Application of Design Combinatorial Theory to Scenario-Based Software Architecture Analysis

Chung-Horng Lung
Department of Systems and Computer Engineering
Carleton University, Ottawa, Ontario, Canada
Email: chlung@sce.carleton.ca

Marzia Zaman
Cistel Technology Inc., Ottawa, Ontario
Email: marzia@cistel.com

Abstract. Design combinatorial theory for test-case generation has been used successfully in the past. It is useful in optimizing test cases as it is practically impossible to exhaustively test any software system. The same concept can be applied while doing high level architecture analysis of a software system. In software architecture analysis, the architect often analyzes different scenarios that a system may experience during its lifecycle to ensure that all or most possible scenarios are covered in the design. Usually, the analysis is conducted manually in an ad-hoc fashion and scenarios are executed separately. However, some important use cases that involve multiple concurrent scenarios may be overlooked with this approach. Software architecture analysis is critical, especially for real time telecommunications systems. More formalism or robustness needs to be considered in the evaluation process, particularly for reliability. This paper demonstrates application of the design combinatorial theory based technique and tool to software architecture reliability analysis of a practical real-time software system.

1. Introduction

Software architectural analysis [Kazman96, Lung00, Clements02] is an important step in software development process. It is critical to ensure all or most functional and non-functional attributes of the software to be developed are well defined, captured and understood by the design team early in the life cycle. If neglected, major problems may surface at the time of implementation which may cause tremendous rework in the later part of the development life cycle. However, it may not always be possible to design the software as intended because the requirements may change over time and the design change may be unavoidable. Also, one may need to conduct reverse engineering in order to enhance the reliability or performance of the software, which is not always built-in. Whether it is done as part of forward engineering or reverse engineering, the architectural analysis is something that needs to be conducted.

Identifying different scenarios is critical and commonly accepted in the architectural analysis. One must understand different user and system requirements to come up with the scenarios. The scenarios may be triggered by the users and/or some other external or internal inputs or events. Also, the fact that one scenario can influence or change the behaviour of some other scenarios, and multiple scenarios can occur together at the same time, makes an explosive set of scenarios and makes the scenario generation challenging.

Traditionally, software architecture analysis is based on executing scenarios, often separately. But the analysis is often conducted in an ad-hoc manner and is heavily dependent on the analyst's experience. With this approach, some scenarios, especially, combinations of scenarios that may happen concurrently in real life may be missed. For instance, Lung, et al. [Lung98] reported a case study for performance improvement as a result of executing multiple concurrent scenarios. The system behaved well while three scenarios were conducted independently. However, performance degraded significantly when these three scenarios happened simultaneously. The use case where these three scenarios could happen at the same time was initially missed, which caused performance problem in a real-time telecommunications system.

For applications that have a large number of scenarios and many possible combinations of various scenarios, a formal or semi-formal approach to scenario analysis is advocated. Manual effort alone is time consuming and error prone. In this paper, we

demonstrate how scenarios or multiple scenarios can be derived in a step by step procedure with the support of the design combinatorial theory. Although the manual effort and time will always be needed, the process can be more formalized. The formal or semi-formal approach will allow the designers to focus on identifying the scenarios that are of concern to the system. However, combination of scenarios is mostly generated by the design combinatorial technique, which increases the coverage of multiple scenarios and reduces the chance of overlooking important cases.

The concept of design combinatorial theory has been studied intensively. The approach has been applied to software testing by many researchers and industry practitioners. Colbrourn [Colbourn04b] and Grinda, et al. [Grindal04] present a thorough study on this topic. It is not our intention of this paper to discuss the theory in detail. Rather, we demonstrate the application of the theory to software architecture analysis of a real telecommunications system with emphasis on reliability, which, to our knowledge, has not been discussed much. Software architecture analysis can be benefited from this concept due to its formalism, coverage and simplicity.

The rest of the paper is organized as follows: Section 2 briefly describes the design combinatorial theory. Section 3 illustrates an application of the design combinatorial theory to software architecture reliability analysis of a real time telecommunications system. Finally, section 4 gives a summary.

2. Design Combinatorial Theory

Design combinatorial involves experimental design where statistical techniques are used for planning experiments such that one can extract maximum possible information from as few experiments as possible. This technique has been used extensively in a wide range of applications from planning medical experiments to industrial experiments [Cohen94].

Depending on the application, different designs are proposed in the past [Cochran50]. In this paper, we focus on the design of pair-wise combinations as it seems to be very effective in test case generation [Burr98, Cohen94, Cohen96, Colbourn04a, Colbourn04b, Dalal99, Grindal04, Kuhn02]. The design of pairwise combination requires that for any pair of parameters, all combination of input values must occur at least once.

Consider a situation with four parameters A,B,C,D; each having four possible values, say 1, 2, 3, 4. It would require 4^4=256 combinations to cover all possible combinations. The set of exhaustive combination will be {(1,1,1,1),(1,1,1,2),(1,1,1,3)…(4,4,4,4)}. However, in practice, most of the combinations are redundant and/or do not provide any additional value to testing. Therefore, the same amount effectiveness can be achieved if there is a way to select the combinations that are of interest to us. Hand-picking the interesting or more useful combination from the exhaustive set is not practical. Even if it is possible for a small system, it is still rather time-consuming. Pairwise combination, on the other hand, requires that all pairwise combinations of the two input values between two parameters are guaranteed. In this case, an algorithm which constructs an optimized set of all pair-wise combinations would be useful.

For example, the following set shown in Table 1 has only 16 combinations that guarantee all pairwise combinations of input values between any pair of parameters. In other words, given any two parameters, say A and C, all combinations of values between these two parameters, i.e., {(1,1),(1,2),(1,3),…,(4,2),(4,3),(4,4)} will be covered in the set given below. The pairwise technique is found to be very useful in many software testing applications as it has been seen that most field faults are occurred due to the interaction of one or two parameters. The design combinatorial technique attempts to provide a set of combinations such that it is optimal in size, i.e., the table has optimum number of rows.

Table 1: Pairwise combinations

#	A	B	C	D
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	1	4	4	4
5	2	1	2	3
6	2	2	1	4
7	2	3	4	1
8	2	4	3	2
9	3	1	3	4
10	3	2	4	3
11	3	3	1	2
12	3	4	2	1
13	4	1	4	2
14	4	2	3	1
15	4	3	2	4
16	4	4	1	3

In this case, the amount of reduction in number of combinations would be from 256 to 16. The gain is

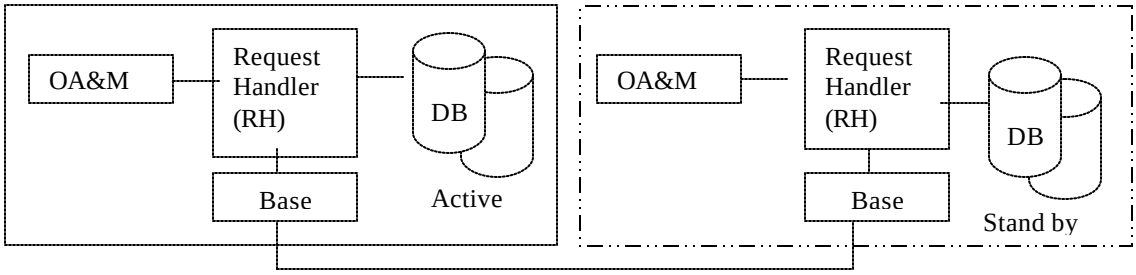

Figure 1: System under study

more significant when the number of parameters and input values per parameter increases.

3. Software Architecture Analysis Using Design Combinatorial Theory

Software architecture analysis shares similar idea with testing. In software architecture analysis, scenarios or use cases are commonly used; while in testing, test cases are mandatory. Therefore, it is reasonable to believe that it would be as effective in scenario generation as it is in test case generation. This section presents a case study of applying the design combinatorial theory to generating failure scenarios of a real telecommunications system. The case study demonstrates the applicability of design combinatorial theory in supporting software architecture reliability analysis.

The approach is applied in identifying scenarios that are useful in evaluating the robustness aspect of a telecommunications software system. Reliability is crucial to telecommunications software. The standard reliability requirement for this type of systems is 99.999%. Therefore, it is extremely critical to conduct thorough software architecture reliability analysis for a product.

3.1. System Under Study

The system under study is a big server in a network. The server will receive many incoming messages concurrently from various sources and it has to process the messages in real-time to satisfy the reliability requirement. Along with system commands/ messages, the server is also responsible for receiving user's query or update messages to the database. The main components of the system include the Base, SCS (Service Control System), SH and RH (Service and Request Handler), Database, SIBBs (Service Independent Building Blocks), OA&M (Operation, Administration, and Management). The Base deals with external communications of the message. The SH and RH provide specific services in response to the incoming messages. The SIBBs are composed of many reusable software components that can be used to build a specific service. Database contains millions of subscriber and support records. OA&M contains many sub-components to monitor the network and resources, keep track of the log and raise alarm in the event of exceptions, and so on.

The system under study is designed to support various configurations. For example, it can be used with or without a hot standby system as well as with or without a mirrored database in both active and standby system. The system under study is shown partially in the following Figure 1.

3.2. Classification of faults

To support software architecture analysis, the approach adopted was similar to traditional approaches [Kazman96, Lung00, Clements02]. The architecture is captured with various views [Kruchten95, Lung00, Nord04]. A list of scenarios are then identified to walk through the architecture. However, extra efforts are needed to evaluate the reliability aspect of the target system, since the system is complicated and the reliability requirement is extremely high. The evaluation objective from the reliability perspective is first identified as:

Reliable software with built-in fault tolerance must be able to handle faults gracefully.

The first step in this study is to categorize the different classes of faults that can happen to the system. The followings show the major different classes of failures: (i) hardware, (ii) network, (iii) software process and (iv) resource exhaustion or congested messages.

Next step is to identify different mechanisms by which the failure can occur under each class. This includes identifying different locations (e.g. hardware, software processes) as well as the different types of failure under each failure class. For instance, the hardware failures can happen in the active system as well as in the standby system. The following

demonstrates some possible scenarios from the reliability perspective.

1. Hardware failures
   1.1 Power failure. The Base encounters a power outage. This could be either a switch off or an unexpected outage.
   1.2. CPU failure.
   1.2.1. CPU failure occurs at the active card.
   1.2.2. CPU failure occurs at the hot standby card.
2. Network failures
   2.1. Connection between the Base Framework and the system management fails.
   2.2. Connection between the active side and the hot standby side fails.
3. Software process/thread failures
   3.1. A process exceeds the execution time limit.
   3.2. A service control block is corrupted.
   3.3. high CPU consumption by a non-real-time process
4. Resource management and other
   4.1. Disk space exhaustion.
   4.2. Memory exhaustion.
   4.3. CPU utilization exceeds the QoS threshold.

As can be seen, it is extremely difficult to handcraft all realistic scenarios that may concurrently happen. It is fairly easy though to generate all possible failure scenarios; however, most of the scenarios may not be feasible and/or worth considering. Also, the number of scenarios would be huge. In this case study we have used the design combinatorial approach to generate the failure scenarios. It serves two purposes: (i) provides a formal mechanism for creating scenarios that may have been omitted if the analysis is conducted manually; (ii) provides a smaller set of scenarios which is more effective than all possible combinations and/or any randomly generated combinations.

3.3. Architectural Analysis Approach

Step 1: For each major class of fault, we have identified the parameters and different values for each parameter. The combinations of the values of the parameters will determine a specific failure scenario. For example, the hardware failure scenario can be modeled in terms of the parameters and the values as shown in Table 2. Each column header of Table 2 represents the parameter and each line under the column represents a value for that particular parameter.

The models for all major fault classes are created. It is worth-mentioning that this step will require significant domain knowledge about the system. In this case study, we have created two types of software failure classes – (i) deals with database operation failure related to the database record, e.g., record not found, corrupted and/or exceeds some limit as shown in Table 3 and (ii) related to various software process failures as shown in Table 4. The failure type in these processes could be different as well; for example, the process could be either dead or timed out.

Step 2: Once the individual models are created, some combined models are created by choosing the various possible interactions among the various failure classes. Special attention is given to the quality of service (QoS) aspect as it plays an important role in reliability. For example, if the software is involved in credit card transaction or any transaction that requires database update, it is critical to handle any failure scenario more gracefully than if the software is doing some other non-critical message processing. Table 5 shows the different message types that may be received The system configuration may also be of interest in architectural scenario analysis. The system under study is designed such a way that it can be configured in a full-duplex mode meaning with a hot stand-by system with a mirrored database in both active and stand-by systems

Table 2: Model for generating hardware failure scenarios

Failure Type	Card	OA&M State
CPU DISK MEMORY General Communications	Active Standby	Power off Out of service

Table 3: Model for generating software failure scenarios - I

Process	Failure Types
RH SIBB TCB DB (Database) OA&M	Dead Timeout

Table 4: Model for generating software failure scenarios - II

Record Type	Failure Type
Subscriber Support	NotFound ExceedsLimit Corrupted

Table 5: Message types

Message ID	Message Type
MSG1	Update
MSG2	Begin
MSG3	End
MSG4	Continue
MSG5	Query

Table 6: System configurations

Config ID	System	Database
CFG1	Both Active & Standby	Mirror present
CFG2	Both Active & Standby	Mirror not present
CFG3	Active only	Mirror present
CFG4	Active only	Mirror not present

and/or other combinations of the databases and active/standby systems. The possible combinations are shown in Table 6.

Step 3: Using design combinatorial theory smaller set of failure scenarios are created from the combined model and the possible configurations. The step is described using an example in details in the following section.

3.4. Techniques and Tools Used

As mentioned earlier, the method based on design combinatorial theory is found to be useful and effective in generating test cases. Pairwise interaction produces a reasonable size as well as an effective test set[Cohen94]. The technique used in this study is based on the similar concept. It is not our focus to compare these methods. Rather, the main focus is to demonstrate the applicability of the concept to software architecture analysis early in the life cycle.

We have developed a prototype tool which generates pairwise combinations, given a number of parameters and the possible values for each parameter. The prototype tool, SmartTC, takes a simple file format as input where the input, i.e., the data model is given in terms of parameters and possible values. It is an important step to come up with a good model and a few iterations are often required to achieve that [Burr98]. The example presented in this section shows a sample input data model which is used to analyze the various scenarios for database related failure. The input data model, consisting of two concurrent database messages is shown below:

```
# Model3: SBDatabase
Standby:Yes,No
MirroredActiveDB:No,Yes
MirroredStandbyDB:No,Yes
RHFailure:None,Timeout,Dead
DBMsg1:Query,Update
DBMsg2:Query,Update
ActiveMainDBFailure:None,SubscriberRecordLimi
tExceed,SupportRecordLimitExceed,Timeout,Dead
```

The output from the tool provides all possible pairwise combinations between any two parameters as shown in Table 7 below. As can be seen from the table below, each row describes a specific scenario. If we were to generate all possible scenarios for this data model, as depicted above, we would have ended up with 2 x 2 x 2 x 3 x 2 x 2 x 5 = 480 scenarios. While it is a paramount task to analyze all these scenarios, it is also not desirable to simply analyze scenarios in an ad-hoc manner as one may easily miss critical scenarios. The pairwise technique is a compromise between ad-hoc and exhaustive analysis. It can be used to complement the conventional practices. The scenarios created by this technique are reasonable in size and they are realistic. For example, row 20 in Table 7 describes the following scenario:

- the system has a standby system
- the system has no mirrored databases
- there is no RH (request handler process) failure
- two concurrent database messages are in the queue – one update and one query
- the DB process is dead

The above scenario is an example of a realistic scenario and needs to be analyzed to see whether the system can recover gracefully from this failure situation. From the user point of view, a query can wait and/or even be dropped under some circumstances. However, the update must happen immediately to ensure data consistency. From our experience, we have found that there was indeed an issue with database update. Although the server was designed to update the database immediately, it did not happen right away as expected due to the page buffering mechanism supported in the third party operating system. After the scenario based analysis, the issue was revealed and notified to the third party operating system vendor to resolve. It is worth-mentioning here that we cannot simply use all possible and/or only pairwise combinations. Pairwise or higher order combinations can be used as a core optimization strategy for generating scenarios when the number of all possible combinations is very high. However, some adjustments are needed in order to ensure important scenarios are included as well as some scenarios that are not feasible or invalid are excluded. The tool has a feature called ”constraints” to incorporate the domain knowledge by including and excluding user given combinations.

4. Summary

Scenarios are commonly used in architectural analysis. However, scenario generation often is conducted in an ad-hoc manner based on practitioners’ expertise and experience. This approach may not work well due to possible complicated cases. This paper incorporated more formalism to the architectural analysis process, which is vital to real-time telecommunications systems

Table 7: Generated failure scenarios

#	Standby	Mirror (Active)	Mirror (Standby)	RHFailure	DBMsg 1	DBMsg 2	DBFailure
1	Yes	No	No	None	Query	Query	None
2	Yes	No	No	None	Query	Query	SubscriberRecordLimitExceed
3	Yes	No	No	None	Query	Query	SupportRecordLimitExceed
4	Yes	No	No	None	Query	Query	Timeout
5	Yes	No	No	None	Query	Query	Dead
…							
20	Yes	No	No	None	Query	Update	Dead
…							
33	No	No	No	None	Query	Update	SubscriberRecordLimitExceed
34	No	No	No	None	Update	Query	SubscriberRecordLimitExceed
35	No	No	No	Timeout	Query	Query	SubscriberRecordLimitExceed
36	Yes	No	No	None	Update	Query	SupportRecordLimitExceed

with extremely high reliability requirements. The design combinatorial theory based technique has been presented in many literatures and found to be useful in testing late in the life cycle. In the case study, we demonstrated how the same concept could be used early in scenario-based architectural analysis with special attention to the reliability requirements. The technique helped improve scenario coverage, especially for multiple concurrent scenarios. The technique when used with domain expertise can add tremendous value early in the design.

References

[Burr98] K. Burr and W. Young, "Combinatorial test techniques: Table-based automation, test generatio and code coverage", *Proc. of the International Conference on Software Testing, Analysis, and Review (STAR'98), October 26-28, 1998*, 1998.

[Clements02] P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison Wesley, 2002.

[Cochran50] Cochran, *Experimental Design*, Willey, New York, 1950.

[Cohen94] D.M. Cohen, S.R. Dalal, A. Kajla, and G.C. Patton, "The automatic efficient test generator", *Proc. IEEE Int. Symposium Software Reliability Eng*, pp. 303-309, 1994.

[Cohen96] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton, "The Combinatorial Design Approach to Automatic Test Generation", *IEEE Software*, vol. 13 (5), pp. 83-89, Sept. 1996.

[Colbourn04a] C.J. Colbourn, M.B. Cohen, and R.C. Turban, "A Deterministic Density Algorithm for Pairwise Interaction Coverage", *Proceedings of the IASTED International Conference on Software Engineering (SE 2004)*, Feb 2004, pp. 245-252.

[Colbourn04b] C.J. Colbourn, "Combinatorial Aspects of Covering Arrays", Le Matematiche (Catania), Sept 2004.

[Dalal99] S.R. Dalal, A. Jain., N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton, B.M. Horowitz, "Model Based Testing in Practice", *Proc. of Int'l Conf on Software Eng.*

[Grindal04] M. Grindal, J. Offutt, S. Andler, "Combination Testing Strategies: A Survey", *Technical Report ISE-TR-05-05*, Geroge Mason Univ., July 2004.

[Kazman96] R. Kazman, G. Abowd, L. Bass, and P. Clements, "Scenario-based analysis of software architecture", *IEEE Software*, Nov 1996.

[Kuhn02] D.R. Kuhn and M.J. Reilly, "An investigation of the Applicability of Design of Experiments to Software Testing", *Proceedings of the 27th NASA/IEEE Software Engineering Workshop*, NASA Goddard Space Flight Center, 4-6 December, 2002.

[Kruchten95] P. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, vol. 12 , no, 6, Nov 1995, pp.42-50

[Lung98] C.-H. Lung, A. Jalnurpukar, and A. El-Rayess, "Performance-Oriented Software Architecture Analysis", *Proc. of the Int'l Workshop on Software Performance Eng. (WOSP)*, pp. 191-196, 1998.

[Lung00] C.-H. Lung and K. Kalaichelvan, "An approach to quantitative software architecture sensitivity analysis", *Int'l Journal of Software Eng and Knowledge Eng*, vol. 10, no. 1, Feb 2000, pp. 97-114.

[Nord04] R. Nord, W. G. Wood, and P. C. Clements, *Integrating the Quality Attribute Workshop (QAW) and the Attribute-Driven Design (ADD) Method*, Technical Note, CMU/SEI-2004-TN-017, July 2004.