Engineering Reliable Service Oriented Architecture:

Managing Complexity and Service Level Agreements

Nikola Milanovic
Model Labs – Berlin, Germany

Information Science
REFERENCE

Senior Editorial Director: Kristin Klinger
Director of Book Publications: Julia Mosemann
Editorial Director: Lindsay Johnston
Acquisitions Editor: Erika Carter
Development Editor: Joel Gamon
Production Coordinator: Jamie Snavely
Typesetters: Keith Glazewski & Natalie Pronio
Cover Design: Nick Newcomer

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Engineering reliable service oriented architecture : managing complexity and service level agreements / Nikola Milanovic, editor.
p. cm.
Includes bibliographical references and index.
Summary: "This book presents a guide to engineering reliable SOA systems and enhances current understanding of service reliability"--Provided by publisher.
ISBN 978-1-60960-493-6 (hardcover) -- ISBN 9781609604943(ebook) 1. Service-oriented architecture (Computer science) 2. Computer networks--Reliability. I. Milanovic, Nikola.
TK5105.5828.E54 2011
004.6--dc22
2010033596

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 14
Complexity Analysis at Design Stages of Service Oriented Architectures as a Measure of Reliability Risks

Muhammad Sheikh Sadi
Curtin University of Technology, Australia

D. G. Myers
Curtin University of Technology, Australia

Cesar Ortega Sanchez
Curtin University of Technology, Australia

ABSTRACT

Tremendous growth in interest of Service oriented Architectures (SOA) triggers a substantial amount of research in its reliability assurances. To minimize the risks of these types of systems' failure, it is a requirement to flag those components of SOA that are likely to have higher faults. Clearly, the degree of protection or prevention of faults mechanism is not same for all components. This chapter proposes the usage of metrics that are simply heuristics and are used to scan the system model and flag complex components where faults are more likely to take place. Thus the metric output is some priority or it is a measure of likelihood of faults in a component. This chapter then suggests the designers for possible changes in the design if there remains any risk(s) of degradation of desired functionalities.

INTRODUCTION

Service Oriented Architecture (SOA), which is prompting a variable shift in the distributed computing history, is subsisted in modern computing environments and is at critical risks due to permanent and transient faults in computing structures (Lakhal, Kobayashi, & Yokota, 2006). Permanent faults such as node stuck-at-1/0, transistor open, shorted transistors, etc., arise during fabrication or result from aging, and destroy the intended function of the circuit (Timor, Mendelson, Birk,

DOI: 10.4018/978-1-60960-493-6.ch014

& Suri, 2008). Transient faults, in contrast, do not damage the chips physically but are catastrophic for desired functionalities of the system (Mukherjee, Emer, & Reinhardt, 2005), (F. Wang, 2008), (Iyer, Nakka, Kalbarczyk, & Mitra, 2005). Both of these faults are severe for those SOA where reliability is a great concern(Narayanan & Xie, 2006), (Tosun, 2005). For example, online banking transactions where a single bit change (1→0) in the most significant bit of the data storing register, may cause a huge difference in balance. Due to processor scaling, reduction in operation voltages, exponential growth of number of transistors in a single chip, increase in clock frequencies, and/or device shrinking, the rate of these faults are moving upwards day by day (Saggese, Wang, Kalbarczyk, Patel, & Iyer, 2005), (Crouzet, Collet, & Arlat, 2005).

Prior research to cope with transient faults (which in turn create soft errors) mostly focuses on post-design phases, such as circuit level solutions, logic level solutions, spatial redundancy, temporal redundancy, and/or error correction codes. Early detection and correction of such problems during the design phase is much more likely to be successful than detection once the system is operational (Cortellessa et al., 2005). Estimating reliability (or at least identifying failure-prone components) early in the life-cycle of a design is therefore preferable (Jurjens & Wagner, 2005), (A. Bondavalli, 2001). From a pure dependability viewpoint, complex components attract more attention of soft errors tolerant approaches than others do, since reliability of a system is correlated with the complexity of the system (Khoshgoftaar, 1996), (Yacoub & Ammar, 2002). To minimize the risks of system failure, it is a requirement to flag those components of SOA that are likely to have higher faults. Clearly, the degree of protection or prevention of faults mechanism is not same for all components. Hence, an approach is needed at the design stage to highlight those complex components and suggest the designers for possible changes in the design if there remains any risk of affecting desired functionalities.

This chapter flags complex components at early design stage and investigates how to encourage the designer to explore changes that could be made in the existing model. For example, how the complexities of the components could be minimized, or how these components could be replaced with alternatives and/or with less complex components are examined. The objective is to keep the functionality and other constraints of the system unaffected or to make a trade-off between them, with the goal to minimize the reliability risks. Case studies illustrate the effectiveness of the proposed approach in determining components' complexity ranking and then lowering their complexities. The model is expressed in Unified Modeling Language (UML) since this allows the modeler to describe different views on a system, including the physical layer (Wood, Akehurst, Uzenkov, Howells, & McDonald-Maier, 2008), (L. Wang, Wong, & Xu, 2007).

EXISTING WORK ON SOFT ERRORS RISKS MINIMIZATION

Software based approaches to tolerate soft errors include redundant programs to detect (Mukherjee, Kontz, & Reinhardt, 2002), (Reinhardt & Mukherjee, 2000), (Rotenberg, 1999), (Smolens et al., 2004) and/or recover from the problem (Vijaykumar, Pomeranz, & Cheng, 2002), duplicating instructions (Oh, Shirvani, & McCluskey, 2002), (Reis, Chang, Vachharajani, Rangan, & August, 2005), task duplication (Xie, Li, Kandemir, Vijaykrishnan, & Irwin, 2004), dual use of super scalar data paths (Ray, Hoe, & Falsafi, 2001), and Error detection and Correction Codes (ECC) (Chen & Hsiao, 1984). Chip level Redundant Threading (CRT) (Mukherjee et al., 2002) used a load value queue such that redundant executions can always see an identical view of memory. Although the load value queue

produced an identical view of memory for both leading and trailing threads, integrating this into the chip multiprocessor environment requires significant changes. In (Reinhardt & Mukherjee, 2000), the authors described the concept of sphere of replication in aiding the design and discussion of fault tolerant Simultaneously and Redundantly Threaded (SRT) processors. The parts of the computer system that fall outside the sphere are not replicated and must be protected by other means such as information redundancy. AR-SMT (Active-stream/Redundant-stream Simultaneous Multithreading) (Rotenberg, 1999) increases the memory requirement and bandwidth pressure two times, since both threads required accessing the cache and individual memory. Doubling the memory may stress the memory hierarchy and degrade performance. Walcott et al. (Walcott, Humphreys, & Gurumurthi, 2007) used redundant multi threading to determine the architectural vulnerability factor, and Shye et al. (Shye, Blomstedt, Moseley, Janapa Reddi, & Connors, To be Appeared) used process level redundancy to detect soft errors. In redundant multi threading, two identical threads are executed independently over some period and the outputs of the threads are compared to verify the correctness. EDDI (Oh et al., 2002), and SWIFT (Reis et al., 2005) duplicated instructions and program data to detect soft errors. Both redundant programs and duplicating instructions create higher memory requirements and increase register pressure. Error detection and Correction Codes (ECC) (Chen & Hsiao, 1984) adds extra bits with the original bit sequence to detect error. Using ECC to combinational logic blocks is complicated, and requires additional logic and calculations with already timing-critical paths.

Hardware solutions for soft errors mitigation mainly emphasize circuit level solutions, logic level solutions and architectural solutions. At the circuit level, gate sizing techniques (Park & Kim, 2008), (Miskov-Zivanov & Marculescu, 2006), (Quming & Mohanram, 2004) increasing capacitance (Oma, Martin, Rossi, & Metra, 2003), (STMicroelectronics, 2003), resistive hardening (Rockett, 1992) are commonly used to increase the critical charge (Q_{crit}) of the circuit node as high as possible. However, these techniques tend to increase power consumption and lower the speed of the circuit. Logic level solutions (S. Mitra, 2006), (Ming Zhang, 2006), (M. Zhang et al., 2006) mainly propose detection and recovery in combinational circuits by using redundant or self-checking circuits. Architectural solutions mainly introduce redundant hardware in the system to make the whole system more robust against soft errors. They include dynamic implementation verification architecture (DIVA) (Austin, 1999), and block-level duplication used in IBM Z-series machines (Meaney, Swaney, Sanda, & Spainhower, 2005). DIVA (Austin, 1999) in its method of fault protection assumed that the checker is always correct and it proceeds using the checker's result in case of a mismatch. So, faults in the checker itself must be detected through alternative techniques.

Hardware and software combined approaches (Gold et al., 2005), (Krishnamohan, 2005), (Vijaykumar et al., 2002), (Mohamed, Chad, Vijaykumar, & Irith, 2003), (Xie et al., 2004), (Srinivasan, Adve, Bose, & Rivers, 2004), (Rashid, Tan, Huang, & Albonesi, 2005) use the parallel processing capacity of chip multiprocessors (CMPs) and redundant multi threading to detect and recover the problem. (Mohamed et al., 2003) shows Chip Level Redundantly Threaded Multiprocessor with Recovery (CRTR), where the basic idea is to run each program twice, as two identical threads, on a simultaneous multithreaded processor. One of the more interesting matters in the CRTR scheme is that there are certain faults from which it cannot recover. If a register value is written prior to committing an instruction, and if a fault corrupts that register after the committing of the instruction, then CRTR fails to recover from that problem. In Simultaneously and Redundantly Threaded processors with Recovery (SRTR) scheme (Vijaykumar et al., 2002), there is a probability of

fault corrupting both threads since the leading thread and trailing thread execute on the same processor. Others (Krishnamohan, 2005), (Xie et al., 2004), (Srinivasan et al., 2004), (Rashid et al., 2005) have followed similar approaches. However, in all cases the system is vulnerable to soft error problems in key areas. In software-based approaches, the complex use of threads presents a difficult programming model. In hardware-based approaches, duplication suffers not only from overhead due to synchronizing duplicate threads, but also from inherent performance overhead due to additional hardware. Moreover, these post-functional design phase approaches can increase time delays and power overhead without offering any performance gain.

Few approaches (Chidamber & Kemerer, 1994), (Harrison, Counsell, & Nithi, 1998) dealt with the static complexities of the system as a risk assessment methodology to minimize the risks of faults. (McCabe, 1976) introduced Cyclomatic complexity, which is measured based on program graphs. However, these static approaches do not deal with the matter of how a module functions in its executing environment. A fault may not manifest itself into a failure if never executed. (Cortellessa et al., 2005), and (Yacoub & Ammar, 2002) defined dynamic metrics that include dynamic complexity and dynamic coupling metrics to measure the quality of software architecture. To assess the severity of the components they have defined only three levels of system failure. However, in real life scenarios, only three severity levels are not sufficient to represent several possible failure modes. Criticality analysis at the sub-system level along with failure Mode and Effect Analysis (FMEA) is also becoming popular in fault tolerant research. A few common methods for assessing criticality in FMEA are Risk Priority Number (RPN) (Bowles, 2004), the MIL_STD 1629A Criticality Number ranking (author, 1984), and the multi-criteria Pareto ranking (Bowles, 1998). However, difficulties in calculating failure rate values or probability of failure make Criticality Number ranking, and the multi-criteria Pareto ranking unpopular to researchers. (Sherer, 1988) has shown a risk assessment methodology by measuring the consequences of errors in different modules. However, the high complexity of the method in real-life applications makes it obsolete. Moreover, the method is applied at the later stages of the system design, which can mean a huge cost increase.

A METHODOLOGY TO MEASURE AND REDUCE COMPONENT'S COMPLEXITY

Complexity analysis does not measure the impact of components in system functionality, but rather shows the rank of likelihood of encountering errors among the components. Some empirical studies have found a correlation between the number of errors in a system and the complexity of the system (Khoshgoftaar, 1996), (Ammar, Nikzadeh, & Dugan, 1997). (Cortellessa et al., 2005) also pointed out that the probability of encountering errors is proportional to the complexity of the system. To minimize the reliability risks, it is therefore necessary at the early design phase to flag complex components that are likely to have higher faults. This paper highlights these components by an assessment of execution time via simulation and the Message-In-and-Out frequency. The details of these metrics are given below.

Execution Time during Simulation

The Failure-In-Time (FIT) of a system due to soft error is proportional to the fraction of time in which the system is susceptible to soft error if the circuit type, transistor sizes, node capacitances, temperature and so forth are kept constant (Nguyen, Yagil, Seifert, & Reitsma, 2005). Hence, the fractional time that a component uses in the execution of a system can flag the soft error proneness of that component. Using Execution Time

(ET) during simulation to measure a component's complexity is a novel approach. Components are executed for a specific operation. Users can specify any operation that seems to be involved with all components. The longer duration to perform the selected operation implies that the component is being used more frequently and/or that it is experiencing many state changes. A soft error occurs at any access point of these components can spread towards all communicating components through the large number of behavioural linkages until the soft error affected component remains in execution. Hence, the likelihood of soft error may be increased if the component takes a longer ET. The method of measuring ET during simulation (to perform an operation by a component) can be shown as follows. Component state S is a function of time: S (t) where t denotes time. An external function F () is required to be executed to perform the operation $F(S(t_i)) \rightarrow S(t_j)$): where $S(t_i)$ is the state of a component at t_i and $S(t_j)$ is the state of that component at t_j. Hence, ET, to execute the function F () that changes the state of the pth component from $S(t_i)$ to $S(t_j)$, is:

$$ET_p\ (F\ (S\ (t_i))\ \rightarrow\ S\ (t_j)) = \sum_{j=1}^{n} d_{pj} \qquad (1)$$

where n is the total number of state changes in the pth component's behaviour execution and d_{pj} is the duration in the jth slot of changing states of pth component.

Since UML does not specify an action model, Telelogic Rhapsody (Telelogic, 2009) is used to gain execution data via simulation. The model is executed in tracing mode. Several tracing commands are used to execute the model. The state transition times for the components are saved to a log file. At the end of the simulation, that log file is analysed to calculate the total ET of the components to perform a selected operation.

Message-In-And-Out Frequency

In object-oriented designs, components are often interdependent. Hence, a failure or error can easily propagate to other components. The malfunctioning behaviour of a component in a high interdependent design cannot be easily isolated. Therefore, this dependence is considered as a valuable measurement for both "a posteriori" and "a priori" analysis (Hitz & Montazeri, 1995). A posteriori analysis is conducted to trace those design aspects that were more likely to produce errors and hence correlate errors with design quality metrics. A priori analysis makes use of this dependence measurement to assess the reliability of designs in an early development phase. This research accepts a priori analysis since it saves both costs and time. In a system model (assumed in UML), components communicate with each other by message passing among them. The number of messages from and to a component shows the measure of dependence with other components. Components with more dependence could easily manifest themselves into failure of the system because services of these components are frequently accessed by other components (Yacoub & Ammar, 2002).

To determine the error proneness, a component's Message-In-and-Out frequency (MIO), which is the ratio of number of messages from and to a component in a scenario and the total number of messages in that scenario, is calculated. More specifically, a component with higher Message-In-and-Out frequency (MIO) is more likely to cause changes in the whole system if there arise any architectural or behavioral change in that component. Define MIO_{i_k} as the MIO for ith component in kth scenario. $M_{(i,j)}$ is the message between component i and component j (where j=1,….,m, $i \neq j$, and m is the number of messages from ith component to other components) in kth scenario, and n_k is the total number of messages,

communicating among all the components, in that scenario. Then, MIO_i can be derived as:

$$MIO_{i_k} = \frac{|\sum_{j=1}^{m} M_{(i,j)} \mid i \neq j|}{n_k} \quad (2)$$

For each component, Total MIO (TMIO) in all possible different scenarios can be calculated using (3). TMIO for ith component is:

$$TMIO_i = \sum_{k=1}^{n'} P(Sc_k) MIO_{i_k} \quad (3)$$

where n' is the total number of scenarios in Handset system, $P(Sc_k)$ is the probability of kth Scenario in that system, and MIO_{i_k} is the MIO for ith component in kth scenario.

Overall Complexity

The Overall Complexity of the ith Component (OCC_i) is the summation of different complexity factors for that component. The equation is:

$$OCC_i = ET_i + TMIO_i \quad (4)$$

where ET_i and $TMIO_i$ are Execution Time, and Message-In-and-Out frequency for the ith component. Since, ET_i and $TMIO_i$ are independent on each other, OCC_i is calculated using the summation of these two factors. For simplicity, the weights of ET and TMIO in measuring total value of complexities are assumed as equal.

Lowering the Complexities of the Components

Component complexity suggests to the designer where in the system design, changes are necessary or helpful to minimize soft errors risk. These changes can be made by applying a suitable approach where he/she may change the architecture or behavioural model of the component to lower its complexity. Refactoring is a good candidate for this type of approach. The purpose of refactoring is to alter the model based on the user's requirements by keeping the functionality and other constraints of the system unaffected. In software engineering, "refactoring source code" means improving it without changing its overall results and is sometimes informally referred to as "cleaning it up" (Wikipedia, 2009). Refactoring neither fixes bugs nor adds new functionality, though it might precede either activity; rather, it improves the understandability of the code, changes its internal structure and design, and removes dead code. UML model refactoring is the equivalent of source code refactoring at the model level with the objective of preserving the model's behaviour (Hosseini & Azgomi, 2008), (Gerson, Damien, Yves Le, & Jean-Marc, 2001). It re-structures the model to improve quality factors, such as maintainability, efficiency, fault tolerance, etc., without introducing any new behaviour at the conceptual level. As the software and hardware system evolves, almost each change of requirements imposed on a system requires the introduction of small adaptations to its design model (Dobrzanski & Kuzniarz, 2006), (Boger, Sturm, & Fragemann, 2003, Revised Papers (Lecture Notes in Computer Science Vol.2591)). However, the designers face challenges to this adaptation by a single modification in the model. A possible solution to this problem can be to provide designers with a set of basic transformations so maintaining model functionality. This set of transformations is known as refactoring, which can be used gradually to improve the design (Dobrzanski & Kuzniarz, 2006). A detailed taxonomy of model transformations has been presented by (Mens & Van Gorp, 2006), (Mens, 2006). Model refactoring can be made by replacing components with ones that are more elegant, merging/splitting the states keeping the behaviour unchanged, altering code readability or understandability, formal concept analysis,

Figure 1. An example Statechart of 'user's access to server' before refactoring

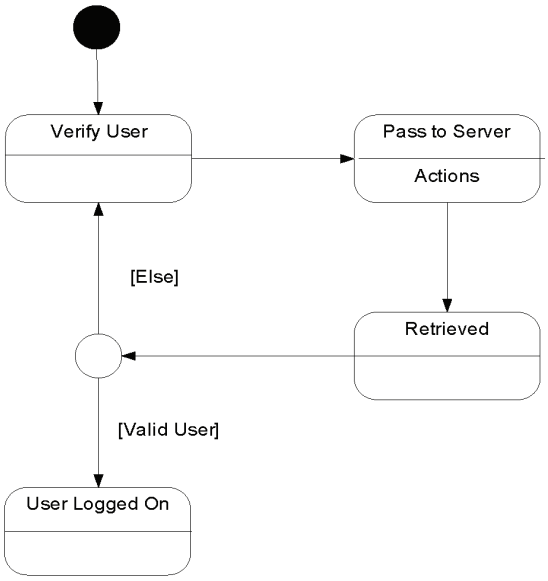
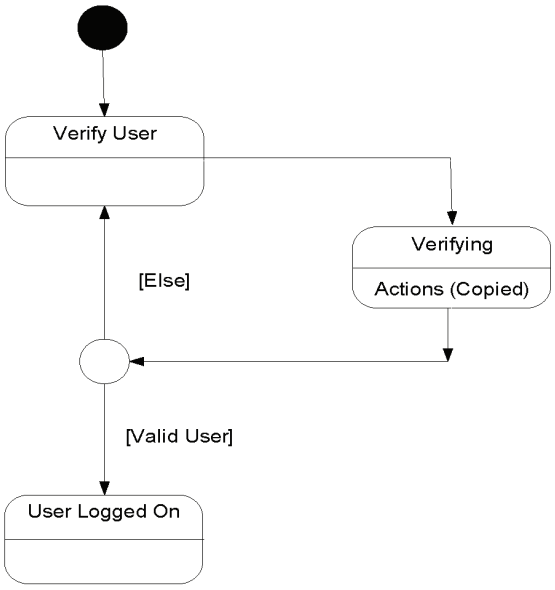


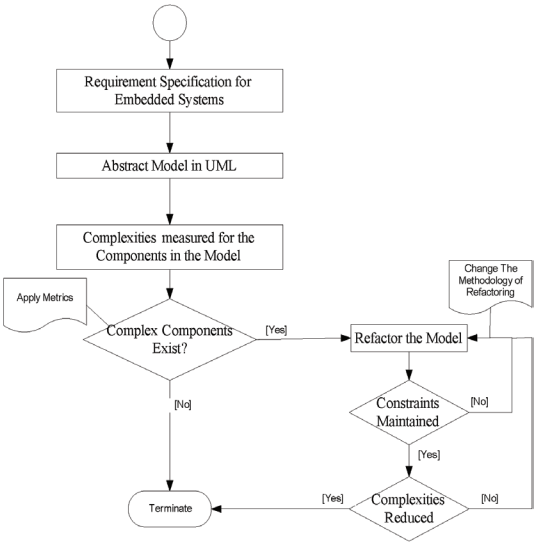
Figure 2. An example Statechart of 'user's access to server' after refactoring



graph transformation, etc. Model refactoring can be detailed by using an example, which consists of Figure1 and Figure2. Figure1 shows an example statechart of a user's access to the server, and Figure2 shows this statechart after refactoring. Two states in Figure1, named as "Pass to Server", and "Retrieved", are merged into one state, "Verifying", in the refactored statechart (Figure 2). The actions used in "Pass to Sever" are copied into the "Verifying" state. Once the complexity ranking is returned, a model can be refactored with the goal of reducing the complexities of the components. Refactoring can be applied on the architecture or behavioural model of the component to lower the complexity, and/or severity, and/or propagation of failure of the components. The methodology of lowering the complexities of components by refactoring is shown in Figure3. As shown in Figure3, initially, the abstract model (in UML) is created from the given specifications. The model is then analysed to measure the complexities of its components. Component complexities need to be compared with a threshold value that users need to determine (for simplicity, the threshold value is ignored in this example).

The large variations among components' complexities are taken as the guideline for flagging the components as complex. If complex

Figure 3. Methodology to lower the complexities of the components by refactoring



components exist in the model, then the model is analysed to be refactored to lower the components' complexities. Special attention needs to be given to the top-ranked components to lower their complexities. Other components can be examined in turn later according to their complexity ranking. Several trial and error iterations are needed to achieve the goal of lowering a component's complexity. In each trial, checks must be made to ensure the refactoring does not interfere with the functionality of the system; otherwise, the model will have to go through another refactoring method. If these constraints are maintained, then the lowering process will check whether components' complexities are sufficiently reduced or not. If the check is successful, then the process will terminate. If not, another iteration of the above steps will occur.

CASE STUDY

Two applications illustrate how the metrics can be applied to measure the complexity of the components. These are an Automated Rail Car system (ARCS), and a wireless telephony Handset System. The first is a safety critical application and the latter is not. Both must meet real-time criteria and they were chosen as they are illustrative of a broad class of systems that must have high reliability.

ARCS Model

A high-level object-model diagram for ARCS and a more detailed diagram of the composites — Terminal and Car — are shown in Figure4. ARCS assumes each pair of adjacent stations is connected by two rail tracks, one for clockwise and one for counter-clockwise travel. Several railcars are available to transport passengers between terminals. A control centre receives, processes, and sends system data to various components. In the proposed ARCS, there are four terminals and eight cars. Passengers can be in any number. A Car has four main parts: ProximitySensor, Cruiser, DestPanel, and OccupancySensor; and a terminal has six main parts: CarHandler, PlatformManager, CallCarButton, Entrance, Exit, and ExitManager. The car is to maintain maximum speed as long as it never comes within 80 meters of any other car. A stopped car will continue its travel only if the smallest distance to any other car is at least 100 meters. A car has its own destination panel. The

Figure 4. (a) High level object-model diagram for ARCS, and (b) More detailed diagrams of the components: terminal and car

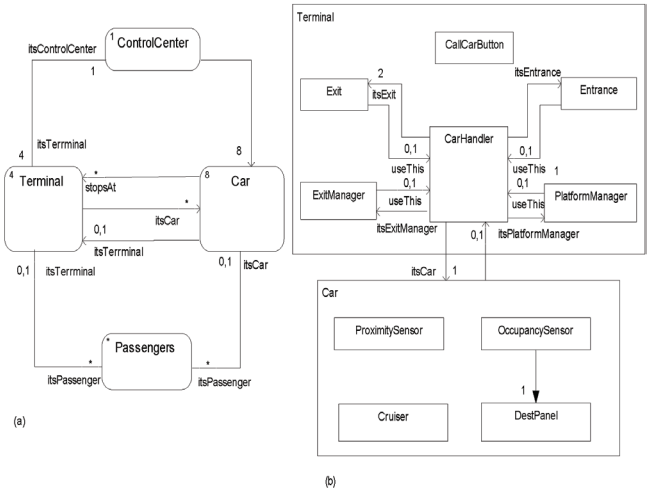
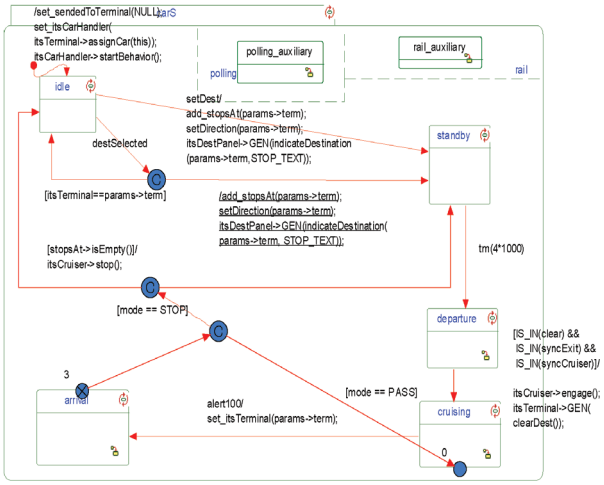


Figure 5. Statechart diagram of car



control centre communicates with various system components—receiving, processing, and providing system data. The ARCS model was created based on the analysis in (Harel & Gery, 1997).

ET Analysis of the Components in ARCS

The state changes of the Car used to measure the ET of the components in the ARCS, are shown in Figure5.

The Car stays at 'Idle' state at any terminal. If the event is generated to move the car from its source to destination then it reaches to its 'Departure' state where it continues its travel only if the smallest distance to any other car (in front) is at least 100 meters. When the car departs from its source, instantly it moves to 'Cruising' state where it continuously uses cruise-controller for maintaining speed. The cruiser can be off, engaged, or disengaged, and the car is to maintain maximum speed as long as it never comes within 80 meters of any other car. It leaves the Cruising state when it receives an event from ProximitySensor alerting the Car that it is within 100 meters from any Terminal. Reaching the Terminal is represented by its 'Arrival' state where it checks whether the current Terminal is in the set of terminals, and if it stops at or not. The Car also provides its own identity and the direction it is traveling to the system. If the Car reaches its destination, it then becomes 'Idle' again. Otherwise, it goes back to 'Cruising' state. Calculated ET for the different components to take a car from Terminal [0] to Terminal [3] is shown in Table 1. Car [0] was assumed (initially) at Terminal [0]. It was se-

Table 1. ET of the Components to Move a Car from Terminal [0] to Terminal [3]

Components	ET	Normalized Values
Car	31s 45 ms	.933
ProximitySensor	47 ms	.023
Cruiser	15 ms	.0074
DestPanel	10 ms	.005
OccupancySensor	8 ms	.004
Terminal	3 ms	.0015
CarHandler	28 ms	.014
CallCarButton	4 ms	.002
Entrance	8 ms	.004
Exit	5 ms	.0025
ExitManager	6 ms	.003
Platform Manager	15 ms	.0074
ControlCenter	1 ms	.0005

lected to move from its source (assumed as Terminal [0]) to its destination (assumed as Terminal [3]). Here, this test case was chosen randomly. The user can choose any possible test case. The system was executed to move it to 'Idle' state. Then the event to move the car to Terminal [3] (from Terminal [0]) was triggered. Since, before moving to 'Idle' state, some components change their behaviour, to calculate the ET, the time for moving to 'Idle' state was also considered with the time to move to destination from 'Idle' state. ET for each component was calculated by:

$$TR_p\ (\{\text{Automated Rail Car System}\}\ (\{\ \text{Car [0] is at Rest at Terminal [0]}\} \rightarrow \{\text{Car [0] is at Terminal [3]}\}) = \sum_{j=1}^{n} d_{pj} \quad (5)$$

where, the symbols are defined in (1).

MIO and TMIO Analysis in ARCS

There are two scenarios in ARCS. These are: i) CarApproachesATerminal, and ii) CarDepartsATerminal. MIO in each scenario and Total MIO (TMIO) in two scenarios for all components are calculated using Equation (2) and Equation (3). The values of MIO and TMIO for all components are shown in Table 2. In Table 2, to simplify the representation of column headings, CarApproachesATerminal Scenario, CarDepartsATerminal Scenario, and Overall Complexities of the Components are abbreviated as CAATS, CDATS, and OCC respectively. The probabilities of the two scenarios are assumed as equal since, if a car departs, there is an equal probability that it will arrive at another Terminal. The blank cells in Table 2 indicate no message from the corresponding component in the corresponding scenario. Car has the highest value of TMIO as it communicates mostly with other components. TMIO for other components are also shown in Table 2.

Overall Complexities of the Components in ARCS

The total complexities of all components are calculated using Equation (4). The last column in Table 2 shows the measured values of each component's complexity. Their values are normalized by taking the ratio between them and the total value. As shown in Table 2, Car is the most complex

Table 2. MIO, TMIO and overall complexities of all components in ARCS

Components	MIO in CAATS	MIO in CDATS	TMIO	OCC
Car	0.64	0.5	1.14	2.073
ProximitySensor	0.21	-	0.21	0.233
Cruiser	0.29	0.21	0.5	0.5074
DestPanel	-	-	-	0.005
OccupancySensor	-	-	-	0.004
Terminal	0.14	0.07	0.21	0.2115
CarHandler	0.43	0.64	1.07	1.084
CallCarButton	-	-	-	0.002
Entrance	0.14	-	0.14	0.144
Exit	-	0.14	0.14	0.1425
ExitManager	-	0.21	0.21	0.213
PlatformManager	0.14	0.07	0.21	0.2174
ControlCenter	-	-	-	0.0005

component and is followed by CarHandler, and Cruiser. The highest value of ET and largest communication dependencies (with other components) took the complexity value of Car to the highest value. In the two scenarios, not all components participated. Hence, MIO and TMIO for some components are not measured in this example. ET alone is used to measure the overall complexities of those components.

Validating the Complexities of the Components in ARCS

To validate the component's complexity measurement, trials are conducted whereby transient faults are injected into the components of the selected system. Transient faults are injected at each component, into one bit at a time. The reason is that transient faults change the value of one bit at a time and the probability of changing two bits and/or two transient faults are almost zero. The fault injection is made by changing one bit of the parameter value, or anywhere in code or in the parameter name. The probabilities of occurrences of soft errors in the components are calculated by taking the ratios between total number of soft error occurrences and total number of fault injections. This ratio can be defined as the Error/Fault injections (E/F) Ratio. In this paper, only those soft errors are counted that cause any degradation or failure in system functionality. If the soft error does not create any degradation in the system then it is not taken as a matter of concern here. Ten trials are made for transient fault injection into every component. The more trials are performed, the better the expected result. However, for this large example model, it is expected that ten trials in each component would be able to give a good idea about their probabilities of soft error proneness. Table 3 shows the E/F ratio for this example.

If these ratios are ranked in an ascending order then it is observed that Table 2 has a similar ranking to Table 3 until the Cruiser component. The next ratio is equal for ProximtySensor, PlatformManager, and ExitManager where, in Table 2, their complexity values differ a little. If that slight difference is neglected, then the complexity ranking for these components shows similar results in these tables. Other results also show a very similar complexity order as in Table 2. Hence, it can be concluded that complexity analysis is able to measure the likelihood of soft error proneness among the components of ARCS.

Table 3. E/F Ratios of the components in ARCS

Components	Number of Transient Faults Injections	Number of Soft Errors	E/F Ratio
Car	10	8	0.8
ProximitySensor		4	0.4
Cruiser		5	0.5
DestPanel		1	0.1
OccupancySensor		1	0.1
Terminal		3	0.3
CarHandler		7	0.7
CallCarButton		1	0.1
Entrance		3	0.3
Exit		3	0.3
ExitManager		4	0.4
PlatformManager		4	0.4
ControlCenter		1	0.1

Table 4. Comparison among dynamic complexity, static complexity, and E/F Ratios of the components of ARCS

Components	E/F Ratio (Normalized)	Dynamic Complexities of the Components (Normalized)	Static Complexities (Normalized)
Car	1	1	1
ProximitySensor	0.5	0.112	0.625
Cruiser	0.625	0.245	0.75
DestPanel	0.125	0.002	0.25
OccupancySensor	0.125	0.0019	0.25
Terminal	0.375	0.102	0.25
CarHandler	0.875	0.523	0.375
CallCarButton	0.125	0.001	0.25
Entrance	0.375	0.069	0.25
Exit	0.375	0.069	0.25
ExitManager	0.5	0.103	0.25
PlatformManager	0.5	0.105	0.25
ControlCenter	0.125	0.0002	0.25

Comparison with Static Complexity Analysis

This comparison shows the contribution of dynamic metrics to measure the complexities of the components over static metrics. The dynamic complexities of the components for ARCS are obtained as outlined in the section of the Methodology of Complexity Analysis. Static complexities are calculated by using McCabe's Cyclomatic complexity theorem (McCabe, 1976). The last Column in Table 4 shows the Cyclomatic Complexities of the components in ARCS.

Table 4 also allows comparison among the dynamic complexity, static complexity, and E/F ratios of the components. The values shown in Table 4 are all normalized to make the comparison possible. The normalization is done by dividing each element in each column with the highest value in the corresponding column. Since in all three columns Car has the maximum value, the normalized value for Car is obtained as '1'. The results show that both dynamic complexity and E/F ratio return a similar ranking. Static complexity, on the other hand, returned a completely different ranking and, in most cases, it failed to distinguish among the complexities of the components. Static complexity analysis, for instance, returned the same complexity value for DestPanel, OccupancySensor, Terminal, CallCarButton, Entrance, Exit, ExitManager, PlatformManager, ControlCenter. Hence, dynamic complexity is more significant than static complexity in component complexity analysis.

Lowering the Complexities of the Components in ARCS

That part of the model dealing with Car behaviour is carefully examined to determine refactoring possibilities to lower its complexity. All the states and their internal and/or external codes used in triggers, as well as actions, are checked. These areas are where refactoring could achieve the goal of reducing Car's time complexity while keeping its functionality unaffected. Two states and their internal codes of Car are merged to reduce the time complexity. Comparison among

the calculated normalized ET of the components of the refactored model and existing model (to take a car from Terminal [0] to Terminal [3]) is shown in Table 5. A lower ET will result in lower complexity as well as lower complexity of the components. Table 5 shows that refactoring the model lowered the ET of Car and ProximitySensor to a measurable extent. Others, except for OccupancySensor and PlatformManager, are also lowered. The increase in complexity value of OccupancySensor and PlatformManager are not so large. For this reason, the increases in these two components can be viewed as negligible. In summary, applying refactoring is effective in lowering the complexities of the components.

A Wireless Telephony Handset System

The Handset system provides voice and data services to users by placing and receiving calls.

Table 5. Comparison among the calculated normalized ET of the components of refactored model and existing model

Components	Normalized ET of Refactored Model	Normalized ET of Existing Model
Car	0.899	.995
Proximity-Sensor	0.00051	.0015
Cruiser	.00048	.00048
DestPanel	.00016	.00032
Occupancy-Sensor	0.00035	.00026
Terminal	.00013	.000096
CarHandler	.00089	.00089
CallCarBut-ton	.000032	.00013
Entrance	0.00022	.00026
Exit	.00016	.00016
ExitManager	.000096	.000192
Platform Manager	0.00064	.00048
ControlCen-ter	.000032	.000032

To deliver services, the wireless network must receive, set up, and direct incoming and outgoing call requests, track and maintain the location of users, and facilitate uninterrupted service when users move within and outside the network. When the wireless user initiates a call, the network receives the request, and validates and registers the user; once registered, the network monitors the user's location. The wireless telephone must send acceptable signal strength to the network to receive the call. When the network receives a call, it directs it to the appropriate registered user. The high-level architectural diagram (black box approach) of the Handset System is shown in Figure6.

ET Analysis of the Sub-Systems in the Handset System

The 'Call Control' Statechart diagram of CM in the Handset system was used for ET analysis and is shown in Figure7.

The statechart identifies the state-based behaviour of instances of 'Call Control' when the system receives call requests from users and connects calls. It has two main states: 'Idle' and 'Active'. Two other states: 'ConnectConfirm' and 'Connected' are nested in 'Active' state. 'Call Control' waits for an incoming call in the 'Idle' state. When an incoming call is received, it forwards the message through its 'cc_mm' port to the MM by sending a 'PlaceCallReq' event. MM, in co-operation with the DL, processes this signal and sends a call confirmation to CM. If CM does not receive a confirmation within thirty seconds then it returns to the 'Idle' state by sending a 'Disconnect' event to MM. If it receives a confirmation, the call connects, and remains connected until it receives a message to disconnect MM. DL also undergo behavioural changes during these operations. When the operation succeeds, the time of executing the 'Place Call' event at 'Idle' state, and the time when the system reached at 'Connected' state of 'Call Control' statechart

Figure 6. High level architecture of the handset system

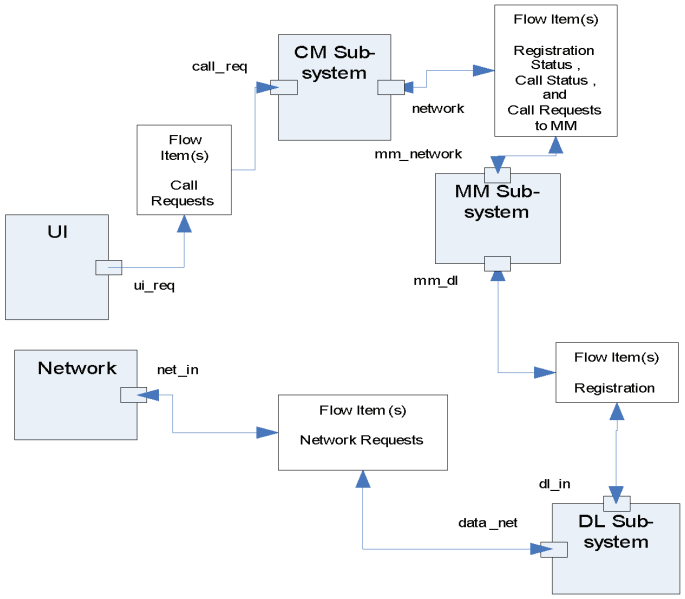
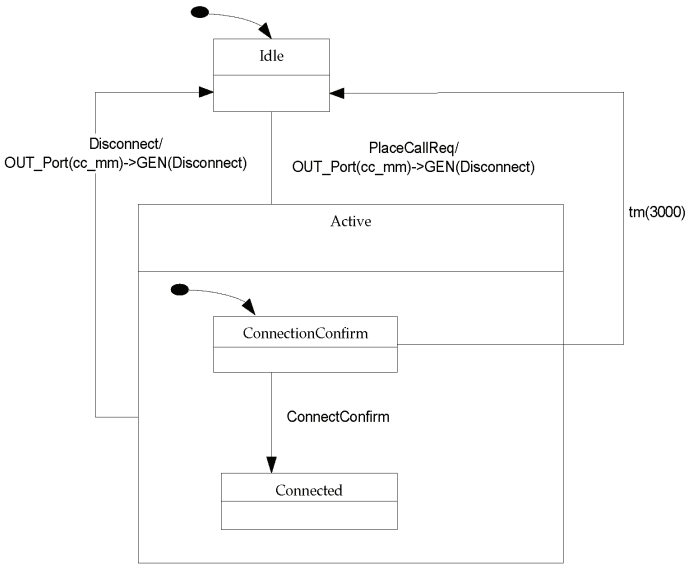


Figure 7. Call control Statechart diagram at the beginning of execution



were recorded to calculate the ET of these sub-systems. ET for each sub-system was calculated by Equation (6) and this equation is derived from equation (1).

$$\mathrm{TR}_p\ (\{\text{ConnectionManagement-}\rangle\text{CallControl-}\rangle\text{GEN[PlaceCallReq]}\}\ \{\text{Idle}\} \rightarrow \{\text{Connected}\}) = \sum_{j=1}^{n} d_{pj} \tag{6}$$

where the symbols are defined in equation (1). The pseudo code to activate the connection in the Handset system is shown in Box 1. PlaceCallReq event's triggering status (from UI) was taken as input, and tCM (Total), tMM (Total), tDL (Total) were returned as output where they represented the total time taken by CM, MM, and DL respectively in connecting the call.

Table 6 shows ET of the sub-systems and their normalized values. As shown in the table, MM

Box 1. Pseudo code to activate the connection in the handset system

```
Input: PlaceCallReq event's triggering status
Output: tCM (Total), tMM (Total), tDL (Total)
Detail:
Start: The system is at Idle state
Step 1. Initialize tCM (Total), tMM (Total), tDL (Total)
Step 2. Input: PlaceCallReq event's triggering status
Step 3. If UI generates PlaceCallReq event in CM then
Step 4. CM sends PlaceCallReq to MM
Step 5. Update tCM (Total): tCM (Total) +=tCM (Step 3)
Step 6. CM entered at ConnetionConfirm state
Step 7. Update tCM (Total): tCM (Total) +=tCM (Step 5)
Step 8. CM sets tm (3000) at ConnetionConfirm state
Step 9. Update tCM (Total): tCM (Total) +=tCM (Step 7)
Step 10. If not MM returns Take Event PlaceCallReq to CM then CM goes to Idle
state
Exit Sub
Else
Update tMM (Total): tMM (Total) +=tMM (Step 9)
CM remains in ConnetionConfirm state
Step 11. MM checks signal strength
Step 12. Update tMM (Total): tMM (Total) +=tMM (Step 10)
Step 13. MM sends Registration Request to DL
Step 14. Update tMM (Total): tMM (Total) +=tMM (Step 12)
Step 15. DL sends Channel open confirmation to MM
Step 16. Update tDL (Total): tDL (Total) +=tDL (Step 14)
Step 17. MM updates location
Step 18. Update tMM (Total): tMM (Total) +=tMM (Step 16)
Step 19. MM sends ConnectConfirm event to CM
Step 20. Update tMM (Total): tMM (Total) +=tMM (Step 18)
Step 21. CM goes to Connected state
Step 22. Update tCM (Total): tCM (Total) +=tCM (Step 20)
End if
End if
Step 23. Output: tCM (Total), tMM (Total), tDL (Total)
End       The system is at Connected state
```

Table 6. ET of the sub-systems to activate the connection

Sub-Systems	ET	Normalized Values
CM	**6**	**0.29**
DL	**1**	**0.05**
MM	**14**	**0.67**

took the longest time in the selected operation. DL, comparatively, took negligible time, since in the selected operation; DL's interference was much lower than MM, and CM.

MIO and TMIO of the Sub-Systems in the Handset System

The probabilities of the occurrences of three scenarios in the Handset system — i) Place Call Request Successful, ii) Network Connect, and iii) Connection Management Place Call Request Success — were assumed as 0.45, 0.30, and 0.25 respectively. The assumptions were made with respect to their usage in real life scenarios.

MIO and TMIO for three different sub-systems were calculated for three different sequence diagrams using Equation (2) and Equation (3). All values of MIO and TMIO for the three sub-systems are shown in Table 7. To simplify the representation of column headings in Table 7, Place Call Request Successful Scenario, Network Connect Scenario, and Connection Management Place Call Request Success Scenario are abbreviated as PCRSS, NCS, and CMPCRSS respectively. The results shown refer to the level of communication dependency of each sub-system with other sub-systems in the Handset system. DL has the largest communication dependency among all three sub-systems followed by CM, and MM, as shown in Table 7.

Overall Complexities of the Sub-Systems in the Handset System

Overall complexities of three sub-systems were calculated using Equation (4), and the last column in Table 7 shows their overall complexities. Overall complexity is the summation of ET (during simulation), and Message-in-and-out-frequencies of the sub-systems. Though MM has the highest value of ET, and DL has the same for TMIO, considering both of the complexities, CM is the most complex sub-system in the Handset system. Overall complexities of DL and MM are almost equal.

Validating Complexities of the Sub-Systems in Handset System

To validate the component's complexity measurement, trials were conducted whereby transient faults were injected into each sub-system of the Handset system. Error/Fault injections (E/F) ratios were calculated for each sub-system. As mentioned earlier, in this paper, only those soft errors were counted that cause any degradation or failure in system functionality, and if the soft error did not create any degradation in the system then it was not taken as a matter of concern. Ten trials were made for transient fault injection into each sub-system. The more trials performed, the better the expected result. However, for this large example model, it was expected that ten trials in each sub-system would provide a good idea about

Table 7. MIO and TMIO of the sub-systems in the handset system

Sub-System Packages	MIO in PCRSS	MIO in NCS	MIO in CMPCRSS	TMIO	Overall Complexities
CM	0.7	0.33	0.83	0.62	0.91
DL	0.6	1	0.67	0.74	0.79
MM	0.2	-	0.17	0.13	0.80

Table 8. E/F ratios of the sub-systems in the handset system

Components	Number of Fault Injections	Number of Faults	Ratio
CM	10	6	0.6
MM		5	0.5
DL		3	0.3

the likelihood of soft errors. Table 8 shows the E/F ratios for this example.

If these ratios are ordered in ascending then it is observed that the order of E/F ratios shown in Table 8 is similar to the complexity order shown in Table 7. Hence, it can be concluded that complexity analysis is able to measure the likelihood of soft errors occurrences among the sub-systems of the Handset system.

Comparison with Static Complexity

This comparison shows the contribution of dynamic metrics to measuring the complexities of the components over static metrics. Static complexities were calculated using McCabe's Cyclomatic complexity theorems (McCabe, 1976). The 4th column of Table 9 shows the measured static complexities of the sub-systems in the Handset system.

Table 9 also allows comparison between the dynamic complexity, static complexity, and E/F ratios of the components. The values, shown in Table 9, were all normalized to make comparison easier. The normalization was done by dividing each element in each column with the highest value in the corresponding column. Since in all three columns CM had the maximum value, the normalized value for CM was obtained as '1'. The results show that both dynamic complexity and E/F ratios returned CM as the top ranked complex component followed by MM and DL. Static complexity on the other hand returned MM as the most complex component followed by CM and DL. Hence, dynamic complexity shows more significance than static complexity in component complexity analysis.

Lowering Complexities of the Sub-Systems in Handset System

The behaviour models of all three sub-systems were carefully examined to be refactored. All the states and their internal and/or external codes used in triggers, actions and so forth were checked to ensure refactoring could achieve the goal of reducing complexity of the sub-systems while keeping its functionality unaffected. The functionality was affected by any change made in the behavioural diagrams of CM, and DL sub-systems. The MMCallControl activity diagram and the InCall sub-activity diagram of MM sub-system could be refactored by maintaining the constraints. Two states, their internal codes, and all the forks of MMCallControl activity diagram were merged, the CheckSignal state of InCall sub-activity diagram in MM sub-system was removed, and the internal codes in CheckSignal state were merged with the VoiceData state to lower its time complexity. The calculated normalized ET of the sub-systems of

Table 9. Comparison among dynamic complexity, static complexity, and E/F ratios of the sub-systems in the handset system

Components	E/F Ratios	Dynamic Complexities of the Components	Static Complexities
CM	1	1	0.71
DL	0.5	0.87	0.57
MM	0.83	0.88	1

refactored model and existing model (to establish a handset connection) is shown in Table 10.

Lower ET will result in lower complexity of the sub-systems. Table 10 shows that refactoring the model is able to lower the ET of the CM, and MM sub-systems significantly. The ET for DL is constant. DL is the least complex sub-system in the Handset system and its complexity is so low that it does not attract any attention by designers. Hence, applied refactoring is acceptable in lowering the complexities of the sub-systems.

DISCUSSION

The case study results show the effectiveness of the proposed method in measuring the complexities of the components in two chosen examples — ARCS and the Handset system. The comparisons drawn between dynamic complexity (the approach of this research) and static complexity (McCabe's Cyclomatic complexity) indicate that, for a number of components, static complexity failed to return variations among their complexity, whereas, this chapter was able to show the variations. E/F ratios showed similar results with obtained complexity results that validate the proposed method. Static complexity, on the other hand, returned a completely different ranking to E/F ratios. Study results also showed how the application of refactoring made changes in the existing design to lower the complexities of the components to minimize the risks of soft errors. It investigated how to encourage designers to explore changes that could be made in the existing models of embedded systems to lower the dependability risks.

CONCLUSION

This chapter flags those components of a system model which are complex in architecture and in behaviour and where there is high probability of fault occurrences. Amendment in the early stages of design saves both cost and time, and it is easier for the designers to flag and defend the risk issue at the modelling level than the system is already implemented or at the later stages of design. The investigation began with the measurement of likelihood of faults in the system. These metrics are: (1) assessment of execution time during simulation, and (2) Message-In-and-Out frequency. Both of the metrics are obtained from the UML specifications that can be used in the early design phase of a system. These are dynamic metrics; that is, they work on the execution phase of the model. Developed metrics are validated by calculating the E/F ratios for the components. This chapter then developed the ways to encourage designers to explore changes that could be made in the existing model to lower the complexities of the components. Later on, the designer should take some corrective actions in those portions of the model or take some special measures like error correction code and/or duplication of hardware or software at only those portions during post design phases to reduce the risks of desired functionality degradation of SOA.

Table 10. Comparison among ET of the components of refactored model and those of existing handset model

Components	Normalized ET of Refactored Model	Normalized ET of Existing Model
CM	0.29	0.228
MM	0.05	0.0393
DL	0.67	0.67

There are open scopes to extend this paper. Some possible future directions are outlined shortly as follows.

In the current approach (for simplicity) the weights of ET and TMIO in measuring total value of complexity by Equation (4) are assumed as equal. The relative weight of different factors in measuring the complexity could be different for different application. For a specific application, depending on individual factor's influence on the whole functionality, its appropriate weight can be generated. Alternatively, weight vectors could be introduced to capture user preferences automatically based on users' selection patterns. Further extension to this paper can be made by finding the appropriate weights of ET and TMIO in measuring the complexity of each component.

Like the weight vectors, complexity threshold is a relative measure. Depending on the type of system and the type of application, the complexity of a component may vary. Every complexity pass threshold is a matter of concern for the designer. Complexity threshold should be derived or defined by the user. This paper considers that the top-ranked complex components have high probability of faults. However, there should be a threshold value of a component's complexity, which could flag whether the component is crossing the complexity boundary or not. The scope is there to measure the complexity threshold, specific to an application, to better categorize the complex components.

To lower the complexity, the current paper applies refactoring, which re-structures the model to improve fault tolerance. Alternative solutions (rather than refactoring) could be examined to achieve the best solution to lowering the complexity of the components, and that of the whole system as well.

REFERENCES

Ammar, H. H., Nikzadeh, T., & Dugan, J. B. (1997). *A methodology for risk assessment of functional specification of software systems using colored Petri nets.* Paper presented at the Fourth International Software Metrics Symposium, Los Alamitos, CA, USA.

Austin, T. M. (1999). *DIVA: A reliable substrate for deep submicron microarchitecture design.* Paper presented at the 32nd Annual International Symposium on Microarchitecture.

Boger, M., Sturm, T., & Fragemann, P. (2003). *Refactoring browser for UML.* Paper presented at the International Conference on Objects, Components, Architectures, Services, and Applications for a Networked World, Berlin, Germany. (LNCS 2591).

Bondavalli, A., Latella, D., Majzik, I., Pataricza, A., & Savoia, G. (2001). Dependability analysis in the early phases of UML based system design. *Journal of Computer Systems Science and Engineering*, *16*(5), 265–275.

Bowles, J. B. (1998). *The new SAE FMECA standard*. Paper presented at the International Symposium on Product Quality and Integrity.

Bowles, J. B. (2004). An assessment of RPN prioritization in a failure modes effects and criticality analysis. *Journal of the IEST*, *47*, 51–56.

Chen, C. L., & Hsiao, M. Y. (1984). Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of Research and Development*, *28*(2), 124–134. doi:10.1147/rd.282.0124

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493. doi:10.1109/32.295895

Cortellessa, V., Goseva-Popstojanova, K., Appukkutty, K., Guedem, A. R., Hassan, A., & Elnaggar, R. (2005). Model-based performance risk analysis. *IEEE Transactions on Software Engineering*, *31*(1), 3–20. doi:10.1109/TSE.2005.12

Crouzet, Y., Collet, J., & Arlat, J. (2005). *Mitigating soft errors to prevent a hard threat to dependable computing.* Paper presented at the 11th IEEE International On-Line Testing Symposium, IOLTS.

Dobrzanski, L., & Kuzniarz, L. (2006). *An approach to refactoring of executable UML models.* Paper presented at the ACM Symposium on Applied Computing, New York.

Gerson, S., Damien, P., Yves Le, T., & Jean-Marc, J. (2001). Refactoring UML Models. *Proceedings of the 4th International Conference on the Unified Modeling Language: Modeling Languages, Concepts, and Tools* (pp. 134-148). Springer-Verlag.

Gold, B. T., Kim, J., Smolens, J. C., Chung, E. S., Liaskovitis, V., & Nurvitadhi, E. (2005). TRUSS: A reliable, scalable server architecture. *IEEE Micro*, *25*(6), 51–59. doi:10.1109/MM.2005.122

Harel, D., & Gery, E. (1997). Executable object modeling with statecharts. *Computer*, *30*(7), 31–42. doi:10.1109/2.596624

Harrison, R., Counsell, S. J., & Nithi, R. V. (1998). An evaluation of the MOOD set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, *24*(6), 491–496. doi:10.1109/32.689404

Hitz, M., & Montazeri, B. (1995). *Measuring product attributes of object-oriented systems*. Paper presented at the 5th European Software Engineering Conference.

Hosseini, S., & Azgomi, M. A. (2008). *UML model refactoring with emphasis on behavior preservation.* Paper presented at the 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering, Piscataway, NJ, United States.

Iyer, R. K., Nakka, N. M., Kalbarczyk, Z. T., & Mitra, S. (2005). Recent advances and new avenues in hardware-level reliability support. *IEEE Micro*, *25*(6), 18–29. doi:10.1109/MM.2005.119

Jurjens, J., & Wagner, S. (2005). *Component-based development of dependable systems with UML*. (LNCS 3778), (pp. 320-344).

Khoshgoftaar, J. M. T. (1996). Software metrics for reliability assessment. In Lyu, M. (Ed.), *Handbook of software reliability engineering* (pp. 493–529).

Krishnamohan, S. (2005). *Efficient techniques for modeling and mitigation of soft errors in nanometer-scale static CMOS logic circuits.* Unpublished doctoral thesis, Michigan State University, United States-Michigan.

Lakhal, N. B., Kobayashi, T., & Yokota, H. (2006). *Dependability and flexibility centered approach for composite Web Services modeling*. Berlin, Germany.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, *SE-2*(4), 308–320. doi:10.1109/TSE.1976.233837

Meaney, P. J., Swaney, S. B., Sanda, P. N., & Spainhower, L. (2005). IBM z990 soft error detection and recovery. *IEEE Transactions on Device and Materials Reliability*, *5*(3), 419–427. doi:10.1109/TDMR.2005.859577

Mens, T. (2006). *On the use of graph transformations for model refactoring*. (LNCS 4143), (pp. 219-257).

Mens, T., & Van Gorp, P. (2006). A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, *152*(1-2), 125–142. doi:10.1016/j.entcs.2005.10.021

Miskov-Zivanov, N., & Marculescu, D. (2006). *MARS-C: Modeling And Reduction of Soft errors in Combinational circuits.* Paper presented at the Proceedings of the Design Automation Conference, Piscataway, NJ, USA.

Mitra, S. M.Z., Seifert, N., Mak, T.M. & Kim, K. (2006). *Soft error resilient system design through error correction.* Paper presented at the International Conference on Very Large Scale Integration and System-on-Chip.

Mohamed, A. G., Chad, S., Vijaykumar, T. N., & Irith, P. (2003). Transient-fault recovery for chip multiprocessors. *IEEE Micro*, *23*(6), 76–83. doi:10.1109/MM.2003.1261390

Mukherjee, S. S., Emer, J., & Reinhardt, S. K. (2005). *The soft error problem: An architectural perspective.* Paper presented at the 11th International Symposium on High-Performance Computer Architecture, San Francisco, CA, USA.

Mukherjee, S. S., Kontz, M., & Reinhardt, S. K. (2002). *Detailed design and evaluation of redundant multi-threading alternatives.* Paper presented at the 29th Annual International Symposium on Computer Architecture.

Narayanan, V., & Xie, Y. (2006). Reliability concerns in embedded system designs. *Computer*, *39*(1), 118–120. doi:10.1109/MC.2006.31

Nguyen, H. T., Yagil, Y., Seifert, N., & Reitsma, M. (2005). Chip-level soft error estimation method. *IEEE Transactions on Device and Materials Reliability*, *5*(3), 365–381. doi:10.1109/TDMR.2005.858334

Oh, N., Shirvani, P. P., & McCluskey, E. J. (2002). Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, *51*(1), 63–75. doi:10.1109/24.994913

Oma, M., Rossi, D., & Metra, C. (2003). *Novel transient fault hardened static latch.* Paper presented at the IEEE International Test Conference (TC), Charlotte, NC, United States.

Park, J. K., & Kim, J. T. (2008). A soft error mitigation technique for constrained gate-level designs. *IEICE Electronics Express*, *5*(18), 698–704. doi:10.1587/elex.5.698

Quming, Z., & Mohanram, K. (2004). *Cost-effective radiation hardening technique for combinational logic.* Paper presented at the Proceedings of the International Conference on Computer Aided Design, Piscataway, NJ, USA.

Rashid, M. W., Tan, E. J., Huang, M. C., & Albonesi, D. H. (2005). Power-efficient error tolerance in chip multiprocessors. *IEEE Micro*, *25*(6), 60–70. doi:10.1109/MM.2005.118

Ray, J., Hoe, J. C., & Falsafi, B. (2001). *Dual use of superscalar datapath for transient-fault detection and recovery.* Paper presented at the 34th ACM/IEEE International Symposium on Microarchitecture.

Reinhardt, S. K., & Mukherjee, S. S. (2000). *Transient fault detection via simultaneous multithreading.* Paper presented at the 27th International Symposium on Computer Architecture.

Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., & August, D. I. (2005). *SWIFT: SoftWare Implemented Fault Tolerance.* Paper presented at the International Symposium on Code Generation and Optimization, Los Alamitos, CA, USA.

Rockett, L. R. Jr. (1992). Simulated SEU hardened scaled CMOS SRAM cell design using gated resistors. *IEEE Transactions on Nuclear Science*, *39*(5), 1532–1541. doi:10.1109/23.173239

Rotenberg, E. (1999). *AR-SMT: A microarchitectural approach to fault tolerance in microprocessors.* Paper presented at the 29th Annual International Symposium on Fault-Tolerant Computing.

Saggese, G. P., Wang, N. J., Kalbarczyk, Z. T., Patel, S. J., & Iyer, R. K. (2005). An experimental study of soft errors in microprocessors. *IEEE Micro*, *25*(6), 30–39. doi:10.1109/MM.2005.104

Sherer. (1988). *Methodology for the assessment of software risk.* PhD Thesis, Wharton School, University of Pennsylvania.

Shye, A., Blomstedt, J., Moseley, T., Janapa Reddi, V., & Connors, D. (in press). PLR: A software approach to transient fault tolerance for multi-core architectures. *IEEE Transactions on Dependable and Secure Computing*.

Smolens, J. C., Gold, B. T., Kim, J., Falsafi, B., Hoe, J. C., & Nowatzyk, A. G. (2004). *Fingerprinting: Bounding soft-error detection latency and bandwidth.* Paper presented at the Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XI, New York, United States.

Srinivasan, J., Adve, S. V., Bose, P., & Rivers, J. A. (2004). *The case for lifetime reliability-aware microprocessors*. Paper presented at the 31st Annual International Symposium on Computer Architecture.

STMicroelectronics. (2003). New chip technology from STmicroelectronics eliminates soft error threat to electronic systems. Retrieved from http://www.st.com/stonline/press/news/year2003/t1394h.htm

Telelogic. (2009). *Homepage information*. Retrieved on January 30, 2009, from http://www.telelogic.com/

Timor, A., Mendelson, A., Birk, Y. & Suri, N. (2008). Using underutilized CPU resources to enhance its reliability. *IEEE Transactions on Dependable and Secure Computing.*

Tosun, S. (2005). *Reliability-centric system design for embedded systems*. Unpublished doctoral thesis, Syracuse University, United States-New York.

Vijaykumar, T. N., Pomeranz, I., & Cheng, K. (2002). *Transient-fault recovery using simultaneous multithreading.* Paper presented at the 29th Annual International Symposium on Computer Architecture.

Walcott, K. R., Humphreys, G., & Gurumurthi, S. (2007). *Dynamic prediction of architectural vulnerability from microarchitectural state.* Paper presented at the Proceedings of the International Symposium on Computer Architecture, New York, United States.

Wang, F. (2008). *Soft error rate determination for nanometer CMOS VLSI logic*. Paper presented at the Proceedings of the Annual Southeastern Symposium on System Theory.

Wang, L., Wong, E., & Xu, D. (2007). *A threat model driven approach for security testing*. Paper presented at the Proceedings of the Third International Workshop on Software Engineering for Secure Systems, SESS' 07, Piscataway, NJ, United States.

Wikipedia. (2009). *Home page*. Retrieved on January 30, 2009, from http://en.wikipedia.org/wiki

Wood, S. K., Akehurst, D. H., Uzenkov, O., Howells, W. G. J., & McDonald-Maier, K. D. (2008). A model-driven development approach to mapping UML state diagrams to synthesizable VHDL. *IEEE Transactions on Computers*, *57*(10), 1357–1371. doi:10.1109/TC.2008.123

Xie, Y., Li, L., Kandemir, M., Vijaykrishnan, N., & Irwin, M. J. (2004). *Reliability-aware co-synthesis for embedded systems.* Paper presented at the 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors.

Yacoub, S. M., & Ammar, H. H. (2002). A methodology for architecture-level reliability risk analysis. *IEEE Transactions on Software Engineering*, *28*(6), 529–547. doi:10.1109/TSE.2002.1010058

Zhang, M. (2006). *Analysis and design of soft-error tolerant circuits.* Unpublished doctoral thesis, University of Illinois at Urbana-Champaign, United States.

Zhang, M., Mitra, S., Mak, T. M., Seifert, N., Wang, N. J., & Shi, Q. (2006). Sequential element design with built-in soft error resilience. *IEEE Transactions on Very Large Scale Integration (VLSI)*. *Systems*, *14*(12), 1368–1378.