defined in the Statemate data dictionary (not shown)[6].

[6] For this case study, four components of the GCS system were assessed including the ARSP, Roll Engine Control Law Processing (RECLP), CP (Communication Processing), and GP (Guidance Processing) components. Each component was evaluated both separately and in an integrated form using our Z/Statecharts approach.

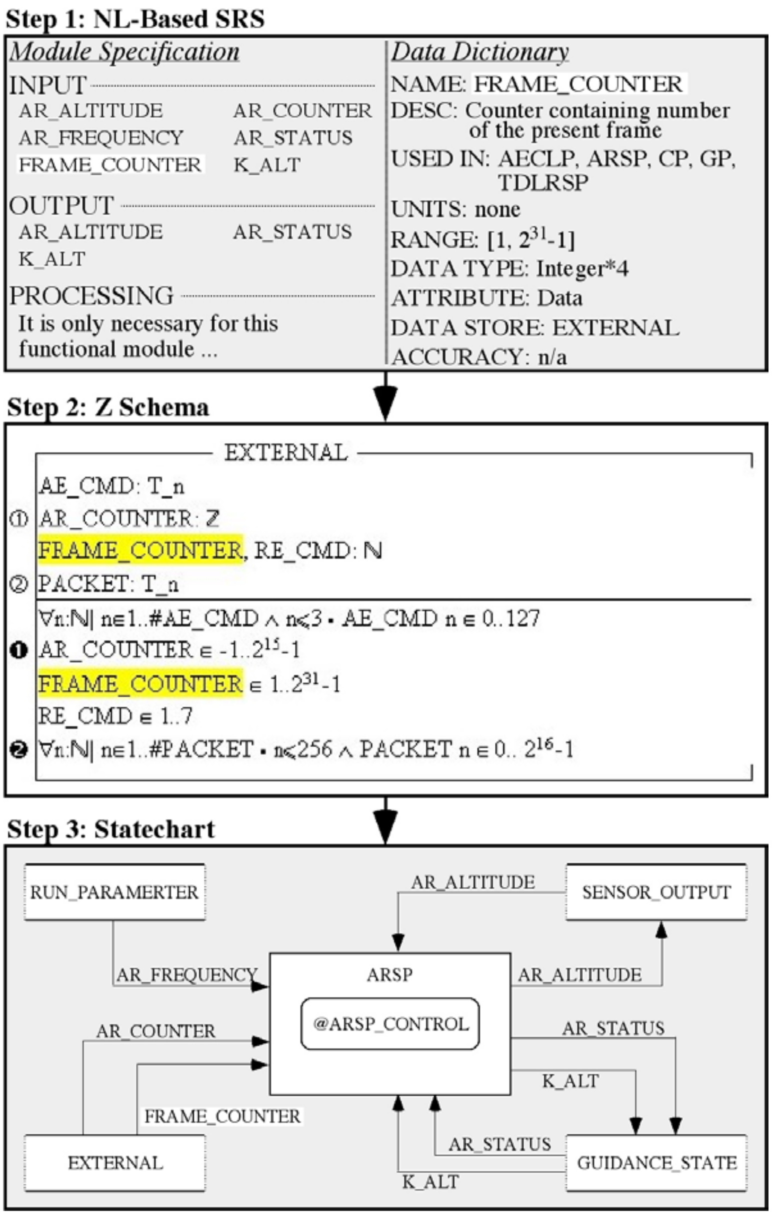
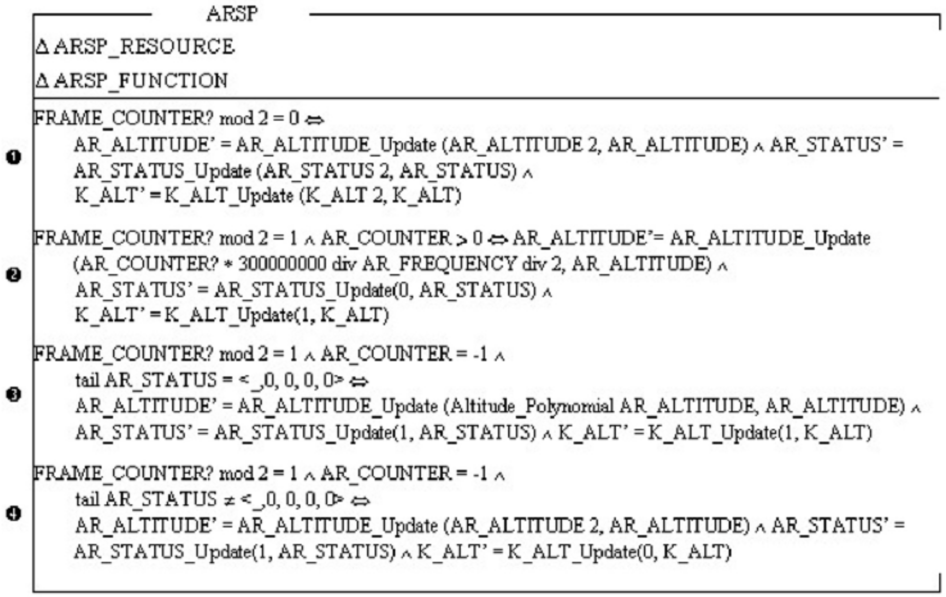


Fig. 6. Translation example from NL-based SRS to Statecharts

The ARSP, as a functional unit, reads the altimeter counter provided by the altimeter radar sensor and converts the data into a measure of distance to the surface of Mars. The ARSP schema (Fig. 7) describes the function of the ARSP unit. The Schema imports the ARSP_RESOURCE and the ARSP_FUNCTION



```
──────── ARSP ──────────────────────────────
Δ ARSP_RESOURCE
Δ ARSP_FUNCTION
────────────────────────────────────────────
FRAME_COUNTER? mod 2 = 0 ⇔
❶    AR_ALTITUDE' = AR_ALTITUDE_Update (AR_ALTITUDE 2, AR_ALTITUDE) ∧ AR_STATUS' =
     AR_STATUS_Update (AR_STATUS 2, AR_STATUS) ∧
     K_ALT' = K_ALT_Update (K_ALT 2, K_ALT)

FRAME_COUNTER? mod 2 = 1 ∧ AR_COUNTER ≥ 0 ⇔ AR_ALTITUDE'= AR_ALTITUDE_Update
❷    (AR_COUNTER? * 300000000 div AR_FREQUENCY div 2, AR_ALTITUDE) ∧
     AR_STATUS' = AR_STATUS_Update(0, AR_STATUS) ∧
     K_ALT' = K_ALT_Update(1, K_ALT)

FRAME_COUNTER? mod 2 = 1 ∧ AR_COUNTER = -1 ∧
     tail AR_STATUS = <_,0, 0, 0, 0> ⇔
❸    AR_ALTITUDE' = AR_ALTITUDE_Update (Altitude_Polynomial AR_ALTITUDE, AR_ALTITUDE) ∧
     AR_STATUS' = AR_STATUS_Update(1, AR_STATUS) ∧ K_ALT' = K_ALT_Update(1, K_ALT)

FRAME_COUNTER? mod 2 = 1 ∧ AR_COUNTER = -1 ∧
     tail AR_STATUS ≠ <_,0, 0, 0, 0> ⇔
❹    AR_ALTITUDE' = AR_ALTITUDE_Update (AR_ALTITUDE 2, AR_ALTITUDE) ∧ AR_STATUS' =
     AR_STATUS_Update(1, AR_STATUS) ∧ K_ALT' = K_ALT_Update(0, K_ALT)
────────────────────────────────────────────
```

Fig. 7. ARSP schema with predicate expressions numbered 1–4

schema for modification[7]. Predicate (1) requires that the current AR_ALTITUDE, AR_STATUS, and K_ALT element values be the same as the predecessors when the FRAME_COUNTER? is even. Predicate (2)–(4) describe the ARSP functional unit in the same manner as for predicate 1.

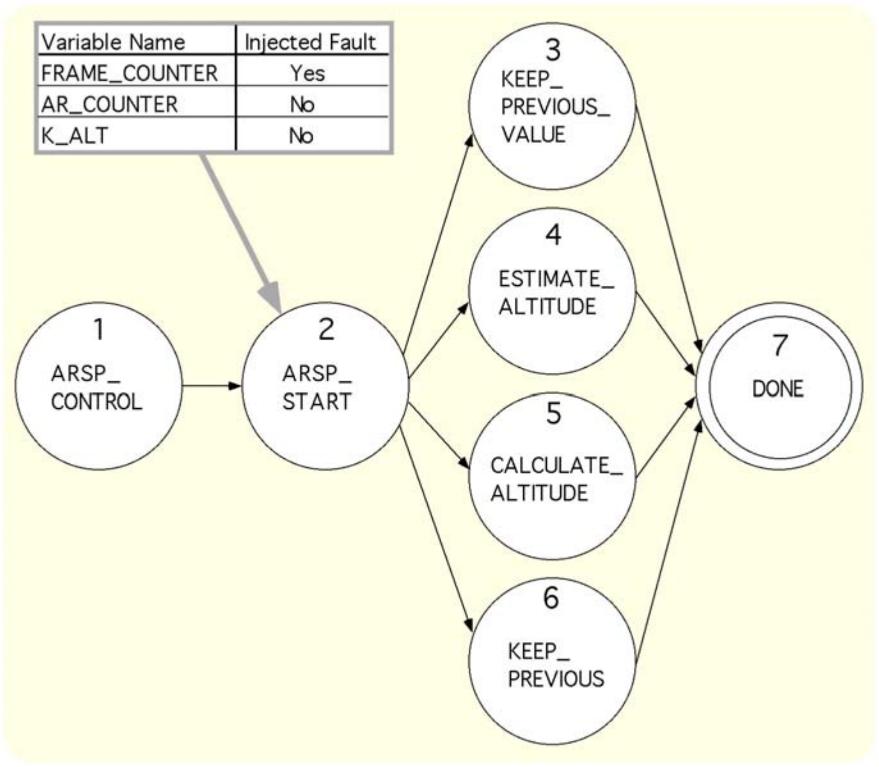
The bottom part of Fig. 6 is the Activity-chart for the ARSP schema shown in Fig. 7 and has one control state linked to a Statechart (@ARSP_CONTROL). This ARSP Statechart model has 4 distinct paths that were tested for fault-tolerance using the fault injection method (described above). The simulation results for each path (i.e., the state transitions shown in Fig. 8) are presented in Table 1. E_1 in Table 1 means that the given state is entered at first when the execution started. The "-" mark in Table 1 indicates that the state is not entered during model execution. The Activity and State names are the names of the activities and states from the Statechart.

Fig. 8 gives the finite state machine representation of the Statecharts model for the ARSP component showing four different state transition *paths*. To appreciate how the fault injection is performed note, for example, the simulation starts from the first state "ARSP_CONTROL". When the simulation process reaches the "ARSP_START" state, the selected variable value is altered (i.e., representing an injected fault, e.g., memory error). The simulation then continues until the "DONE" state is reached. At this point the output values are compared with the expected values.

[7] Note, the various "_Update" functions used in the ASRP schema are defined in the ASRP_FUNCTION schema, which is not shown.

Table 1. ARSP component simulation results

Name of Chart	Activity/State Name	Transition Paths			
		1	2	3	4
ARSP	ARSP	E_1	E_1	E_1	E_1
	@ARSP_CONTROL	E_2	E_2	E_2	E_2
ARSP_CONTROL	ARSP_START	E_3	E_3	E_3	E_3
	KEEP_PREVIOUS_VALUE	E_4	-	-	-
	ESTIMATE_ALTITUDE	-	E_4	-	-
	CALCULATE_ALTITUDE	-	-	E_4	-
	KEEP_PREVIOUS	-	-	-	E_4
	DONE	E_5	E_5	E_5	E_5



Variable Name	Injected Fault
FRAME_COUNTER	Yes
AR_COUNTER	No
K_ALT	No

Fig. 8. State transition diagram showing fault injection locale

Table 2 details the steps used for injecting faults by altering a system state variable (e.g., FRAME_COUNTER) at a certain state or so-called breakpoint (e.g., ARSP_START) during the simulation. "-" implies don't care and "*" indicates an estimated value in the table. The expected values of the output variables are not the same as the actual values of the output due to the state variable change. Again, the expected values are determined based on equations given in the requirements specification.

Table 2. Detailed fault injection data

	Variables	Before Execution	Expected Values	After Execution
Input	FRAME_COUNTER	2	2	2
	AR_STATUS	-	[1,1,0,0,0]	[1/0,1,0,0,0]
	AR_COUNTER	-1	-1	-1
Output	AR_STATUS	[1,0,0,0,0]	[1,1,0,0,0]	[1/0,1,0,0,0]
	K_ALT	[1,1,1,1,1]	[1,1,1,1,1]	[1,1,1,1,1]
	AR_ALTITUDE	[2000, -, -, -, -]	[2000, 2000, -, -, -]	[*, 2000, -, -, -]

Table 3. ARSP fault injection results

	Altered State Variable											
State where fault is injected	FRAME_COUNTER				AR_COUNTER				AR_STATUS			
	Path				Path				Path			
	1	2	3	4	1	2	3	4	1	2	3	4
ARSP_START	**x**	x	x	x	x	x	x	x	x	x	x	x
KEEP_PREVIOUS_VALUE	√	√	√	√	√	√	√	√	√	√	√	√
ESTIMATE_ALTITUDE	√	√	√	√	√	N/A	√	√	√	N/A	√	√
CALCULATE_ALTITUDE	√	√	√	√	√	√	x	√	√	√	√	√
KEEP_PREVIOUS	√	√	√	√	√	√	√	√	√	√	√	√
DONE	√	√	√	√	√	√	√	√	√	√	√	√

The fault injection results are described in Table 3 (bold 'x' indicates the aforementioned example described in Fig. 8). This table shows 72 test results (outputs) from 12 different simulation runs. An "x" indicates an incorrect output, √ indicates no defect and "N/A" indicates not applicable. The "State where fault is injected" column is the same state defined in the Statecharts model (also shown in Fig. 8). The result table indicates that all output values are incorrect when faults are injected to the ARSP_START state[8]. In addition, a fault injected into the CALCULATE_ALTITUDE state produces erroneous outputs. Therefore, one can conclude these two Statechart model states are the most vulnerable.

Based on the simulation results, the SRS was determined to be incomplete. To remedy the situation, the AR_FREQUENCY value must be bounded to prevent the AR_ALTITUDE value from exceeding its limit. To do this, one of the following conditions should be included:

1. $1 \leq AR\_FREQUENCY \leq AR\_COUNTER \times 75000$
2. $AR\_COUNTER = -1 \mid$
   $(0 \leq AR\_COUNTER \leq AR\_FREQUENCY/75000)$

Using Statemate, a GCS Activity-chart (Fig. 9) is developed inside of the GCS project. Activities are represented by rectangles and States are represented by rectangles with rounded corners. Every activity must have only one control state. The GCS activity (representing the GCS schema—see Fig. 10) interacts

[8] Any false/erroneous input given in the initial state causes incorrect output and the ARSP_START state is the initial state for the ARSP component.

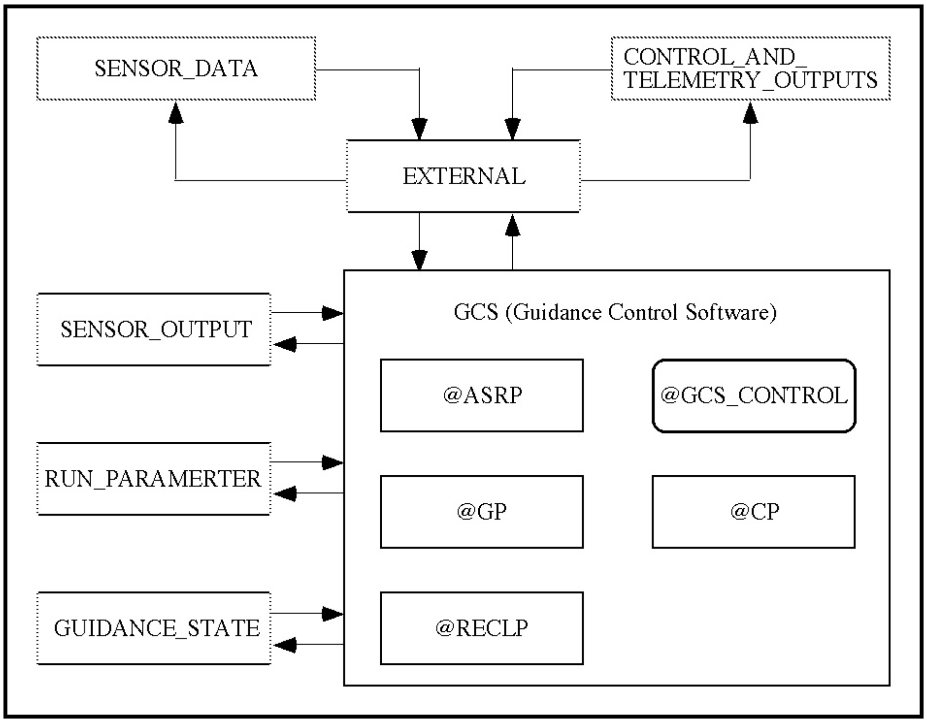
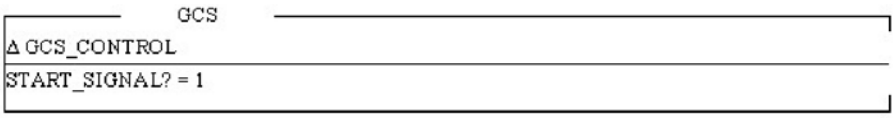


Fig. 9. GCS Activity-chart



GCS

Δ GCS_CONTROL

START_SIGNAL? = 1

Fig. 10. GCS schema

with four data stores (represented by rectangles with dotted vertical edges), which contain data definitions. The data stores contain the same variable definitions as in the corresponding Z schemas. The "@" symbol indicates a chart linked to a particular state or activity. For example, the @GCS_CONTROL state represents a link with the GCS_CONTROL Statechart. The @ARSP, @CP, @ GP, and @ RECLP activities represent the four GCS components and are linked to their own Activity-charts respectively.

The GCS project has GCS Activity-charts and four sub-Activity-charts. Each Activity-chart has one control state that is linked to a Statechart. Most of the Statecharts used for controlling activities are divided into several Statecharts, which use super-states to reduce complexity. Table 4 gives the execution orders of the GCS Statechart model, which are equivalent to the Z specification of the GCS system. The execution test results show that the Statecharts model does

Table 4. GCS excerpt high-level activity or state charts simulation results

Name of chart	Activity/State Name	Activity/State Transition Order					
GCS	@GCS_CONTROL	En_1	En_{33}				
	@ARSP	En_4	En_7				
	@GP	En_{14}	En_{17}				
	@RECLP	En_{24}	En_{27}				
	@CP	En_9	En_{12}	En_{19}	En_{22}	En_{29}	En_{31}
GCS_CONTROL	INITIALIZATION	En_2	En_3				
	@SUBFRAME1	En_5	En_{13}				
	@SUBFRAME2	En_{15}	En_{23}				
	@SUBFRAME3	En_{25}	En_{33}				
SUBFRAME1	RUN_ARSP	En_6	En_8				
	RUN_CP	En_{10}	En_{11}				
SUBFRAME2	RUN_GP	En_{16}	En_{18}				
	RUN_CP	En_{20}	En_{21}				
SUBFRAME3	RUN_RECLP	En_{26}	En_{28}				
	RUN_CP	En_{30}	En_{32}				

not have absorbing states or activities. Moreover, all of the activities and states are reachable and there is no inconsistency in the model. This result indicates that it is feasible to assess the overall structure of components integration using Z and Statecharts for completeness and consistency. The approach provides a way to deal with a complex set of requirements for a component based embedded control system symbiotically utilizing verification and validations tools (Z/Eves from ORA Canada [31] and Statemate from ilogix [19]).

4.2 Reliability Assessment of the ABS of a Passenger Vehicle

The increasingly common use of software embedded in critical systems has created the need to depend on them even more than before, and to measure just how dependable they are. Knowing that the system is reliable is absolutely necessary for safety-critical systems, where any kind of failure may result in an unacceptable loss of human life. This case study uses an analytical approach for estimating the reliability of a CBS system. It demonstrates our approach to estimating the reliability of the system by taking the architecture of the CBS system and the reliabilities of the individual components into consideration.

Anti-Lock Braking System Description. The system under study here is an embedded vehicle sub-system (including both hardware and software components). A complex embedded vehicle system (like the Anti-lock Braking System) is composed of numerous components and the probability that the system survives (efficient or acceptable degraded performance) depends directly on each of the constituent components.

Anti-lock Braking System (ABS) is an integrated part of the total vehicle braking system. It prevents wheel lockup during an emergency stop by modulating the brake pressure and permits the driver to maintain steering control

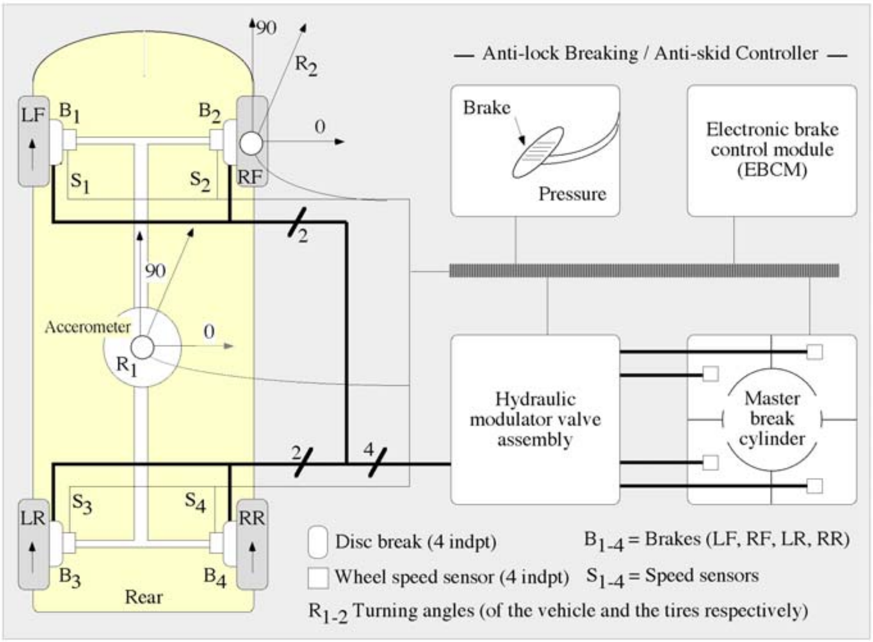

— Anti-lock Breaking / Anti-skid Controller —
Brake
Pressure
Electronic brake control module (EBCM)
Hydraulic modulator valve assembly
Master break cylinder
LF
RF
LR
RR
B1
B2
B3
B4
S1
S2
S3
S4
R1
R2
90
0
Accerometer
Rear
2
4
Disc break (4 indpt)
Wheel speed sensor (4 indpt)
B1-4 = Brakes (LF, RF, LR, RR)
S1-4 = Speed sensors
R1-2 Turning angles (of the vehicle and the tires respectively)

Fig. 11. Top-level schematic showing sensors/actuators and processing

while braking. Fig. 11 shows a top level schematic of the ABS. The ABS of a passenger vehicle is composed of the following components: (*i*) Wheel Speed Sensors—These measure wheel-speed and transmit information to an electronic control unit. (*ii*) Electronic Control Unit (Controller)—This receives information from the sensors, determines when a wheel is about to lock up and controls the hydraulic control unit. (*iii*) Hydraulic Control Unit (Hydraulic Pump)—This controls the pressure in the brake lines of the vehicle. (*iv*) Valves—Valves are present in the brake line of each brake and are controlled by the hydraulic control unit to regulate the pressure in the brake lines.

Stochastic Activity Network (SAN) Model. The ABS is modelled using SANs [3], which are a stochastic formalism used for performance modelling. Tools exist to automatically generate the underlying Markov chains from a high level representation of the system in the form of a SAN model. UltraSAN is an X-window based software tool for evaluating systems that are represented as SANs.

Assumptions. Modelling the ABS using SANs requires a number of simplifying assumptions. To allow a Markov chain analysis, the time to failure of all components is assumed to have an exponential distribution. This signifies that the distribution of the remaining life of a component does not depend on how

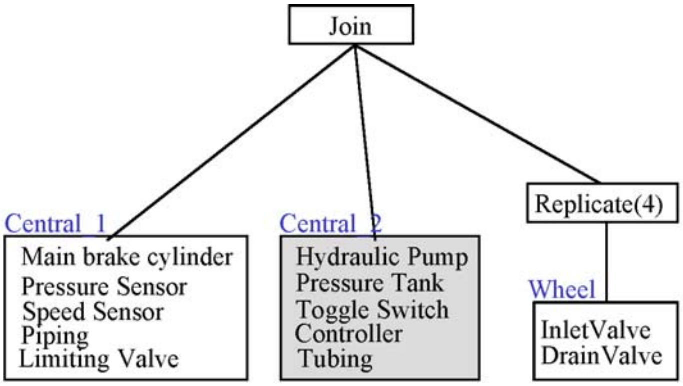


Fig. 12. The ABS composed SAN model

long the component has been operating. To consider the severity of failures, every component is assumed to operate in three modes: normal operation, degraded operation or causing loss of stability. To be able to model coincident failures, some correlation between failures of certain components (like controller and hydraulic pump) is assumed.

Composed SAN Model. The composed ABS model is shown in Fig. 12. The model consists of three individual SAN subnets: *Central_1*, *Central_2* and *Wheel*. The *Wheel* subnet is replicated four times to represent the four wheels of the vehicle. The division into these three categories is done to facilitate the representation of coincident failures. Such a distribution/categorization avoids replicating of subnets where unnecessary (for modelling severity and coincident failures) and thereby prevents the potential state explosion problem.

SAN Subnets Modelling Failure Severity and Coincident Failures. All subnets when combined to form the composed model share some common places: *degraded*, *LOS*, *LOV* and *halted*. The first three places represent the severity of failure, while the *halted* place is relevant in the context of the halting condition (i.e., system failure). The *Central_2* subnet is shown in Fig. 13. The presence of tokens in *degraded*, *LOS* and *LOV* represents the system operation under degraded mode, loss of stability and loss of vehicle respectively. The system is operating normally when there are no tokens in any of these three places.

The subnet is instantiated with a single token in the *central_2* place. The *central2_op* activity fires and deposits a token in each of the five places: *hydraulicPump*, *pressureTank*, *toggleSwitch*, *controller* and *tubing*. The portion of the subnet for the *controller* component is highlighted in Fig. 13 and discussed here in the context of severity of failures. The *controllerFail* activity models the failure of the controller. There are three possible outcomes of this activity. The *controller* either degrades (with probability 0.2, output gate *controllerDe-*

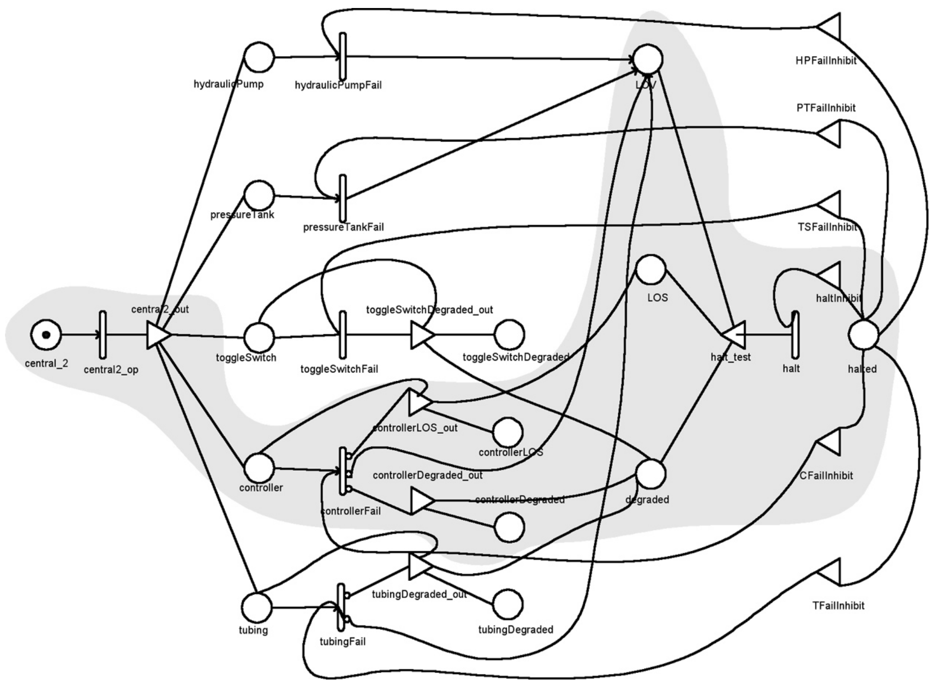


Fig. 13. Central_2 subnet with the controller component highlighted

graded_out), or causes loss of stability (with probability 0.4, output gate *controllerLOS_out*), or causes loss of vehicle (with probability 0.4, output to LOV). In the former two cases the controller continues to operate in a degraded manner, as is evident by the recycling back of the token to the *controller* place. Further, the failure rate in this situation increases by two (for degraded) and four (for loss of stability) orders of magnitude respectively. The code snippet that achieves this is shown in Table 5.

Coincident failures involving two components are represented by causing the failure of one component (degraded operation or loss of stability) to increase the failure rate of the dependent component. The degeneration of a component A to a degraded mode causes the failure rate of a "related" component B to increase by two orders of magnitude. The failure of component A to a lost stability mode causes the failure rate of a "related" component B to increase by four orders of magnitude. Table 5 shows the rates for the activities modelling the failure of the controller and the hydraulic pump (other component failure rates are modelled in a similar manner). Case 1, 2 and 3 represent the probabilities of the failure causing loss of vehicle, loss of stability and degraded mode respectively.

Since UltraSAN requires the failure rate to be specified in a single statement, the conditional operator available in the C programming language is used. Consider the *controllerFail* activity in Table 5. Since a degenerated tubing (i.e., in degraded mode) is assumed to affect the failure rate of the con-

Table 5. Activity rates model severity and coincident failures

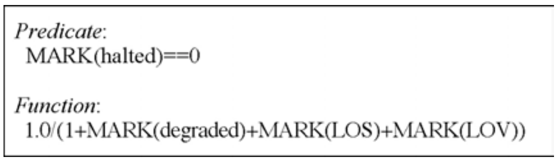
Activity	Rate	Probability		
		Case 1	Case 2	Case 3
controllerFail	MARK(controllerLOS)!=0? controllerRate*10000: (MARK(controllerDegraded)!=0 || MARK(tubingDegraded)!=0 ?controllerRate*100 :controllerRate)	0.4	0.4	0.2
hydraulicPumpFail	MARK(controllerLOS)!=0? hydraulicPumpRate*10000: (MARK(controllerDegraded)!=0 ?hydraulicPumpRate*100 :hydraulicPumpRate)	1.0	-	-

troller, if the number of tokens in the *tubingDegraded* place is not zero (i.e., MARK(tubingDegraded)!=0), the failure rate for the controller increases by two orders of magnitude (i.e., controllerRate*100). Similarly, for the *hydraulicPump-Fail* activity, it is assumed that a failed controller affects the failure rate of the hydraulic pump. Thus, the failure rate for the hydraulic pump increases by four orders of magnitude if the controller has failed causing loss of stability, and increases by two orders of magnitude if the controller is operating in a degraded mode.

Reliability Measure and Halting Condition. The required reliability measure is defined as a reward rate function. The reward rates for the SAN model are defined to take the degraded operation of the system into consideration.

Reward rates are specified using a predicate and a function. The function represents the rate at which the reward is accumulated in the states when the predicate evaluates to true. Fig. 14 shows the reward rate used to calculate reliability. As long as the system is functioning (i.e., not in an absorbing state), the reward accumulates as a function of the number of tokens in the *degraded*, *LOS* and *LOV* places. The function evaluates to 1.0 when there are no tokens in any of those three places indicating normal operation. The reliability is 0 when the system has stopped functioning (in an absorbing state). For all other states, the reliability ranges from 1.0 to 0.0 depending on how degraded the system is (indicated by the number of tokens in those three places).

This SAN model recycles tokens when the system is either operating in normal mode or degraded mode. Thus, it is necessary to explicitly impose a halting condition to indicate an absorbing state. The *halted* place common to all the subnets is used to specify the halting condition. Five or more tokens in *degraded*, or three or more tokens in *LOS*, or one or more token in *LOV*, cause a token to appear in *halted*. The presence of a token in this place is the indication of an absorbing state in the corresponding SAN. This is achieved by having an input condition on each activity stating that the activity is enabled only if there are no



Predicate:
MARK(halted)==0

Function:
1.0/(1+MARK(degraded)+MARK(LOS)+MARK(LOV))

Fig. 14. Reward rate to calculate reliability

tokens in the *halted* place (i.e., MARK(*halted*)==0). The presence of a token in *halted* thus disables all the activities in the model, thereby causing an absorbing state.

Reliability Analysis Results. The reliability of the system at time t is computed as the expected instantaneous reward rate at time t. To determine the reliability of the ABS, transient analysis of the developed SAN models was carried out using the instant-of-time transient solver available in the UltraSAN tool. The reliability was measured between 0 and 5×10^4 hours. The time duration was deliberately conservative, even though the average life span of a passenger vehicle ranges from 3000–9000 hours, the reliability measures were determined for up to 5×10^4 hours.

The reliability measure was predicted at 11 different points along the range of 0 to 5×10^4 hours. The interval between the points did not remain constant along the entire time range and therefore the X-axis is not linear and should be taken into account when viewing the results graphs. The expected values of reliability at various time instances were plotted as a function of time. In Fig. 15, the Y-axis gives the measure of interest—the reliability; while the time range (0 to 5×10^4 hours) is shown along the X-axis. As expected, the reliability steadily decreases with time. The dashed line indicates the reliability function when coincident failures are modelled and the complete line indicates the reliability function when coincident failures are not modelled.

The reliability functions diverge perceptibly after around 1000 hours of operation, and the difference continues to increase with time. At 5×10^4 hours, the reliability has dropped down to 0.21 when coincident failures are modelled, and down to 0.30 when coincident failures are not modelled, a difference of 0.09 in reliability in the two cases within 5×10^4 hours. Considering the time period approximately around the expected lifetime of the vehicle (3,000-9,000 hours), the difference in reliability after 5000 hours of operation is approximately 0.0253 and after 10^3 hours is 0.0493. This clearly indicates that representing severity and coincident failures in the model contributes to predicting the system reliability that may be closer to how the real system will behave considering the underlying assumptions.

The Mean Time to Failure calculated at 5×10^4 hours in the case where coincident failures are not modelled is approximately 29,000 hours, and in the case where coincident failures are modelled is approximately 25,000 hours, a difference of 4,000 hours. It is important to realize that these results are only for

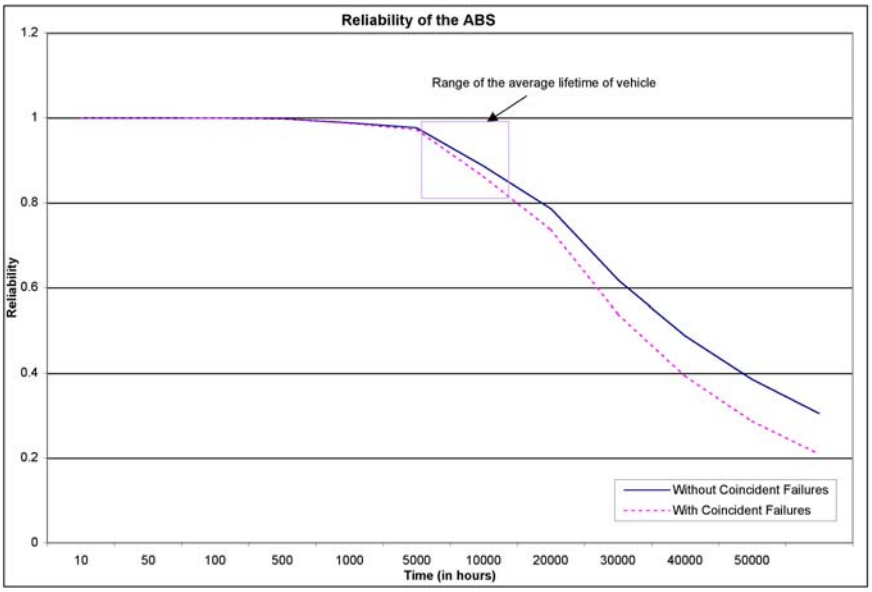


Fig. 15. Reliability results for severity and coincident failures

the limited number of coincident failures and levels of severity that have been modelled. Clearly, modelling severity and coincident failures have a significant contribution in determining the system reliability at any given instant of time.

Validity Concerns. A model is always a compromise between precision and simplicity. How closely a model mirrors its originator or the vision of the system is in direct conflict with how easily and efficiently the model can be analyzed (i.e., solved with respect to its predicted behavior). The models described were built incrementally to achieve the best balance between faithfulness to the real system and keeping the model tractable at the same time. As a result, models of higher fidelity (more realistic) were created progressively.

5 Challenges

The CBSD paradigm has emerged from the concept of building software out of components. Using components is not such a new concept, as traditional design methods have always tried to identify parts (modules, classes, functions, etc.) that are appropriate and effectively address the principle of separation of concerns (moderated by suitable measures of cohesion and coupling). Moreover, the notion of packaging software in such a way that makes it reusable is not new either (e.g., generic packages/instantiation, inheritance/polymorphism, etc.). Notwithstanding, the CBSD paradigm, as a new sub-discipline of software engineering, has been recognized as an important new development that brings support for developing dependable and maintainable high integrity systems as assemblies of components as well as strategies for developing components as reusable entities that are flexible, and extensible. CBSD faces many challenges, some of which include [6]: component trustworthiness and certification [28], composition predictability [40], requirements management and component selection [18, 22, 23], long-term management of CBS systems [6], development process models, component configurations, versioning and hierarchy (e.g., nesting that causes lack of conceptual integrity [5]), dependable safety-critical systems and CBS Engineering including trustworthy, scalable, cost-effective tool support. These are some of the current challenges. The success of CBSD will heavily depend on further research and the emergence of standard formalized frameworks (e.g., FMB methods) that can endure the aforementioned challenges in the critical disciplines that support those essential activities related to CBS systems development.

6 Conclusions and Future Work

As the demand for more flexible, adaptable, extensible, and robust high integrity CBS systems accelerates, adopting new software engineering methodologies and development strategies becomes critical. Such strategies will provide for the construction of CBS systems that assemble flexible software components written at different times by various developers. Traditional software development strategies and engineering methodologies, which require development of software systems from scratch, do not adequately address these needs. CBSD works by developing and evolving software from selected reusable software components, then assembling them within appropriate software architectures. CBSD relies heavily on explicitly defined architectures and interfaces, which can be evaluated using our FMB framework. CBSD has the potential to:

- Significantly reduce the development cost and time-to-market of enterprise CBS systems,
- Enhance the reliability of CBS systems using FMB methods where each reusable component is assessed in terms of covering and satisfying requirements (e.g., complete, consistent, etc.) and undergoes reliability analysis as deemed necessary, especially for high integrity system deployment,
- Improve the maintainability of CBS systems by allowing new, higher-quality components to replace old ones; and
- Enhance the quality of CBS systems where application-domain experts develop components, while software engineers specializing in CBSD, assemble the components.

CBSD is in the very first phase of maturity. CBSD using FMB methods is even less mature. Nevertheless, formal approaches are recognized as powerful tools that can significantly change the development of software and software use in general. Tools and frameworks for building applications and systems by means of component assembly will be tantamount in meeting the challenges ahead. Standardization of domain-specific components on the interface level will make it possible to build applications and systems from components purchased from different vendors. Work on standardization in different domains continues, (e.g., the OPC Foundation [30], is working on a standard interface to make possible interoperability between automation and control applications, field systems and devices and business and office applications)[9].

The result of the first study showed how to construct a complete and consistent specification using this method (Z-to-Statecharts). The process uncovered incomplete and inconsistent requirements that were associated with ambiguities (i.e., a reader's interpretation of the natural language and its inherent lack of precision). We have demonstrated our approach can help to identify ambiguities that result in incorrectly specified artifacts (i.e., in this case, requirements).

In the second study, the characteristics of failure severity and coincident failures were successfully incorporated into the model developed for the ABS of a passenger vehicle. The models evolved over successive iterations of modelling, increasingly refined in their ability to represent different factors that affect the measure of interest (i.e. system reliability). This refinement process, we claim, gives a (potentially) more realistic model. For example, the analyses showed that the reliability predictions were different (i.e., deteriorated) when the non-functional characteristics of severity and coincident failures were incorporated. However, because the model is an abstraction of the real world problem, predictions based on the model should be validated against actual measurements observed from the real phenomena. This study can be the basis of numerous other studies, building up on the foundation provided and investigating other areas of interest (e.g., validating predictions against field observations, or finding a more realistic level of abstraction combined with a higher degree of complexity, and solving the models using supercomputers).

Acknowledgments

The authors wish to thank Dr. Tom Potok, who is the Applied Software Engineering Laboratory Group Lead at ORNL, and Dr. Stefan Greiner at Daimler-Chrysler (RIC/AS) for their help and critique. Also, Kelly Hayhurst, who is a research scientist at NASA Langley, working in the area of design correctness and certification, provided immeasurable, crucial and essential support with respect to the GCS Requirements Specification. Her help and encouragement is *deeply* appreciated.

References

1. Arlat, J., K. Kanoun, and J.-C. Laprie, Dependability Modeling and Evaluation of Software Fault-Tolerant Systems. IEEE Transactions on Computers, 1990. 39(4):504–513.

[9] Support for the exchange of information between components, applications, and systems distributed over the Internet will be further developed. Works related to XML [13] will be further expanded.

2. Clements, P., Bass, L., Kazman, R., and Abowd, G., Predicting Software Quality by Architecture-Level Evaluation. Component-Based Software Engineering: Selected Papers from the Software Engineering Institute, 1995: pp. 19–25.
3. Couvillion, J., Johnson, R., Obal II, W. D., Qureshi, M. A., Rai, M., Sanders, W. H., and Tvedt, J. E., Performability Modeling with UltraSAN. IEEE Software, 1991. 8(5):69–80.
4. Cox, P.T. and B. Song. A Formal Model for Component-Based Software. in Proc. of 2001 IEEE Symosium on Visual/Multimedia Approaches to Programming and Software Engineering. 2001. Stresa, Italy: IEEE. pp. 304–311.
5. Crnkovic, I., Larrson, M., Kuster, F. J., and Lau, K., Databases and Information Systems, Fourth International Baltic Workshop, Selected Papers. 2001: Kluwer Academic Publishers. pp. 237–252.
6. Crnkovic, I., Component-based Software Engineering—New Challenges in Software Development. Software Focus, 2002. 2(4):127–133.
7. Czerny, B., Integrative Analysis of State-Based Requirements for Completeness and Consistency, in Computer Science. 1998, Michigan State University.
8. Dugan, J.B. Experimental analysis of models for correlation in multiversion software. in Proc. of 5th Int'l Symposium on Software Reliability Engineering. 1994. Los Alamitos, CA: IEEE Computer Society. pp. 36–44.
9. Eckhardt, D.E. and L.D. Lee, Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors. IEEE Transactions on Software Engineering, 1985. 11(12):1511–1517.
10. Fitch, D., Software Safety Engineering (S2E) Program Status. 2001. `sunnyday.mit.edu/safety-club/fitch.ppt`.
11. Gay, F.A., Performance Evaluation for Gracefully Degrading Systems. in Proc. of 9th Annual Int'l Symposium on Fault-Tolerant Computing (FTCS-9). 1979. Madison, Wisconsin: IEEE Computer Society. pp. 51–58.
12. Glass, R.L., Software Reliability Guidebook. 1979, Englewood Cliffs, New Jersey: Prentice-Hall. pp. 241.
13. Griss, M.L. and G. Pour, Accelerating Development with Agent Components. IEEE Computer, 2001. 34(5):37–43.
14. Hamlet, D., D. Mason, and D. Woit. Theory of Software Reliability Based on Components. In Proc. of 23rd International Conference on Software Engineering (ICSE'01). 2001. Toronto, Canada: IEEE Computer Society. pp. 361–370.
15. Harel, D., Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 1987. 8:231–274.
16. Harel, D. and M. Politi, Modeling Reactive Systems with Statecharts. 1998: McGraw Hill.
17. Hecht, M., D. Tang, and H. Hecht. Quantitative Reliability and Availability Assessment for Critical Systems Including Software. 12th Annual Conference on Computer Assurance. 1997. Gaitherburg, Maryland.
18. Heineman, G. and W. Councill, Component-based Software Engineering: Putting the Pieces Together. 1 ed. 2001, Boston: Addison Wesley. 818.
19. I-Logix, I-Logix Inc. Website. 2002. `www.ilogix.com`.
20. Jacky, J., The Way of Z: practical programming with formal methods. 1997: Cambridge University Press.
21. Kanoun, K. and M. Borrel. Dependability of Fault-Tolerant Systems—Explicit Modeling of the Interactions Between Hardware and Software Components. in Proc. of 2nd Int'l Computer Performance and Dependability Symposium (IPDS). 1996. Urbana-Champaign: IEEE Computer Society. pp. 252–261.

22. Kim, H.Y., Validation of Guidance Control Software Requirements Specification for Reliability and Fault-Tolerance, in School of Electrical Engineering and Computer Science. 2002, Washington State University: Pullman. pp. 76.
23. Kotonya, G. and A. Rashid. A strategy for Managing Risks in Component-based Software Development. in Proc. of 27th Euromicro Conference. 2001. Warsaw, Poland: IEEE Computer Society. pp. 12–21.
24. Leveson, N., Safeware – system safety and computers. 1995: Addison Wesley.
25. Littlewood, B. and D.R. Miller, Conceptual Modeling of Coincident Failures in Multiversion Software. IEEE Transactions on Software Engineering, 1989. 15(12):1596–1614.
26. Littlewood, B. and L. Strigini. Software reliability and dependability: a roadmap. in Proc. of International Conference on Software Engineering. 2000. Limerick, Ireland: ACM Press. 22. pp. 175–188.
27. Lo, J.-H., Kuo, S.-Y., Lyu, M. R., and Huang, C.-Y. Optimal Resource Allocation and Reliability Analysis for Component-Based Software Applications. Computer Software and Applications Conference, COMPSAC'02. 2002. Oxford, England: IEEE Computer Society.
28. Morris, J., Lee, G., Parker, K., Bundell, G., and Chiou, P. L., Software Component Certification. IEEE Computer, 2001. 34(9):30–36.
29. Nicola, V.F. and A. Goyal, Modeling of Correlated Failures and Community Error Recovery in Multiversion Software. IEEE Transactions on Software Engineering, 1990. 16(3):350–359.
30. OPC Foundation. 2002. `http://www.opcfoundation.org`.
31. ORA, ORA Canada Website. 2002. `http://www.ora.on.ca/`.
32. Popstojanova, K.G. and K. Trivedi. Stochastic Modeling Formalisms for Dependability, Performance and Performability. Performance Evaluation: Origins and Directions. 2000: Springer-Verlag. LNCS 1769. pp. 403–422.
33. Pradham, D.K., Fault-Tolerant Computer System Design. 1996: Prentice Hall.
34. Sahner, R.A. and K. Trivedi. A hierarchical, combinatorial-Markov model of solving complex reliability models. in Proc. of ACM/IEEE Fall Joint Computer Conference. 1986. Dallas, Texas: IEEE Computer Society. pp. 817–825.
35. Sedigh-Ali, S. and R.A. Paul. Metrics-guided quality management for component-based software systems. in Proc. of Computer Software and Applications Conference, COMPSAC'01. 2001. Chicago: IEEE Computer Society. pp. 303–308.
36. Sherif, M.Y., C. Bojan, and H.A. Hany. A Component-Based Approach to Reliability Analysis of Distributed Systems. in Proc. of the 18th IEEE Symposium on Reliable Distributed Systems. 1999. Lausanne, Switzerland. pp. 158–167.
37. Veryard, R., Software Component Quality. 1997.
`http://www.users.globalnet.co.uk/~rxv/CBDmain/DIPQUE.htm`.
38. Voas, J. and J. Payne, Dependability Certification of software components. Journal of Systems and Software, 2000. 52:165–172.
39. Wallin, C., Verification and Validation of Software Components and Component Based Software Systems, in Building Reliable Component Based Systems, I. Crnkovic and M. Larrson, Editors. 2002, Artech House.
40. Wallnau, K. and J. Stafford. Ensembles: Abstractions for a New Class of Design Problem. in Proc. of 27th Euromicro Conference. 2001. Warsaw, Poland: IEEE Computer Society. pp. 48–55.
41. Woodcock, J. and J. Davies, Using Z: Specification, Refinement, and Proof. Series of Computer Science. 1996: Prentice Hall International.