Traffic-aware Stress Testing of Distributed Real-Time Systems based on UML Models using Genetic Algorithms

By

Vahid Garousi

A thesis submitted to
The Faculty of Graduate Studies and Research
In partial fulfillment of the requirements of the degree of
Doctor of Philosophy

Ottawa-Carleton Institute of Electrical and Computer Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, K1S 5B6
Canada

August 2006

Copyright © 2006 by Vahid Garousi

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
Acceptance of the thesis

Traffic-aware Stress Testing of Distributed Real-Time Systems based on UML Models using Genetic Algorithms

Submitted by
Vahid Garousi

In partial fulfillment of the requirements for the degree of

Doctor of Philosophy

____

Professor L. C. Briand (Co-Supervisor)

____

Professor Y. Labiche (Co-Supervisor)

____

Professor V. C. Aitken (Department Chair)

Carleton University
August 2006

Abstract

A stress test methodology aimed at increasing chances of discovering faults related to distributed traffic in distributed systems is presented. The technique uses as input a specified UML 2.0 model of a system, augmented with timing information, and yields stress test requirements composed of specific Control Flow Paths along with time values to trigger them.

We propose different variants of our stress testing methodology to test networks and nodes of a system under test according to various heuristics. Using a real-world system specification, we design and implement a prototype distributed system and describe, for that particular system, how the stress test cases are derived and executed using our methodology.

We report the results of applying our stress test methodology on the prototype system and discuss the usefulness of the technique. Results indicate that the technique is significantly more effective at detecting distributed traffic-related faults when compared to standard test cases based on an operational profile. Furthermore, a sophisticated stress test technique based on Genetic Algorithms is proposed to handle specific constraints in the context of Real-Time distributed systems.

Acknowledgments

I would like to express my sincere gratitude to all the people who have been an integral part of my graduate experience. First and foremost I would like to thank my thesis advisors: Prof. Lionel C. Briand and Dr. Yvan Labiche for their help and guidance throughout my PhD career. Lionel and Yvan, thanks for supervising my work with continuous encouragement, interest and enthusiasm. Our discussions always gave me thinking material for a few days of mental digestion, resulting in an extremely stimulating guidance. I am greatly indebted for your genuine interest in my work and for the high quality of your advices. *Merci beaucoup tous les deux!*

I would like to thank my thesis committee members: Dr. Jeff Offutt, Dr. Alan Williams, Dr. Nicola Santoro, Dr. Greg Franks, and Dr. Murray Woodside for taking the time to read this thesis and providing me with valuable comments.

I would like to thank Siemens Corporate Research in New Jersey and the Canada research chair fund in Software Quality Engineering for supporting my PhD work financially.

I would also like to thank everyone at the SQUALL Lab for a great work environment. I had the pleasure to be a colleague of the following individuals (in alphabetical order): Zaheer Bawar, Michael G. Bowman, Hongyan Chen, Jim Cui, Daniel Di Nardo, Bin Dong, Wojciech Dzidek, Maged Elaasar, Wladimir de Lara Araujo Filho, Joanne Leduc, Qing Lin, Xuetao Liu, Yanhua Liu, Reymes M. Rivera, Samar Mouchawrab, Alex Sauve, Mike Sowka, Tao Yue, and Gregory Zoughbi. My thanks also goes to the friendly staff at our department office: Blazenka Power, Jennifer Poll, Darlene Hebert, and Anna Lee for providing support in many occasions.

On a personal note, even if I was more than 9,084 km away from my homeland (Miyana, Azerbaijan), I had any moment the possibility to find a piece of Azerbaijani culture, humor or spirituality thanks to the community of Azerbaijanis in Ottawa and in Canada (in large): Şovqi Mürsəlov, Fəriba Zəmanı, Şəhruz Torfax, Faxtə Zəmanı, Əli Bəxşı, Əkbər Macidov, Nazila Isgəndərova, Şəhriyar Rəhnəmayan, Fəxrəddin Qurbanov, Ceyhun Şahverdiyev, and Orxan Haciyev; to name among many. Thank you all for the good time I spent in your company. Şovqi, thanks for devoting your time to teach me how to play Tar, and thanks for inviting us all to dinner parties at your place to feel at home in many occasions. Şəhruz, thanks for your real friendship. *Hammınız, sağ olun!*

I would also like to express my appreciation to my master's supervisor, Prof. Amir Keyvan Khandani, and his wife, Dr. Ladan Tahvildari (faculty members of the University of Waterloo), who provided me with academic and personal encouragements in many occasions. *Kheili mamnoon!*

Last but not least, I am indebted to Xiao Han whose love, encouragement and support in the past several years made all this work possible. Furthermore, my family members were always sources of support from a long distance: my parents Sürəyya and Yusif, my sisters Sonya, Röya and Gülara, and my brother Nəvid. I could not have achieved this without their unlimited help and encouragements. *Sizə bır ömur can sağlıqı, uğurlar və sələmət arzulayıram!*

Vahid Garousi
August 2006
Ottawa, Canada

To my parents:
Surayya Shahbazi and Yusif Garousi
and my homeland: Azerbaijan

Ana və atama:
Sürəyya Şəhbazlı and Yusif Gəruslı
və ana yurduma: Azərbaycan

Table of Contents

CHAPTER 1 INTRODUCTION 1
1.1 MOTIVATION AND GOAL 1
1.2 APPROACH 3
1.3 CONTRIBUTIONS 4
1.4 STRUCTURE 5
CHAPTER 2 BACKGROUND 7
2.1 RELATED WORKS 7
2.2 PROBLEM STATEMENT (INITIAL) 12
2.3 TERMINOLOGY 12
2.4 UML PROFILE FOR SCHEDULABILITY, PERFORMANCE, AND TIME 14
2.5 AN OVERVIEW ON UML 2.0 SEQUENCE DIAGRAMS 17
CHAPTER 3 AN EXTENDED FAULT TAXONOMY FOR DISTRIBUTED REAL-TIME SYSTEMS 20
3.1 AN EXTENDED FAULT CLASSIFICATION FOR DISTRIBUTED REAL-TIME SYSTEMS 21
3.1.1 Persistency of Faults 23
3.1.2 Distribution 25
3.1.3 Time Criticality 30
3.1.4 Concurrency 30
3.1.5 Resource-Usage Orientation 31
3.1.6 Location of Creation or Occurrence 32
3.2 CHAIN OF DISTRIBUTION FAULTS 32
3.3 CLASS OF FAULTS CONSIDERED IN THIS WORK 33
CHAPTER 4 OVERVIEW OF THE STRESS TEST METHODOLOGY 34
4.1 STRESS TEST METHODOLOGY 34
4.2 INPUT AND INTERMEDIATE MODELS IN OUR STRESS TEST METHODOLOGY 36
4.2.1 Input System Design Models 36
4.2.2 Test Models 38
CHAPTER 5 INPUT SYSTEM MODEL 41
5.1 SEQUENCE DIAGRAM 42
5.1.1 Timing Information of Messages in SDs 44
5.2 CLASS DIAGRAM 48
5.3 MODIFIED INTERACTION OVERVIEW DIAGRAMS 49
5.3.1 Existing Representations to Model Inter-SD Constraints 51
5.3.2 Modified Interaction Overview Diagrams 55
5.4 CONTEXT DIAGRAM 56
5.5 NETWORK DEPLOYMENT DIAGRAM 58
5.5.1 Extending the Notation of UML 2.0 Deployment Diagrams 61
5.5.2 Network Interconnectivity Graph 63
5.6 MODELING REAL-TIME CONSTRAINTS 64
CHAPTER 6 CONTROL FLOW ANALYSIS OF SEQUENCE DIAGRAMS 69
6.1 AN OVERVIEW OF OUR CONTROL FLOW ANALYSIS TECHNIQUE 70
6.1.1 Challenges of SDs' CFA 70
6.1.2 Towards a CFM for SDs 72
6.1.3 Concurrent Control Flow Graph: a Control Flow Model for SDs 74

6.1.4 Consistency Mapping Rules from SDs to CCFGs............*77*
6.1.5 Concurrent Control Flow Paths............*83*
6.2 INCORPORATING DISTRIBUTION AND TIMING INFORMATION IN CCFPS............84
6.3 FORMALIZING MESSAGES............85
6.4 DISTRIBUTED CCFP............87
6.5 TIMED INTER-NODE AND INTER-NETWORK REPRESENTATIONS OF DCCFPS............88
CHAPTER 7 CONSIDERING INTER-SD CONSTRAINTS............91
7.1 INDEPENDENT-SD SETS............93
7.1.1 Definitions............*94*
7.1.2 Derivation of Independent-SD Sets............*95*
7.1.3 Algorithm Complexity............*97*
7.2 CONCURRENT SD FLOW PATHS, CCFP AND DCCFP SEQUENCES............98
7.2.1 Concurrent SD Flow Paths............*98*
7.2.2 Concurrent Control Flow Paths Sequence............*100*
7.2.3 Duration of a Concurrent Control Flow Path Sequence............*102*
CHAPTER 8 RESOURCE USAGE ANALYSIS OF DISTRIBUTED TRAFFIC..105
8.1 ESTIMATING THE DATA SIZE OF A DISTRIBUTED MESSAGE............106
8.1.1 Effect of Inheritance............*109*
8.1.2 Messages with Indeterministic Sizes............*110*
8.2 FORMALIZING RELATIONSHIPS BETWEEN NODES AND NETWORKS............111
8.2.1 Node-Network and Network-Network Memberships............*112*
8.2.2 Network Paths Function............*113*
8.3 DISTRIBUTED TRAFFIC USAGE ATTRIBUTES............113
8.3.1 Location: Nodes vs. Networks............*114*
8.3.2 Direction (for nodes only): In, Out, Bidirectional............*116*
8.3.3 Type: Amount of Data vs. Number of Network Messages............*117*
8.3.4 Duration: Instant vs. Interval............*119*
8.4 ASPECTS TO CONSIDER WHEN ESTIMATING NETWORK TRAFFIC USAGE............121
8.4.1 Effects of Multiple Network Paths between Nodes............*121*
8.4.2 Delay in Network Transmissions............*124*
8.4.3 Traffic Distribution of Messages with Durations more than a Time Unit............*126*
8.4.4 Effect of Concurrent Processes............*127*
8.5 A CLASS OF TRAFFIC USAGE ANALYSIS FUNCTIONS............128
8.5.1 Naming Convention............*128*
8.5.2 Functions............*129*
CHAPTER 9 TIME-SHIFTING STRESS TEST TECHNIQUE............136
9.1 PROBLEM STATEMENT: REVISITED............137
9.2 STRESS TEST HEURISTIC............137
9.3 AN EXAMPLE TO ILLUSTRATE THE HEURISTIC............138
9.4 EXCERPTS............141
CHAPTER 10 GENETIC ALGORITHM-BASED STRESS TEST TECHNIQUE143
10.1 TYPES OF ARRIVAL PATTERNS............145
10.2 ANALYSIS OF ARRIVAL PATTERNS............149
10.3 ACCEPTED TIME SETS............154
10.4 FORMULATING THE PROBLEM AS AN OPTIMIZATION PROBLEM............159
10.5 IMPACT OF ARRIVAL PATTERNS ON STRESS TEST STRATEGIES............159

10.5.1 Impact on Instant Stress Test Strategies *160*
10.5.2 Impact on Interval Stress Test Strategies *160*
10.5.3 How Arrival Patterns are Addressed by Stress Test Strategies *164*
10.6 CHOICE OF THE OPTIMIZATION METHODOLOGY: GENETIC ALGORITHMS 165
10.7 TAILORING GENETIC ALGORITHM TO DERIVE INSTANT STRESS TEST REQUIREMENTS 167
10.7.1 Chromosome *168*
10.7.2 Constraints *171*
10.7.3 Initial Population *172*
10.7.4 Determining a Maximum Search Time *175*
10.7.5 Objective (Fitness) Function *182*
10.7.6 Operators *184*
10.8 INTERVAL STRESS TEST STRATEGIES ACCOUNTING FOR ARRIVAL PATTERNS 191
10.9 IMPACTS OF UNCERTAINTY IN TIMING INFORMATION ON OUR STRESS TEST METHODOLOGY 197
10.10 WAIT-NOTIFY STRESS TEST TECHNIQUE 199
CHAPTER 11 AUTOMATION AND ITS EMPIRICAL ANALYSIS 202
11.1 GALIB 202
11.2 GARUS 204
11.2.1 Class Diagram *204*
11.2.2 Activity Diagram *206*
11.2.3 Input File Format *207*
11.2.4 Output File Format *211*
11.3 VALIDATION OF TEST REQUIREMENTS GENERATED BY GARUS 212
11.3.1 Satisfaction of ATSs by Start Times of DCCFPs in the Generated Stress Test Requirements *215*
11.3.2 Checking the extent to which ISTOF is maximized *216*
11.3.3 Repeatability of GA Results across Multiple Runs *218*
11.3.4 Convergence Efficiency across Generations *220*
11.3.5 Our Strategy for Investigating Variability/Scalability *224*
11.3.6 Impacts of Test Model Size (Scalability of the GA) *234*
11.3.7 Impacts of Arrival Pattern Types *245*
11.3.8 Impacts of Arrival Pattern Parameters *252*
11.3.9 Impact of Maximum Search Time *264*
CHAPTER 12 CASE STUDY 268
12.1 AN OVERVIEW OF TARGET SYSTEMS 269
12.2 CHOOSING A TARGET SYSTEM AS CASE STUDY 271
12.2.1 Requirements for a Suitable Case Study *271*
12.2.2 None of the Systems in our survey Meets the Requirements *272*
12.3 OUR PROTOTYPE SYSTEM: A SCADA-BASED POWER SYSTEM 274
12.3.1 SCAPS Specifications *274*
12.3.2 SCAPS Meets the Case-Study Requirements *276*
12.3.3 SCAPS UML Design Model *277*
12.3.4 Stress Test Objective *291*
12.3.5 Implementation *292*
12.3.6 Hardware and Network Specifications *294*

12.4 Stress Test Architecture 295
12.5 Running Stress Test Cases 297
12.6 Building the Stress Test Model for SCAPS 298
12.6.1 Network Interconnectivity Tree 298
12.6.2 Control Flow Analysis of SDs 299
12.6.3 Derivation of Distributed Concurrent Control Flow Paths 302
12.6.4 Derivation of Independent-SD Sets 303
12.6.5 Derivation of Concurrent SD Flow Paths 304
12.6.6 Data Size of Messages 306
12.7 Stress Testing SCAPS by Time-Shifting Stress Test Technique 307
12.8 Stress Testing SCAPS by Genetic Algorithm-based Stress Test Technique308
12.8.1 Using GARUS to Derive Stress Test Requirements 308
12.8.2 Test Results 318
12.8.3 Conclusions 324
CHAPTER 13 GENERALIZATION OF OUR STRESS TEST METHODOLOGY TO TARGET OTHER TYPES OF FAULTS 325
13.1 Targeting other Types of Resources 325
13.1.1 Resource Usage Analysis of other Types of Resources 326
13.1.2 CPU 329
13.1.3 Memory 333
13.2 Targeting other Types of Faults 335
13.2.1 Distributed Unavailability Faults 336
13.2.2 Resource Unavailability Faults 338
13.2.3 Concurrency Faults 339
CHAPTER 14 SUMMARY 343
14.1 Conclusions 343
14.2 Future Research Directions 345

List of Tables

Table 1-A stereotype to model SRT constraints. 67
Table 2-A stereotype to model HRT constraints. 67
Table 3-Tagged values of SRT and HRT stereotypes. 67
Table 4- Mapping rules from SDs to CCFGs. 78
Table 5-Data size of some of the primitive data types in Java (adopted from [75]).......... 109
Table 6-A set of heuristics to identify a suitable MST ($MST_{suitable}$). 179
Table 7-(a): Durations of several CCFPs. (b): Arrival patterns of several SDs. 196
Table 8-Output format of 10 schedules generated by GARUS. 217
Table 9-Descriptive statistics of the maximum ISTOF values over 1000 runs. Values are in units of data traffic (e.g. KB). 218
Table 10-Summary of GARUS results for five runs. 223
Table 11- Variability parameters for experimental test models. 225
Table 12-Experimental test models with different sizes. 230
Table 13-Experimental test models with variations in SD arrival patterns. 232
Table 14-Execution time statistics of 1000 runs of *tm1…tm6*. 235
Table 15-Descriptive statistics of the maximum ISTOF values for each test model over 1000 runs. Values are in units of data traffic. 239
Table 16-Descriptive statistics of the maximum stress time values for each test model over 1000 runs. Values are in time units. 243
Table 17-Minimum, maximum and average values of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of each test model. 245
Table 18-Execution time statistics of 1000 runs of *tm7…tm11*. 246
Table 19-Descriptive statistics of the maximum ISTOF values for each test model over 1000 runs. Values are in units of data traffic. 249
Table 20-Descriptive statistics of the maximum stress time values for each test model over 1000 runs. Values are in time units. 250
Table 21-Minimum, maximum and average values of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of each test model. 252
Table 22-Execution time statistics of 1000 runs of *tm8, tm9, tm10, tm12,…tm18*. 255
Table 23-Descriptive statistics of the distributions in Figure 109. 257
Table 24-Descriptive statistics of the distributions in Figure 110. 259
Table 25-Descriptive statistics of the distributions in Figure 111. 262
Table 26-Descriptive statistics of the distributions in Figure 112. 267
Table 27-Mean data sizes of the entity data classes of SCAPS. 307
Table 28-An operational profile for SCAPS. 320
Table 29-Probabilities of taking DCCFPs of SDs *OC* and *PRNF* according to the operational profile given in Table 28. 321
Table 30-Quantiles of the distributions in Figure 144. 323

List of Figures

Figure 1-The structure of the GRM Framework of the UML-SPT profile [12]. 15
Figure 2-Part of the deployment architecture of a chemical reactor system.................... 16
Figure 3-Example of time modeling using UML-SPT profile... 17
Figure 4-UML 2.0 Sequence Diagram Metamodel. 18
Figure 5-An example illustrating the new features of the UML 2.0 SDs. 19
Figure 6-The fundamental chain of dependability threats. 21
Figure 7-Tree of Generalized Fault Classes for Distributed Systems.............................. 23
Figure 8-Occurrences of Distributed Unavailability Faults (DUF). 27
Figure 9-An example scenario showing how a distributed traffic fault might happen..... 29
Figure 10- Overview of our model-based stress test methodology (a UML activity diagram). 35
Figure 11- Metamodel of input and intermediate test models in our stress test methodology. 37
Figure 12-Modeling the deployment node of an object using *node* tagged value. 43
Figure 13- Probabilistic representation of uncertainty in a task's execution times. 47
Figure 14-The approach in which the different SD constraint types are considered by the two optimization algorithms in this work. 50
Figure 15- Use Case Sequential Constraints for the Librarian actor (adopted from [60]). 52
Figure 16-Interaction Overview Diagram (IOD) of a simplified ATM system............... 54
Figure 17- Modified Interaction Overview Diagram (MIOD) of a simplified ATM system. 55
Figure 18-A controller system made of several sensors. 57
Figure 19-(a): Modeling concurrent instances of SDs inside MIOD. (b): Equivalent in meaning to (a). 58
Figure 20-A simple network deployment for an online shopping service. 59
Figure 21-A metamodel for network topologies.. 60
Figure 22-A network topology.. 60
Figure 23-Using a Network Deployment Diagram (NDD) to model the network topology of Figure 22. 62
Figure 24-Modeling network interconnectivity of a university network. 63
Figure 25-Network Interconnectivity Graph (NIG) of the topology in Figure 22. 64
Figure 26- Examples usages of the «SRTaction» and «HRTaction» stereotypes in a SD and in a MIOD. 68
Figure 27-A SD with asynchronous messages.. 71
Figure 28- A SD with *par* operator. 72
Figure 29- CCFG metamodel... 75
Figure 30-CCFG of the SD in Figure 27. 79
Figure 31-(a)-Part of the CCFG instance *ccfg* mapped from the SD in Figure 27, satisfying the consistency rule #2. (b)-Part of the CCFG, corresponding to the instance shown in (a)... 81
Figure 32-(a):Part of the CCFG instance *ccfg* mapped from the SD in Figure 27, satisfying the consistency rule #3. (b): Part of the CCFG, corresponding to the instance shown in (a)... 83
Figure 33-CCFPs of the CCFG in Figure 30. 83

Figure 34- DCCFPs of the CCFPs in Figure 33. 88
Figure 35-Timed inter-node representation of *DCCFP*(ρ_2) in Figure 34. 89
Figure 36-A simple system NIG. 89
Figure 37-Timed inter-network representation of a DCCFP. 90
Figure 38- The MIOD of a library system. 93
Figure 39-The Independent SD Graph (ISDG) corresponding to the MIOD in Figure 38. The ISDS=*{A,B,G,H}* is shown with dashed edges. 96
Figure 40-A MIOD with a multi-SD construct. 97
Figure 41-An example MIOD and the CCFG of one of its SDs. 99
Figure 42-The call tree of the recursive algorithm *Duration* applied to $CCFPS_1$. 104
Figure 43-A class diagram showing three classes with data fields. 109
Figure 44-A Network Interconnectivity Graph (NIG). 112
Figure 45-A system made up of four nodes and three networks. 115
Figure 46-Timed inter-node and inter-network representations of three DCCFPs. 116
Figure 47-A typical system composed of two nodes and four processes. 118
Figure 48-Network traffic diagram (data traffic) of $DCFP_2$ in Figure 47. 118
Figure 49-Network traffic diagram (number of distributed messages) of $DCFP_2$ in Figure 47. 119
Figure 50-"In-data" traffic diagram of a node, highlighting difference between *instant* and *interval* (3ms) traffic. 120
Figure 51-Example Network Interconnectivity Graph (NIG). 122
Figure 52-A Network Deployment Diagram (NDD) annotted with network transmission delay information. 125
Figure 53-The data traffic diagram of a node with two processes. 127
Figure 54-Naming convention for the traffic usage functions. 128
Figure 55-A simple system NIG. 138
Figure 56-Heuristic to stress test instant data traffic on a network. 140
Figure 57-Activity diagram of stress test strategy *StressNetInsDT(net)*. 141
Figure 58- Pseudo-code to check if the arrival pattern *AP* is satisfied by an arrival time. 150
Figure 59-Accepted Time Intervals (ATI) of a *bounded* arrival pattern (*'bounded', (4, ms), (5, ms)),* i.e. *MinIAT=4ms, MaxIAT=5ms.* 151
Figure 60-Accepted Time Intervals (ATI) of the bursty arrival pattern (*'bursty', (5, ms), 2)*. 152
Figure 61-Accepted Time Points (ATP) of the irregular inter-arrival pattern (*'irregular', (1, ms), (5, ms), (6, ms), (8, ms), (10, ms))*. 153
Figure 62-Accepted Time Intervals (ATI) of the periodic inter-arrival pattern (*'periodic', (5, ms), (1, ms))*. 153
Figure 63-Probability Distribution Function (PDF) of (*'poisson', (5, ms))* arrival pattern. 154
Figure 64-(a): Accepted Time Set (ATS) metamodel. (b): Three instances of the metamodel. 155
Figure 65- Illustrating the overlap of two ATSs' intervals. 158
Figure 66-Example intersections of two ATSs. 158
Figure 67-Formulating the problem of generating stress test requirements as an optimization problem. 159

Figure 68-Impact of arrival patterns on instant (a)-(b) and interval (c)-(d) stress test strategies. 162
Figure 69-SD arrival pattern constraints. 164
Figure 70-(a): Metamodel of chromosomes and genes in our GA algorithm. (b): Part of an instance of the metamodel. 169
Figure 71- Constraint #1 of the GA (an OCL expression) 171
Figure 72-Constraint #2 of the GA (an OCL function). 172
Figure 73-Pseudo-code to generate a chromosome for the GA's initial population 173
Figure 74-An example where the ATS intersection of all SDs is null, but they can overlap. 175
Figure 75-Impact of maximum search time on exercising the time domain. 177
Figure 76- The ATSs of three irregular APs. 178
Figure 77-Illustration showing the heuristic of choosing a suitable maximum search time. 181
Figure 78-Computing the Instant Stress Test Objective Function (ISTOF) value of a chromosome. 184
Figure 79-Crossover operators. 187
Figure 80-Two example uses of the crossover operators 188
Figure 81- *DCCFPMutation* operator. 191
Figure 82- *startTimeMutation* operator. 191
Figure 83- *ISDSMutation* operator. 191
Figure 84-Impact of arrival patterns on the duration of a CCFP. 195
Figure 85-Call tree of the recursive algorithm *minAPDuration* applied to a CCFPS. ... 197
Figure 86-Heuristics of the Wait-Notify Stress Test Technique (WNSTT). 201
Figure 87-Basic GAlib class hierarchy (adopted from [99]). 203
Figure 88-Simplified class diagram of GARUS. 205
Figure 89-Overview activity diagram of GARUS. 207
Figure 90-GARUS input file format. 208
Figure 91-An example input file of GARUS. 210
Figure 92-An example DTUP of a DCCFP. 210
Figure 93-(a): Stress test requirements format in GARUS output file. (b): An example. 212
Figure 94-ATSs of the SDs in the TM in Figure 91, and a stress test schedule generated by GARUS. 215
Figure 95-Modified DCCFPs of the test model in Figure 91. 217
Figure 96-(a): Histogram of maximum ISTOF and stress time values for 1000 runs of test model corresponding to the input file in Figure 91. (b): Corresponding max stress time values for one of the frequent maximum ISTOF values (72 units of traffic). 219
Figure 97-Histogram of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of the test model corresponding to the input file in Figure 91 by GARUS. 221
Figure 98-Simplified activity diagram of our random test model generator. 228
Figure 99-Visualization of the average values in Table 14. 235
Figure 100-Histograms of maximum ISTOF values (y-axis) for 1000 runs of each test model. The y-axis values are in traffic units. 239

Figure 101- Probability of the event that at least one test requirement with an ISTOF value in $group_{70}$ is yielded in a series of *n* runs of GARUS. 240
Figure 102-Histograms of maximum stress time values for 1000 runs of each test model. The y-axis values are in time units. 242
Figure 103- Histograms of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of each test model. 244
Figure 104-Visualization of the average values in Table 18. 246
Figure 105-Histograms of maximum ISTOF values for 1000 runs of each test model. The y-axis values are in traffic units. 248
Figure 106-Histograms of maximum stress time values for 1000 runs of each test model. The y-axis values are in time units. 250
Figure 107- The intersection of several periodic ATSs is a *discrete* unbounded ATS... 251
Figure 108- Histograms of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of each test model. 252
Figure 109-Histograms of maximum ISTOF values for 1000 runs of each test model. The y-axis values are in traffic units. 256
Figure 110- Histograms of maximum stress time values for 1000 runs of each test model. The y-axis values are in time units 260
Figure 111- Histograms of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of each test model. 263
Figure 112- Impact of variations in maximum search time on the GA's behavior and outputs. 265
Figure 113-A typical architecture of SCADA systems. 271
Figure 114- Modified SCAPS Use-Case Diagram including a Timer actor. 278
Figure 115- SCAPS network deployment diagram. 279
Figure 116-SCAPS partial class diagram. 280
Figure 117- SDs *OM_ON* and *OM_QC* (Overload Monitoring). 282
Figure 118-SD *queryONData(dataType)*. 283
Figure 119-SD *queryQCData(dataType)*. 283
Figure 120- SD *OC* (Overload Control). 285
Figure 121-SD *DSPS_ON* and *DSPS_QC* (Detection of Separated Power System). 285
Figure 122-SD *PRNF* (Power Restoration after Network Failure). 286
Figure 123-SCAPS Modified Interaction Overview Diagram (MIOD). 290
Figure 124-A screenshot of the main screen of SCAPS. 293
Figure 125-Overview of SCAPS Stress Test Architecture. 295
Figure 126- SCAPS Network Interconnectivity Graph (NIG). 298
Figure 127-CCFG(OM_ON). 299
Figure 128-CCFG(OM_QC). 300
Figure 129-CCFG(OC). 300
Figure 130-CCFG(DSPS_ON). 301
Figure 131-CCFG(DSPS_QC). 301
Figure 132-CCFG(PRNF). 301
Figure 133-CCFP and DCCFP sets of SDs in SCAPS. 302
Figure 134-(a):Independent-SDs Graph (ISDG) corresponding to the MIOD of Figure 123. (b), (c) and (d): Three of the maximal-complete subgraphs of the ISDG (shown with dashed edges), yielding three ISDSs. 304

Figure 135-A grammar to derive CSDFPs from SCAPS' MIOD................................. 305
Figure 136-Some of the CSDFPs of SCAPS derived from the grammar in Figure 135. 306
Figure 137- Input File containing SCAPS Test Model for a GASTT Test Objective.... 310
Figure 138-Relationship between ATSs of SCAPS SDs and their execution durations, *d(sd_name)*, to each other in 50 time units. 311
Figure 139--Relationship between ATSs of SCAPS SDs and their execution durations, *d(sd_name)*, to each other in 200 time units. 313
Figure 140-Histograms of 100 GARUS Outputs for a SCAPS Test Objective............. 314
Figure 141- Probability of the event that at least one test requirement with an ISTOF value in $group_A$ is yielded in a series of *n* runs of GARUS for SCAPS. 316
Figure 142-Five Different Test Requirements (TR) generated by GARUS for a SCAPS Test Objective. 317
Figure 143-Part of *CCFG(OC),* annotated with probabilities of paths after decision nodes. 320
Figure 144-Execution time distributions of test suites corresponding to SRT constraint *SRTC1* by running operational profile test (OPT) and stress test cases from GASTT (ST). 323
Figure 145-Overview activity diagram of the Model-Based Resource Usage Analysis (MBRUA) technique.. 327
Figure 146-Modeling CPU usage example... 331
Figure 147-Memory usage analysis example. 334
Figure 148-Four SDs with distributed messages. 337
Figure 149-Heuristics for the application of the barrier scheduling heuristic in the context of UML-based stress testing. 341
Figure 150-GA chromosome terminology.. 359
Figure 151-Illustration of crossover operator (*single point* crossover). 360
Figure 152-Illustration of mutation operator. 360
Figure 153-Activity diagram of the most general form of genetic algorithms (concept from [87]).. 361
Figure 154-Formalization of bounded APs time properties. 363
Figure 155-Two overlapping ATIs. 364

List of Equations

Equation 1- Distributed Traffic Usage (DTU) functions for different types of messages. 108
Equation 2-Node-network membership function. 112
Equation 3-Network-network membership function. 112
Equation 4-A function to calculate the shares of a network in data transmissions between two nodes. 123
Equation 5-A function to calculate the data transmissions delay of network between a source node and a network (or a destination node). 126
Equation 6-Intersection of two ATSs. 157
Equation 7-A formula to calculate the maximum ATP (*maxATP*) of an irregular AP... 179
Equation 8-A formula to calculate the *Unbounded Range Starting Point (URSP)* of a bounded AP, given the ATIs of the AP. 180
Equation 9-A formula to calculate the *Unbounded Range Starting Point (URSP)* of a bounded AP, given the minimum and maximum inter-arrival times (*minIAT* and *maxIAT*) of the AP. 180
Equation 10- Instant Stress Test Objective Function (ISTOF). 183
Equation 11- Function returning the earliest arrival time of a SD based on its arrival pattern. 195
Equation 12-RUD and RUM for CPU resource. 332
Equation 13-RUD and RUM for memory resource. 335

List of Acronyms

AD	Activity Diagram
AOCS	Attitude and Orbit Control System
AP	Arrival Pattern
APC	Arrival-Pattern Constraint
ATI	Accepted Time Interval
ATP	Accepted Time Point
ATS	Accepted Time Set
BCET	Best-Case Execution Time
BLOB	Binary Large OBject
BNF	Backus-Nauer Form
CCFG	Concurrent Control Flow Graph
CCFP	Concurrent Control Flow Path
CCFPS	Concurrent Control Flow Path Sequence
CFA	Control Flow Analysis
CFG	Control Flow Graph
CFM	Control Flow Model
CFP	Control Flow Path
CPU	Central Processing Unit
CSDFP	Concurrent SD Flow Paths
DBMS	DataBase Management System
DCCFP	Distributed Concurrent Control Flow Path
DCCFPS	Distributed Concurrent Control Flow Path Sequence
DCS	Distributed control system
DRTS	Distributed Real-Time Systems
DT	Data Traffic
DUF	Distributed Unavailability Fault
ECC	Error Correcting Codes
GARUS	GA-based test Requirement tool for distribUted Systems
GASTT	Genetic Algorithm-based Stress Test Technique
GRM	General Resource Modeling
HMI	Human-Machine Interface
HRT	Hard Real-Time
IDE	Integrated Development Environment

IOD	Interaction Overview Diagram
IVSDS	Invalid SD Schedule
IPMCS	Industrial Process Measurement and Control System
ISDS	Independent-SD Sets
ISTOF	Instant Stress Test Objective Function
LAN	Local Area Network
MIOD	Modified Interaction Overview Diagram
MT	Message Traffic
NDD	Network Deployment Diagram
NIG	Network Interconnectivity Graph
NRI	Network Resources Index
OCL	Object Constraint Language
OMG	Object Management Group
OO	Object-Oriented
OPTC	Operation Profile-based Test Case
OPTR	Operation Profile-based Test Requirement
OSI	Open Systems Interconnection
PDF	Probability Distribution Function
PLC	Programmable Logic Controller
PRI	Performance Requirements Index
PSTN	Public Switched Telephone Network
RAD	Rapid Application Development
RandTMGen	Random Test Model Generator
RT	Real-Time
RTCOST	Real-Time Constraint-Oriented Stress Test
SCADA	Supervisory Control and Data Acquisition
SCAPS	A SCADA-based Power System
SD	Sequence Diagram
SDNUM	SD-Network Usage Matrix
SHR	Synchronized Hyperedge Replacement
SPE	Software Performance Engineering
SRT	Soft Real-Time
STPE	Stress-Test Performance Engineering
SUT	System Under Test

TC Te1e-Control unit

TM Test Model

TSSTT Time-Shifting Stress Test Technique

UC Use Case

UCM Use-Case Map

UML Unified Modeling Language

UML-SPT UML profile for Schedulability, Performance, and Time

VSDS Valid SD Schedule

WAN Wide Area Network

WCET Worst-Case Execution Time

WNSTT Wait-Notify Stress Test Technique

Chapter 1

INTRODUCTION

1.1 Motivation and Goal

Distributed Real-Time Systems (DRTS for short) are becoming more important to our everyday life. Examples include command and control systems, aircraft aviation systems, robotics, and nuclear power plant systems [1]. However as described in the literature, the development and testing of a DRTS is difficult and takes more time than the development and testing of a distributed system without real-time constraints or a non-distributed system, one which runs on a single computer.

System testing has been the topic of a myriad of research in the last two decades or so. Most testing approaches target system functionality rather than performance. However, Weyuker and Vokolos point out in [2], that a working system more often encounters problems with performance degradation as opposed to system crashes or incorrect system responses. In other words, not enough emphasis is generally placed on performance testing. In hard real-time systems, where stringent deadlines must be met, this poses a serious problem. Because hard real-time systems are often safety critical systems,

performance failures are intolerable. Deadlines that are not adhered to can in some applications lead to life-threatening risks. The risk of this occurring can be greatly reduced if enough performance testing is done before deploying the system. Performance degradation and consequent system failures due to this degradation usually arise in stressed conditions. For example, stressed conditions can be attained in a DRTS when many users are concurrently accessing a system or when large amounts of data are transferring through a network link.

In a recent paper by Kuhn [3], sources of failures in the United States' Public Switched Telephone Network (PSTN), as a very big DRTS, were investigated. It was reported that in the time period of 1992-1994, although only 6% of the outages were overloads, they led to 44% of the PSTN's service downtime in the respected time frame. In the system under study, overload was defined as the situation in which service demand exceeds the designed system capacity. So it is evident that although overload situations do not happen frequently, the failure consequences they result into are quite expensive.

Furthermore, in the context of *Distributed Control Systems (DCS)* (e.g., [4]) and *Supervisory Control And Data Acquisition (SCADA)* systems (e.g., [5]), reports such as [6], [7], [8], [9] indicate the high risk of failures due to network overload.

The motivation for our work can be stated as follows: because DRTS are by nature concurrent and are often real-time, there is a need for methodologies and tools for stress testing and debugging DRTS under stressing conditions, such as heavy user loads and intense network traffic. The systems should be tested under stress before being deployed in the field. In this work, our focus for stress testing is on the network traffic in DRTS, one of the fundamental factors affecting the behavior of DTRS. Distributed nodes of a

DTRS regularly need to communicate with each other to perform some of the system's functionalities. Network communications are, however, not always successful and timely. Problems such as congestion, transmission errors, or delays might occur in a network. On the other hand, many real-time and safety-critical systems have hard deadlines for many of their operations, where catastrophic consequences may result from missing deadlines. Furthermore, a system might behave well with normal network traffic loads (in terms of either amount of data or number of requests), but the communication might turn to be poor and unreliable if many network messages or high loads of data are concurrently transmitted over a particular network or towards a particular node.

1.2 Approach

We propose a technique to derive test requirements to stress the robustness of a system to network traffic problems in a cost-effective manner. It is based on models, rather than code, describing, among other things, interactions between distributed objects. This is a difficult problem as, for a given DRTS where several concurrent processes are running on each distributed node and processes communicate frequently with each other, the size of the set of all possible network interaction interleavings is unbounded, where a network interaction interleaving is a possible sequence of network interactions among a subset of all processes running on a subset of all nodes.

The Unified Modeling Language (UML) [10] is increasingly used in the development of DRTS systems. Since 1997, UML has become the de facto standard for modeling object-oriented software and is used, in one way or another, by nearly 70 percent of IT industry [11]. The new version of UML, version 2.0 [10], was finalized by the OMG in August 2003. UML 2.0 offers an improved modeling language compared to UML 1.x versions:

enhanced architecture modeling, improved extensibility, support for component-based development, modeling of relationships and model management [11]. As we expect UML to be increasingly used for DTRS, it is therefore important to develop automatable UML model-driven, stress test techniques and this is the main motivation for the work reported here.

Assuming that the UML design model of a DTRS is in the form of Sequence Diagrams (SD) annotated with timing information, and the systems' network topology is given in a specific modeling format, we propose a technique to derive test requirement to stress the DTRS with respect to network traffic in a way that will likely reveal robustness problems. We introduce a systematic technique to automatically generate a network interaction interleaving that will stress the network traffic on a network or a node in a System Under Test (SUT) so as to analyze the system under strenuous but *valid* conditions, e.g., triggering the Withdraw sequence diagram before (or together with) the Login in an ATM system is invalid. If any network traffic-related failure is observed, designers will be able to apply any necessary fixes to increase robustness before system delivery.

1.3 Contributions

The contributions of this work can be summarized as follows:

- An extended faults taxonomy for DRTS (Chapter 3)
- A control flow analysis technique based on UML 2.0 SDs (Chapter 6)
- A resource usage analysis technique for network traffic usage in DRTS (Chapter 8)

- A family of automated stress testing techniques (Chapter 9) aiming at increasing chances of discovering faults related to network traffic in DTRS. Based on a specific UML 2.0 system model, and analysis of the control flow in SDs, it yields stress test requirements composed of specific CFPs (Control Flow Paths) to be invoked and a schedule according to which to trigger each CFP. In addition to sequence diagrams. Two different approaches are discussed for the identification and scheduling of CFPs: the first one is based on a heuristic (Chapter 9); the second one, more general, re-expresses the objectives as an optimization problem and solves it with a Genetic Algorithm (Chapter 10).

1.4 Structure

The remainder of this document is structured as follows. Relevant background information is given in Chapter 2, where we discuss the related works and define the main terminology used throughout the document. Chapter 3 presents an extended fault taxonomy for DRTS so that the types of faults we target are well defined. Chapter 4 presents an overview of the stress test methodology. The assumed input system models for the methodology are precisely described in Chapter 5. From Chapter 6 to Chapter 8, we describe in detail how a stress test model is built to support automation. Chapter 6 describes a technique for the control flow analysis of UML 2.0 sequence diagrams. Chapter 7 discusses how sequential and conditional constraints among sequence diagrams (or their corresponding use cases) can be analyzed when generating stress test requirements. A resource usage analysis technique for network traffic usage is then presented in Chapter 8. Chapter 9 proposes the simpler version of our stress test technique which should be applicable for a large proportion of DTRS. A more

sophisticated version of the technique, which takes into account complex arrival patterns for internal and external system events, is presented in Chapter 10. This technique re-expresses our objectives as an optimization problem and uses Genetic Algorithms to derive test requirements. Chapter 11 discusses how the stress test methodology can be fully automated using a prototype tool we have developed to generate stress test requirements. This tool is carefully assessed by an experiment. A comprehensive case study is presented in Chapter 12 in order to assess the usefulness of our overall methodology on a realistic example. Chapter 13 discusses how our stress test methodology can be generalized to target other types of faults than the one targeted in Chapter 9 and Chapter 10, or other types of resources (e.g. CPU or memory) instead of network traffic. Finally, Chapter 14 concludes this document and discusses some of the future research directions.

Chapter 2

BACKGROUND

This section presents related works (Section 2.1), a detailed problem statement (Section 2.2), the basic terminology used in this thesis (Section 2.3), and a brief introduction to the UML profile for Schedulability, Performance, and Time (UML-SPT) [12] (Section 2.4). As UML 2.0 sequence diagrams are used as the main behavior model, an overview on SDs is presented in Section 2.5.

2.1 Related Works

There has not been a great deal of work published on systematic generation of stress test suites for software systems. The works in [13-17] are notable exceptions. On a different note, there are reports that highlight the high cost of system outages and damages due to high loads and systems' malfunction under stressing conditions. For example, Kuhn [3] investigated the sources of failures in the United States' Public Switched Telephone Network (PSTN)─a very large distributed system. It was reported that in the time period of 1992-1994, although only 6% of the outages were overloads, they led to 44% of the PSTN's service downtimes in the studied time frame.

Authors in [15] propose a class of load test case generation algorithms for telecommunication systems which can be modeled by Markov chains. The black-box techniques proposed are based on system operational profiles. The Markov chain that represents a system's behavior is first built. The operational profile of the software is then used to calculate the probabilities of the transitions in the Markov chain. The steady-state probability solution of the Markov chain is then used to guide the generation process of test cases according to a number of criteria, in order to target specific types of faults. For instance, using probabilities in the Markov chain, it is possible to ensure that a transition in the chain is involved many times in a test case so as to target the degradation of performance due to large numbers of calls/requests that can be accepted by the system. From a practical standpoint, targeting only systems whose behavior can be modeled by Markov chains can be considered a limitation of this work. Furthermore, using only operational profiles to test a system may not lead to stressing situations.

Yang proposed a technique [13] to identify potentially load sensitive code regions to generate load test cases. The technique targets memory-related faults (e.g., incorrect memory allocation/de-allocation, incorrect dynamic memory usage) through load testing. The approach is to first identify statements in the module under test that are load sensitive, i.e., they involve the use of *malloc()* and *free()* statements (in C) and pointers referencing allocated memory. Then, data flow analysis is used to find all Definition-Use (DU)-pairs that trigger the load sensitive statements. Test cases are then built to execute paths for the DU-pairs.

Briand et al. [16] propose a methodology for the derivation of test cases that aims at maximizing the chances of deadline misses within a system. They show that task

deadlines may be missed even though the associated tasks have been identified as schedulable through appropriate schedulability analysis. The authors note that although it is argued that schedulability analysis simulates the worst-case scenario of task executions, this is not always the case because of the assumptions made by schedulability theory. The authors develop a methodology that helps identify performance scenarios that can lead to performance failures in a system. It combines the use of external aperiodic events (ones that are part of the interface of the software system under test, i.e., triggered by events from users, other software systems or sensors) and internally generated system events (events triggered by external events and hidden to the outside of the software system) with a Genetic Algorithm.

Zhang et al. [14] describe a procedure, similar to ours, for automating stress test case generation in multimedia systems. The authors consider a multimedia system consisting of a group of servers and clients connected through a network as a SUT. Stringent timing constraints as well as synchronization constraints are present during the transmission of information from servers to clients and vice versa. The authors identify test cases that can lead to the saturation of one kind of resource, namely CPU usage of a node in the distributed multimedia system. The authors first model the flow and concurrency control of multimedia systems using Petri-nets [18] coupled with temporal constraints. Allen's interval temporal logic [19] was used by the authors to model temporal relationships. For example, given two media objects, *VideoA* and *VideoB*, the representation: $\alpha VideoB = \beta VideoA + 4$ (where $\alpha VideoB$ and $\beta VideoA$ denote the begin time of *VideoB* and end time of *VideoA* respectively) is used to express the starting of *VideoB* four time units after the end of *VideoA*. In their model, Zhang and Cheung first identify a reachability graph of the

Petri net representing the control flow of multimedia systems. This graph is quite similar to a finite state machine where the states are referred to as markings and the transitions correspond to the transitions in the Petri-net. Each marking on the reachability graph is composed of a tuple representing all the places on the Petri-net along with the number of tokens held in each. It is important to note that only reachable markings (that is ones that can be reached by an execution of the Petri-net) are included in the reachability graph. From there, the authors identify test coverage of their graph as a set of sequences that cover all the reachable markings. These sequences, or paths in the reachability graph, are referred to as firing sequences. Firing sequences are composed of a transition and a firing time, represented as a variable. From there, each sequence is formulated into a linear programming problem and solved, outputting the actual firing times that maximize resource utilization.

The proposed technique can not be easily generalized to generate test cases for different stress testing strategies of a distributed system. Some of the limitations of their technique are:

- They assume constant resource utilization (called as *weight* by the authors) for each media object. While in most DRTS, the resource usage of each object (system component) varies with time.
- Only instant stress testing (happening in one time instant) is supported. But a system may only exhibit failures if stress test is prolonged for a period of time.
- The temporal relationships and control flow model of the system should be modeled using Petri-nets [18] and Allen's interval temporal logic [19]. Although these two notations have solid mathematical foundations, they are not widely used

by software developers. It would be much better if the required temporal relationships and control flow information could be extracted from the UML model of a system.

- The proposed technique can not be easily generalized to generate test cases for different stress testing strategies, i.e., testing networks vs. nodes, stress direction (i.e., towards a node vs. from a node). This will be discussed in detail in our system model and methodology sections.

Several techniques have been proposed to find concurrent faults, such as [20-23] which aim at finding data-race related faults. For example, Ben-Asher et al. [21] propose a set of heuristics to increase the probability of manifesting data-race related faults. The goal is to increase the chance of exercising data-races in the program under test and thus increase the chance of manifesting concurrency faults that are data-race related. The proposed technique first orders global shared variables according to the number of times they are accessed by different processes. Then data-race based heuristics are used to change the runtime interleaving of threads so that the probability of fault manifestation increases. One of the proposed heuristics in [21] is called *barrier scheduling,* in which barriers are installed before and after accessing a particular shared variable. A barrier causes the processes accessing the variable to wait just before accessing it. When a predefined number of processes are waiting, the heuristic then simultaneously resumes all the waiting processes to access the shared variable, for example using *notifyAll()* in Java.

The existing techniques to find concurrent faults do not distinguish between local or distributed concurrent processes. However since a set of concurrent processes can run on

distributed locations, the existing methods to find concurrency faults can also be potentially used in a distributed system, which is implicitly concurrent as well.

2.2 Problem Statement (Initial)

We first define the problem we tackle in a general way, without providing details on the modeling and formalisms which will be proposed later on in this thesis.

> Assuming that the UML design model of a DRTS is given, the problem is to find a systematic technique which automatically generates a set of test requirements to stress the network traffic of the system nodes and network links such that the probability of exhibiting network traffic-related faults increases. The UML design model of the SUT is assumed to include at least the system's sequence diagrams (annotated with start and end timing information of each message), class diagram(s) and a *system network interconnectivity package diagram* which will be introduced in Section 5.5 and shows the interconnectivity of the system's nodes and network links. There can be several concurrent processes running on each system node where processes communicate with other processes located on the other nodes.

The above problem statement will be revisited in Section 9.1, where it will be detailed and rephrased using the modeling and formalisms proposed in Chapter 5 to Chapter 7.

2.3 Terminology

Here we define the basic terminology used throughout this paper.

Performance Testing. Performance testing is defined as the testing activity which is conducted to evaluate the compliance of a system or component with specified

performance requirements. By thorough performance testing, it is expected that the risks of performance failures in systems are reduced. If performance is defined in terms of response time, software systems must produce results within acceptable time intervals. For example, most users of desktop systems will be annoyed with response times longer than a few seconds. In hard real-time systems, the deadlines to accept and respond to an input are measured in small time units such as milliseconds [24]. In all of these applications, the inability to meet response time requirements is no less a bug than incorrect outputs or a system crash.

Stress Testing. Stress testing is defined as the testing process by which a software system is put under heavy stress and demanding conditions in an attempt to increase the probability of exhibiting failures. A stress test pushes the SUT to its design limits and tries to cause failures under extreme *but valid conditions*. This kind of testing will reveal two kinds of faults: lack of fail-safe behavior and load-sensitive bugs. The stress test suites may increase the number of simultaneous actions and cause resources to be used in unexpected way. This may reveal faults on rare conditions, in exception handlers, and in restart/recovery features of a software system [24].

Distributed system: A collection of autonomous, geographically-dispersed computing nodes (hardware or software) connected by some communication medium: one or more *networks*.

Distributed node: A geographically-dispersed computing node, which is a part of a distributed system and is part of a *network*.

Network: A network is the communication backbone for a set of nodes in a distributed system. A network may be a subnet of another network or the supernet of several other networks.

2.4 UML Profile for Schedulability, Performance, and Time

The UML standard has been used in a large number of time-critical and resource-critical distributed systems [25-29]. Based on this experience, a consensus has emerged that, while a useful tool, UML is lacking some modeling notations in key areas that are of particular concern to distributed system designers and developers. In particular, it was noticed that the lack of a quantifiable notion of time and resources was an obstacle to its broader use in the distributed and embedded domain. To further standardize the use of UML in modeling complex distributed systems, the OMG (Object Management Group) adopted a new UML profile named "*UML Profile for Schedulability, Performance and Time*" (SPT) [12] (referred to as the UML-SPT).

The UML-SPT profile proposes a framework for modeling real-time systems using UML. The profile was finalized on Sept. 2003 and is becoming popular in the research community [30-34] and the industry [35]. The profile provides a uniform framework, based on the notion of quality of service (QoS), for attaching quantitative information to UML models. Specifically, QoS information models, either directly or indirectly, the physical properties of the hardware and software environments of the application represented by the model. This framework is referred to as the *General Resource Modeling* framework (GRM) by the UML-SPT profile. The structure of the GRM framework is shown in Figure 1 [12].

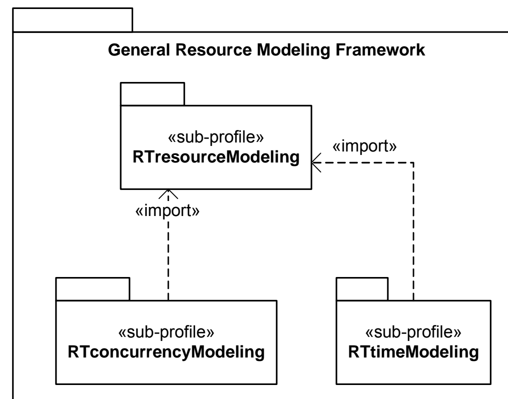


Figure 1-The structure of the GRM Framework of the UML-SPT profile [12].

According to the UML-SPT profile's specification [12], sub-profiles are defined as profile packages dedicated to specific aspects and modeling analysis techniques. As shown in Figure 1, the *RTtimeModeling* sub-profile imports the *RTresourceModeling* sub-profile since time can be considered as a resource in a system. The *RTtimeModeling* sub-profile provides means for representing time and time-related mechanisms that are appropriate for modeling real-time software systems. It is composed of several packages that introduce the following separate but related groups of concepts:

- Concepts for modeling time and time values, included in the *TimeModel* package.
- Concepts for modeling events in time and time-related stimuli, included in the *TimedEvents* package.
- Concepts for modeling timing mechanisms (clocks, timers), included in the *TimingMechanisms* package.
- Concepts for modeling timing services, such as those found in real-time operating systems, included in the *TimingServices* package.

As we will see in the faults taxonomy related to time constraints in distributed systems (Section 3.1.1), we will mostly use the concepts for modeling events in time and time-

related stimuli in the context of this work. Those concepts are included in the *TimedEvents* package of the *RTtimeModeling* sub-profile. The modeling of the *TimedEvents* package is shown in Section 4.1.3 of the UML-SPT profile [12].

As an example, part of the deployment architecture of a typical chemical reactor system is shown in Figure 2, where a sensor controller node (n_{sc}) is supposed to get the sensor data from sensors n_{s1} and n_{s2}, and then send the data to be updated in the control server (n_{cs}).

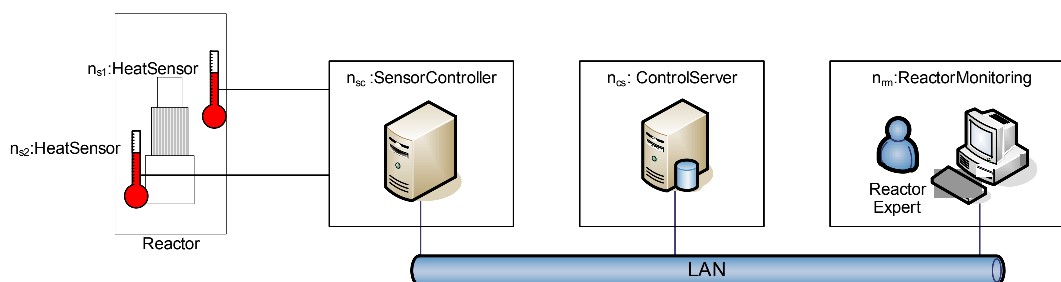


Figure 2-Part of the deployment architecture of a chemical reactor system.

The sequence diagram (SD) in Figure 3 shows the realization of the data update process. The SD is using time modeling constructs in the *TimeModel* package of the UML-SPT profile and the UML 2.0 [10] notations. The timing information of messages has been modeled using the UML-SPT. For example, *«RTstimulius»* denotes that the first message is a RT stimulus with an arrival pattern specified by the tagged-value *«RTArrivalPattern»*.

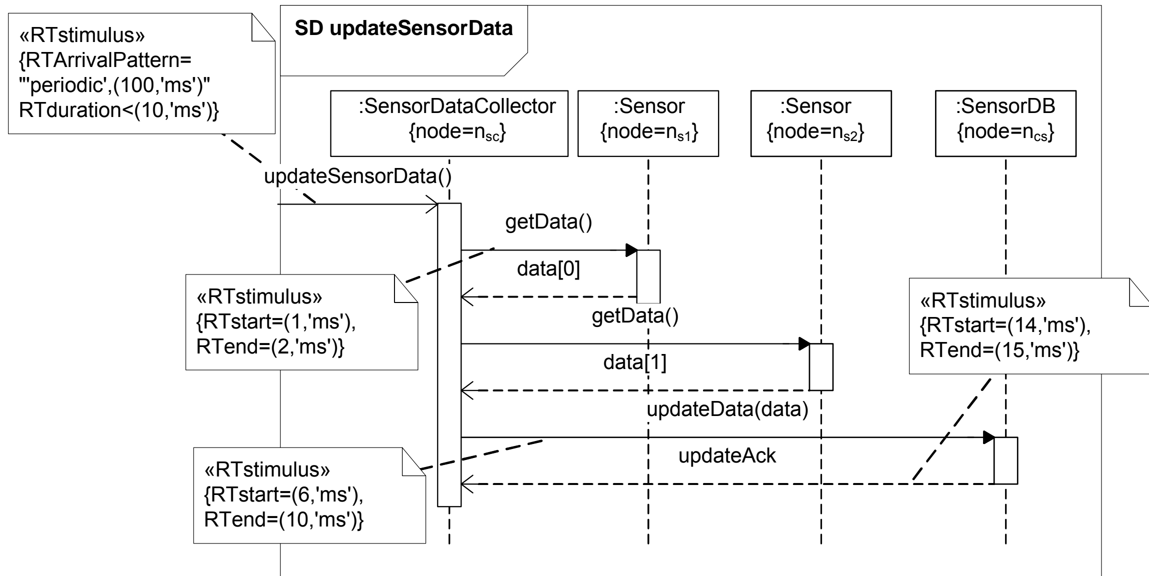


Figure 3-Example of time modeling using UML-SPT profile.

The system is obviously a safety-critical one, where an inadequate response time of the system might have life-threatening consequences. In other words, the temperature of the system should be measured and checked according to the timing notations in Figure 3 and prompt corrective actions should be carried out if the temperature is higher than a pre-specified threshold. Note that the decision of time units (e.g., *ms* in Figure 3) modeled in SDs are based on the precisions of the estimates.

2.5 An Overview on UML 2.0 Sequence Diagrams

The UML 2.0 [10] syntax of SDs is used in this work. Comparing to UML 1.x [36, 37], UML 2.0 have proposed a set of new features to SDs. Some of the new features are illustrated with an example in Figure 5. The SD metamodel showing the class diagram of SD features is shown in Figure 4.

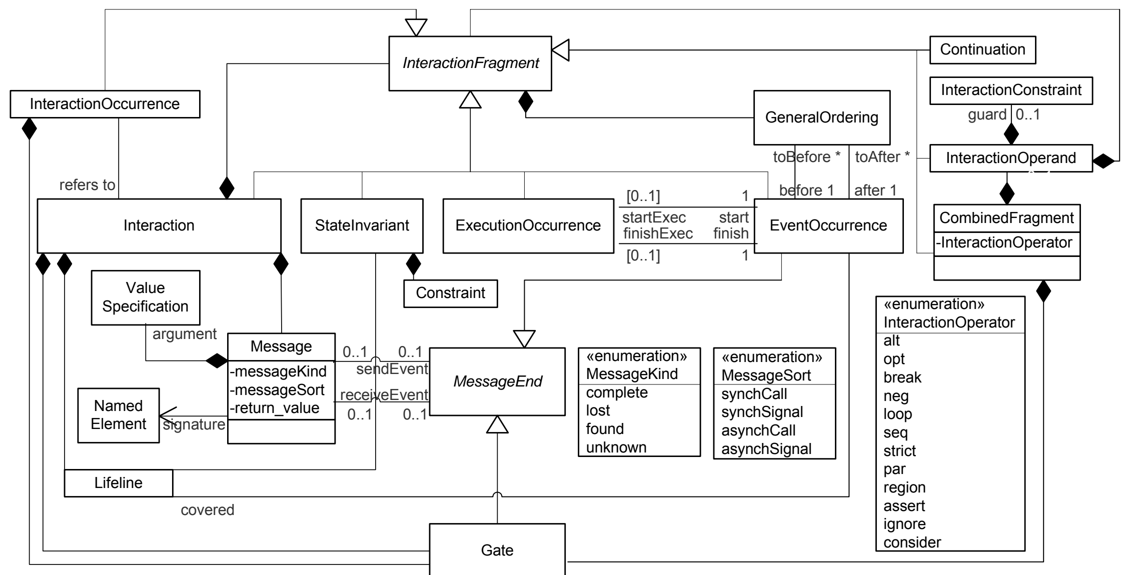


Figure 4-UML 2.0 Sequence Diagram Metamodel.

A message in a SD is the basic form of communication in interactions. Communication can raise a signal, invoke an operation, and create or destroy an object instance. UML 2.0 no longer draws a distinction between message and stimulus as UML 1.x did. In the new version, a message can be one of the following two types:

- *Operation call*: which expresses the invocation of an operation on the receiving object. An operation call must match the signatures of an operation on the target object (receiver of the message).
- *Signal*: which represents a message object sent out by one object and handled by the other object that is equipped to respond to it.

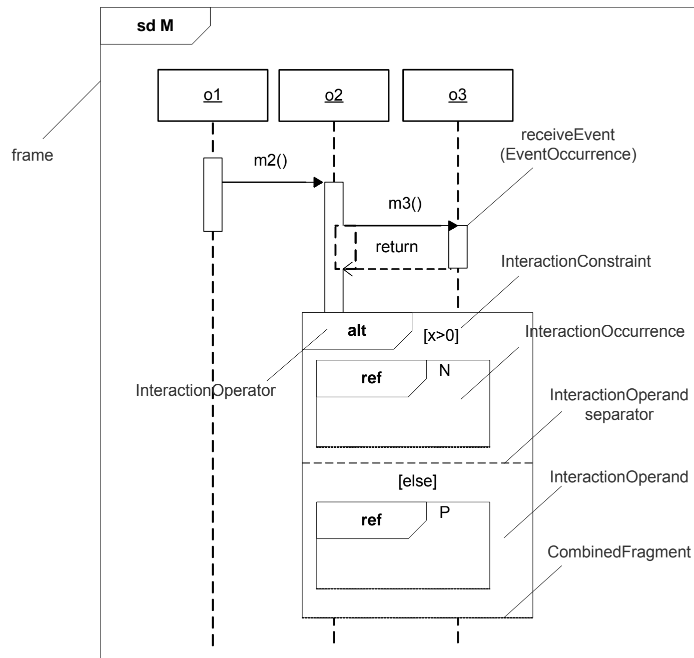


Figure 5-An example illustrating the new features of the UML 2.0 SDs.

UML also provides four varieties (or *sorts* as UML 2.0 calls them) for a message. Message sorts identify the sort of communication reflected by a message. The sorts of messages supported are defined in an enumeration called *MessageSort* (in Figure 328 of [10]) as:

- *SynchCall*: synchronous call
- *AsynchCall*: asynchronous call
- *SynchSignal*: synchronous signal
- *AsynchSignal*: asynchronous signal

Chapter 3

AN EXTENDED FAULT TAXONOMY FOR DISTRIBUTED REAL-TIME SYSTEMS

Before devising any type of testing technique, one needs to clearly specify the target set of fault types. Since the scope of our testing technique is distributed real-time systems, we present in this chapter a fault taxonomy for such systems, which is an extension to the fault taxonomy presented in [38].

Section 3.1 presents our fault classification for DRTS. Section 3.2 discusses the chain of distribution faults, i.e., how a specific fault may recursively lead to other faults (possibly of other types than the initial fault). Finally, the classes of faults considered in this work, a sub-set of what is described in Section 3.1, are described in Section 3.3. This will clarify the objectives of the stress testing methodology we present in Chapter 9 and Chapter 10.

3.1 An Extended Fault Classification for Distributed Real-Time Systems

To operate successfully, most large distributed systems depend on software, hardware, and human operators and maintainers to function correctly. Failure of any one of these elements can disrupt or bring down an entire system.

According to the terminology used in system dependability, a system may fail either because it does not comply with the specification, or because the specification did not adequately describe its function [38]. Three fundamental categories of threats exist in the dependability theory presented in [38]: *failures, errors,* and *faults*. A system failure occurs when the delivered service deviates from fulfilling the system function. An error is the part of the system state that may cause a subsequent failure. A fault is the adjudged or hypothesized cause of an error. A fault is active when it produces an error; otherwise it is dormant. Failures, errors, and faults are closely related. The chained causality relationship between these threats is shown by Avizienis et al. [38], as depicted in Figure 6.

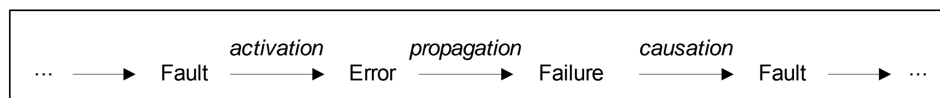


Figure 6-The fundamental chain of dependability threats.

The arrows in this chain express a causality relationship between faults, errors and failures. From the users viewpoint, a malfunction in a system is observed via a failure, which itself has been caused by an error and that by a fault. Therefore in terms of system granularity, failures are in a higher level than errors and those are in a higher level than faults. For example in a typical web-based email system such as Yahoo, which most

probably uses parallel/distributed web servers to serve huge number of clients at the same time, a typical failure from a user standpoint might be: "*Yahoo! mail doesn't let me log in*". This failure might be due to an error such as: "*the user database can not be reached*" in the system, which in turn, might be caused by a distributed fault like: "*congestion in a database server's request queue has resulted in an unavailability of the server*".

Adapting the concept of dependability to our context, i.e., distributed systems, it makes sense to account for specific faults which occur specially in distributed systems and thus extend the taxonomy introduced in [38]. Our proposed additions are presented in Figure 7, where the faults classes on the gray background are our proposed additions, while those on a white background were discussed by Avizienis et al. in [38].

We have added five top-most categories: distribution, time criticality, concurrency, resource-usage orientation, and location of creation or occurrence. By this extension, we believe that the fault taxonomy can be used in the context of DRTSs.

The technique proposed in this thesis aims to target transient faults (in term of persistency) since stressing conditions are by nature transient. Thus, we first revisit the persistency of faults in Section 3.1.1. In terms of the other fault categories described in [38] (domain, intent, boundary, cause and phase), the objective of our technique is not specific to a specific category (e.g., internal or external for the faults boundary). Thus, we do not discuss those fault categories in detail here.

Sections 3.1.2 to 3.1.6 discuss our five extended categories: distribution, time criticality, concurrency, resource-usage orientation, and location of creation or occurrence.

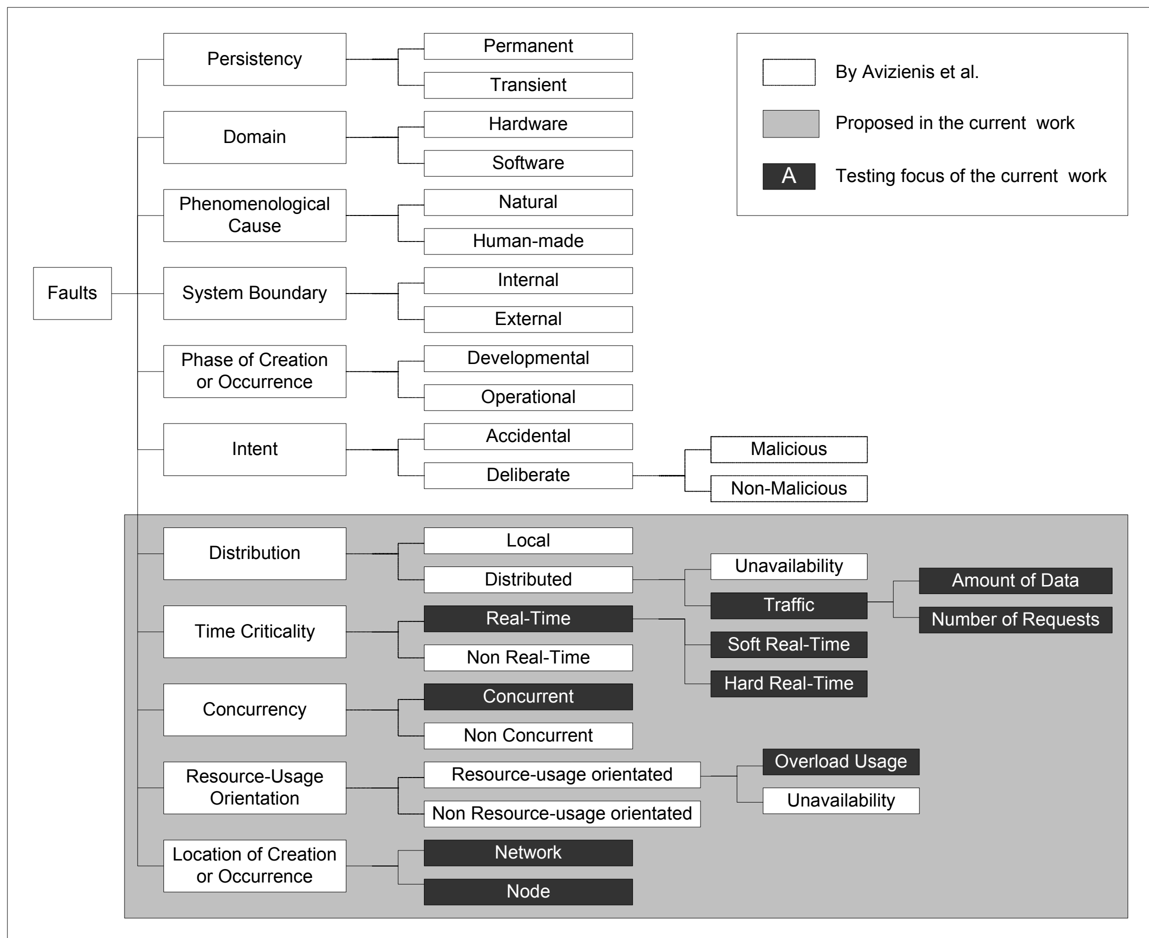


Figure 7-Tree of Generalized Fault Classes for Distributed Systems.

3.1.1 Persistency of Faults

Some studies have suggested that since software is not a physical entity and hence not subject to transient physical phenomena (as opposed to hardware), software faults are permanent in nature [39]. Some other studies classify software faults as either permanent or transient [38]. Permanent faults are essentially design faults which can be identified quite easily and can be removed during the testing and debugging phase (or early deployment phase) of the software life cycle. Transient faults, on the other hand, belong to the class of temporary internal faults and are intermittent. They are essentially faults

whose conditions of activation occur rarely or are not easily reproducible. Hence these faults result in transient failures, i.e., failures which may not recur if the software is restarted or is run in normal load conditions. Some typical situations in which transient faults might surface are high usage loads, improper or insufficient exception handling and interdependent timing of various events. It is for this reason that transient faults are difficult to identify through regular testing techniques. Hence a mature piece of software in the operational phase, released after its development and testing stage, is more likely to experience failures caused by transient faults than due to permanent faults.

Some studies on failure data have reported that a large proportion of software failures are transient in nature [40, 41], caused by phenomena such as overloads or timing and exception errors [42, 43]. For example, a study of failure data from a fault tolerant system, called Tandem, indicated that 70% of the failures were transient failures, caused by faults like race conditions and timing problems [44].

Altogether, depending on the system under study, we might be able to list some of the situations in which transient faults might happen:

- Overloads
- Race conditions on shared resources
- Interdependent timing of various events
- Improper or insufficient error handling
- Memory leaks

3.1.2 Distribution

Since nodes are geographically distributed in a distributed system, there should be a communication medium connecting them. We can thus classify faults based on their distribution aspect. We identify faults pertaining to communication among nodes under the class of *distributed* faults. The contrary category is called *local* faults.

An important point to mention here is that since both the SUT and the test system run in the application layer of the OSI (Open Systems Interconnection)'s 7-layer network architecture [45], we only consider faults which are of relevance to the application layer and not the lower OSI layers, such as bit transmissions errors which are handled and corrected by the Error Correcting Codes (ECC) in the data link layer. In the context of testing distributed systems, we identify two types of faults with a distributed nature:

- Distributed unavailability faults
- Distributed traffic faults

We discuss each of the above fault categories in the following sections.

3.1.2.1 Distributed Unavailability Faults

Distributed unavailability faults relate to the *availability* (readiness for correct service) and *reliability* (continuity of correct service) attributes of a system. The specification of most distributed systems usually dictates that the system's network links and nodes should be highly available and reliable. For example, in a safety-critical system like a distributed air traffic control, the flight and runway information should be updated frequently in the system's central database. Failing to do so, which might be caused for

example by a network unavailability fault between a radar and the controller, might result in disastrous consequences.

A distributed unavailability fault happens when a system component is no longer available and can not provide services to other components in the system. This equally applies to networks and nodes. For example, a distributed message from a source node may not reach the destination node because one of the network links in the path from the source to the destination node is exhibiting a distributed unavailability fault. Since there are essentially three parties (network, the source and the destination nodes) in every communication, therefore in our definition, this fault might happen in either a network or in a node, which can be described using the "Location of Creation or Occurrence" fault class, as shown in Figure 7. This justifies why Unavailability, an orthogonal concept to "Location of creation or Occurrence", is sub-type of type Distributed.

Network links between any two distributed nodes might become unavailable at any time during the system activity. As we will assume in the system model in Chapter 5, any arbitrary network link in the network path between any two nodes in the system might be unavailable while the other links are functioning well. Therefore, all different types and combinations (e.g., the sender node, any of the network links, or the receiver node of a message) of unavailability faults have to be accounted for if we want to test all possibilities of unavailability in a system. The reason why we would like to distinguish the unavailability fault in terms of its location of creation or occurrence is that the system's overall behavior might be different when a network link, the source or the destination nodes exhibit unavailability faults. A schematic notation of possible

distributed unavailability faults in a simple distributed message scenario is shown in Figure 8.

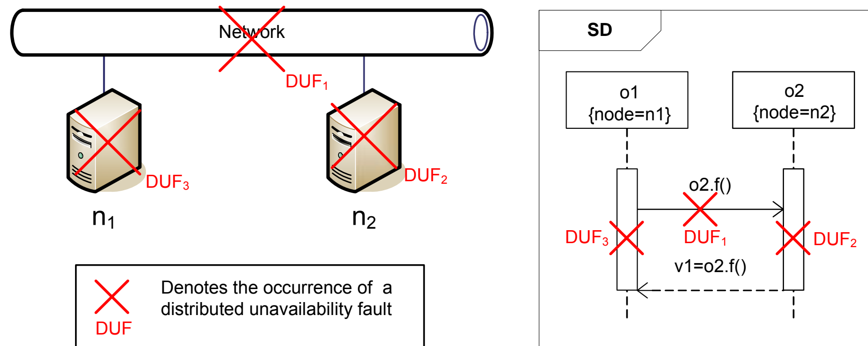


Figure 8-Occurrences of Distributed Unavailability Faults (DUF).

In the simple distributed message scenario of Figure 8, object o_1 on node n_1 invokes a remote procedure call *f()* from object o_2 on node n_2 and subsequently receives the return value. A distributed unavailability fault (DUF) might happen anywhere in this scenario. We have identified three of all possible DUFs as shown with DUF_i's in Figure 8. Suppose DUF_1 happens on the network connecting two nodes and just after the message $o_2.f()$ is sent from o_1 to o_2. DUF_2 occurs in n_2 (e.g. node n_2 crashes) after message $o_2.f()$ has arrived in o_2 and while o_2 (node n_2) is busy processing function *f()*. DUF_3 is a DUF which takes place in n_1 before receiving the reply message ($v_1=o_2.f()$). The time and location where a DUF happens might cause different failures and subsequent faults in a system. Therefore to achieve full coverage in terms of DUFs, all different times and locations of DUFs have to be tested in a system.

Distributed unavailability faults might happen due to a variety of reasons, such as: physical damage to a network cable, dead node, dead router/switch/hub in the network path, and network or application software malfunction.

3.1.2.2 Distributed Traffic Faults

A distributed traffic fault occurs when at least one of the system components does not function correctly under heavy network traffic, but remains available. Distributed traffic faults can be due, for instance, to network congestions, buffer overflows, or processing delay in software modules. This, again, also equally applies to networks and nodes. A detailed discussion of the root causes for distributed traffic faults is outside the scope of this document since such discussions will be mostly related to computer and communication networks literature. However, among the main causes, we consider two cases: large amounts of data transmitted by networks, and high number of requests handled by nodes.

There have been many studies in the area of network traffic and researchers have used many analytical models. One of the most common models to represent traffic in networks is to use queuing theory [46]. Those group of works build mathematical models (such as Markov chains) to analyze a network's behavior under stressed conditions. Most of such techniques work in the lower layers of the OSI architecture and do not analyze stressed conditions from the application layer (software) point of view. Thus, a detailed discussion of those group of works is outside the scope of this document as they are mostly in the field of computer and communication networks. The interested reader is referred to the extensive literature in these areas.

As an example of a scenario when a distributed traffic fault might happen, consider the network schematic shown in Figure 9. Let us suppose the nodes in $Network_A$ (n_1, n_2, n_3) send messages to nodes in $Network_B$ (n_4, n_5, n_6) simultaneously, where each message contains a large amount of data. All of these messages have to go through $Network_{AB}$

which connects $Network_A$ and $Network_B$. If the total size of the simultaneous data sent over $Network_{AB}$ is larger than its capacity, there will probably be a delay or other network faults that can be referred to "distributed traffic faults". This fault may cause an error and subsequently a failure in the system, which in turn might lead to other classes of faults according to the fundamental chain of dependability threats shown in Figure 6. Distributed database and multimedia servers are examples of systems where large amounts of data are usually exchanged between nodes and distributed traffic faults might occur.

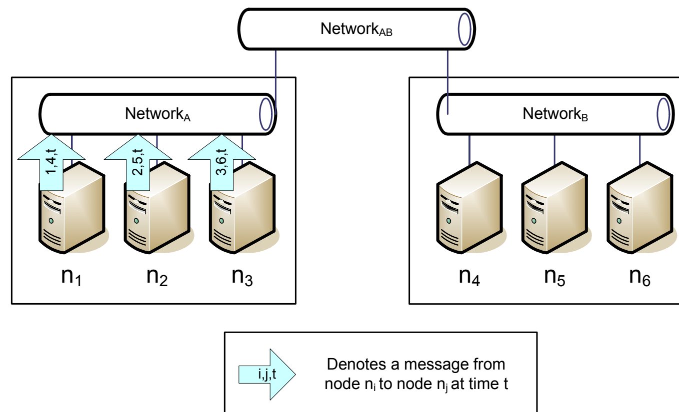


Figure 9-An example scenario showing how a distributed traffic fault might happen.

As discussed previously, in addition to the amount of data transmitted over a network or from/to a node, we further acknowledge that high number of simultaneous messages might also be a potential cause of traffic faults. Considering the example scenario in Figure 9, assume each of the concurrent processes on the nodes n_1, n_2, and n_3 (inside $Network_A$) send messages to processes on nodes n_4, n_5, and n_6 (inside $Network_B$) all at the same time. Since there can be large number of concurrent processes on each node, there might be scenarios where high number of distributed messages go over the network

$Network_{AB}$. This, subsequently, might cause a distributed traffic fault in the network and/or any of the nodes.

3.1.3 Time Criticality

In the context of DRTSs, faults can also be categorized based on their real-time (time criticality) aspect. A real-time fault occurs when a real-time deadline is missed. As discussed earlier, safety-critical systems often have time constraints which they should react on time. As usually categorized in the literature, real-time deadlines (constraints) are of two types: *hard* and *soft* deadlines. Hard deadlines are constraints that absolutely must be met [47]. A missed hard deadline results in a system failure. A system with hard deadlines is called a hard real-time system. On the other hand, *soft* real-time systems are characterized by time constraints (soft deadlines) which can (a) be missed occasionally, (b) be missed by small time derivations, or (c) occasionally skipped altogether. Usually, these permissible variations are stochastically characterized. Another common definition for soft real-time systems is that they are constrained only by average time constraints. Examples include on-line databases and flight reservation systems. Therefore, in soft real-time systems, *late* data may still be *good* data, depending on some measure of the severity of the lateness.

3.1.4 Concurrency

A concurrency fault occurs if the root cause of a system failure is due to the concurrency among processes. There might be, for example, a shared resource that is accessed by several processes in a system. The synchronization scheme and order in which a shared

resource is accessed might lead to a concurrency fault. Some types of concurrency faults are: deadlock, livelock, starvation and data-races.

A deadlock is a situation where two or more processes cannot proceed because they are all waiting for the other to release some shared resource. Livelock happens when processes are blocked with reasons other than waiting for a shared resource, for example a busy waiting on a condition that can never become true [48]. Resource starvation is a more subtle form of a deadlock state. A process may have large resource requirements and may be overlooked repeatedly because it is easier for the resource management system to schedule other processes with smaller resource requirements [48]. Data-race is an anomaly of concurrent accesses by two or more threads to a shared variable when at least one is writing. Programs which contain data-races usually demonstrate unexpected and even non-deterministic behavior. The outcome might depend on specific execution order (a.k.a. threads' interleaving). Rerunning the program may not always produce the same results. Thus, programs with data-races are hard to test and debug.

3.1.5 Resource-Usage Orientation

Resource-usage orientated (resource, in short) faults relate to usage of resources (e.g. network bandwidth, CPU, or memory) in a system. We identify two types of resource faults : (1) overload usage of a resource, and (2) unavailability of a resource.

An overload resource fault might happen if the amount of usage from a resource exceeds its limits (capacity). For example, several processes may try to allocate more memory space than it is available in a system. Resource unavailability faults relate to the *availability* (readiness for correct service) and *reliability* (continuity of correct service)

attributes of resources. Note that there is an overlap in the concept between resource unavailability and distributed unavailability faults. The focus of the former is on resource aspect of faults, while the later deals with distribution aspect of faults in a DRTS. Distributed unavailability faults can be considered as a special type of resource unavailability faults if networks and nodes (distributed components of a DRTS) are considered as resources.

3.1.6 Location of Creation or Occurrence

We propose this new classification for faults in DRTS to specify location of creation or occurrence. We consider two possibilities for the location of a fault: network or node. Considering a distributed system to be a set of networks and nodes, a fault might occur in any of the nodes or networks.

3.2 Chain of Distribution Faults

As shown in the fundamental chain of dependability threats in Figure 6, a fault with a specific type may recursively lead to other faults with different types. For example, a distributed fault such as data traffic fault might lead to a real-time fault, where a process might miss its assigned deadline to perform a particular task. This chained causality can be rephrased as: when a process does not receive the data it was waiting for, on time (by a specific deadline), it is not able to perform its action on time. Therefore, when studying the root cause of faults in a system, it is important to order the faults according to the order they occur and cause the next one in the faults chain: the data traffic fault is the first fault in the chain and the real-time one is the second in the above example.

3.3 Class of Faults Considered in this Work

The dark boxes in Figure 7 depict the classes of faults targeted by the stress testing methodology described in this document:

In other words, our methodology targets traffic-related faults of distributed nature, which can occur on either nodes or networks in a DRTS. Such a traffic-related fault is supposed to be related with network bandwidth as resource type. The following can be an example of such a fault: *congestion in a database server's request queue.*

To investigate if our test methodology can be generalized to target other types of faults, we will need to discuss its details first through the next several chapters. We will discuss in Chapter 13 how our methodology can be either tailored to target other types of resources (e.g. CPU or memory) instead of network bandwidth, or generalized to target other types of faults.

Chapter 4

OVERVIEW OF THE STRESS TEST METHODOLOGY

In this chapter, we present in one view the different pieces of information and steps/activities of our entire approach, and at the same time indicating in which coming chapter more details are provided. We also describe succinctly the purpose of each part of the input model and the purpose of each intermediate representation. Note the although some input (and intermediate) models (in our approach) are not standard UML diagrams, we propose them to stay in the UML world, thus facilitating tool support.

Section 4.1 presents the overview of our model-based stress test methodology. An overview of the input (and intermediate) models used in our stress test methodology is discussed in Section 4.2.

4.1 Stress Test Methodology

The overview of our model-based stress test methodology is shown using an Activity Diagram (AD) in Figure 10. Note that the steps after test requirements generation, i.e., deriving test cases and executing them, are not addressed by this thesis. However, we discuss those steps for the specific case of our case study. A UML model of a SUT, following specific but realistic requirements, is used in input. A test model (TM) is then

built to facilitate subsequent automation steps. The TM and a set of stress test parameters (objectives) set by the user are then used by an optimization algorithm to derive stress test requirements. Test requirements can finally be used to specify test cases to stress test a SUT. The TM consists of three sub-models: a control flow analysis model (Chapter 6), inter-SD constraints (Chapter 7) and network traffic usage pattern (Chapter 8).

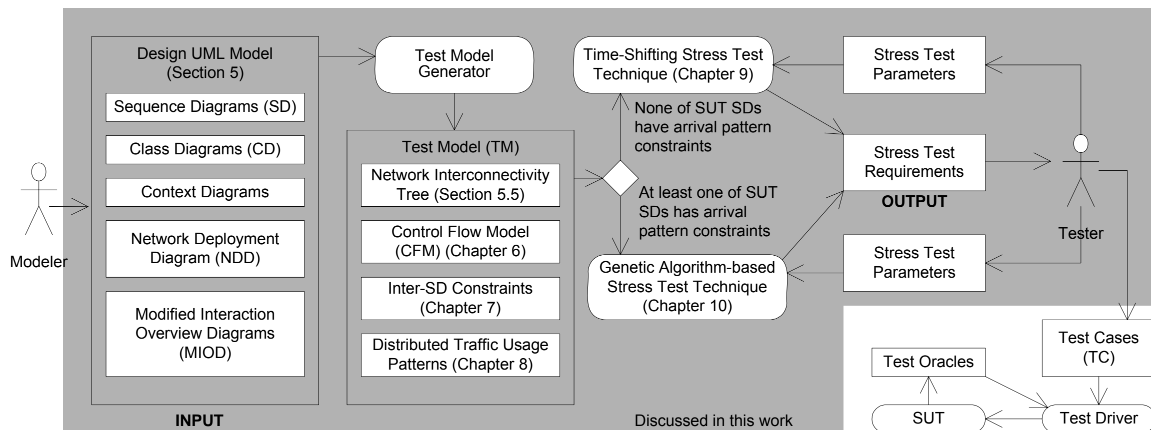


Figure 10- Overview of our model-based stress test methodology (a UML activity diagram).

As we will discuss in Section 5.3, triggering SDs may not be allowed in any time instant. These types of constraints are called *arrival-patterns*. If none of a SUT's SDs has arrival-pattern constraints, we use a simple optimization algorithm (Chapter 9) to derive stress test requirements from a TM. Otherwise, if at least one of SDs has arrival pattern constraints, we show in Chapter 10 that a more sophisticated optimization algorithm is needed and present one based on Genetic Algorithms.

Test requirements are the outputs of our technique, which can be used by a tester to derive test cases. A test driver can be utilized to feed the derived test cases to the SUT, monitor its behavior, check the output with test oracles and generate test verdicts.

4.2 Input and Intermediate Models in our Stress Test Methodology

An overview of the input and intermediate test models in our stress test methodology is shown by their respective high level metamodels in Figure 11. The metamodels are grouped into two packages: input system design models, and test models, which are described next.

4.2.1 Input System Design Models

The design UML model of a SUT should consist of the following diagrams which are described extensively in Chapter 5.

- Sequence diagrams (SD): SDs model the behavior of a SUT. To enable our stress test methodology to derive stress test requirements (as schedules to trigger SDs), we also require the messages in SDs to be annotated with timing information (e.g., Figure 3).
- Class diagrams (CD): CD of a SUT will be used to estimate the data size of messages in SDs. This will enable our stress test methodology to derive stress test requirements with maximum possible network traffic.
- Network Deployment Diagram (NDD): A NDD is an extended diagram to UML 2.0 deployment diagrams will models the network topology of a SUT, and will let

our stress test methodology to derive *localized* stress test requirements (i.e., on a specific node or network).

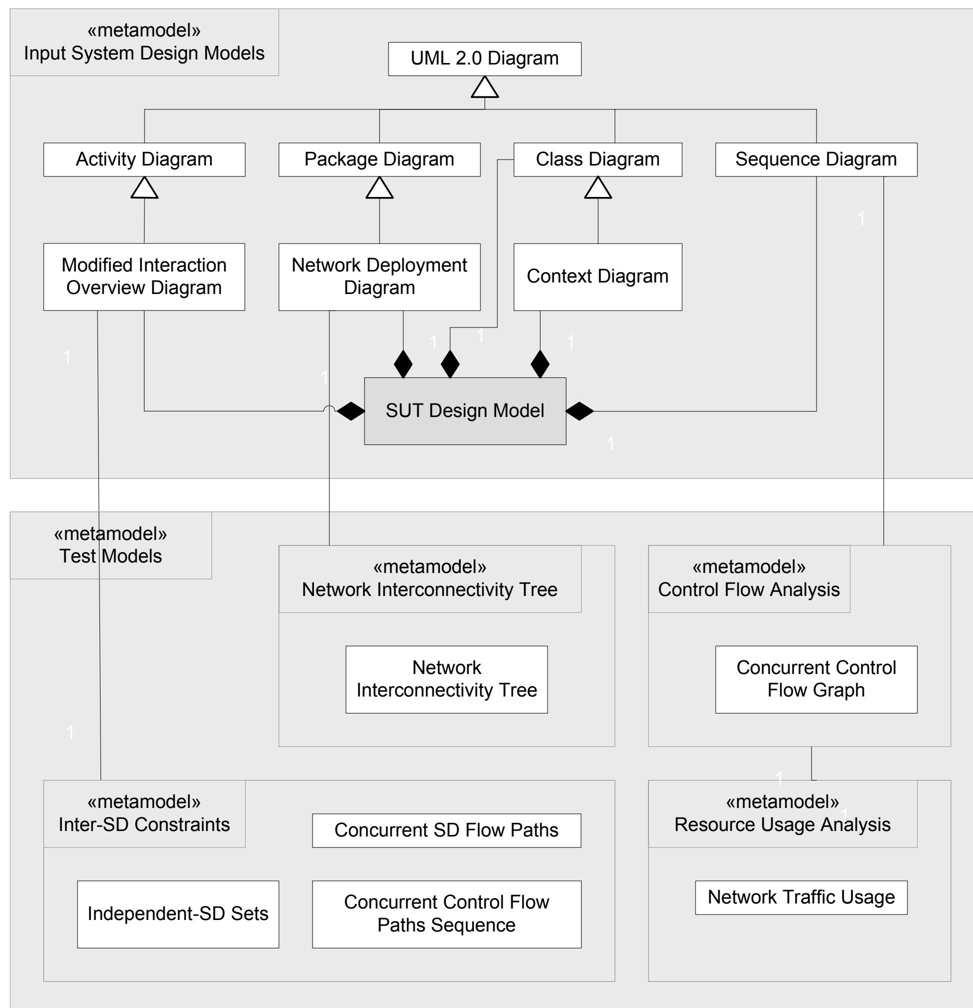


Figure 11- Metamodel of input and intermediate test models in our stress test methodology.

- Context Diagram [49]: Context diagrams are an extension to class diagrams and will provide the number of multiple invocations of a SD, relevant in the context of DRTSs.

- Modified Interaction Overview Diagram (MIOD): A MIOD models the sequential and conditional constraints among SDs. This will enable modelers to specify the valid sequences of SDs in a SUT, and our stress test methodology to derive stress test requirements complying with such constraints.

SDs and CDs are standard UML diagrams, while NDDs, Context diagrams and MIODs are not standard diagrams, but they extend standard UML diagrams as explained in detail in Chapter 5.

4.2.2 Test Models

The Test Model (TM) package consists of four sub-packages: (1) control flow analysis model, (2) network traffic usage model, (3) network interconnectivity tree and (4) inter-SD constraints, which are described in the next subsections.

4.2.2.1 Control Flow Analysis

In UML 2.0 [10], SDs may have various program-like constructs such as conditions (using *alt* combined fragment operator), loops (using *loop* operator), and procedure calls (using interaction occurrence construct). As a result, a SD is composed of Control Flow Paths (CFP), defined as a sequence of messages in a SD. Furthermore, as we discussed in [50], asynchronous messages and parallel combined fragments entail concurrency inside SDs.

In a SD of a DS, some messages are *local* (sent from an object to another on the same node), while others are *distributed* (sent from an object on one node to an object on another node). Furthermore, different CFPs can have different sequences of messages and each message can have different signatures and a different set of parameters. Therefore,

the network traffic usage pattern of each CFP can be different from other CFPs. Thus, comprehensive model-based stress testing should take into account the different CFPs of a SD.

As we will discuss in Chapter 6, synchronous and asynchronous messages should be handled differently in the control flow analysis of a SD. We will propose a CFM (Control Flow Model) for SDs, referred to as CCFG (Concurrent Control Flow Graph). OCL consistency-rules will be used to define the mapping between a SD and its equivalent CCFG (Concurrent Control Flow Graph). CCFGs will support asynchronous messages and concurrency in SD. Similar to the concept of Control Flow Paths (CFP), we will propose Concurrent Control Flow Paths (CCFP), which can be derived from a CCFG. To consider distributed messages, between two objects on two different nodes, in a SD, Distributed Concurrent Control Flow Paths (DCCFP) will be defined. The process to build a CFM will be discussed in Chapter 6.

4.2.2.2 Resource Usage Analysis

We define the resource usage analysis metamodel to enable resource usage analysis of messages in SDs. We only consider network traffic resource usage in this work. Quantifying network traffic usage is done by measuring the amount of traffic entailed by a message and assigning the value to the flow node (in CCFP) corresponding to a message. Therefore, the resource usage analysis is done at the message-level in this work.

We consider four abstract classes for network traffic usage: *type, duration, direction,* and *location*. These classes will be discussed in further detail in Chapter 8. A technique to formally analyze network traffic usage of a system based on a given UML model will be

proposed in Chapter 8. The resource model will be formalized in a way to facilitate the stress testing of network traffic in a SUT.

4.2.2.3 Network Interconnectivity Graph

A Network Interconnectivity Graph (NIG) is an internal data structure derived from a NDD using the technique presented in Section 5.5.2. As we discuss in Chapter 9, it facilitates tree-based manipulations on a SUT's topology, such as extracting the network path between two nodes.

4.2.2.4 Inter-SD Constraints

These constraints are derived from a Modified Interaction Overview Diagram (MIOD), which models constraints among SDs of a SUT. We will propose in Chapter 7 four concepts to analyze such constraints in our methodology.

- Independent-SD Sets (ISDS): An ISDS is a set of SDs that can be executed concurrently, i.e. there are no sequential constraints between any two of the SDs in the set to prevent it.
- Concurrent SD Flow Paths (CSDFP): A CSDFP is a sequence of SDs from a start to an end node of a MIOD. In other words, a CSDFP is a sequence of SDs that are allowed to be executed in a system (according to the constraints modeled in a MIOD).
- Concurrent Control Flow Paths Sequence (CCFPS): A CCFPS is derived from a CSDFP by substituting each SD by one of its CFPs.

Chapter 5

INPUT SYSTEM MODEL

In this work, stress test input data is assumed to be the UML 2.0 [10] design model of a SUT. As discussed in Chapter 1, UML has become the de-facto standard for modeling object-oriented software for nearly 70 percent of IT industry since 1997 [11]. The new version, UML 2.0 [10], proposed by the OMG in August 2003, offers an improved modeling language. As we expect UML to be increasingly used for DRTS, it is therefore important to develop automatable UML model-driven, stress test techniques.

We describe in this chapter the modeling information required. The rationale for using the following five modeling diagrams by the methodology are described next:

- Two standard UML 2.0 diagrams: sequence diagrams (Section 5.1), and class diagrams (Section 5.1.1.3)
- A modified UML 2.0 diagram: modified interaction overview diagram (Section 5.3) - our specific diagram used in our test methodology
- A context diagram [49] (Section 5.4)

- A specialized UML 2.0 package structure, referred to as Network Deployment Diagram (NDD) (Section 5.5)

Furthermore, two tagged-values (specialized from the UML-SPT tagged-values) for modeling hard and soft Real-Time constraints in UML behavior diagrams are described in Section 5.6.

5.1 Sequence Diagram

The goal in this work is to systematically stress test a SUT and we need to find some particular test requirements, based on the behavior of the SUT, to feed into the SUT. Therefore the dynamic behavior of the SUT should be analyzed to derive such test requirements. According to the UML 2.0 specification [10], seven UML diagrams can be used to specify the behavior of a system. As shown in Appendix A of [10], they are Activity, Sequence, Collaboration (or called Communication in Section 14 of [10]), Interaction Overview, Timing, Use case and State machine diagrams. Among all those diagrams, only sequence and communication diagrams provide message-level details of a program, which are needed for the Control Flow Analysis (CFA) needed for stress testing. Furthermore, among the last two, SDs have been more popular than communication diagrams in modeling dynamic behavior of systems, as they provide a richer set of behavior modeling constructs (e.g. loops and conditions).

SDs have been accepted as essential UML artifacts for modeling the behavioral aspects of systems [51]. They show interacting objects as well as the control flow of those interactions (e.g., if-then conditions, repetitions) [52]. These diagrams are particularly well-suited for object-oriented software as it is now well recognized that the complexity

of object-oriented software lies in objects interactions much more than in individual operations. Moreover, SDs have been the basis for several approaches for testing object-oriented software [24, 51, 53, 54], although not in a distributed or real-time context. These are the reasons why we choose SDs as the source of information for dynamic behavior of a SUT.

Because SDs can specify varying flows of control (e.g., thanks to conditional constructs), network-traffic stress conditions may happen in only subsets of those SD control flows. Thus, we need to analyze control flow in SDs to derive network-aware stress test requirements. We have presented a control flow analysis technique based on SDs in [50], which we will use in this work. An overview of this technique will be given in Chapter 6.

Since each of the participating objects of a SD may be deployed on a different node, we need to model this information in SDs. We use a node tagged value to specify this information, as illustrated in Figure 12: objects o_1, o_2, and o_3 are deployed on three different nodes in the network, n_1, n_2, and n_3, respectively..

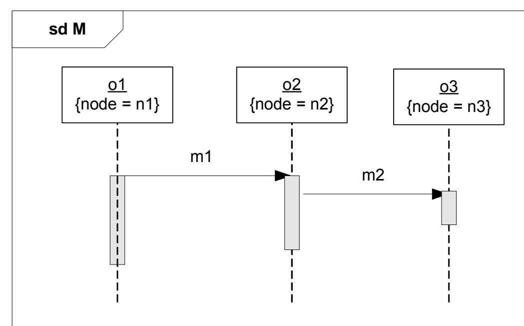


Figure 12-Modeling the deployment node of an object using *node* tagged value.

5.1.1 Timing Information of Messages in SDs

As mentioned in Chapter 1, real-time systems often have real-time constraints that have to be met at runtime or real-time faults will occur. In Chapter 3, we also mentioned that that a fault can trigger other subsequent faults as well. For instance, a network traffic fault might trigger a real-time fault. Therefore, our overall heuristic in this work is to schedule the SD's of a SUT such that all possible distributed messages with maximum data sizes on a particular network link or a node happen at the same time. As we will see in the next sections, this will maximize the chance of exhibiting network traffic faults and consequently any other faults dependent on them.

In order to devise precise test requirements (from time point of view) that yield such a stress test scenario of network traffic in a SUT, our stress test methodology requires that the timing information of messages in SDs is available and as precise as possible. The more precise the timing information of messages, the more precise (and thus potentially more stressing) the test requirements will be to stress test a SUT. By timing information of a message, we basically mean the start and end times of a message. As discussed in Section 3.1.3, messages in a typical DRTS might have hard or soft real-time constraints. There might be also messages that do not possess any real-time constraints. In terms of timing information, there can be two types of messages which are discussed next: (1) messages with predictable timing (to a degree of uncertainty), and (2) messages with unpredictable timing information.

5.1.1.1 Estimating Execution Times

Three of the approaches used in the literature to estimate timing information of messages in DRTSs are:

- Static analysis and manual estimations (such as [55])
- Runtime monitoring (such as [56])
- Benchmarks (such as [49])

With manual estimations, different schedulability analysis techniques are used and the program code is analyzed by hand. The possible execution paths which lead to extreme execution times are then derived. Static analysis methods are limited when the test object contains program control structures such as loops. In this case, loop bounds must be specified manually or can only be estimated. This process is resource-consuming, error-prone, and is not scalable. Furthermore, the BCET (Best-Case Execution Time) can be too optimistic and the WCET (Worst-Case Execution Time) too pessimistic due to manual estimations.

Another common approach for estimating timing information is runtime monitoring techniques (such as [56]), which can be utilized to get a statistical overview of the time length of such messages at runtime prior to the testing phase. Statistical distributions of start and end times of such messages can be derived by running the system before testing and their expected values can then be used by the stress test technique in this paper. However, due to the statistical (and hence indeterministic) nature of the timing values, such timing information might not lead to precise stress scenarios. We assume that a time

measurement technique has been used for the messages in the SUT and that such information is already available.

Estimating timing information may also be possible in late design stages by using, for example, heuristics similar to the ones used in the COMET (Concurrent Object Modeling and Architectural Design with UML) [49] object-oriented life cycle, where Gomaa proposed a heuristic to estimate time durations of messages based on benchmarks defined on previously-developed similar messages. Such an approach can be adapted to the estimation of number and types of local variables of a method by comparing the functionality/role of a method at hand with benchmarks of previously-developed methods' local variables (in the same target programming language). It should be acknowledged that this is in general a complex task which would require extensive experience and skills. Such information should then be provided by modelers in an appropriate way, for example by using specific tagged-values.

5.1.1.2 Uncertainty of Timing Information

Regardless of the type of technique used by engineers to estimate timing information, uncertainty (impreciseness) in timing information is inevitable. Different models have been used in the literature to model time uncertainty (e.g., [57, 58]). The core part of such models which incorporates uncertainty of time are similar. The work in [58] formalizes such a concept as follows.

Uncertainty in execution time of a task t is taken care of by letting its execution time ET be a random variable characterized by a probability mass function (PMF) PET. PET is defined over K+1 points, denoted by et_k, $k \in 0 \ldots K$. By definition, a) the values et_k are

assumed to be in the ascending order with increasing k, but with the restriction that et_k,≤WCET of task t, b) $et_0 = 0$ and $PET\ (ET = et_0) = 0$, and c) $PET(ET = et_k) \neq 0$ for $k \in 1 \ldots K$. Assumptions (b) and (c) are intended to produce a minimal set of mass values in the initial specification of uncertainties, while excluding zero as a possible execution time. For example, the probabilistic representation of a task execution requirement is depicted in Figure 13. The sequence of execution times and their probabilities are derived by applying schedulability analysis techniques, and are referred to as *Timed Sequence of Probabilities (TSP)* [58]. As it is shown, the task's execution time is assumed to be 2, 3 and 6 units of time with the probabilities of 0.3, 0.6, and 0.1, respectively.

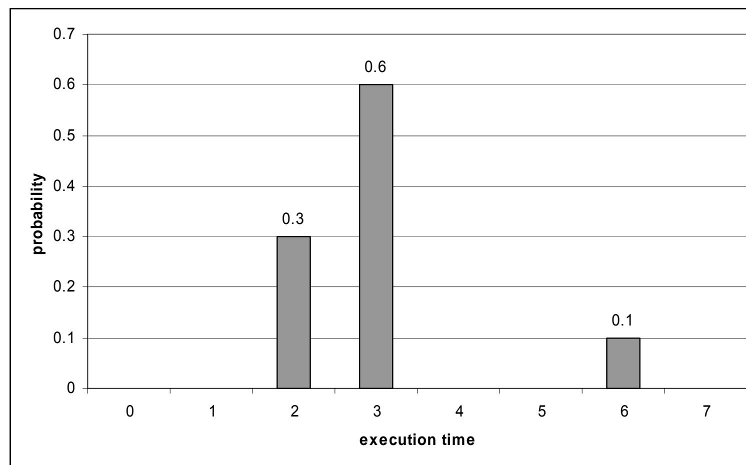


Figure 13- Probabilistic representation of uncertainty in a task's execution times.

To further investigate the impact of uncertainty in timing information on our quantitative stress test methodology (Chapter 9), we need to present the methodology itself first. Thus, such this topic will be revisited in Section 10.9.

5.1.1.3 Messages with Unpredictable Execution Times

After using different schedulability analysis techniques to estimate execution times of messages in a SUT, there might still be messages whose execution times are unpredictable: no WCET/BCET can be found for them. Examples of such messages are those with data-intensive parameters whose data sizes can not be estimated early enough.

Estimating the execution times of such messages will lead to great amounts of uncertainty in such time values, which might lead to great deals of indeterminism in our stress test methodology and the output test requirements it will generate. Thus, we will present in Section 10.10 a different version of our stress test methodology, referred to as *Wait-Notify Stress Test Technique (WNSTT)*, which will in specific target systems with at least one message with unpredictable execution time.

5.2 Class Diagram

The class diagram is at the heart of the object modeling process. The class diagram models the resources used to build and operate the system. It models each resource in terms of its structure, relationships and behavior [11].

The stress testing technique in this document will use the class diagram(s) of a system for the following two purposes:

- To achieve full coverage criteria for polymorphism in control flow analysis of SDs, as explained in Section 5 of [50, 59], and
- To estimate the data size of a distributed message in a SD (either a call or a reply message), as explained in Section 8.1.1.

No specific constraints is imposed on the use of the UML 2.0 class diagram notation.

5.3 Modified Interaction Overview Diagrams

Executing any arbitrary sequence of use cases (UCs) (i.e., their corresponding SDs) in a SUT might not be always valid or possible. Business logic of a SUT might enforce a set of constraints on the sequence (order) of SDs, requiring that certain conditions be satisfied before a particular SD can execute.

Different types of SD constraints can be considered:

- *Sequential* constraints [60] define a set of valid SD sequences: e.g., the *Login* SD of an ATM system should be executed before the *Withdrawal* SD.
- *Conditional* constraints are related to sequential constraints and indicate the condition(s) that have to be satisfied before a sequence of SDs can be executed. For example, the *Login* SD should be executed "successfully" before the *Withdrawal, Transfer* and *Deposit* SDs, or the *RenewLoan* SD of a library system can be invoked up to "two times" for an instance of a loan.
- *Arrival-pattern* constraints relate to timing of SDs. The time instant when a SD can start running might be constrained in a system. Each SD might be allowed to execute only in some particular time instants. For example in a replicated distributed database server system, where the data on the main server is mirrored (copied) to the replicated servers, the policy may be to run the *Mirror* SD every hour and not on every transaction (maybe since the SD deals with enormous amounts of data). Another scenario in which a SD can have an arrival-pattern constraint is when the SD is triggered by an event and the event is periodic.

Accounting for those constraints when deriving stress test requirements is paramount if one wants to obtain valid stressing executions. If no constraint or only sequential and conditional constraints apply to SDs, we have developed a simple heuristic approach for stress test requirement generation (Chapter 9). If arrival-pattern constraints exist, a more complicated approach, based on a Genetic Algorithm is proposed in Chapter 10. This is illustrated in Figure 14.

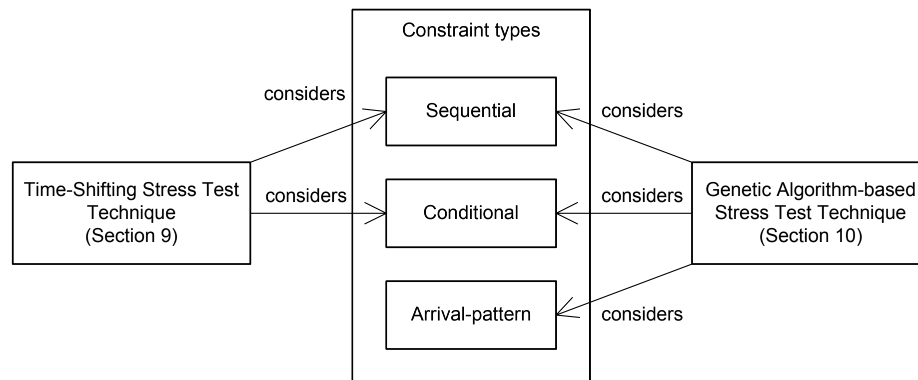


Figure 14-The approach in which the different SD constraint types are considered by the two optimization algorithms in this work.

In order to analyze and take the above three types of constraints into account when conducting any type of testing on a SUT, the constraints should be modeled. Arrival patterns apply to each SD, and hence are not inter-SD constraints. Arrival patterns can be modeled using the *RTArrivalPattern* tagged-value of the UML-SPT profile, as explained in Section 2.4. Refer to the SD in Figure 3 for an example.

Sequential and conditional constraints are between SDs. Therefore, we refer to them as *inter-SD* constraints. In the following, we first discuss existing techniques and representations to model and formalize inter-SD constraints (Section 5.3.1) and we then choose the one which suits best our context (Section 5.3.2).

5.3.1 Existing Representations to Model Inter-SD Constraints

In this section, we present a brief review of the existing representations to model SD constraints, in both UML and non-UML contexts. We briefly discuss some of them and focus on the ones in the context of UML.

Before UML became a standard, an OO-development method called Fusion [61] proposed the notion of *life-cycle model* which bears some similarity in concepts to what we call SD sequential constraints. Such constraints are the direct result of the logic of the business process the system purports to support. Such processes could be specified by life-cycle models.

Use-Case Maps (UCMs) [62] are also one of the notations which bears some similarities to UML activity diagrams. The UCM notation aims to link behavior and structure in an explicit and visual way. *UCM paths* are architectural entities that describe *causal* relationships between *responsibilities* which are bound to underlying *organizational structure* of *components*. UCM paths represent scenarios that intend to bridge the gap between requirements (use cases) and detailed design [62].

Allen's interval temporal logic [19] is also one of the models proposed for modeling temporal constraints among a group of objects. This temporal logic was used by Zhang and Cheung in [14] to model the temporal constraints among objects in multimedia presentations. Having modeled these temporal constraints, the authors presented a technique to stress test the CPU load of a multimedia system using linear programming optimization technique.

Petri-nets [18] can also be used to model sequential constraints among SDs. For example, Zhang and Cheung [14] model the flow and concurrency control of multimedia objects using Petri-nets. The advantage of Petri-nets is that it a well-founded formal notation that has been widely used for the modeling of dynamic behavior.

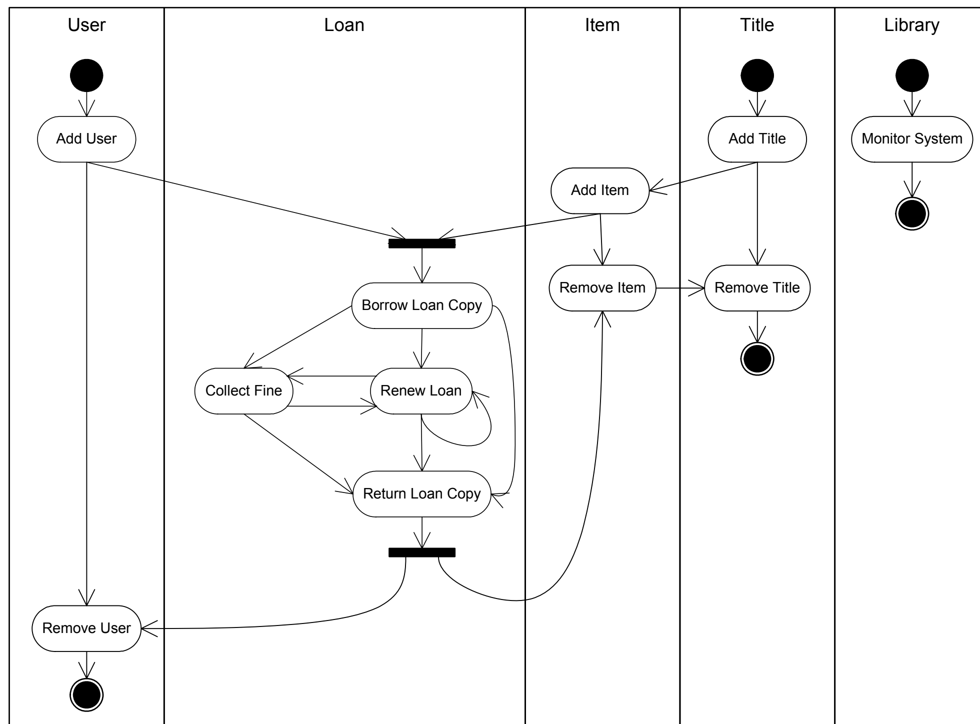


Figure 15- Use Case Sequential Constraints for the Librarian actor (adopted from [60]).

In the context of UML and SDs, there have also been techniques and representations to model and formalize constraints among SD, [60] and [63] for instance. When modeling the behavior of a system, a SD is usually modeled to realize a particular UML UC. Briand and Labiche [60] report that when planning test cases for UCs, all possible execution *sequences* for UCs have to be identified. In order to do that, they represent the sequential dependencies of UCs as an activity diagram in which vertices are UCs and

edges are sequential dependencies between UCs. An edge between two UCs (from a tail UC to a head UC) specifies that the tail UC must be executed in order for the head UC to be executed, but the tail UC may be executed without any execution of the head UC. In addition, specific situations require that several UCs be executed independently (without any sequential dependencies between them) for another UC to be executed, or after the execution of this other UC. This is modeled by *join* and *fork* synchronization bars in the activity diagram, respectively. The authors illustrated the technique on a library system: Figure 15 shows the UC sequential constraints for the Librarian actor (parameters of the UCs such as a *title_id* for *AddTitle* are not shown for clarity).

Nebut et al. [63] propose a contract language for functional requirements expressed as parameterized use cases. They also provide a method, a formal model and a prototype tool to automatically derive both functional and robustness test cases from the parameterized use cases enhanced with contracts. In this technique, pre- and post-conditions (i.e., logical expressions) are attached as UML notes to each use case in the use case diagram. The sequential constraints among SDs can then be deduced from the set of contracts: e.g., if the postcondition of use case A implies the precondition of use case B then sequence A-B is legal.

The OMG introduced a new UML diagram in UML 2.0, namely *Interaction Overview Diagram* (IOD) (Section 14.4 of [10]). IODs "define interactions through a variant of activity diagrams in a way that promotes overview of the control flow" [10]. IODs are specializations of Activity Diagrams (AD) where object nodes are either *Interactions* or *InteractionOccurrences*. Recall that following the UML 2.0 terminology, a sequence diagram is an Interaction and referring to an existing sequence diagram is an

InteractionOccurrence. In other words, an IOD is used to show how flows of executions of sequence diagrams. IODs are similar to the activity diagram notation suggested in [60] for use case sequential constraints, except that in an IOD, the designer can use powerful Activity Diagram notations such as conditions and loops.

As an example, the IOD of an ATM system is depicted in Figure 16. The IOD is composed of interaction occurrences which refer to specific SDs (*Insert Card*, *Login*, *Display Menu*, etc.). The flow of control between interaction occurrences is modeled using AD flows. The control can take on different paths as modeled by decision nodes. For example, if only login is successful, the interaction occurrence *Display Menu* is invoked.

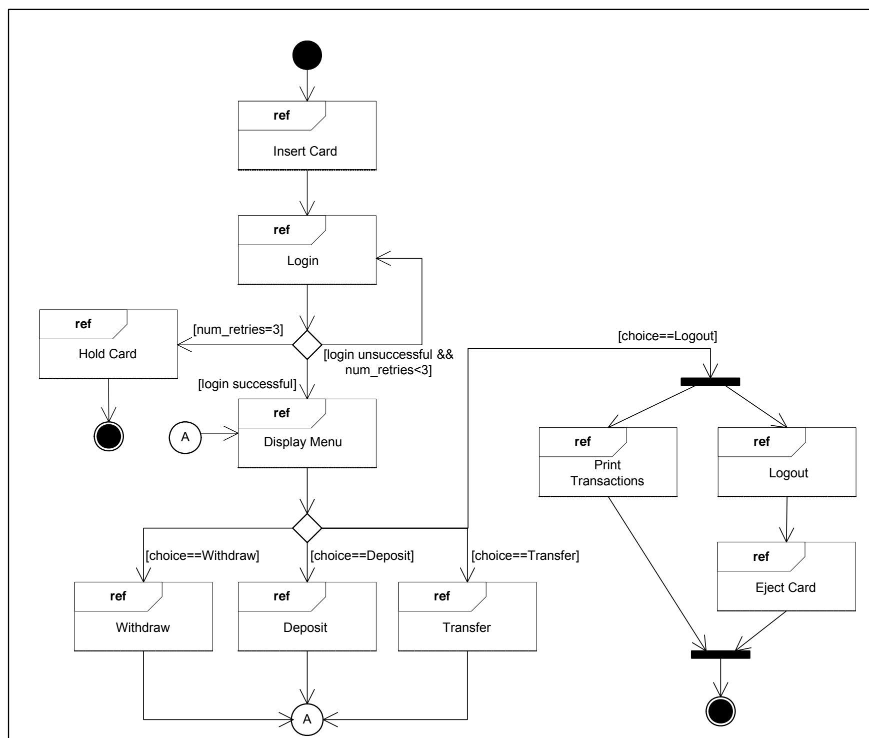


Figure 16-Interaction Overview Diagram (IOD) of a simplified ATM system.

5.3.2 Modified Interaction Overview Diagrams

One fundamental constraint for the choice of a representation for sequence diagram constraints is that the entire system modeling should be performed using UML. The IOD notation is therefore a suitable representation for our needs.

UML 2.0 specification does not explicitly discuss swimlanes for IODs. However, as an IOD is basically a specific AD, it can therefore have swimlanes. Thus, to model which actor or sub-system invokes a particular SD, we explicitly include the concept of activity partitions using the AD notation for swimlanes in IODs, and refer to such IODs as *Modified Interaction Overview Diagrams (MIOD)*. Thus, MIODs are our specific diagrams used in our test methodology. In a MIOD, SDs are grouped into swimlanes according to the actors triggering them. For example, the MIOD of the IOD in Figure 16 is shown in Figure 17.

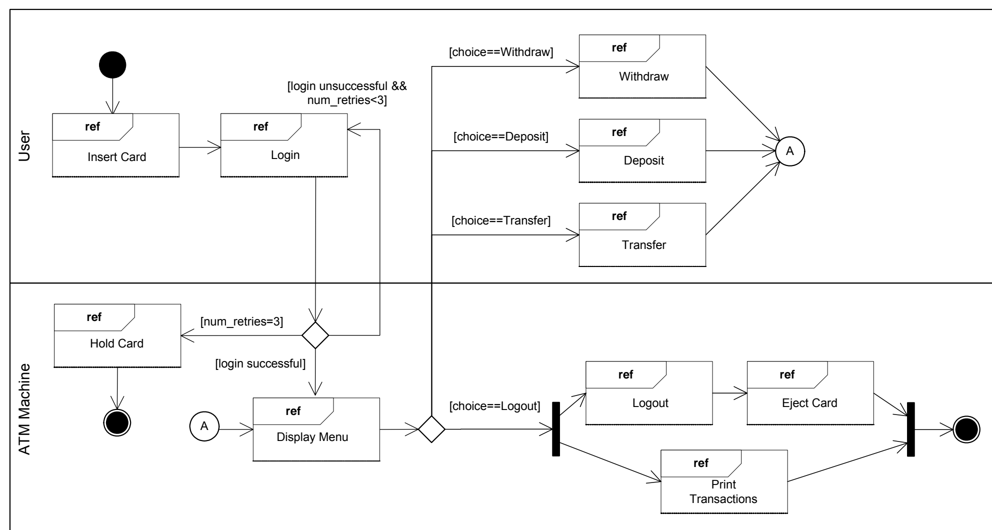


Figure 17- Modified Interaction Overview Diagram (MIOD) of a simplified ATM system.

The differences of our MIOD modeling notation with the use-case sequential-constraints modeling done in [60] are: (1) the MIOD is a notation for system-wide sequential constraint modeling for SDs, while the notation in [60] was per actor. (2) the MIOD takes into account the conditional constraints (defined in Section 5.3) among SDs, while the work in [60] did not explicitly support such constraints.

In Chapter 7, we will discuss the SD constraints in more detail and we will see why modeling those constraints is needed and in the current work for the purpose of stress testing.

5.4 Context Diagram

In a DRTS, there are often cases that lead to multiple concurrent invocations of a SD. For example, there might be several sensors which, as actors, trigger a particular SD at the same time in a controller system. Having multiple concurrent invocations of a SD rather than once can potentially have a different effect on the amount of network traffic in the system. Such a case should be modeled and be provided to our stress test technique.

To model concurrent invocations of SDs, we use the information provided in a Context Diagram [49], a concept originally proposed in the COMET (Concurrent Object Modeling and Architectural Design Method) framework [49]. A Context Diagram is a particular class diagram where the system being developed is modeled as class and is associated with the actors it interacts with. For example, a context diagram is shown in Figure 18-(a), where a controller system is made of three sensors. On the other hand, a sensor is the actor which can trigger the SD *UpdateData*, Figure 18-(b). Therefore, at one time instant, up to three concurrent instances of the SD can be executed.

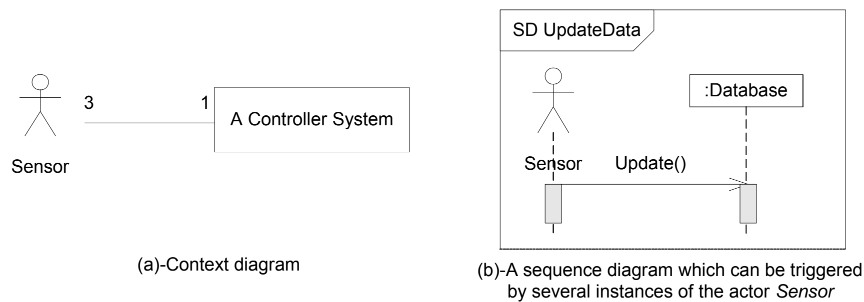


Figure 18-A controller system made of several sensors.

Alternatively, the number of concurrent instances of a SD may be modeled inside a MIOD. We propose a modeling notation, referred to as *multi-SD,* similar to the concept of multi-objects in UML. The multi-SD construct is used in MIODs to model multiple instances of a SD. Furthermore, a tagged-value titled *instances* is used to model the number of concurrent instances. An example is shown in Figure 19-(a). SD *UpdateData* is a multi-SD, where three instances of which can be executed concurrently. *SD1* and *SD2* are arbitrary SDs which are modeled before and after SD *UpdateData* according to business logic of the system. Note that the MIOD in Figure 19-(a) is equivalent to Figure 19-(b): a multi-SD can be replaced by a fork/join construct and multiple instances of the multi-SD in-between. The number of the SDs between fork and join are equal to the number modeled by the tagged-value *instances*.

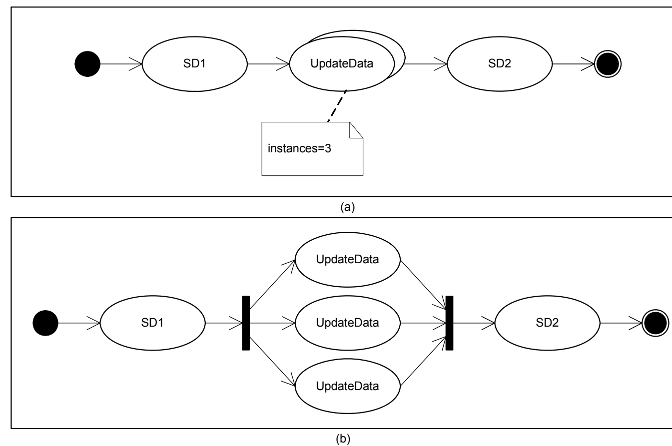


Figure 19-(a): Modeling concurrent instances of SDs inside MIOD. (b): Equivalent in meaning to (a).

Our test technique accepts both of the above two modeling approaches to model multiple instances of SDs. Number of concurrent invocations of a SD can be easily extracted if the multi-SD construct of MIODs is used. On the other hand, if a CD is used to model such information, our technique needs to look and match the SDs actors with the actors in the CD of a system to extract the information.

5.5 Network Deployment Diagram

Since we are dealing with nodes and networks which can be connected in any arbitrary fashion to each other in a SUT, and we further intend to use UML 2.0 models as the source for testing, we should find a proper notation in UML 2.0 to model networks/nodes interconnectivity, i.e., the system topology.

In UML 2.0 [10], there has been a significant change in support for modeling application architecture, nodes and communication paths, compared to UML 1.x [11]. Modelers can model complicated deployment scenarios such as nested and generalized nodes. Network topology modeling has also enhanced. *CommunicationPath*, Section 10.3.2 of [10],

generalized from standard UML's "Association", is a new concept for modeling the communication path between distributed nodes of a system. As defined in [10]: "A communication path is an association between two nodes, through which nodes are able to exchange signals and messages." For example, Figure 20 represents a simple network deployment of an online shopping system where client workstations, servers and printers are collaborating.

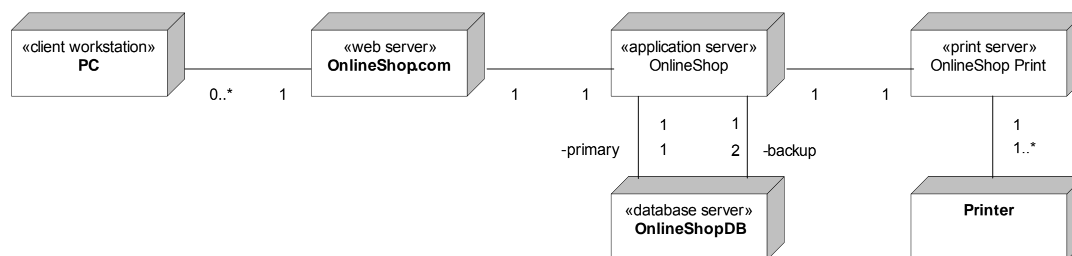


Figure 20-A simple network deployment for an online shopping service.

However, to the knowledge of the authors; modeling a hierarchical set of networks, network paths and their inter-connectivity, such as the one shown in Figure 22, is not directly stated in the UML 2.0 specification [10].

The structure of the distributed architecture of a SUT as we need it to be described is shown in Figure 21 as a metamodel. A distributed SUT consists of two or more distributed nodes and one or more networks. As described in terminology (Section 2.2), a node is a geographically-dispersed computing node, which is a part of a system. A node is part of a network in a system. A *network* is the communication backbone for a set of nodes in a system. A network may be subnet of another network, and at the same time it can be the supernet of several other networks. For example, a typical network topology is shown in Figure 22.

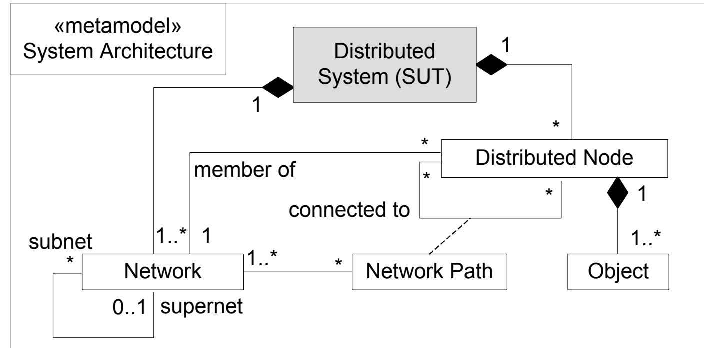


Figure 21-A metamodel for network topologies.

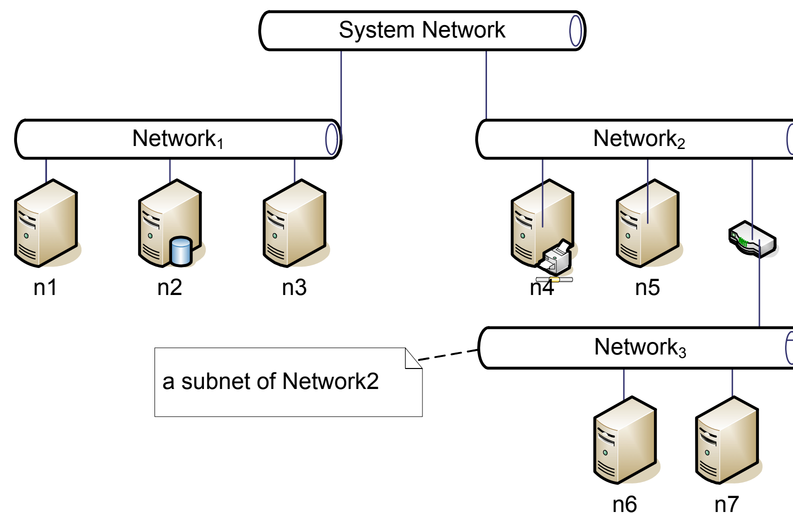


Figure 22-A network topology.

In the example of Figure 22, there are four networks in the system: *System Network*, *Network*$_1$, *Network*$_2$ and *Network*$_3$. Each network has several nodes (n_i) or networks as shown. For example, *Network*$_2$ has two nodes (n_4 and n_5) and one network *Network*$_3$, which itself has is the owning network of two other nodes (n_6 and n_7). It is assumed that there is at least one network in every distributed system and that is named as *System Network* which connects the highest level networks and nodes to each other.

In order to traverse from a node to another in the system, there can be (in general) several *network paths* between each two nodes. A network path between two nodes is an ordered

set denoting the unique path of networks between the sender and receiver nodes of a message extracted from the network topology. For example, the network path from n_1 and n_6 in Figure 22 is <*Network*$_1$, *SystemNetwork*, *Network*$_2$, *Network*$_3$>. A function to derive the network paths between two nodes will be described in Section 8.2. An extension to the UML 2.0 deployment diagram, referred to as *Network Deployment Diagram (NDD)*, described next, will be used to model a network topology.

5.5.1 Extending the Notation of UML 2.0 Deployment Diagrams

We want to describe a distributed architecture using UML 2.0 so as to be able to use it as an input for our dependency analysis in the context of UML-based development. Modeling a hierarchical set of networks and their inter-connectivity is not directly addressed in the UML 2.0 specification [10]. We therefore extend UML 2.0 deployment diagrams by adding two stereotypes to the node notation: *«network»* and *«node»*. We thus identify the type of an entity as a network or a node. Furthermore, association roles *supernet* and *subnet* are used to model the containment relationships between super and sub-networks. As an example, the architecture in Figure 22-(a) is modeled by the NDD in Figure 23.

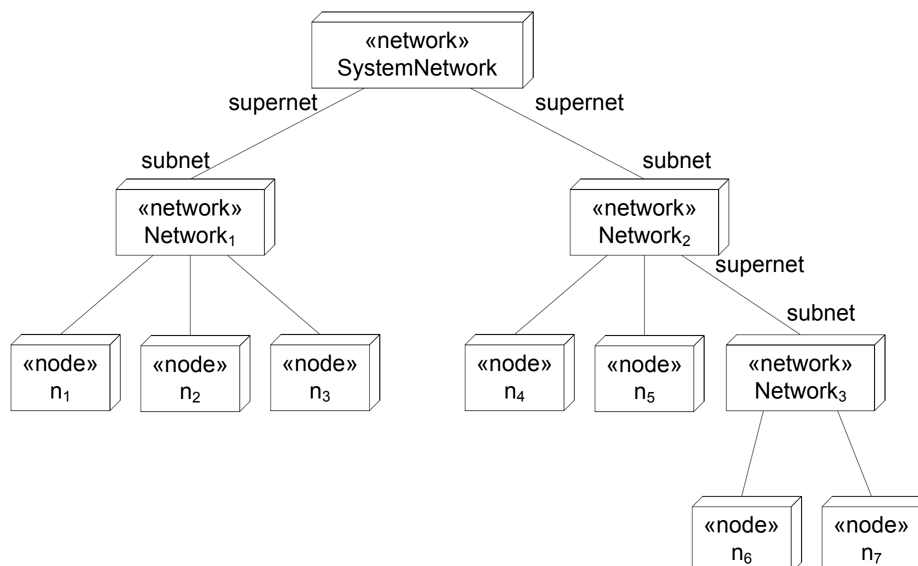


Figure 23-Using a Network Deployment Diagram (NDD) to model the network topology of Figure 22.

In order to model and quantify bandwidth (capacity) values of each network, we define a *bandwidth* tagged value for the «network» stereotype in the above notation. The format of the *bandwidth* tagged value is *{bandwidth=(bw, u)}* where *bw* is the bandwidth value in unit *u,* e.g. *{bandwidth=(100,kbps)}, kbps:* kilo bits per second. Furthermore, since the bandwidth of the network interface of a node connected to a network, as well as that of a switch/router/gateway connecting two different networks might be different than the two connected networks, we can also optionally model the bandwidth values of those model elements using *bandwidth* tagged value. We can assume that if the bandwidth value of a node's network interface (or a network) is not specified, its value is defined to be the value of the network the node is a member of (or the supernet of the network). The way to model *bandwidth* tagged values is shown by an example in Figure 24, which depicts the network interconnectivity of nodes and networks in a typical university setting. The system is deployed in three buildings ($Building_i$), each of which has its own subnet per

floor. Each node (workstation) is represented as *w(building_number). (floor_number). (node_number)*, such as *w3.1.2*. The bandwidth of the network interface of node *w1.1.1.* has been specified to be 100 kbps.

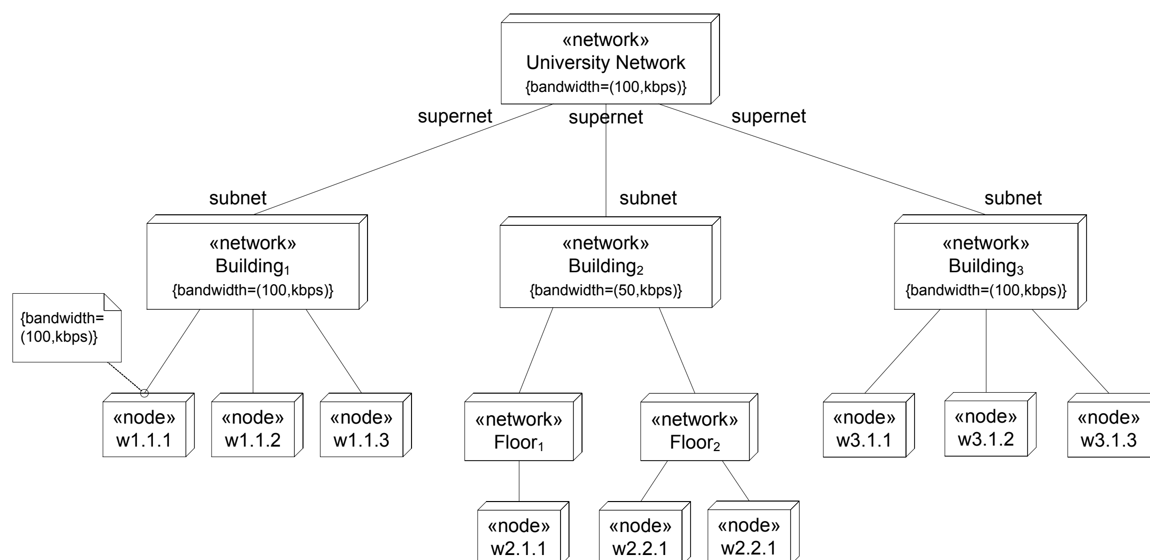


Figure 24-Modeling network interconnectivity of a university network.

Therefore, we assume that the network interconnectivity model of the SUT is modeled using the above notation. As a more efficient, tool-specific, representation which will be used by our testing technique, we propose a tree data structure for representing such interconnectivity. We refer to the new notation as *Network Interconnectivity Graph (NIG)*, which is described next.

5.5.2 Network Interconnectivity Graph

A Network Interconnectivity Graph (NIG) is an equivalent data structure to the NDD notation, which is used internally in our methodology. In a NIG, networks and nodes are shown as rectangles and circles, respectively. For example, the NIG of the network interconnectivity model of the Figure 22 (or equivalently Figure 23) is shown in Figure

25. The rationale of having NIG is to enable the test technique to easily find the subset of nodes and networks for deriving stress test cases and also to find the network paths between any two given nodes. For example, if a tester's goal is to stress test only the network *Network*$_2$ in the system shown in Figure 25, the test strategy will only look for messages going through *Network*$_2$ in the NIG tree and will generate the test cases by considering only those messages.

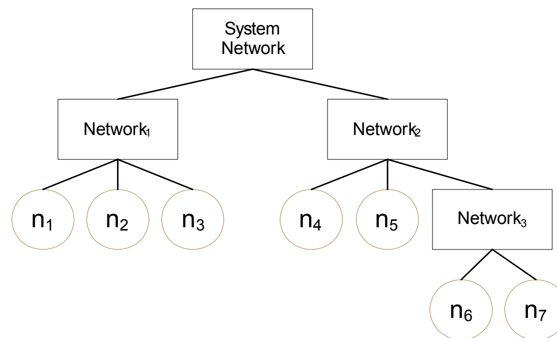


Figure 25-Network Interconnectivity Graph (NIG) of the topology in Figure 22.

Generating the NIG from a NDD is an easy procedure. The root node will be the system's overall network (*System Network*). Then, all the high-level subnets of the system will be the children of the root. This repeats for all the nested subnets in the system. We finally put the distributed nodes of the system as leaf nodes. The bandwidth values of different components, modeled by the bandwidth tagged value in the design UML model, can also be stored in NIG data structure's elements (rectangles for networks and circles for nodes) and edges (representing switch/router between networks).

5.6 Modeling Real-Time Constraints

As discussed in Section 3.1.3, Real-Time (RT) constraints are of two types: *soft* and *hard*. Hard RT constraints are constraints that absolutely must be met. A missed hard deadline

results in a system failure. Soft RT constraints are those which can be missed occasionally, i.e., the probability that they can be missed is usually limited by a threshold.

Furthermore, as discussed in Section 2.4, the UML profile for Schedulability, Performance, and Time (UML-SPT) [12] proposes comprehensive modeling constructs to model timing information. Although UML-SPT briefly mentions soft and hard RT constraints (Section 2.2.3 of [12]), it doesn't propose any specific stereotypes to distinguish between hard and soft RT constraints in UML models. There is an *SAction* stereotype (*S* for Schedulability) in the UML-SPT, and one of its tagged-values is *laxity*: which specifies the type of deadline: hard or soft. But the stereotype is primarily intended for schedulability purposes, by having some other schedulability-specific tagged-values.

On the other hand, explicit distinction of soft and hard RT constraints when modeling can be beneficial. This can help analysts, developers and testers to distinguish between the two types and perform necessary actions for each of them. For example, stress testing hard RT constraints is of higher priority compared to the soft constraints. We will see in Chapter 9 how our stress testing technique deals with the two types of RT constraints.

In order to model hard and soft RT constraints, we propose an extension to the *RTaction* stereotype of the UML-SPT referred to as *HRT* (Hard RT Constraints) and *SRT* (Soft RT Constraints). Furthermore, in order to model the statistical threshold probability up to which SRT constraints can be missed, we consider a tagged value referred to as *missProb* for SRT constraints. Similarly, we consider a tagged value referred to as *criticality* for HRT constraints. Criticality is a real number in the range [0..1] indicating the degree to which the consequences of missing a hard deadline are unacceptable: the closer to one the criticality of a HRT constraint, the more severe the consequences of missing it. For

example, if violating a HRT constraint may cause life-threatening situations, it would be better to set criticality to 1. Conversely, if for example the cost of violating a HRT constraint is just an increase in the temperature of a water hydro plant (which will not immediately lead to catastrophic results), then this constraint would have a lesser value of criticality. *HRTaction* and *SRTaction* stereotypes are presented in Table 1 and Table 2, which are simila r to the representation used in the UML-SPT [12]. In DRTSs where some messages are identified by analysts to have higher priorities than other messages, the *HRTaction* and *SRTaction* stereotypes can be used to model those priorities.

Table 1 and Table 2 define two new stereotypes, «SRTaction» and «HRTaction», which can be applied to any of the four UML modeling concepts listed (*Message*, *MessageSequence*, *Action*, and *ActionSequence*) or to their respective subclasses. *Message* corresponds to messages in SDs. A *MessageSequence* is an ordered sequence of SD messages. *Action* corresponds to actions in activity diagrams (AD). A *ActionSequence* is an ordered sequence of AD actions. For further details on these base classes, refer to [12]. The «SRT» and «HRT» stereotypes have two associated tagged values each, which are defined in Table 3.

Stereotype	Base Class	Tags
SRTaction	Message	RTduration RTmissProb
	MessageSequence	
	Action	
	ActionSequence	

Table 1-A stereotype to model SRT constraints.

Stereotype	Base Class	Tags
HRTaction	Message	RTduration RTcriticality
	MessageSequence	
	Action	
	ActionSequence	

Table 2-A stereotype to model HRT constraints.

Tag	Type	Multiplicity
RTduration	RTtimeValue	[0..1]
RTmissProb	Real [0…1]	[0..1]
RTcriticality	Real [0…1]	[0..1]

Table 3-Tagged values of SRT and HRT stereotypes.

Table 3 defines the type of each tag. An *RTduration* tagged value is an instance of the *RTtimeValue* data type (Section 4.2.2.4 of [12]). *RTmissProb* and *RTcriticality* are real value in the range of [0…1]. Each tag also has a multiplicity indicating how many individual values can be assigned to each tag. A lower bound of zero implies that the tagged value is optional.

«SRTaction» and «HRTaction» stereotypes can be used either in a SD or a MIOD. In the former case, the RT constraint is applied to a *Message* or a *MessageSequence,* while in

the latter, the constraint is applied to an *Action*, or an *ActionSequence* (since MIOD is a subtype of activity diagrams). Examples usages of the «SRTaction» and «HRTaction» stereotypes in a SD and in a MIOD are demonstrated in Figure 26.

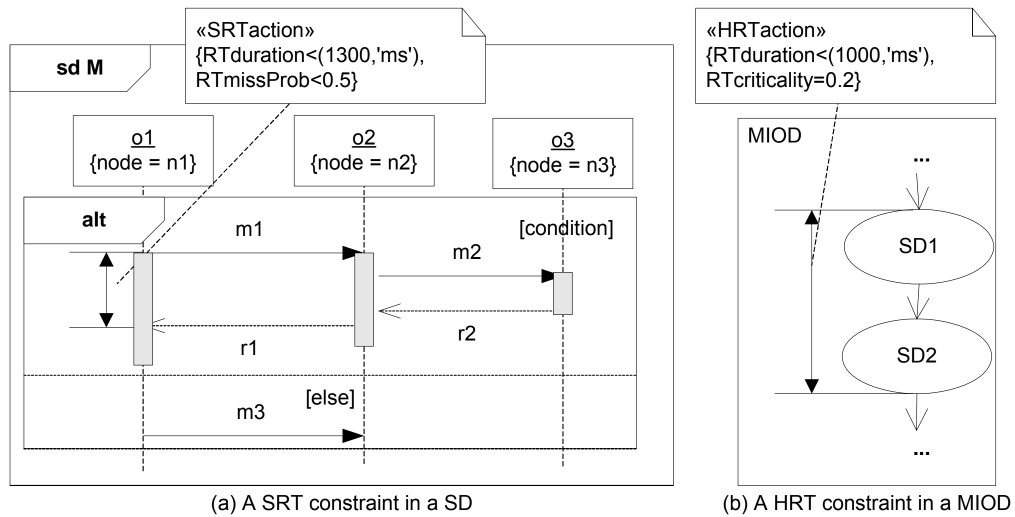


Figure 26- Examples usages of the «SRTaction» and «HRTaction» stereotypes in a SD and in a MIOD.

Chapter 6

CONTROL FLOW ANALYSIS OF SEQUENCE DIAGRAMS

We presented a Control Flow Analysis (CFA) technique in [50] to analyze control flow in SDs. We presented Concurrent Control Flow Graph (CCFG) as a Control Flow Model (CFM) for SDs. If we consider the UML 2.0 SDs metamodel (Figure 4), asynchronous messages and *par* interaction operator entail intra-SD concurrency. However, such concurrency cannot be analyzed by conventional CFGs (Control Flow Graphs). Concurrency resulting from the above two modeling features has to be taken into account when analyzing the control flow in SDs. The impacts of the above two modeling features, leading to concurrency inside SDs, were discussed in [50].

We review in Section 6.1 some of the discussions from [50] which we use in this thesis, such as Concurrent Control Flow Paths (CCFPs). More details on our control flow analysis technique can be found in [50]. Section 6.2 discusses how the distribution and timing can be incorporated in control flow information. In order to precisely define how we will perform traffic analysis of SDs (Chapter 8) based on their distribution and timing

information, we formally define SD messages. in Section 6.3. A special type of CCFPs (Distributed CCFP), which will be used in our methodology later on, is presented in Section 6.4. In order to help developers visualize the behavior of DCCFPs with respect to time over the system nodes and networks, a timed inter-node and inter-network representations of distributed CCFPs are presented in Section 6.5, which will be used in future chapters.

6.1 An Overview of our Control Flow Analysis Technique

We review in this section some of the discussions from [50] which we use in this thesis. We first briefly identify the challenges of SDs' CFA in Section 6.1.1. By surveying some of the existing Control Flow Model (CFM) in the literature, Section 6.1.2 explains our choice of a suitable CFM. The detail of our CFM is presented in Section 6.1.3. Section 6.1.4 presents a set of OCL-based mapping rules from SDs to our CFM, and illustrates them on an example SD. Based on the our CFM's metamodel, Section 6.1.5 discusses how different control flow paths of a SD can be formally represented.

6.1.1 Challenges of SDs' CFA

Conventional CFA techniques [64] are usually applied to sequential programs, and are thus not easily applicable when concurrency has to be accounted for.

Asynchronous messages and the *par* interaction operator in SDs entail intra-SD concurrency. Figure 27 shows an example SD that we will use later in the article to illustrate our approach. It contains two asynchronous messages, namely *addToQueue()* and *process()*, labeled C and E respectively. (The other messages are synchronous.) Figure 27 also shows one of the new constructs in UML 2.0 SDs, namely combined

fragments. The combined fragments are labeled *opt* and *loop*, which are respectively used to specify options (alternatives) and loops. The interaction occurrence (*ref*) is used to refer to other SDs. *par* is another combined fragment used to illustrate asynchronous communications of groups of messages: for instance, in Figure 28, messages *m1*, and *m2* and *m3* are handled in parallel. The reader not familiar with those new constructs is referred to [65] for further details.

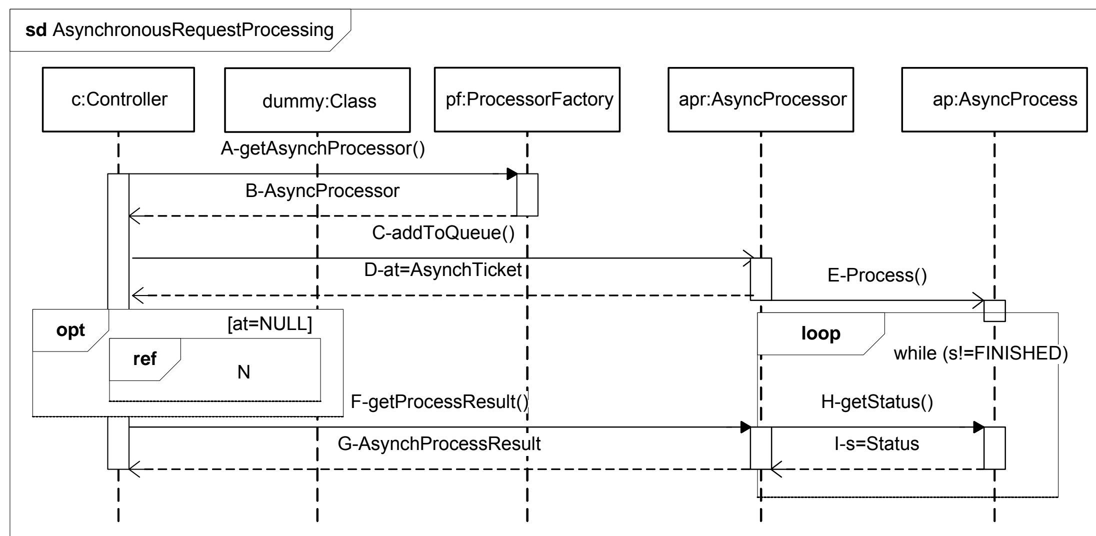


Figure 27-A SD with asynchronous messages.

Although traditional CFG models have constructs (i.e., nodes and edges) to specify branching and sequences of executions, they do not possess specific constructs to specify asynchronous messages or the concurrent sequences of executions.

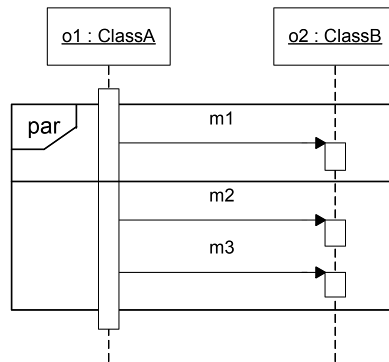


Figure 28- A SD with *par* operator.

6.1.2 Towards a CFM for SDs

Various CFMs have been proposed for CBCFA of concurrent programs in the literature, such as: [66], [67], [68], [18], [69] and [70]. The first three CFMs are used in the context of compilers, languages and formal methods. Although they are well-defined, they are difficult to adapt to the UML notation. Thus, they cannot be easily used in the context of UML and metamodel-based transformations. The work in both [69] and [70] reports on Petri net-based CFMs. Petri-nets [18] contain specific constructs to specify sequences of executions, either synchronous or not, where fork and join nodes (bars) are used to model synchronization of concurrent executions. One advantage of Petri-nets is that they have a well-established formal notation that has been widely used for the modeling of dynamic behavior, as well as extensive tool support.

Among the CFMs used in the previous MBCFA techniques (i.e., IRCFG [71], CRE [72], and LGSPN [31]), only LGSPN (a Petri-net based model) takes into account concurrent control flow.

UML has adopted a Petri-net like semantics for control and object flow modeling referred to as *Activity Diagrams* (AD). ADs have been in UML since its early 1.x versions . They

account for both sequential and concurrent control flow and data flow. As UML 2.0 points out (Section 12.1 of [65]): "*These [UML activities] are commonly called control flow and object flow models*." Among the alternative representations of LGSPN, TIG [69], TIG-based Petri-nets [70], Petri-net and UML AD, we choose the latter for the CFA of SDs. The reasoning for this decision is threefold:

1. AD already has a well-defined metamodel, which is needed by our MBCFA technique and satisfies our needs.
2. SD and AD are both in the context of UML. Both metamodels are part of a large collection (UML) and they have been designed in a similar methodology. Therefore, our technique may potentially benefit from the fact that both metamodels are part of the same context.
3. Furthermore, the generated CFM (a slightly modified version of ADs in our case) can be easily visualized/analyzed by standard UML 2.0 CASE tools.

UML 2.0 proposes six different but dependent activity packages (Section 12 of [65]): BasicActivities (*BA*), StructuredActivities (*SA*), IntermediateActivities (*IA*), CompleteStructuredActivities (*CSA*), ExtraStructuredActivities (*ESA*), and CompleteActivities (*CA*). Based on the analysis of the metamodel and class descriptions of the activity packages, we decide to use the *IA* package as the starting point towards a CFM for our MBCFA approach. The reasons for this selection are: (1) The *IA* package fits the needs of MBCFA, since it supports modeling both concurrency (by being Petri-net like, i.e., including ForkNode and JoinNode) and structured control constructs, and (2) The *IA* package is simpler than the other three packages (*CSA*, *ESA* and *CA*), as it does not include the modeling features needed for advanced data flow modeling. The *BA*

and *SA* packages are not chosen since they are too simple, i.e. they do not include the ForkNode and JoinNode constructs. However, due to specific requirements in our MBCFA, which will be explained in the next section, we need to extend the *IA* package to a new activity package called *CCFG*.

6.1.3 Concurrent Control Flow Graph: a Control Flow Model for SDs

We merge[1] the Concurrent CFG (CCFG) package from the *IA* package by adding new associations and sub-classes as a CFM for SDs. A CCFG (activity class in the metamodel) will be generated for one SD. In the case where a SD calls (refers to) another SD, there will be control flow edges connecting their corresponding CCFGs. We can refer to this concept as *Inter-SD CCFG*, similar to the concept of inter-procedural CFG [64].

The CCFG metamodel is shown in Figure 29. Extensions are made to four of the classes in the *IA* metamodel: *Activity*, *ActivityNode*, *ExecutableNode*, and *ActivityPartition*, which are described next. Furthermore, since the *Activity* class of the *IA* metamodel is extended, its sub-classes (*ControlNode*, *ExecutableNode* and their sub-classes as well) in the *CCFG* metamodel are also implicitly extended from their corresponding classes in the *IA* metamodel.

---

[1] "Merge" is a terminology (stereotype) used on associations between two UML AD packages (Figure 175 of [65] OMG, "UML 2.0 Superstructure Final Adopted specification," 2003.). The classes of the AD package on the tail of the "merge" association extend the classes of the AD package on the head of the association.

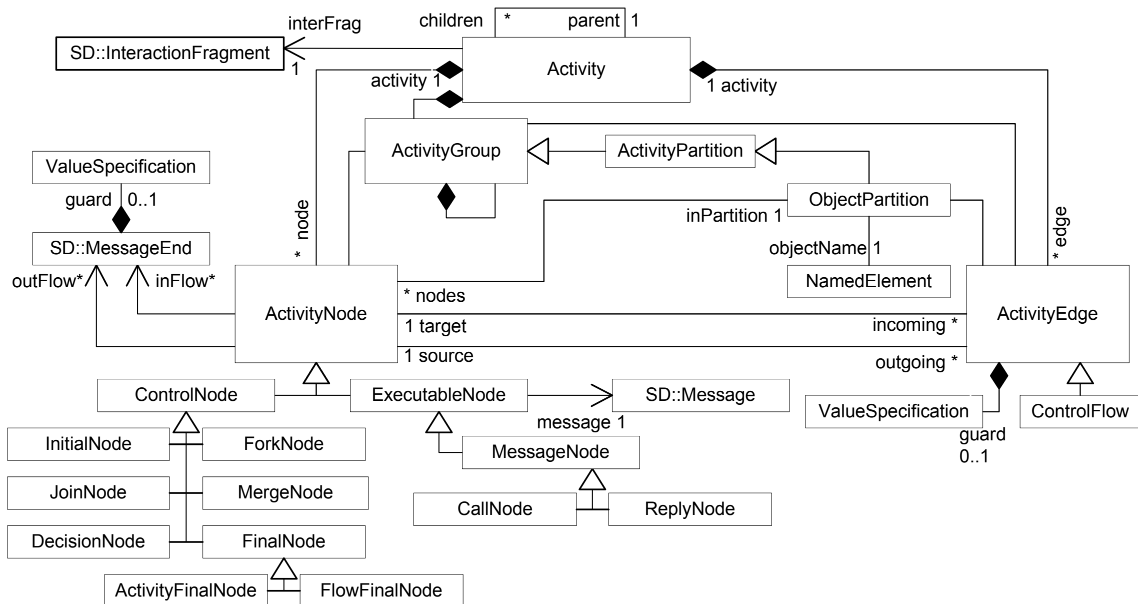


Figure 29- CCFG metamodel.

Each instance of the *Activity* class in the CCFG metamodel corresponds to an instance of the *InteractionFragment* class in the SD metamodel. Therefore, in order to access the interaction fragment associated with an activity, we need to add an association from the activity class in the CCFG metamodel to the interaction fragment class in SD metamodel. Note that the activity and CCFG classes of the CCFG metamodel are used interchangeably in this article. Furthermore, since SD interaction fragments can be nested, their corresponding activities have to be nested too. Therefore, we add a reflexive bidirectional association from the activity class to itself with role names *parent* and *child*. Each CCFG has one parent CCFG and can have multiple child CCFGs.

The need for an extension to *ActivityNode* arose when we started to design our mapping approach (Section 6.1.4). In order to build all control flows (edges) among all activity nodes of a CCFG, we need to make associations between activity nodes and their

corresponding messages' message ends. In other words, we add two associations to *ActivityNode: inFlow* and *outFlow*, which are both targeted to the *SD::MessageEnd* class. These two associations will keep track of in and out flows of an activity node, to be used later in control flow connection. The reason why we assign a zero to many multiplicity (*) for these two associations is that there might be cases in which more than one in/out flows have to be built towards/from a node. Consider node *F* in Figure 30 as an example, which has two in flows. Our strategy for control flow connection is to store send and receive events of each message in the in and out flow sets of its corresponding executable node. In case when a message is inside an *alt* or *loop* combined fragment, the message ends are stored inside in and out flows of the decision node, responsible for controlling the flow of the combined fragment. The guard association of the *MessageEnd* is intended for storing guard conditions when storing in/out flows of a node. The guard will be later copied to the corresponding activity edge. More details on how the in and out flows are handled are provided in Section 6.1.4 and [59]. Note that the subclasses of *ActivityNode* in the CCFG metamodel extend *ActivityNode* in similar ways like subclasses of *ActivityNode* in the IA metamodel. For example, *CCFG::ControlNode* extends *CCFG::ActivityNode* (which itself extends *IA::ActivityNode*) and *IA::ControlNode*. We also change the semantics of activity final nodes in a way that they can have outgoing edges. This is needed to be done since when a SD calls another SD using an interaction occurrence, there needs to be a control flow from the activity final node of the CCFG corresponding to the called SD to a message node in the CCFG corresponding to the caller SD (see Figure 30 for example).

We define new *CallNode* and *ReplyNode* classes in a CCFG which correspond to a call/reply message in the corresponding SD. Distinguishing between call and reply messages (and their corresponding activity nodes) can enrich our MBCFA technique. These two classes in a CCFG are generalized by a new abstract class *MessageNode* which itself is being generalized by *ExecutableNode* in *IA*. One more extension is to make it possible to access the corresponding message of an *ExecutableNode*. In order to do this, we add an association to *ExecutableNode*, which is entitled *message* and is targeting the *SD::Message* class.

An extension to *ActivityPartition* is made by adding a subclass named *ObjectPartition*. This is to group a CCFG's activity nodes based on the receiver object of their corresponding messages: swimlanes in an AD then correspond to classifiers in the corresponding SD.

6.1.4 Consistency Mapping Rules from SDs to CCFGs

Using OCL [73], we propose a consistency rule-based approach to map SDs into CCFGs. The mapping rules are useful in several ways: (1) they provide a logical specification and guidance for our transformation algorithms that derive a CCFG from a SD (both being instances of their respective metamodels), and (2) they help us ensure that our CCFG metamodel is correct and complete with respect to our control flow analysis purpose, as the OCL expression composing the rules must be based on the metamodels.

We have derived fourteen consistency rules, expressed in OCL, that relate different elements of an instance of a SD metamodel to different elements of an instance of the CCFG metamodel. They are all listed in Table 4 and are illustrated them with the

example SD of Figure 27. However, due to space constraint, we only describe a subset of them in the remaining sections.

#	**SD feature**	**CCFG feature**
1	Interaction	Activity
2	First message end	Flow between InitialNode and first control node
3	SynchCall/SynchSignal	CallNode
4	AsynchCall or AsynchSignal	(CallNode+ForkNode) or ReplyNode
5	Message SendEvent and ReceiveEvent	ControlFlow
6	Lifeline	ObjectPartition
7	*par* CombinedFragment	ForkNode
8	*loop* CombinedFragment	DecisionNode
9	*alt/opt* CombinedFragment	DecisionNode
10	*break* CombinedFragment	ActivityEdge
11	Last message ends	Flow between end control nodes and FinalNode
12	InteractionOccurrence	Control Flow across CCFGs
13	Polymorphic message	DecisionNode
14	Nested InteractionFragments	Nested CCFGs

Table 4- Mapping rules from SDs to CCFGs.

To demonstrate the feasibility of our approach, we have applied the consistency rules to the SD of Figure 27 and the resulting CCFG is shown in Figure 30. Each message node in CCFG of Figure 30 is labeled with the corresponding message name in SD of Figure 27. Two fork nodes in the CCFG are created because of the two asynchronous messages in the SD of Figure 27. Due to space limitations, we describe in the next sections only two of the rules (#2 and #3) and how they are applied to the SD of Figure 27. In the rule descriptions below, symbol → means mapping from a SD feature to a CCFG feature. Further details on applying each consistency rule and the OCL expressions of the other consistency rules are described in [59].

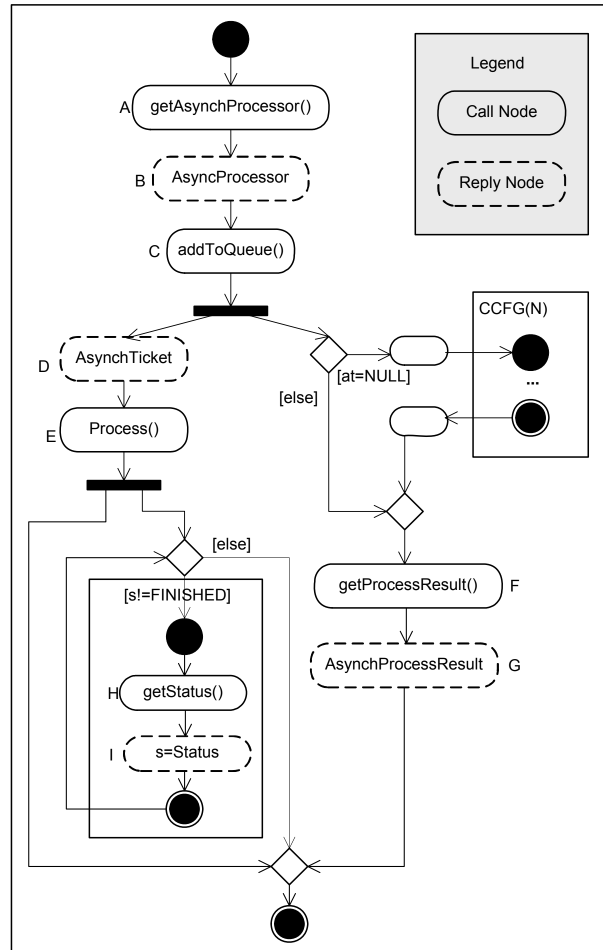


Figure 30-CCFG of the SD in Figure 27.

6.1.4.1 First Message End → Flow between InitialNode and first Control Node

This consistency rule checks if there is a flow from the initial node of every CCFG to its first control node. The first control node of a CCFG is the one that corresponds to the first message of the corresponding SD.

OCL Mapping

```
1 SD::InteractionFragment.allInstances->forAll(interFrag:InteractionFragment|
2   CCFG::InitialNode.allInstances->exits(in:InitialNode|
3     in.activity=Utility::Util.getCCFG(interFrag) and
4     in.outgoing->includes(flow:ControlFlow|
5       getCCFG(interFrag).node->exits(an:ActivityNode|
6         an.inFlow->includes(Utility::Util.getFirstMessage
7         (interFrag).sendEvent) and flow.target=an
8       )
9     )
10  )
11)
```

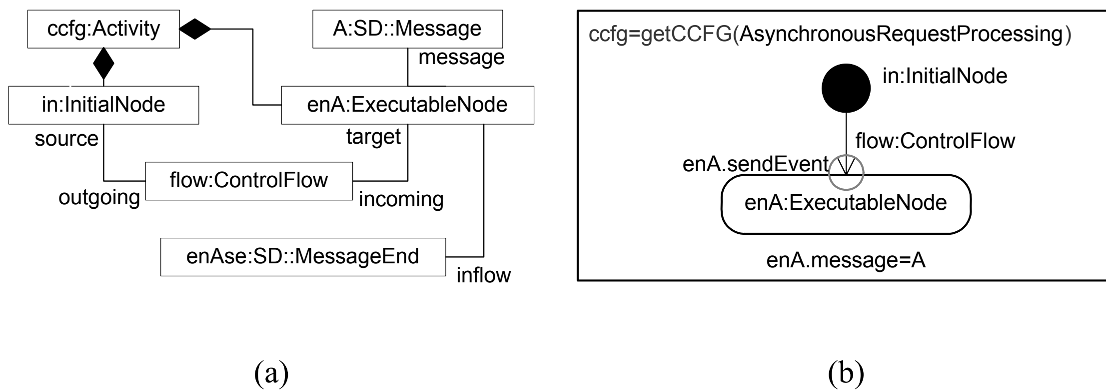
The CCFG of each interaction fragment is checked to have an initial node (lines 1-2) with specific characteristics. There should be a control flow from the initial node to the activity node having the *sendEvent* of the first message of the interaction fragment in its *inflow* (lines 4-7).

To reduce the complexity of our consistency rules, we have defined several utility functions inside a utility class *Util*, such as *getCCFG()* and *getFirstMessage()*, as they are used in the above rule. *getCCFG()* returns the *CCFG::Activity* instance associated with an instance of *SD::InteractionFragment*. *getFirstMessage()* returns the first message of a given interaction fragment according to the ordering provided by its *GeneralOrdering*. *GeneralOrdering* is the SDs mechanism to order messages. More details are provided in [59].

Example

Figure 31-(a) shows part of the CCFG instance that satisfies the above consistency rule based on Figure 27. The corresponding part of the resulting CCFG is represented in Figure 31-(b). Executable node *enA* corresponds to the message *A* (*getAsynchProcessor)*

in Figure 27. *enAse* is the *sendEvent MessageEnd* of the message *A*. *enAse* is already in the inflow of node *enA* because of the rule #3. Furthermore, *getFirstMessage(ccfg)* returns message *A,* and in this way, the appropriate control flow connection between the *ccfg*'s initial node and node *enA* is checked to exist.



(a) (b)

Figure 31-(a)-Part of the CCFG instance *ccfg* mapped from the SD in Figure 27, satisfying the consistency rule #2. (b)-Part of the CCFG, corresponding to the instance shown in (a).

6.1.4.2 SynchCall/SynchSignal→CallNode

This rule maps synchronous (call or signal) messages of a SD to call nodes of a CCFG. (These messages are identified thanks to enumeration values *synchCall* and *synchSignal* of message attribute *messageSort*.)

OCL Mapping

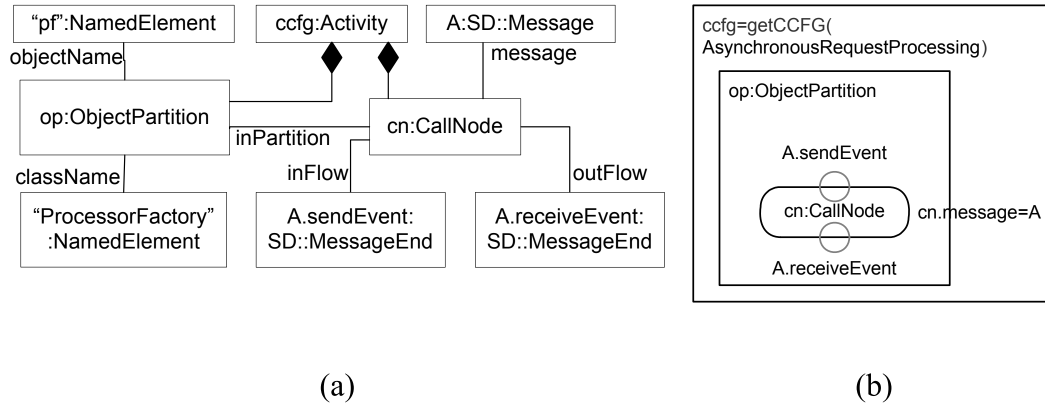
```
1 SD::Message.allInstances->forAll(m:Message|
2   (m.messageSort=SD::MessageSort.synchCall) or
3   (m.messageSort=SD::MessageSort.synchSignal)
4   implies
5     CCFG::CallNode.allInstances->exits(cn:CallNode|
6       cn.message=m and
```

```
        -- check object partition

7       cn.inPartition= Utility::Util.getObjectPartition(
          m.receiveEvent.covered.connectable_element_name) and

        -- make sure cn is prepared for control flow (edge) connections

        -- m.SendEvent/m.ReceiveEvent should be in inFlow/outFlow of cn

8       cn.inFlow->includes(m.sendEvent) and
9       cn.outFlow->includes(m.receiveEvent) and
10      Utility::Util.getCCFG(m.interaction).node->includes(cn)
11    )
12)
```

The *synchCall* and *synchSignal* messages of a SD are selected in lines 1-3. The existence of a corresponding call node *c* is checked in lines 5-10. The call node *cn*'s message association value should be *m* (line 6), its object partition should be lifeline of message *m* (line 7), and its inflow and outflow sets should include only *m*'s send and receive events, respectively (lines 8-9). These inflow and outflow information are needed since the control flow consistency rule (#5) will use them to connect nodes to each other to form the control flow. Line 10 makes sure that node *cn* is a part of the CCFG corresponding to interaction containing message *m*. *getObjectPartition()* is another utility function, which returns the object partition instance associated with a lifeline (using consistency rule #6).

Example

Figure 32-(a) shows part of the CCFG instance corresponding to message *A* in Figure 27 that satisfies the consistency rule #3. The corresponding part of the resulting CCFG is represented in Figure 32-(b). Call node *cn* corresponds to message *A* in Figure 27. Since the receiver lifeline of message *A* is *pf:ProcessorFactory*, the *inPartition* association of *cn* corresponds to *ObjectPartition* instance *op* which is associated with object and class names *pf* and *ProcessorFactory*, respectively.



(a) (b)

Figure 32-(a):Part of the CCFG instance *ccfg* mapped from the SD in Figure 27, satisfying the consistency rule #3. (b): Part of the CCFG, corresponding to the instance shown in (a).

6.1.5 Concurrent Control Flow Paths

The concept of Concurrent Control Flow Paths (CCFPs) is similar to the conventional Control Flow Paths (CFPs), except that they consider concurrent control flows as they are derived from CCFGs [50, 59]. We presented a grammar in [50, 59] to derive all different CCFPs of a CCFG.

For example, by using such grammar, some of the CCFPs of the CCFG in Figure 30 can be derived as shown in Figure 33. The symbol ρ will be used in the rest of this article to refer to CCFPs.

$$\rho_1 = ABC\begin{pmatrix} DE\begin{pmatrix} \\ \end{pmatrix} \\ FGHI \end{pmatrix} \qquad \rho_2 = ABC\begin{pmatrix} DE\begin{pmatrix} JK \\ \end{pmatrix} \\ FGHI \end{pmatrix}$$

$$\rho_3 = ABC\begin{pmatrix} DE\begin{pmatrix} (JK)^2 \\ \end{pmatrix} \\ FGHI \end{pmatrix} \qquad \rho_4 = ABC\begin{pmatrix} DE\begin{pmatrix} (JK)^3 \\ \end{pmatrix} \\ FGHI \end{pmatrix}$$

Figure 33-CCFPs of the CCFG in Figure 30.

Four CCFPs for the CCFG in Figure 30 are due to the decision node (corresponding to a loop) in the CCFG. According to the grammar of CCFPs (Equation 1 of [50, 59]), a loop can either be bypassed (ε) – if possible, taken only once, a representative or average number, and a maximum number of times. These possibilities have derived the four CCFPs: ρ_1, ρ_2, ρ_3 and ρ_4. The loop is bypassed in ρ_1, taken once in ρ_2, repeated twice in ρ_3, and a maximum number (m) of times in ρ_4. Each CCFP is made of several message nodes of a CCFG. Each message node corresponds to a message in the corresponding SD of the CCFG. In the rest of this article, we will refer to CCFP messages and nodes interchangeably.

6.2 Incorporating Distribution and Timing Information in CCFPs

The discussions in [50, 59] about CCFPs described generic CCFPs in a sense that they can be used to analyze control flow of SDs with distributed or non-distributed messages. In the current context, we consider SDs with distributed messages and we saw in Section 5.1 that the node on which a SD object is deployed can be modeled using *node* stereotype. Since only distributed messages of a SD are of interest to our testing technique, therefore we need to incorporate the distribution data of messages inside CCFPs. As the sender/receiver objects and nodes of a message are already modeled in SDs, we can easily access those information from a CCFP, which is a set of messages.

Furthermore, as discussed in Section 5.1.1, we assumed that timing information of messages in a SD are modeled using the *RTstart* and *RTend* tagged values of the UML-SPT profile [12].We can also easily access such information of each message in a CCFP.

Following the above discussion, we can derive all the above information along with message signature and returns list of messages from SDs during the CFA phase. To facilitate our mathematical relations in the next sections, we consider the following format for the call and reply messages of each CCFG and CCFP.

6.3 Formalizing Messages

In order to precisely define how we perform traffic analysis of SDs, we formally define SD messages. Similar to the tabular representation of messages, proposed by UML 2.0 [10], each message annotated with timing information (using the UML-SPT profile [12]) can be represented as a tuple:

message=(sender, receiver, methodOrSignalName, parameterList, returnList, startTime, endTime, msgType)

where

- *sender* denotes the sender of the message and is itself a tuple in the form *sender=(object, class, node*), where:
  - *object* is the object (instance) name of the sender.
  - *class* is the class name of the sender.
  - *node* is where the sender object is deployed.
- *receiver* denotes the receiver of the message and is itself a tuple in the same form as *sender.*
- *methodOrSignalName* is the name of the method or signal on the message.
- *parameterList* is the list of parameters for call messages. *parameterList* is a sequence in the form $parameterList=<(p_1, C_1, in/out), ..., (p_n, C_n, in/out)>$,

where p_i is the *i*-th parameter with class type C_i and *in/out* determines the kind of parameter p_i. For example if the message is $m(o_1{:}C_1,\ o_2{:}C_2)$, then the ordered parameters set will be $parameterList=<(o_1,\ C_1,\ in),\ (o_2,\ C_2,\ in)>$. If the method call has no parameter, this set will be empty.

- *returnList* is the list of return values on reply messages. It is empty in other types of messages. UML 2.0 assumes that there may be several return values by a reply message. We show *returnList* in the form of a sequence $returnList=<(var_1=val_1,C_1),\ \ldots,\ (var_n=val_n,C_n)>$, where val_i is the return values for variable var_i with type C_i.
- *startTime* is the start time of the message (modeled by UML-SPT profile's *RTstart* tagged value).
- *endTime* is the end time of the message (modeled by UML-SPT profile's *RTend* tagged value).
- *msgType* is a field to distinguish between signal, call and reply messages. Although the *messageSort* attribute[2] of each message in the UML metamodel can be used to distinguish signal and call messages, the metamodel does not provide a

---

[2] The *messageSort* attribute of a message specifies the type of communication reflected by the message [10] Object Management Group (OMG), "UML 2.0 Superstructure Specification," 2005., and can be any of these values: *synchCall* (synchronous call), *synchSignal* (synchronous signal), *asynchCall*, or *asynchSignal*

built-in way to separate call and reply messages. Further explanations on this and an approach to distinguish between call and reply messages can be found in [50].

6.4 Distributed CCFP

Distributed CCFP is a CCFP where CCFP messages (call or reply) are distributed. A CCFP message is distributed if its sender and receiver are located in two different nodes. Formally, using the definitions of call and reply node from Section 6.2 a CCFP message *msg* is distributed if:

$$msg.sender.node \neq msg.receiver.node$$

where *msg* can be either a call or a reply message. In other words, a distributed CCFP message is one whose corresponding SD message goes to a different receiver node than its sender node. Similarly, Distributed CCFP (DCCFP) is a CCFP that only includes distributed CCFP messages. A DCCFP is built from a given CCFP ρ by removing all local messages and keeping the distributed ones. As an example, let us assume the CCFPs given in Figure 33. In order to derive their DCCFPs, we should first judge each messages as local or distributed. According to the corresponding SD (Figure 27), all the messages except the messages *A* and *B* are distributed. Therefore, in the CCFG of Figure 30, only control nodes *A* and *B* are local, and the rest are distributed. Hence, the DCCFPs corresponding to the CCFPs given in Figure 33 are shown in Figure 34.

$$DCCFP(\rho_1) = C\begin{pmatrix} DE\begin{pmatrix} \\ \end{pmatrix} \\ FGHI \end{pmatrix} \quad , DCCFP(\rho_2) = C\begin{pmatrix} DE\begin{pmatrix} JK \\ \end{pmatrix} \\ FGHI \end{pmatrix}$$

$$DCCFP(\rho_3) = C\begin{pmatrix} DE\begin{pmatrix} (JK)^2 \\ \end{pmatrix} \\ FGHI \end{pmatrix} \quad , DCCFP(\rho_4) = C\begin{pmatrix} DE\begin{pmatrix} (JK)^3 \\ \end{pmatrix} \\ FGHI \end{pmatrix}$$

Figure 34- DCCFPs of the CCFPs in Figure 33.

6.5 Timed Inter-Node and Inter-Network Representations of DCCFPs

In this section, we provide a timed inter-node (and inter-network) representation of DCCFPs. This representation can help to visualize the behavior of DCCFPs with respect to time over the system nodes and networks. This will help to better understand our discussions in the remainder of this article.

UML 2.0 introduces a new interaction diagram called *Timing Diagrams* (Section 14.4 of [10]). As defined by UML 2.0: “Timing Diagrams are used to show interactions when a primary purpose of the diagram is to reason about time. Timing diagrams focus on conditions changing within and among lifelines along a linear time axis.” We use the basic concepts of UML 2.0 timing diagrams and propose a model for timed inter-node and inter-network representations of DCCFPs. These two representations of a DCCFP can be useful to represent a timeline view of the flow and occurrence of distributed messages by a DCCFP in node and network levels. These representations are 2-dimentioanl charts where the X-axis is a linear time axis and the Y-axis is the set of all nodes referenced at least once by the control nodes of a given DCCFP.

For example, let us consider the SD of Figure 27 and $DCCFP(\rho_2)$ in Figure 34. Timed inter-node representation of $DCCFP(\rho_2)$ in shown in Figure 35, where the message ends

correspond to the type of corresponding messages (synchronous/asynchronous call or reply) in the SD. Let us also assume that the start and end times of all control nodes (*A*…*K*) are given using UML-SPT profile stereotypes (Section 5.1.1) with the values as shown. In this representation, the X-axis represents time and the Y-axis lists the nodes of the DCCFP messages.

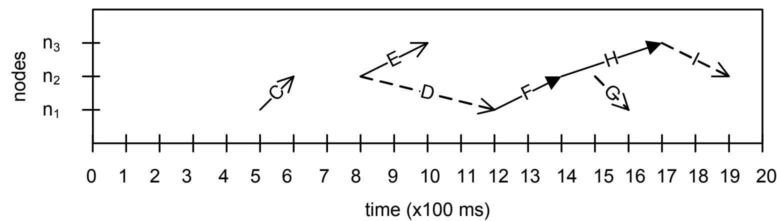


Figure 35-Timed inter-node representation of *DCCFP*(ρ_2) in Figure 34.

Suppose the NIG of this system is as the one shown in Figure 36. The inter-network representation of the *DCCFP*(ρ_2) can be derived using the node information in SD, the inter-node representation (Figure 35) and the system NIG.

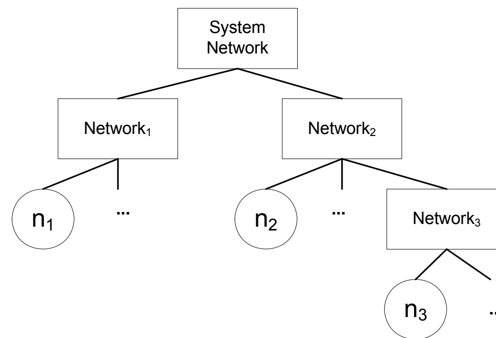


Figure 36-A simple system NIG.

The inter-network representation of the *DCCFP*(ρ_2) is drawn in Figure 37. Start and end networks of each message in this representation are derived by finding the networks where the message's sender and receiver nodes are members. For example, the sender

and receiver nodes of message (call node) *C* are nodes n_1 and n_2, which are members of $Network_1$ and $Network_2$, respectively. In addition to the traffic imposed on networks they start and end, messages like *C* have an implicit traffic on networks that are not their immediate parent in NIG, but are in the network paths from their start to end nodes. For example, *C* entails an implicit traffic on *SystemNetwork* in addition to $Network_1$ and $Network_2$. Other cases like this can also be identified from NIG.

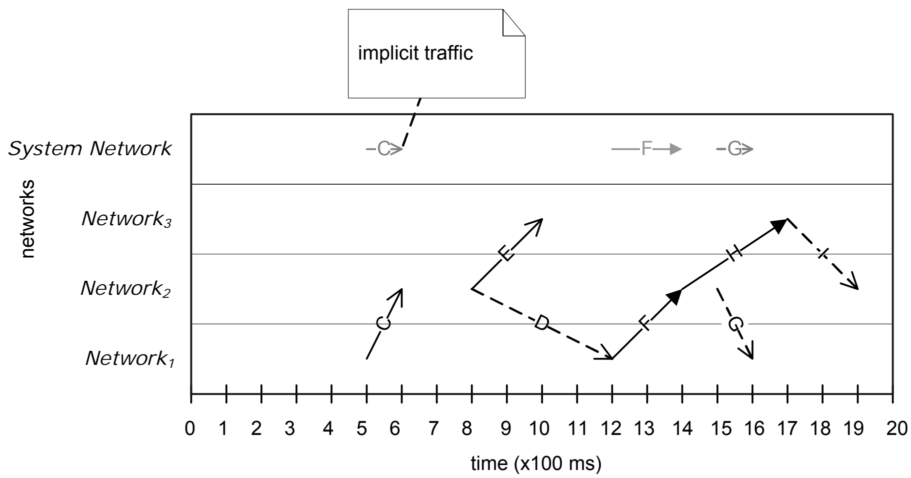


Figure 37-Timed inter-network representation of a DCCFP.

Chapter 7

CONSIDERING INTER-SD CONSTRAINTS

As discussed in Section 5.3, executing any arbitrary sequence of use cases (and thus their corresponding SDs) in a SUT might not be always valid or possible. This might be due to the constraints enforced by the business logic of a SUT on the sequence (order) of SDs and also the conditions that have to be satisfied before a particular SD can be executed. Modified Interaction Overview Diagrams (MIOD) were proposed in Section 5.3 to model sequential and conditional inter-SD constraints. We discussed how such constraints can be modeled by a MIOD.

As we will discuss in Chapter 9, our stress test technique will identify the most data-centric messages of each SD and will try to either run SDs concurrently or will run a sequence of SDs which impose the maximum amount of network traffic. However, test requirements should comply with the inter-SD constraints.

In the following sections, we propose two methods to consider inter-SD constraints in our stress testing context, assuming that a MIOD is given. The method in Section 7.1 will be used to derive the *Independent-SD Sets* (*ISDSs*) in a SUT. An ISDS is a set of SDs, in

which any two SDs are independent, thus the entire set can be run concurrently. In other words, there are no inter-SD sequential constraints between any two of the SDs in an ISDS to prevent from doing so. Our stress test technique in Chapter 9 will make use of ISDS by calculating the maximum traffic of each ISDS by adding the maximum traffic of its SDs. Then, among all ISDS of a MIOD, the ISDS with maximum traffic will be chosen as the ISDS which entails the maximum stress. Then after, the SDs of the chosen ISDS will be scheduled in a way to maximize the instant traffic in a particular time instant.

The method proposed in Section 7.2 will be used to derive the *Concurrent SD Flow Paths* (*CSDFP*) and *CCFP/DCCFP Sequences* (*CCFPS*/*DCCFPS*). Similar to the concept of CCFP, a CSDFP is a path from a MIOD's start node to a final node. The CSDFPs of a MIOD specify the allowed sequences of SDs in a system. According to this definition, any sequence of SD in a SUT which is not a CSDFP is not allowed to be executed.

On the other hand, we defined CCFP and DCCFP in Chapter 6 and saw that each SD can have one or more such paths. We define CCFP/DCCFP Sequences *(CCFPS/DCCFPS)* as the sequences of CCFPs/DCCFPs which are built from a CSDFP. Further explanations are provided in Section 7.2. A variation of our stress test technique in Chapter 9 will make use of CSDFP by calculating the maximum traffic of each CSDFP. Then, among all CSDFPs of a MIOD, the CSDFP with maximum traffic will be chosen as the CSDFP which entails the maximum stress.

7.1 Independent-SD Sets

An *Independent-SD Set* (*ISDS*) is a set of SDs that can be executed concurrently, i.e. there are no sequential constraints between any two of the SDs in the set to prevent it.

Assuming that a MIOD is given, we propose a technique in this section to find all ISDSs of the MIOD. As an example, let us consider the MIOD of a library system as shown in Figure 38. This MIOD is the completed version[3] of the activity diagram shown in [60]. For brevity, the SDs are labeled by capital letters from A to O. The MIOD is modeled using the use case diagram given in Appendix A of [60] and some typical business logic assumptions of the library system.

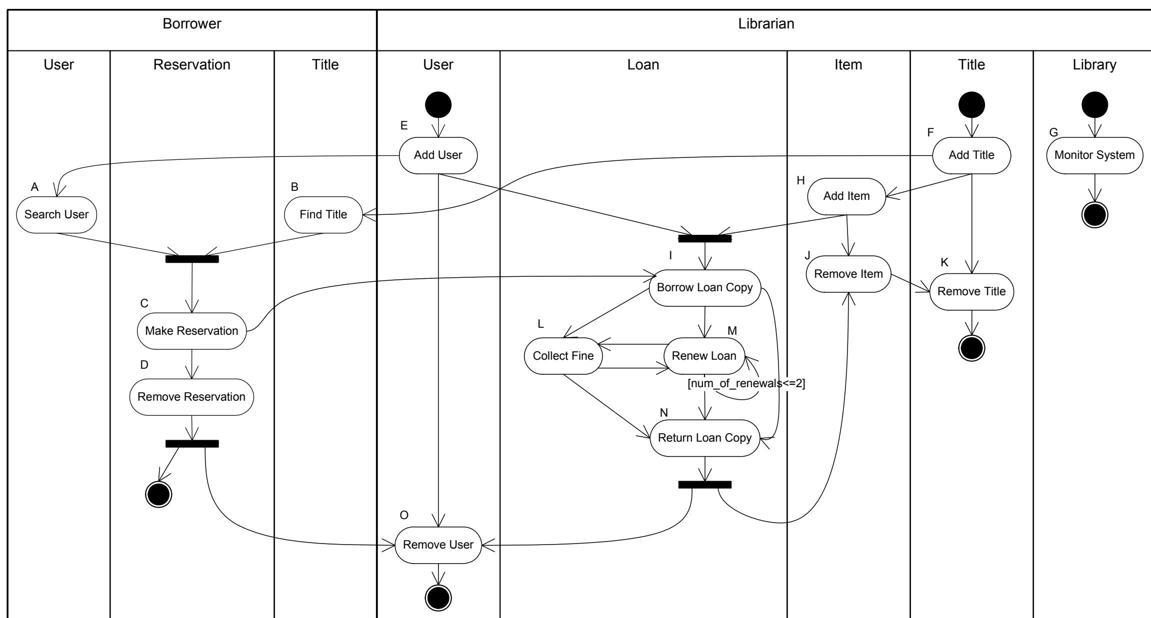


Figure 38- The MIOD of a library system.

---

[3] The sequential constraints of the SDs (use cases) for the actor *Borrower* and the conditional constraint of the SD *RenewLoan* are added.

7.1.1 Definitions

We rephrase here the definition of dependent/independent SDs and an ISDS in the context of a MIOD. A set of SDs are said to be *independent* if there are no inter-SD constraints between any two of the SDs in a MIOD to prevent them from being executed concurrently. As discussed in Section 5.3, sequential and conditional constraints among SDs are modeled in a MIOD. An edge between two SDs (from a tail SD to a head SD) in a MIOD specifies that the tail SD must be executed in order for the head SD to be executed, but the tail SD may be executed without any execution of the head SD. In addition, specific situations require that several SDs be executed independently (without any sequential dependencies between them) for another SD to be executed. This is modeled by *join* and *fork* synchronization bars in a MIOD, respectively.

Therefore, we can define a dependency relationship between any two SDs in a MIOD. Two SDs SD_1 and SD_2 are *dependent* if there is at least one path in the MIOD from one of them to the other one. For example SDs *AddUser* and *ReturnLoanCopy* are dependent in the MIOD of Figure 38. Conversely, two SDs are independent if there is no path in the MIOD from one of them to the other one. For example SDs *AddUser* and *AddTitle* are independent in the MIOD of Figure 38. Similarly, two sets of SDs are said to be independent if all the SDs of one of them are independent from all the SDs of the other one.

In a MIOD, an *Independent-SD Set (ISDS)* is a *maximal* set of independent SDs. By maximal, we mean that no other SD can be added to the set. For example, the set of SDs {*AddUser*, *AddTitle*} is a set of independent SDs in Figure 38, however it is not maximal according to our definition, since, for instance, SD *MonitorSystem* can be added to this

set while the independence relationship still holds among all the SDs in the set. In fact, {*AddUser*, *AddTitle*, *MonitorSystem*} is an ISDS.

7.1.2 Derivation of Independent-SD Sets

According to the discussions in the previous section, ISDSs of a MIOD can be derived by examining SDs of a MIOD and deriving all possible maximal sets of SDs that are independent. Identifying ISDSs from a MIOD is a graph-based problem. If we first build, from a MIOD, a graph where nodes are SDs and edges link independent SDs, then finding ISDSs amounts to finding maximal-complete subgraphs[4], a well-known graph-based problem [74] for which there exist efficient solutions (e.g., [74]).

Let us propose a graph notation referred to as Independent SD Graph (ISDG)=*(N,E)*, where N is the set of SDs of a MIOD and there is an edge in E between two SDs if they are independent according to the definition given in the previous section. For example, the ISDG corresponding to the MIOD in Figure 38 is shown in Figure 39.

Every *maximal-complete subgraph* of an ISDG is an ISDS. For example, the maximal-complete subgraph *{A,B,G,H}* is shown with dashed edges in the ISDG of Figure 39, which corresponds to a ISDS.

---

[4] A *maximal-complete subgraph* is a *complete subgraph* that is not contained in any other complete subgraph. A *complete subgraph* of a graph is a subgraph in which there exists an edge between any pair of nodes [74] R. E. Tarjan, "Depth-First Search and Linear Graph Algorithms," *Society for Industrial and Applied Mathematics' Journal on Computing*, vol. 1, 1972.

When deriving Independent-SD Sets, the effect of *multi-SDs* (Section 5.4) will be as the following. After an ISDS is derived using the algorithm mentioned above, each SD of the ISDS is checked to see if it is a multi-SD. If yes, the multi-SD is replaced with two parentheses similar to the technique we used in [50, 59] to derive CCFPs of a SD. The number of SDs between two parentheses is equal to the number modeled by the tagged-value *instances* annotated to the multi-SD. For example consider ISDS={*A,B,C*} derived from the MIOD in Figure 40. In this MIOD, SD *A* is a multi-SD where three concurrent instances of it can be executed.

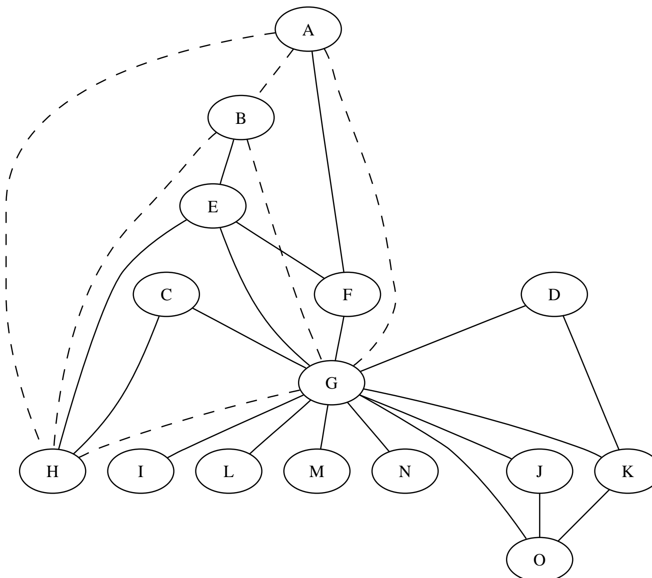


Figure 39-The Independent SD Graph (ISDG) corresponding to the MIOD in Figure 38. The ISDS=*{A,B,G,H}* is shown with dashed edges.

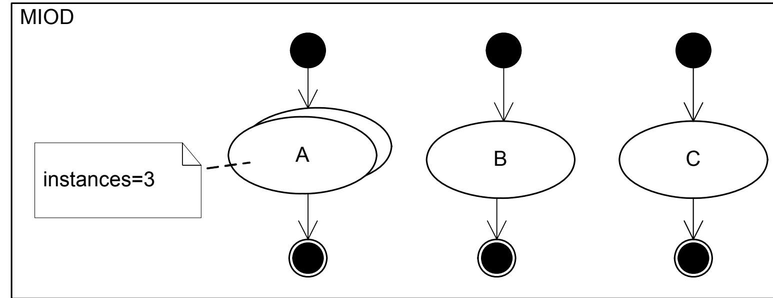


Figure 40-A MIOD with a multi-SD construct.

Since SD *A* is a multi-SD, we modify the ISDS as the following.

$$ISDS = \{ A, B, C \} \Rightarrow ISDS = \{ \begin{pmatrix} A \\ A \\ A \end{pmatrix}, B, C \}$$

The above ISDS transformation means that, if any SD is independent from a multi-SD, it will be independent from its multi instances, too.

7.1.3 Algorithm Complexity

The brute-force algorithm to build an ISDG would be to check all pairs of SDs of a MIOD and build an edge between them in the ISDG if the two SDs are independent. This will have the complexity of $O(n^3)$, where n is the number of SDs.

Tarjan [74] has devised an $O(n)$ algorithm for determining maximal-complete subgraphs of a graph. Therefore consider the complexity to build an ISDG, $O(n^2)$, and the complexity to derive its maximal-complete subgraphs, the overall complexity to derive ISDSs will be $O(n^2)$.

7.2 Concurrent SD Flow Paths, CCFP and DCCFP Sequences

To account for sequential and conditional inter-SD constraints in test cases, we propose Concurrent SD Flow Paths (CSDFP), CCFP and DCCFP Sequences (CCFPS and DCCFPS) in this section.

7.2.1 Concurrent SD Flow Paths

We discussed in Section 5.3 how to model the sequential and conditional constraints among SDs using a MIOD. Similar to the concept of CCFP, which was made from a CCFG, we define a *Concurrent SD Flow Path* (*CSDFP*) to be a sequence of SDs from a start to an end node of a MIOD. In other words, a CSDFP is a sequence of SDs that are allowed to be executed in a system (according to the constraints modeled in a MIOD).

There is a hierarchical relationship between MIODs and CCFGs, and also CSDFPs and CCFPs. To better illustrate this relationship, consider the example given in Figure 41, where a MIOD (a) and the CCFG (b) of one of the SDs in the MIOD are shown.

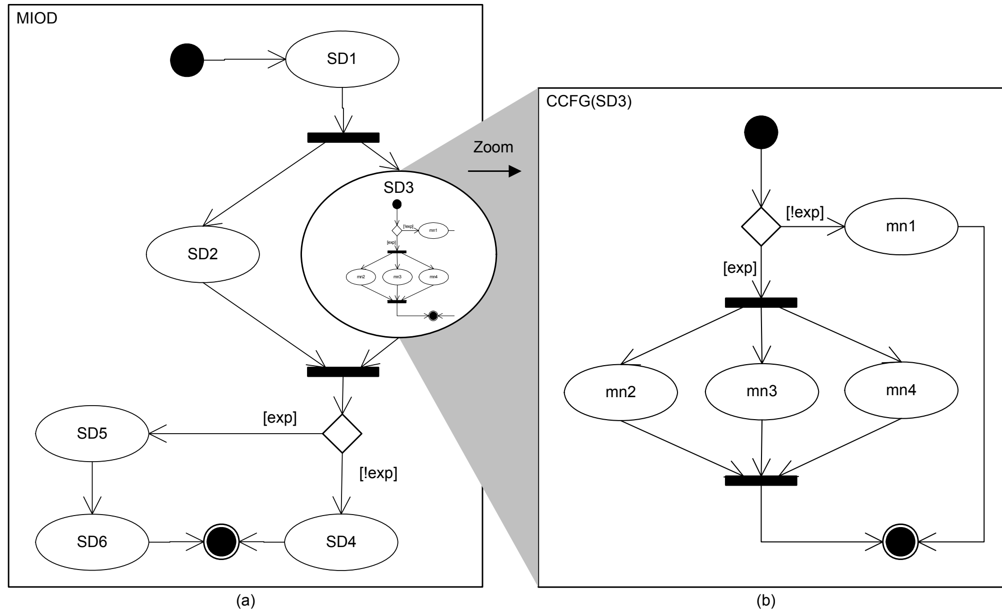


Figure 41-An example MIOD and the CCFG of one of its SDs.

The MIOD shows the system-level flow paths, where the flow paths are built from SDs, e.g., SD_1 and SD_2. In turn, whenever the control is on a SD, the CCFG of the SD determines which control flow should be followed. We have actually enlarged the CCFG of SD_3 in Figure 41 to better represent the hierarchical relationship.

In order to find CSDFPs of a MIOD, we use the same technique as we used in [50, 59] (and discussed in Chapter 6) to derive CCFPs of a SD. This is doable since both MIOD and CCFG are extensions of ADs. For example, the MIOD in Figure 41 has the following two CSDFPs:

$$CSDFP_1 = SD_1 \begin{pmatrix} SD_2 \\ SD_3 \end{pmatrix} SD_4$$

$$CSDFP_2 = SD_1 \begin{pmatrix} SD_2 \\ SD_3 \end{pmatrix} SD_5 SD_6$$

As another example, we list here some of the CSDFPs (out a total of 62) which can be derived from the MIOD in Figure 38:

$$CSDFP_1 = \begin{pmatrix} EA \\ FB \end{pmatrix} CD \quad CSDFP_2 = \begin{pmatrix} EA \\ FB \end{pmatrix} CDO \quad CSDFP_3 = \begin{pmatrix} EA \\ FB \end{pmatrix} CILMLMLNO \quad CSDFP_4 = \begin{pmatrix} E \\ FH \end{pmatrix} ILO$$

$$CSDFP_5 = G \quad CSDFP_6 = FHJK \quad CSDFP_7 = EO \quad CSDFP_8 = FK$$

7.2.2 Concurrent Control Flow Paths Sequence

We defined CSDFP in the previous section. Similar to the concept of control flow paths, a system's set of CSDFPs represents the possible sequences of SDs a system might follow in a typical execution. However, a SD usually contains more than one control flow paths, out of which, only one will execute in a particular run. We discussed CCFP and DCCFP in Chapter 6 as concepts to represent these possible execution paths of a SD. To incorporate CCFP and DCCFP in CSDFPs, we define two new concepts: *CCFPS (Concurrent Control Flow Paths Sequence) and DCCFPS (Distributed CCFPS)* to represent different sequences of scenarios a CSDFP might follow in different executions. A CCFPS can be derived from a CSDFP by substituting each SD by one of its CCFPs. Similarly, a DCCFPS can be derived from a CSDFP by substituting each SD by one of its DCCFPs.

For example, let us consider the example in Figure 41. SD_3 has two CCFPs as:

$$CCFP_{3,1} = mn_1$$

$$CCFP_{3,2} = \begin{pmatrix} mn_2 \\ mn_3 \\ mn_4 \end{pmatrix}$$

where mn_i is the message node corresponding to message *mi (*not shown*)* in SD_3. Suppose $DCCFP_{3,1}$ and $DCCFP_{3,2}$ are the corresponding DCCFPs of the above two CCFPs. Similarly, assume that SD_1, SD_2, SD_4, SD_5 and SD_6 have the following sets of CCFPs. Let us also identify the corresponding DCCFPs by $DCCFP_{i,j}$ (j^{th} DCCFP for SD i).

$$CCFP(SD1) = \{CCFP_{1,1}, CCFP_{1,2}, CCFP_{1,3}\}$$
$$CCFP(SD2) = \{CCFP_{2,1}, CCFP_{2,2}\}$$
$$CCFP(SD4) = \{CCFP_{4,1}, CCFP_{4,2}, CCFP4_{1,3}\}$$
$$CCFP(SD5) = \{CCFP_{5,1}\}$$
$$CCFP(SD6) = \{CCFP_{6,1}, CCFP_{6,2}, CCFP_{6,3}, CCFP_{6,4}\}$$

We derived the CSDFPs of the MIOD in the previous section as:

$$CSDFP_1 = SD_1 \begin{pmatrix} SD_2 \\ SD_3 \end{pmatrix} SD_4$$
$$CSDFP_2 = SD_1 \begin{pmatrix} SD_2 \\ SD_3 \end{pmatrix} SD_5 SD_6$$

By substituting each SD of $CSDFP_1$ by one of their corresponding CCFPs, for example, the following CCFPSs can be derived:

$$CCFPS_1 = CCFP_{1,3} \begin{pmatrix} CCFP_{2,2} \\ CCFP_{3,1} \end{pmatrix} CCFP_{4,3}$$
$$CCFPS_2 = CCFP_{1,1} \begin{pmatrix} CCFP_{2,1} \\ CCFP_{3,2} \end{pmatrix} CCFP_{4,2}$$
$$CCFPS_3 = CCFP_{1,2} \begin{pmatrix} CCFP_{2,2} \\ CCFP_{3,1} \end{pmatrix} CCFP_{4,1}$$

Similarly, the following DCCFPS can be derived from $CSDFP_2$:

$$DCCFPS_1 = DCCFP_{1,3} \begin{pmatrix} DCCFP_{2,2} \\ DCCFP_{3,1} \end{pmatrix} DCCFP_{5,1} DCCFP_{6,4}$$
$$DCCFPS_2 = DCCFP_{1,1} \begin{pmatrix} DCCFP_{2,2} \\ DCCFP_{3,2} \end{pmatrix} DCCFP_{5,1} DCCFP_{6,2}$$

As it can be realized from the definitions of CCFPS and DCCFPS, when the number of SDs and their CCFPs increase, number of CCFPS and DCCFPS can increase exponentially. Ways to cope with this combinatorial explosion problem must be investigated. One such approach is to use available inter-SD control and data flow information to eliminate infeasible CCFPSs, e.g., executing $CCFP_{1,3}$ from SD_1 followed

by $CCFP_{3,2}$ from SD_2 (in $CSDFP_1$) may not be feasible because of the input values to be selected and the resulting system states.

Another important issue is that the current automatic procedure to derive CCFPS (and DCCFPS) may produce infeasible (one could say illegal) CCFPS (DCCFPs). A form of data flow analysis has to be done on the set of derived CCFPS (DCCFPs) to eliminate the infeasible (illegal) ones. This is one of our future works.

7.2.3 Duration of a Concurrent Control Flow Path Sequence

Some of our stress test requirement algorithms in Chapter 9 will need the duration (time length) of a CCFPS. We present Algorithm 1 to recursively calculate the duration of a CCFPS using the time length of the CCFPs in the sequence.

1. Function Duration(*ccfps*: CCFPS): integer
2. if *ccfps* is atomic (only made of one CCFP)
3. return $\max_{\forall m \in ccfps}(m.endTime)$
4. else if *ccfps* is the serial concatenation of several CCFPSs (i.e., $ccfps = ccfps_1 \cdots ccfps_n$)
5. return Duration($ccfps_1$)+…+ Duration($ccfps_n$)
6. else if *ccfps* is the concurrent combination of several CCFPSs (i.e., $ccfps = \begin{pmatrix} ccfps_1 \\ \cdots \\ ccfps_n \end{pmatrix}$)
7. return max(Duration($ccfps_1$),…, Duration($ccfps_n$))
8. End Function

Algorithm 1-Calculating the duration of a Concurrent Control Flow Path Sequence (CCFPS).

Line 2 of Algorithm 1 is the stopping criterion of the recursion. It is when *ccfps* (the given CCFPS) is an atomic CCFPS (only made of one CCFP). In this case, the duration

of *ccfps* is equal to the duration of its one and only CCFP, which is calculated by line 3. As time constraints are modeled in SDs using the UML-SPT profile, the time reference at the beginning of every SD (and hence its CCFPs) is set to zero (see Figure 27 as an example). Therefore, the duration of a CCFP is equal to the end time of its latest message (maximum of *m.endTime*'s). For details on our message formalism, refer to Section 6.3. Lines 4-5 are executed if *ccfps* is a serial concatenation of several other CCFPSs. Since the CCFPSs execute serially in this case, the total duration is the summation of their individual durations. If *ccfps* is a concurrent combination of several other CCFPSs, lines 6-7 will be used. For a concurrent combination of CCFPSs, we assume that all of the CCFPSs start at the same time. Therefore, the duration will be the longest duration of the enclosed CCFPSs.

For example, we calculate the time duration of $CCFPS_1$ discussed in Section 7.2.2. For brevity, we use $p_{i,j}$ for $CCFP_{i,j}$. Suppose the duration of each of the individual CCFPs of $CCFPS_1$ are as the follows: $CCFP_{1,3}$ (2800 ms), $CCFP_{2,2}$ (1300 ms), $CCFP_{3,1}$ (1000 ms), and $CCFP_{4,3}$ (1000 ms). To better illustrate how Algorithm 1 works, the call tree of the recursive algorithm *Duration* applied to $CCFPS_1$ is shown in Figure 42. Since the $CCFPS_1$ is a serial concatenation of three CCFPSs itself, three recursive calls are made, whose results will be added upon return. One of these CCFPSs ($\begin{pmatrix} \rho_{2,2} \\ \rho_{3,1} \end{pmatrix}$), is the concurrent combination of two CCFPs, therefore the maximum value of their durations are returned as the durations of this CCFPS and so on.

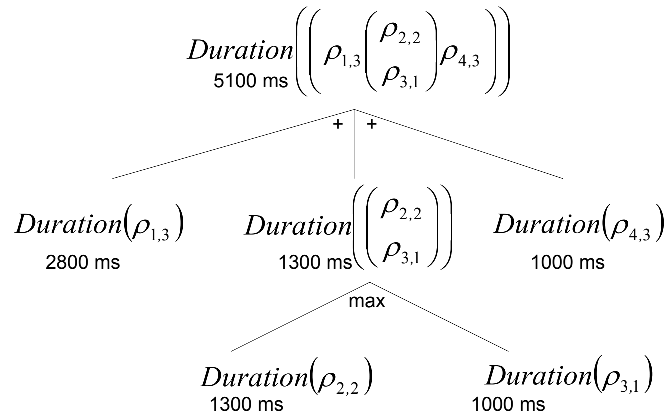


Figure 42-The call tree of the recursive algorithm *Duration* applied to $CCFPS_1$.

Note that the duration of a DCCFPS is equal to duration of its corresponding CCFPS, which is made by replacing all the DCCFPs with the corresponding CCFPs. This is because in order to run a DCCFP, the corresponding CCFP should be executed. As discussed in Section 6.4, a DCCFP is just a filtered CCFP where only distributed messages are selected.

Chapter 8

RESOURCE USAGE ANALYSIS OF DISTRIBUTED TRAFFIC

As we saw in the system model of this work (Figure 11), each node of the system can have several running processes. Different processes often need to communicate with other processes on other nodes of the system to perform a use case. In a typical collaboration between two distributed objects in a SD, the sender object calls an operation of the receiver object via a message (usually with parameters); the message is handled (executed) by the receiver object, and if the message was a call, finally the return values are returned to the sender object as a reply message. Distributed call and reply messages have to go over the network connection between the sender and receiver objects, and entail distributed traffic on the connecting networks. We assume two distributed traffic types: *data* and *message*. Data traffic is the amount of data transferred by distributed messages, which is dependent on the messages sizes. On the other hand, message traffic is the number of messages being transmitted, regardless of their sizes.

In order to study and analyze distributed traffic usage in a current context and to devise network-aware stress test requirement in a SUT, this section aims to formalize the distributed traffic usage of each message and each DCCFP in a system. In order to do so, a method will be proposed in Section 8.1 to estimate data size of a distributed message (a message which goes from a node to a different one). Section 8.2 will provide formal definition of membership relationships between nodes and networks. Different attributes of distributed traffic in our formalism will be proposed in Section 8.3, which will include: Traffic location: nodes vs. networks (Section 8.3.1); Traffic direction (for nodes only): in, out, or bidirectional (Section 8.3.2); Traffic type: data traffic vs. number of messages (Section 8.3.3); Traffic duration: instant vs. interval (Section 8.3.4) – whether traffic is measured in one single time instant or during a period of time. They will allow us to measure traffic usage in different ways, and thus better focus the stress testing activity, for instance on a specific node or network, or on the incoming traffic to a node.

We will then discuss in Section 8.4 the aspects we consider while estimating network traffic usage. The estimations will be formalized by means of traffic functions for DCCFPs in Section 8.5. The resource usage analysis technique presented in this section will be used in Chapter 9 and Chapter 10.

8.1 Estimating the Data Size of a Distributed Message

In order to measure and analyze the amount of traffic every distributed message entails on a network, we need to have a method to estimate the data size of a distributed message. The following representation was presented for messages of a DCCFP in Section 6.2:

message=(sender, receiver, msgSort, methodOrSignalName, parameterList,

returnList, startTime, endTime msgType)

By looking at the above representation, we find out that the most data-centric parts are *parameterList* and *returnList,* respectively, which actually go through a network. These two fields, i.e. *parameterList* and *returnList*, were defined as *parameterList*= $<(p_1,C_1,[in|out]), \ldots, (p_n,C_n,[in|out])>$ and *returnList*=$<(var_1=val_1,C_1), \ldots, (var_n=val_n,C_n)>$, respectively. Therefore, it can be said that the most data centric part of a message are essentially parameters p_i and return values val_i, respectively. Therefore, a simple solution to estimate data size of each message is to find a way to estimate the max (or average) data sizes for each class type C_i in both of sets *parameterList* and *returnList*.

An intuitive way to estimate the data size of a set of classes will be to add up data sizes of all classes in the set. Let us define the data size of a class to be the total summation of sizes of its attributes in bytes. (This is performed recursively if a class attribute has a non-primitive type.) Therefore the total the size of the classes in a *parameterList* and *returnList* can be a rough estimate for the data sizes of call and reply messages. Formally, the *Distributed Traffic Usage (DTU)* functions for different types of messages are presented in Equation 1.

$$
\begin{aligned}
&DTU : Message \rightarrow Real \\
&\forall msg \in Message : DTU(msg) = \begin{cases} SignalDT(msg) & ;\text{if } msg.msgType = 'Signal' \\ CallDT(msg) & ;\text{if } msg.msgType = 'Call' \\ ReplyDT(msg) & ;\text{if } msg.msgType = 'Reply' \end{cases} \\
&SignalDT(msg) = dataSize(msg.methodOrSignalName) \\
&CallDT(msg) = \sum_{C_i \mid (-,C_i) \in msg.parameterList} dataSize(C_i) \\
&ReplyDT(msg) = \sum_{C_i \mid (-,C_i) \in msg.returnList} dataSize(C_i) \\
&\forall C \in classDiagram : dataSize(C) = \sum_{a_i \in C.attributes} dataSize(a_i)
\end{aligned}
$$

Equation 1- Distributed Traffic Usage (DTU) functions for different types of messages.

A dash (-) in Equation 1 indicates that a field can take any arbitrary value (a "don't care" field). Note the format of *parameterList* and *returnList*, as mentioned above: *msg.parameterList (msg.returnList)* is the ordered set of parameters (returns) for a call (reply) message. $dataSize(C_i)$ is a function returning the data size of the class C_i. *C.attributes* denotes the set of attributes of class *C*. $dataSize(a_i)$ is the size of an attribute a_i of class *C,* which can be calculated by its attribute type. If the attribute type is an atomic type, like *int*, *long*, *bool*, its size (in bytes) is dependent on the target programming language. For example, the data sizes of some primitive data types in Java are shown in Table 5 (adopted from [75]). In case an attribute a_i of a class is itself an object with another class type, the size of that attribute, $size(a_i)$, will be the size of its class type and can be calculated recursively.

As an example, suppose a call message msg_1 with $parameterList=<(o_1,A),(o_2,B)>$, where classes *A* and *B* are defined in the class diagram of Figure 43. Using the class specifications of *A* and *B*, we can estimate the size of the message msg_1 as:

$$size(msg_1) = size(A) + size(B) = (8x(100+500)) + (8x(100+500)+8x400) = 12.8KB$$

Data Type	Description	Size
`byte`	Byte-length integer	1 Byte
`short`	Short integer	2 Bytes
`int`	Integer	4 Bytes
`long`	Long integer	8 Bytes

Table 5-Data size of some of the primitive data types in Java (adopted from [75]).

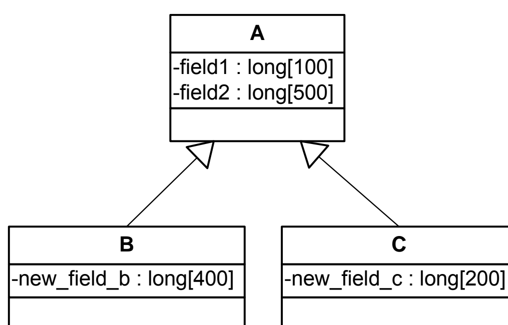


Figure 43-A class diagram showing three classes with data fields.

8.1.1 Effect of Inheritance

While estimating the data size of a class (and the messages using it), one consideration would be to take into account the inheritance relationships the class might be engaged in. This might affect the size of the messages making use of that particular class in their *parameterList* or *returnList*.

For example, suppose the method signature of a method m of a receiver object to be $m(o_1,o_2:A):A$, which basically means that two parameters of class type A are passed to the method m and an object of the type A is returned. Class A is defined in the class diagram of Figure 43. Since B and C are both sub-classes of A, therefore an object of type B or C can also be the actual parameters of the method m at runtime, which in this case will

cause the message to have difference data sizes, since classes *B* and *C* each have an extra local defined attribute. Therefore, the inheritance relationships of classes can be used to find the maximum possible data size of a class while estimating the data sizes.

8.1.2 Messages with Indeterministic Sizes

As mentioned in Section 8.1, the most data centric parts of a message (call or reply) are *parameterList* and *returnList*, respectively. In their formal representation, we assumed that these two lists are ordered sets of tuples of class types together with object values. We saw that the data sizes of such messages can be estimated using Equation 1.

We assume, in this work, having parameter and return values with classes of fixed data size. However there might also be parameters or return values that are not types of classes whose sizes can be measured precisely. For example, an input parameter of a call message might be of type, say, *String* in C++. The size of such an object might change depending on the length of the string assigned to it. As another example, suppose a call message like *store(data:BLOB)* in a distributed database system. This message is a generic example of messages sent between distributed database servers in such system, which asks the receiver of the message to save a big pile of data of type BLOB (Binary Large OBject)[5] in its own local database. Apparently, similar to the case of *String* class

---

[5] A Binary Large OBject (or BLOB) is a collection of binary data stored as a single entity in a database management system. BLOBs are typically images, audio or other multimedia objects, though sometimes binary code is stored as a BLOB [76]

type, a data object of type BLOB may have variant sizes in different situations. Therefore, Equation 1 can not be applied to estimate data size of a message in those cases.

One simple approach to estimate data size of messages having parameter or return lists with items of indeterministic data sizes is to measure sizes in a statistical fashion. Statistical distribution of the size of such messages can be derived by monitoring the message size in different runs, or by using information from *data profiles*, presented as part of an extended operational profile model [77]. Runtime monitoring techniques (such as [56]) can be utilized to monitor and derive such distributions.

8.2 Formalizing Relationships between Nodes and Networks

We saw earlier in Section 5.5 that a tree structure named NIG (Network Interconnectivity Graph) can be generated from a UML network deployment diagram, to represent the interconnection of the nodes and networks in a system. The NIG of a system is shown in Figure 44, where there are seven nodes ($n_1,\ldots,n_7$) and four networks (including system network).

To facilitate traffic usage analysis we formalize in the following sub-sections the following concepts:

- Node-network and network-network membership
- Network paths function

Wikipedia, "Definition of Binary Large OBject (BLOB)," in *http://en.wikipedia.org/wiki/Binary_large_object*, Last accessed: Feb. 2006..

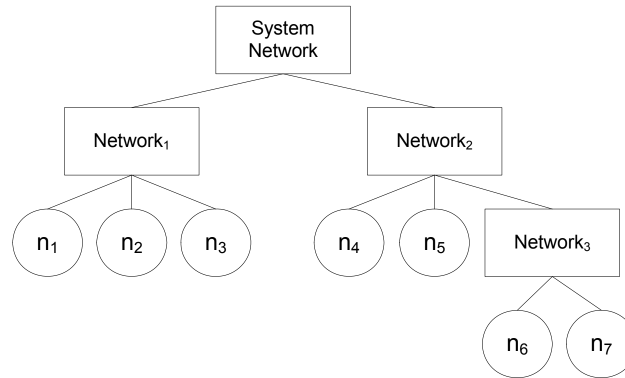


Figure 44-A Network Interconnectivity Graph (NIG).

8.2.1 Node-Network and Network-Network Memberships

To formally specify if a node is a member of a network, we can define function *member_of()* as:

$$isMemberOf(nod, net) = \begin{cases} true; \text{if there is a direct connection between } net \text{ and } nod \text{ in a NIG} \\ false; \text{otherwise} \end{cases}$$

Equation 2-Node-network membership function.

Similarly, a membership function can be defined among networks as:

$$isMemberOf(net_{sub}, net_{super}) = \begin{cases} true & \text{;if there is a direct connection between } net \text{ and } nod \text{ in a NIG with supernet/subnet association rule names} \\ false & \text{; otherwise} \end{cases}$$

Equation 3-Network-network membership function.

For example, the following relations hold in the NIG shown in Figure 44:

- $isMemberOf(n_2, Network_1) = true$
- $isMemberOf(n_3, Network_3) = false$
- $\forall n_i: isMemberOf(n_i, SystemNetwork) = true$

- $isMemberOf(n_7, Network_2)= true$
- $isMemberOf(Network_2, SystemNetwork)= true$

8.2.2 Network Paths Function

A network path function can be defined between any two nodes (the sender and the receiver of a typical distributed message) in a system. Recall from Section 5.5 that given a sender (n_s) and a receiver node (n_r), there can be several network paths between two nodes. Thus, we devise a function to calculate the ordered sets of networks, which a message sent from n_s will go through until it reaches n_r. NIG can be used to derive network paths. For example assuming the NIG of Figure 44, the network paths (only one in this case) between n_4 (as the sender node) and n_6 (as the receiver node) will be:

$$getNetworkPaths\ (n_4, n_6)=\{<Network_2, Network_3>\}$$

Recall that there can generally be several paths (routes) between two nodes in a network. Routing algorithms are usually used to maximize the network bandwidth and/or balance load in large networks by sending data through various paths between two nodes. Thus, we have considered a generalized case for this function to account for several paths between any two given nodes.

8.3 Distributed Traffic Usage Attributes

In the current resource usage analysis, we consider four attributes for distributed traffic usage:

- Traffic location: nodes vs. networks (Section 8.3.1)
- Traffic direction (for nodes only): in, out, or bidirectional (Section 8.3.2)

- Traffic type: data traffic vs. number of messages (Section 8.3.3)
- Traffic duration: instant vs. interval (Section 8.3.4)

8.3.1 Location: Nodes vs. Networks

If the intermediate network nodes (such as routers and gateways) are left out from the system software point of view, distributed traffic can essentially go through two places in a system: nodes or networks. In a typical distributed message scenario, the message is initiated from the sender node, and travels along the network path from the sender to receiver node. The network path (defined in Section 8.2.2) is made up of one or more networks in the system. Finally, the message arrives at the destination node, where it is supposed to be handled appropriately (depending on its type: call or reply). We define traffic location to be the locality of traffic flow in a system, which can be either a network or a node.

Let us consider an example. A system made of four nodes and three networks is shown in Figure 45. Topological and NIG representations of the system's network interconnectivity are shown in this figure. Nodes n_1 and n_2 are members of $Network_1$. Nodes n_3 and n_4 are members of $Network_2$. $Network_1$ and $Network_2$ are connected through *System Network*.

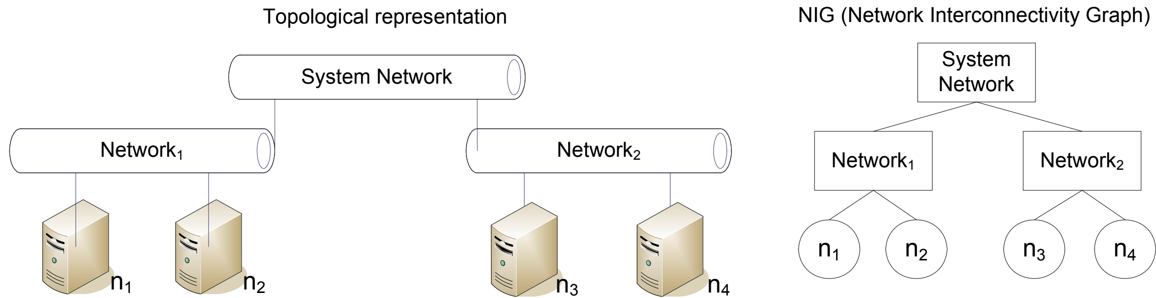


Figure 45-A system made up of four nodes and three networks.

Considering the system topology shown in Figure 45, suppose there are several processes running on each node and several SDs in the system. For simplicity, let us consider only three DCCFPs, which are extracted from SDs by the control flow analysis technique described in Chapter 6. To clarify the difference between traffic location in term of nodes or networks, the timed inter-node and inter-network representations (Section 6.5) of the three mentioned DCCFPs are shown in Figure 46-(a) and (b), respectively.

As it can be seen in Figure 46-(a), $DCCFP_1$ has two call and reply messages between nodes n_1 and n_2, which both are members of $Network_1$ (according to the NIG in Figure 45). Therefore, traffic entailed by $DCCFP_1$ only goes through $Network_1$, as shown in timed inter-network representation of $DCCFP_1$ in Figure 46-(b). $DCCFP_3$ has messages going across $Network_1$ and $Network_2$, which have to go via *SystemNetwork*. This is shown in representation of $DCCFP_3$ in Figure 46-(b), where messages with gray lines represent "implicit traffic", imposed by the original traffic imposed by the message. For example, the first call message of $DCCFP_3$ goes from $Network_1$ (time=1ms) to $Network_2$ (time=3ms) and in addition to traffics made on $Network_1$ and $Network_2$, this message puts an implicit traffic on *SystemNetwork*.

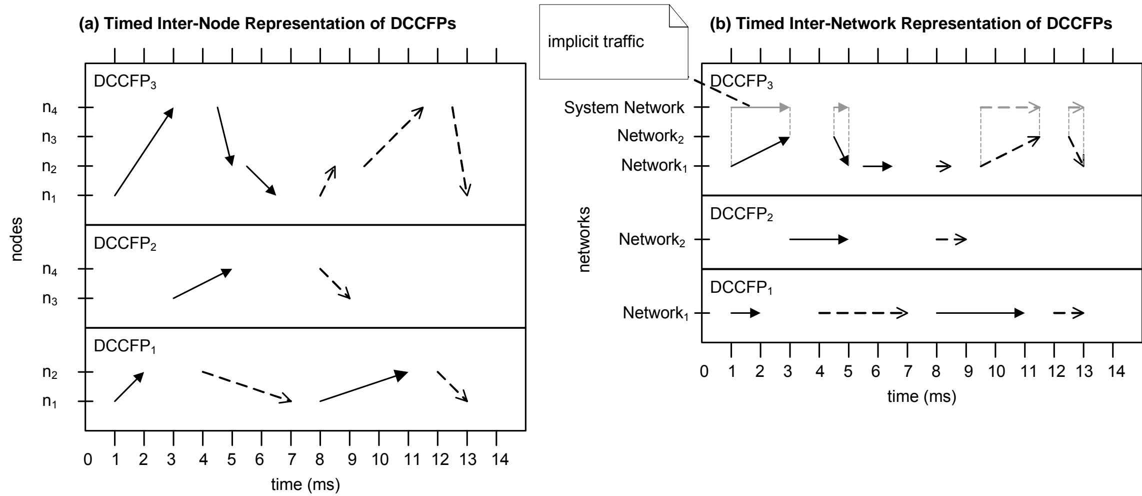


Figure 46-Timed inter-node and inter-network representations of three DCCFPs.

8.3.2 Direction (for nodes only): In, Out, Bidirectional

As discussed above, traffic location can be either a network, or a sender/receiver node. In case of a node, we can think of three traffic measurements in terms of the traffic *direction*. In our definition, traffic direction of a node can be either *in, out* or *bidirectional* form. This is due to the fact that a node is an end point of traffic in the system. Since a network in the system only relays the traffic, i.e., it transmits the traffic to other networks/nodes, we therefore only consider the bidirectional traffic for networks. For simplicity, when we talk about traffic for networks in this report, we implicitly mean the bidirectional traffic for networks.

For example, consider the timed inter-node network representation of $DCCFP_1$ in Figure 46-(a). Node n_1 sends traffic on time intervals (1-2ms) and (8-11ms) (out traffic for n_1), while it receives traffic on time intervals (4-7ms) and (12-13ms) (in traffic for n_1). n_1 is idle (not sending nor receiving any traffic) in other times.

8.3.3 Type: Amount of Data vs. Number of Network Messages

From a system-software point of view, distributed traffic has two types:

1. The amount of data, and
2. The number of distributed messages

For example, consider a simple system made up of two nodes n_A and n_B. Node n_A might rarely communicate with n_B, but when sending a message, n_A sends huge amounts of data to n_B, while n_B frequently sends queries to n_A, and gets replies. However each request from n_B to n_A and the corresponding reply has a small data size going back and forth. Therefore, it is useful to analyze and measure distributed traffic according to both types.

We discussed how to estimate the data size of a distributed message in Section 8.1. For the analysis of distributed traffic imposed by a distributed message in terms of number of messages, the analysis is straight forward and we can just count each distributed message (either call or reply) as one message over a network. To compare data traffic versus message traffic, let us consider the example in Figure 47.

To compare data versus message traffic, let us look at the control flow path $CCFP_2$ in *CCFG(M)* shown in Figure 47. Let us show the DCCFP of $CCFP_2$ as $DCCFP_2$. Note that, for simplicity, only the CCFG nodes inside *CCFG(M)* are shown for $DCCFP_2$ in Figure 47 and not those belonging to *CCFG(P)* and *CCFG(N)*. If we consider data traffic as the distributed traffic, we measure the amount of data (in bytes) sent on the network by $DCCFP_2$. In the time interval shown in the SD *M* (Figure 47), $DCCFP_2$ has one call message *m2(op)* and one reply message *rv2=m2(op)*. Call message *m2(op)* is sent from node n_1 to n_2, where the parameter of the message (*op*) can be of any data size. For

simplicity let us assume that the size of message *m2(op)* is 10 KB and the size of returned message *rv2=m2(op)* is 50 KB (these can be calculated using the method in Section 8.1). With these assumptions, we can draw a distributed traffic diagram showing data traffic for $DCCFP_2$ as shown in Figure 48. The x-axis is time in milliseconds and only the time interval shown in the SD *M* is considered.

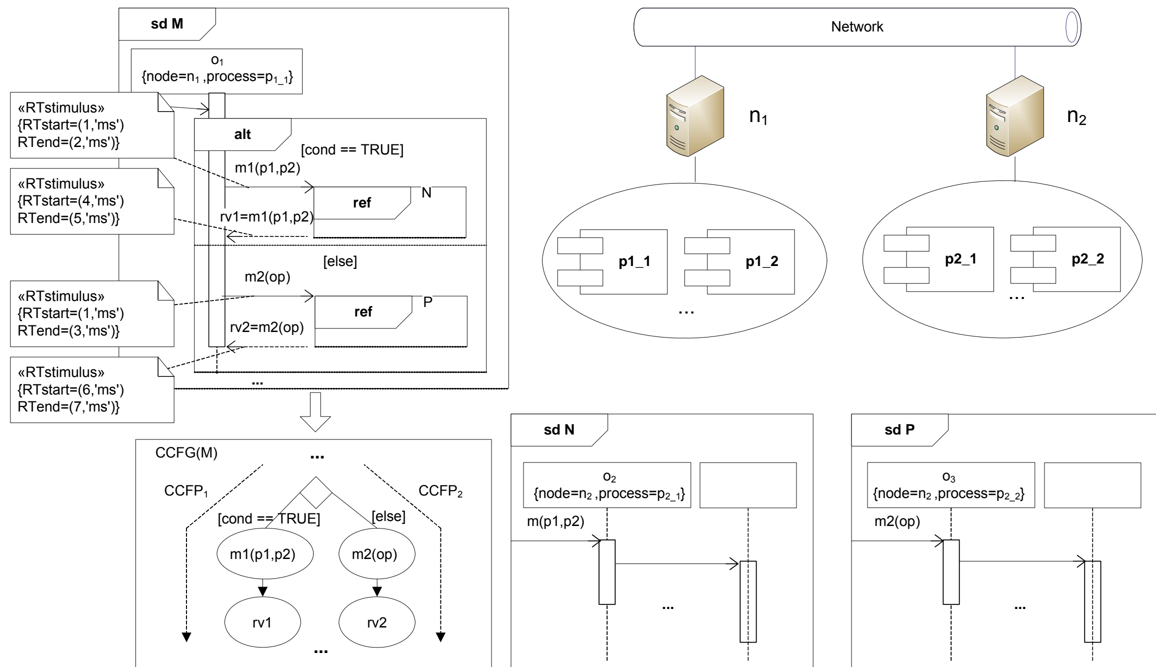


Figure 47-A typical system composed of two nodes and four processes.

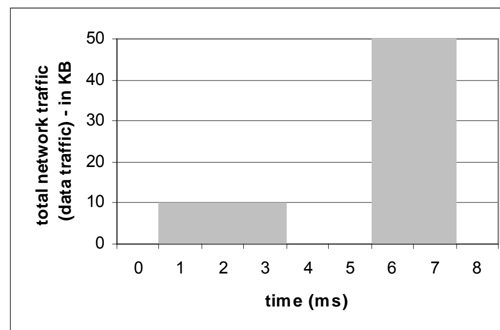


Figure 48-Network traffic diagram (data traffic) of $DCFP_2$ in Figure 47.

On the other hand, if we consider number of distributed messages as the distributed traffic, the distributed traffic diagram of $DCCFP_2$ will be as Figure 49 shows. Each call or reply message counts for one unit of distributed message in this analysis.

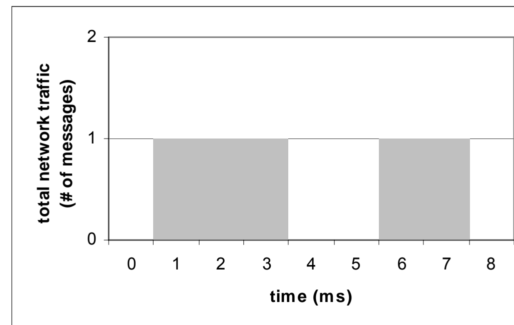


Figure 49-Network traffic diagram (number of distributed messages) of $DCFP_2$ in Figure 47.

Each of the above two distributed traffic types (amount of data vs. number of messages) can be analyzed at different levels of granularity in a system: message-level (in a SD), DCCFP-level (in a SD), SD-level, process-level, node-level, or the entire system. Different levels of granularity can be extracted from the system metamodel as shown in Figure 11. An example of such analysis is given in Section 8.4. The granularity considered in this work is message-level, unless otherwise mentioned.

8.3.4 Duration: Instant vs. Interval

In the previous sections, we analyzed distributed traffic per each time instant. When analyzing traffic, we define two types of time analyses: *instant* and *interval*. Instant traffic is the amount of traffic measured in one time instant. In a similar way, one can analyze the distributed traffic over an interval of time. We refer to this type of traffic as *interval* traffic.

We saw that a DCCFP might have different usage levels of distributed traffic in different time instants. Therefore we can add up instant duration traffic values over a given amount of time to get the traffic value over an interval. For example, data and message traffic diagrams of $DCCFP_2$ were shown in Figure 48 and Figure 49, respectively. Those diagrams show the instant traffic of $DCCFP_2$. Suppose we want to see how much data and message traffic $DCCFP_2$ imposes *during* a given interval of time, say 10 ms. Considering Figure 48, it can be said that $CCFP_2$ imposes 60 KB data traffic and two units of message traffic during the first 10 ms from its start time.

As another example, suppose the data traffic into a node *n* is to be analyzed (in-data traffic). Note that the level of granularity in this case is a node. The node under study has many processes running on it and processes communicate with other nodes in the system. A typical in-data traffic diagram of *n* can be sketched as shown in Figure 50, which is actually derived by adding all message-level traffic values for all the messages sent to *n*.

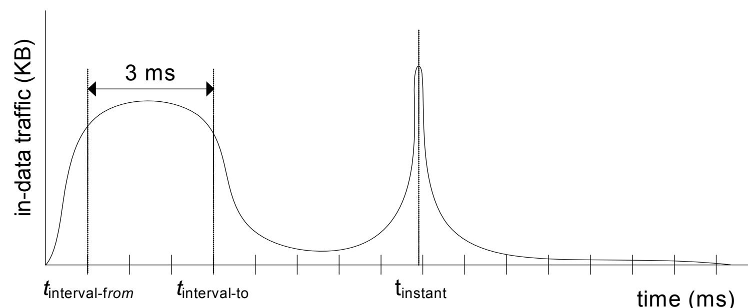


Figure 50-"In-data" traffic diagram of a node, highlighting difference between *instant* and *interval* (3ms) traffic.

According to Figure 50, if one wants to find the time when maximum instant traffic happens in *n*, the answer would be time= $t_{instant}$. However, if the question is to find an

interval of time (say 3 ms) when the maximum interval traffic happen in *n*, then the answer would be ($t_{interval\text{-}from}$, $t_{interval\text{-}to}$).

8.4 Aspects to Consider when Estimating Network Traffic Usage

We consider the following aspects when estimating network traffic usage and discuss their effects on the formalized traffic functions (Section 8.5) in Section 8.5.

- Effects of multiple network paths between nodes (Section 8.4.1)
- Delay in network transmissions (Section 8.4.2)
- Effects of concurrent processes on distributed traffic (Section 8.4.4)

8.4.1 Effects of Multiple Network Paths between Nodes

Recall from Section 8.2.2 that there can be in general several network paths between a pair of nodes. Thus, in general, the data sent from a node to another is actually divided into several parts and is transmitted through several paths. Such a dispatching is handled by networking components of a system (e.g., routers and network protocols). Consider the example Network Interconnectivity Graph (NIG) in Figure 51, where there are three network paths from *srcNode* to *destNode*: $\{<net_a, net_b>, <net_c, net_d, net_e>, <net_c, net_f, net_g>\}$. In the current work, for simplification, we assume that the data sent from a source node (e.g., *srcNod*) to a destination (e.g., *destNode*) is divided into *equal* parts and is transmitted through all the paths between the two nodes. In other words, we assume that the networks of a SUT are not adaptive (do not have intelligent load balancing features). For example, assuming that 300 KB of data is going to be sent from *srcNode* to *destNode* in the NIG in Figure 51, we assume that the data is divided into three equal 100 KB

pieces (by network components) and each of the pieces will be sent via one of the three paths between the two nodes. Considering networks net_a and net_c in this NIG, we can thus conclude that 1/3 and 2/3 of the data are transmitted through each of these networks, respectively.

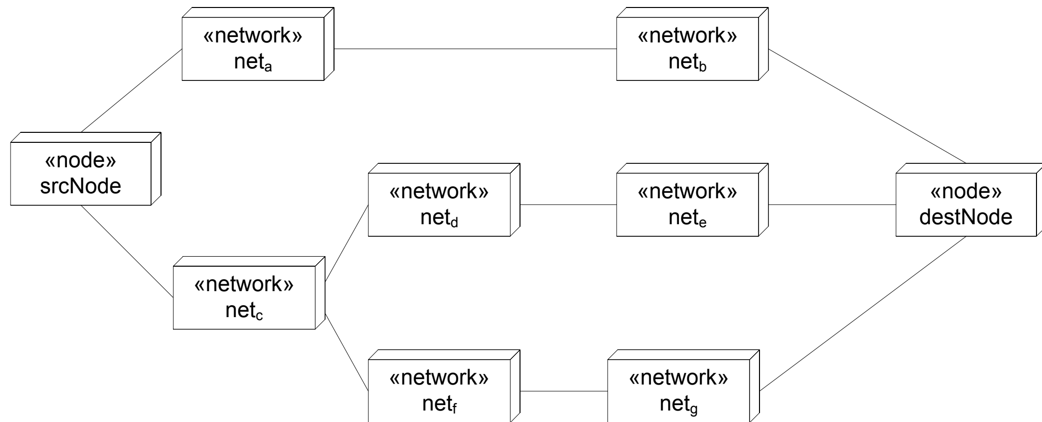


Figure 51-Example Network Interconnectivity Graph (NIG).

Furthermore, we assume that the network paths' dispatching policy does not change during the transmission of a message, i.e., the transmission shares of each of the involved networks stay the same during the entire transmission. For example, the share of net_a and net_c in the above example stay at same ratios of 1/3 and 2/3 during the transmission of a message. In real complex DRTS, adaptive dispatching policies are usually used to balance load in each time instant. For example, assuming that such a policy is in place for the NIG in Figure 51, most of the data of a large message might be transmitted via the net_a net_b path in a first interval of the message transmission, while the policy routes the traffic via the net_c net_d net_e path in the last interval of the transmission. To relax such a limitation, more information from the routing and network protocols involved in a SUT should be provided such that we can calculate the transmission shares of each

network/path in each time instant. This would help to derive more precise stress test requirements.

Therefore, we devise a function called *netTransmissionShare* (Equation 4) based on the above simplifications to calculate such shares. If the given network is a member of at least one path between the two nodes, the function returns the ratio of the number of paths between two nodes in which the network is a member of, to the total number of paths between two nodes. Otherwise, the function returns 0.

$$netTransmissionShare(network, sourceNode, destinationNode) = \begin{cases} |paths| / |allPaths| & \begin{array}{l} ; network \in getNetworkPaths(sourceNode, destinationNode) \wedge \\ paths = \{ path \in getNetworkPaths(sourceNode, destinationNode) \mid network \in path \} \wedge \\ allPaths = getNetworkPaths(sourceNode, destinationNode) \end{array} \\ 0 & ; network \notin getNetworkPaths(sourceNode, destinationNode) \end{cases}$$

Equation 4-A function to calculate the shares of a network in data transmissions between two nodes.

To avoid the above simplification (equal shares for all network paths between two nodes), the *netTransmissionShare* function can be modified in future works to incorporate more realistic cases in adaptive networks and networks with intelligent load balancing. For example, in real-world applications, more traffic is usually sent through networks with higher bandwidths. The *netTransmissionShare* function is used in the network traffic usage formulas in the rest of this article to calculate the amount of traffic on a specific network. This will enable our test methodology to have a rough estimate of anticipated traffic on a network, and thus, to find test requirements, which if executed, will lead to high traffic (stress).

8.4.2 Delay in Network Transmissions

There are usually inevitable delays in network transmissions [78]. For example, after a message is sent to a network to be transmitted, it usually takes a short time for the network to start the actual transmission of data. In the communication networks community, such delays are usually analyzed and measured using probability distributions [79], e.g., normal or gamma distributions. We thus consider such delays and their corresponding probability distributions as a source of imprecision in our quantitative analysis of network traffic usage in this chapter and test requirements derivation process in Chapter 9, which will be based on the quantitative analyses in the current chapter.

Assuming that transmission delay probability distributions have been analyzed and calculated for a SUT, let us discuss how such information can be modeled in a Network Deployment Diagram (NDD). As modeling such information is not mentioned in the UML 2.0 specification, we define a tagged-value called *transmissionDelay* which can be annotated to networks in a NDD. This tagged-value takes values of type *RTtimeValue*, defined in the UML-SPT profile [12]. Two of such value types are probability distributions (discussed in more detail in Section 10.1) and exact timing values (e.g., 2 ms). For example, the NDD in Figure 52 is annotated with network transmission delay information using the *transmissionDelay* tagged-value: networks net_a, net_b and net_c have been specified as having delays with normal distributions characterized by two parameters. For networks with no *transmissionDelay* tagged-values, we assume that transmission delays are 0. In this example, when a message is sent from *srcNode* to *destNode* through the network path $net_a net_c net_d$, the first network packet of the message

starts to be delivered to *destNode*, after 25 (15+10) ms on average, due to delays in two of the involved networks.

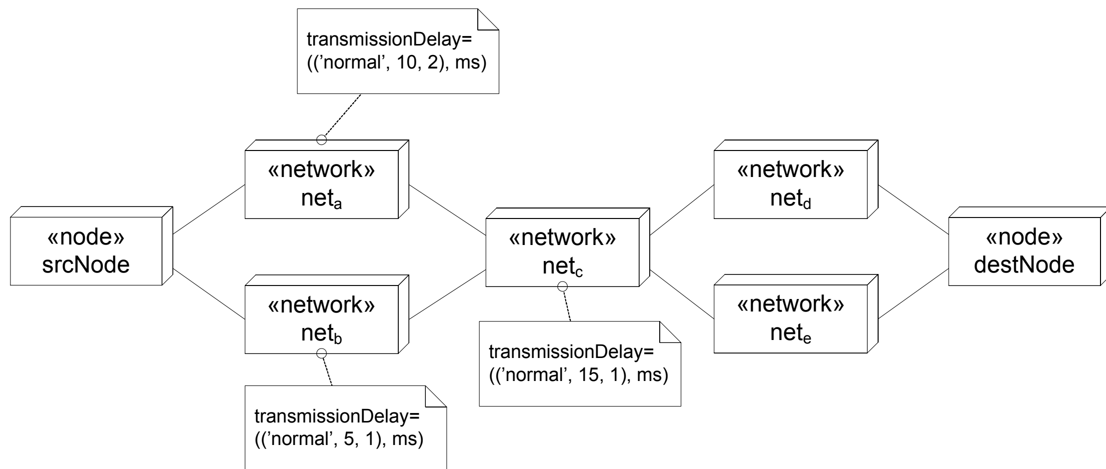


Figure 52-A Network Deployment Diagram (NDD) annotted with network transmission delay information.

Therefore, we define a function called *expNetworkDelay* (Equation 5) based on the above discussions to calculate the *expected cumulative* transmission delay from a source node to a network (or a destination node). Recall from Section 8.4.1 that we assumed multiple network paths between two nodes (or a node and a network). Thus, we estimate the cumulative transmission delay between a source and a destination to be the average of cumulative transmission delays of all paths between them. The *cumulative* transmission delay of a path is calculated by adding the delays of all the networks in the path. Furthermore, since delays are usually measured by probability distributions, we thus use the expected values of such distributions. *EDV* in Equation 5 stands for *Expected Delay Value*, and *EDV(net)* returns the expected value of the transmission delay probability distributions of network *net*.

$$exp\ NetworkDelay(srcNode, dest) = \frac{1}{|P|} \sum_{\forall path \in P} \left(\sum_{\forall net \in path} EDV(net) \right)$$

P = Set of paths from *srcNode* to *dest*

Equation 5-A function to calculate the data transmissions delay of network between a source node and a network (or a destination node).

Note the averaging all the cumulative transmission delays for all paths between a source and a destination in a NDD is a simplification of real-life systems.

To avoid this simplification, the *expNetworkDelay* function (the *1/|P|* expression, in specific) can be modified in future works to incorporate different weighting mechanism for adding up the cumulative delays for all paths between a source and a destination in a NDD, e.g., some paths might be rarely used, thus they should have less impact on the cumulative network delay.

8.4.3 Traffic Distribution of Messages with Durations more than a Time Unit

When the durations of a (data-intensive) distributed message is more than a time unit, analyzing the network traffic entailed by transmission of the message can be complicated. Different mathematical traffic distribution models for large data messages have been studied in the computer and communication networks literature (e.g., [80, 81]). Such models take into account the different physical properties of the deployed network of a SUT and investigate statistically the traffic flow of large messages across the network.

The simplest traffic model studied in the literature is the one with uniform distribution, in which it is assumed that equal portions of a large message will be transmitted in each time instant. For example, consider a distributed message *msg* with data size of 20 MB whose execution time duration (end time – start time) is 10 time units (say 4 s). By using

the uniform distribution as the traffic model, we can estimate that 5 MB portions of the message will be transmitted in each second. It has been shown (e.g., [81]) that uniform distribution can be used as an acceptable approximate traffic model in many applications. Thus, for the sake of simplicity, we choose such a model in this thesis and will use it in the design of our traffic usage analysis functions in Section 8.5.

8.4.4 Effect of Concurrent Processes

According to the SUT model in Figure 11, several processes can run concurrently on a single node. Each of the processes might be in the process of running a method. Therefore, the distributed traffic caused by the node will be the sum of the traffics by all its concurrent processes. For example, the data traffic diagram of a node with two processes *Process*$_1$ and *Process*$_2$ over an interval of 10 milliseconds is shown in Figure 53. It is evident that a node's total traffic in a single time instant is the sum of the traffic caused by each of its processes.

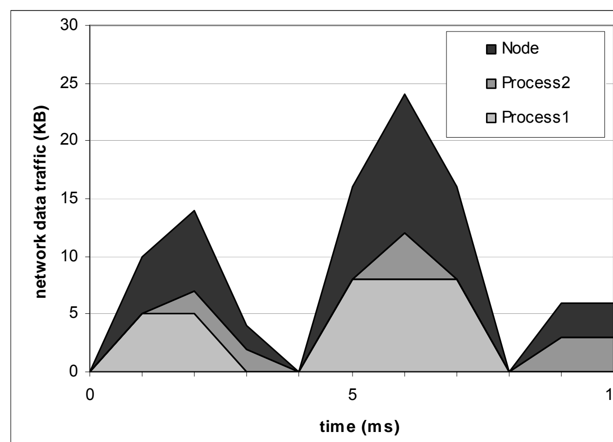


Figure 53-The data traffic diagram of a node with two processes.

8.5 A Class of Traffic Usage Analysis Functions

As discussed in Chapter 6, each SD can have several DCCFPs, where each DCCFP is a path in a SD's CCFG and includes only distributed call and reply messages. Different attributes of distributed traffic were also discussed in Section 8.3 which included: location, direction, type and duration.

In this section, a class of functions is proposed to measure distributed traffic entailed by DCCFPs. The functions aim to take into account the traffic attributes mentioned earlier. First, the naming convention of the functions is given in Section 8.5.1. Formal definitions of the functions are then proposed in Section 8.5.2 along with some examples on how the function values can be calculated.

8.5.1 Naming Convention

A tree structure denoting the traffic functions' naming convention and their input parameters is shown in Figure 54.

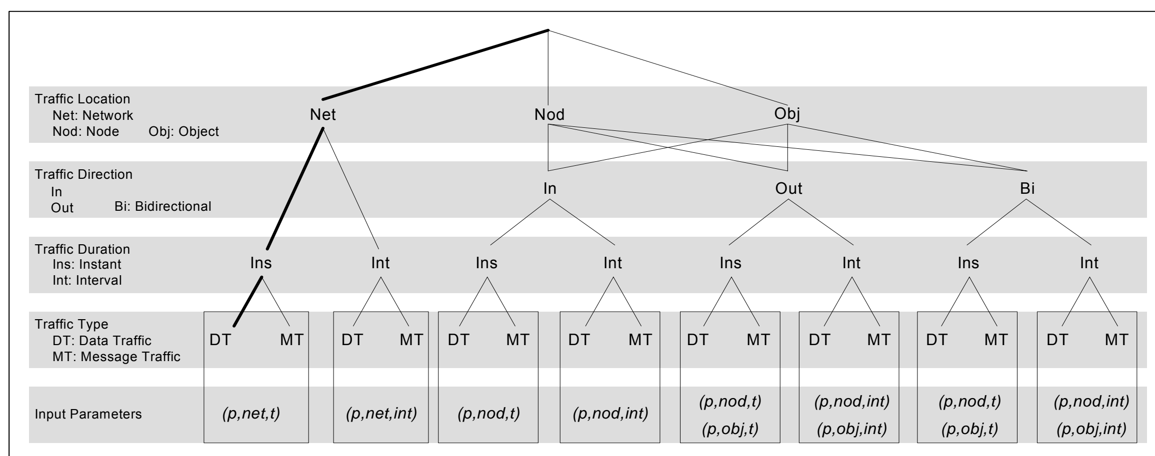


Figure 54-Naming convention for the traffic usage functions.

The root node of the tree has a null label. A function name is determined by traversing from the root to a leaf node and concatenating all the node labels in order. The top four layers of the tree in Figure 54 correspond to the four query attributes discussed in Section 8.3. By counting the number of paths from the root node of the tree to leaf nodes, we would get 28 paths (4 for networks, and 12 for node and object categories each). This means that there will be 28 different traffic functions to be formalized.

The bottom layer in Figure 54 specifics the input parameters of a traffic function with the name made by traversing from the root to a leaf node. For example, consider the path specified by the bold line in Figure 54. This path represents function *NetInsDT*. According to the bottom layer, the input parameters of this function would be (ρ, *net, t*). This function returns the instant (*Ins*) data traffic value (*DT*) of a given DTCCFP (ρ) for a given network (*net*) at a given time (*t*). Input parameters with *int* in the bottom layer in Figure 54 correspond to the functions with interval duration. The start and end times of an interval, i.e., *int=(start, end)*, should be provided for such functions. For functions with node or object traffic location, the input parameters include either *nod* or *obj* as traffic location, respectively.

More detailed descriptions of the functions are given next. Functions with network, node and object traffic location are described in Sections 8.5.2.1, 8.5.2.2 and 8.5.2.3, respectively.

8.5.2 Functions

In this section, traffic functions are listed using the naming convention given in Figure 54. The functions are grouped according to the top layer (traffic location) of the tree in

Figure 54. Mathematical formulas to calculate some traffic functions are provided below. Other traffic functions can be derived in similar fashion. In the following mathematical formulas, for brevity, *msg.start* and *msg.end* stand for *msg.startTime* and *msg.endTime*. *msg.s.n* and *msg.r.n* stand for *msg.sender.node* and *msg.recevier.node*.

8.5.2.1 Traffic Location: Network

1. *NetInsDT*: returns the value of instant data traffic, a given DCCFP entails on a given network from a given time instant *t*.

$$NetInsDT(\rho, net, t) = \begin{cases} \sum_{msg_i} \begin{array}{l} netTrnasmissionShare(\\ net, msg_i.sender.node, msg_i.receiver.node). \\ size(msg_i)/dur(msg_i) \end{array} & \begin{array}{l} ;\exists msg_i \mid msg_i \in \rho \wedge \\ msg_i.start + expNetworkDelay(\\ net, msg_i.sender.node) \leq t \leq msg_i.end \wedge \\ net \in getNetworkPaths(\\ msg_i.sender.node, msg_i.receiver.node) \end{array} \\ 0 & ;\text{otherwise} \end{cases}$$

where *size()* returns the size of a message in bytes as described in Section 8.1. *dur()* returns the time duration of a message which can be calculated as: *dur(m)=m.startTime–m.endTime*. Since a message can span over several time units, our definition for the data traffic value of a message at a time unit is its total data size divided by its duration, which will give the message's traffic per time unit. Function *netTransmissionShare* (Equation 4 in Section 8.4.1) returns a coefficient which denotes the share of the network *net* in data transmissions between the sender and receiver objects of a message. Function *expNetworkDelay* (Equation 5 in Section 8.4.2) returns the delay time distance between net and the sender of each message. This function is used to account for transmission delays in network paths in our quantitative analysis of network traffic usage here and the test requirements derivation process in Chapter 9.

2. *NetInsMT*: returns the value of instant message traffic, a given DCCFP entails on a given network at a given time instant.

$$NetInsMT(\rho,net,t)=\begin{cases}\sum_{msg_i} netTrnasmissionShare(net,msg_i.sender.node,msg_i.receiver.node) & ;\exists msg_i \mid msg_i \in \rho \wedge msg_i.start + exp\,NetworkDelay(net,msg_i.sender.node) \le t \le msg_i.end \wedge net \in getNetworkPath(msg_i.sender.node,msg_i.receiver.node) \\ 0 & ;\text{otherwise}\end{cases}$$

3. *NetIntDT*: returns the value of interval data traffic, a given DCCFP entails on a given network during a given time interval. *NetIntDT* can be calculated using *NetInsDT*.

$$NetIntDT(\rho,net,int)=\sum\nolimits_{t=int.start}^{t=int.end} NetInsDT(\rho,net,t)$$

4. *NetIntMT*: returns the value of interval message traffic, a given DCCFP entails on a given network during a given time interval.

$$NetIntMT(\rho,net,int)=\sum\nolimits_{t=int.start}^{t=int.end} NetInsMT(\rho,net,t)$$

8.5.2.2 Traffic Location: Node

1. *NodInInsDT*: returns the value of instant data traffic, a given node receives by running a given DCCFP at a given time instant. “In” denotes that the traffic direction is towards the node as explained in Section 8.3.2.

$$NodInInsDT(\rho,nod,t)=\begin{cases}\sum_{msg_i} size(msg_i)/dur(msg_i) & ;\exists msg_i \mid msg_i \in \rho \wedge msg_i.start \le t \le msg_i.end \wedge msg_i.receiver.node = nod \\ 0 & ;\text{otherwise}\end{cases}$$

2. *NodInInsMT*: returns the value of instant message traffic, a given node receives by running a given DCCFP at a given time instant.

$$NodInInsMT(\rho, nod, t) = \begin{cases} |msg_i| & ;\exists msg_i \mid msg_i \in \rho \wedge \\ & msg_i.start \le t \le msg_i.end \wedge \\ & msg_i.receiver.node = nod \\ 0 & ;\text{otherwise} \end{cases}$$

3. *NodInIntDT*: returns the value of interval data traffic, a given node receives by running a given DCCFP during a given time interval.

$$NodInIntDT(\rho, nod, int) = \sum_{t=int.start}^{t=int.end} NodInInsDT(\rho, nod, t)$$

4. *NodInIntMT*: returns the value of interval message traffic, a given node receives by running a given DCCFP during a given time interval.

$$NodInIntMT(\rho, nod, int) = \sum_{t=int.start}^{t=int.end} NodInInsMT(\rho, nod, t)$$

5. *NodOutInsDT*: returns the value of instant data traffic, a given node sends by running a given DCCFP at a given time instant. “Out” denotes that the traffic direction is from the node as explained in Section 8.3.2.

$$NodOutInsDT(\rho, nod, t) = \begin{cases} \sum_{msg_i} size(msg_i)/dur(msg_i) & ;\exists msg_i \mid msg_i \in \rho \wedge \\ & msg_i.start \le t \le msg_i.end \wedge \\ & msg_i.sender.node = nod \\ 0 & ;\text{otherwise} \end{cases}$$

6. *NodOutInsMT*: returns the value of instant message traffic, a given node sends by running a given DCCFP at a given time instant.

$$NodOutInsMT(\rho, nod, t) = \begin{cases} |msg_i| & ;\exists msg_i \mid msg_i \in \rho \wedge \\ & msg_i.start \le t \le msg_i.end \wedge \\ & msg_i.sender.node = nod \\ 0 & ;\text{otherwise} \end{cases}$$

7. *NodOutIntDT*: returns the value of interval data traffic, a given node sends by running a given DCCFP during a given time interval.

$$NodOutIntDT(\rho, nod, int) = \sum_{t=int.start}^{t=int.end} NodOutInsDT(\rho, nod, t)$$

8. *NodOutIntMT*: returns the value of interval message traffic, a given node sends by running a given DCCFP during a given time interval.

$$NodOutIntMT(\rho, nod, int) = \sum_{t=int.start}^{t=int.end} NodOutInsMT(\rho, nod, t)$$

9. *NodBiInsDT*: returns the value of instant data traffic, a given node "sends or receives" by running a given DCCFP at a given time instant.

$$NodBiInsDT(\rho, nod, t) = \begin{cases} \sum_{msg_i} size(msg_i)/dur(msg_i) & ;\exists msg_i \mid msg_i \in \rho \wedge \\ & msg_i.start \le t \le msg_i.end \wedge \\ & (msg_i.sender.node = nod \vee msg_i.receiver.node = nod) \\ 0 & ;\text{otherwise} \end{cases}$$

10. *NodBiInsMT*: returns the value of instant message traffic, a given node "sends or receives" by running a given DCCFP at a given time instant.

$$NodBiInsMT(\rho, nod, t) = \begin{cases} |msg_i| & ;\exists msg_i \mid msg_i \in \rho \wedge \\ & msg_i.start \le t \le msg_i.end \wedge \\ & (msg_i.sender.node = nod \vee msg_i.receiver.node = nod) \\ 0 & ;\text{otherwise} \end{cases}$$

11. *NodBiIntDT*: returns the value of interval data traffic, a given node "sends or receives" by running a given DCCFP during a given time interval.

$$NodBiIntDT(\rho, nod, int) = \sum_{t=int.start}^{t=int.end} NodBiInsDT(\rho, nod, t)$$

12. *NodBiIntMT*: returns the value of interval message traffic, a given node "sends or receives" by running a given DCCFP during a given time interval.

$$NodBiIntMT(\rho, nod, int) = \sum_{t=int.start}^{t=int.end} NodBiInsMT(\rho, nod, t)$$

8.5.2.3 Traffic Location: Object

We only present the *ObjInInsDT* and *ObjInInsMT* functions next. The other functions for the object traffic location (*ObjInIntDT, ObjInIntMT, ObjOutInsDT, ObjOutInsMT, ObjOutIntDT, ObjOutIntMT, ObjBiInsDT, ObjBiInsMT, ObjBiIntDT,* and *ObjBiIntMT*) can be derived similar to the functions of the node traffic location.

1. *ObjInInsDT*: returns the value of instant data traffic, a given object receives by running a given DCCFP at a given time instant.

$$ObjInInsDT(\rho, obj, t) = \begin{cases} \sum_{msg_i} size(msg_i)/dur(msg_i) & ;\exists msg_i \mid msg_i \in \rho \wedge \\ & msg_i.start \le t \le msg_i.end \wedge \\ & msg_i.receiver.object = obj \\ 0 & ;\text{otherwise} \end{cases}$$

   where *r* and *o* are shorthand notations for receiver node and object fields of a message.

2. *ObjInInsMT*: returns the value of instant message traffic, a given object receives by running a given DCCFP at a given time instant.

$$ObjInInsMT(\rho, obj, t) = \begin{cases} |\, msg_i \,| & ;\exists msg_i \mid msg_i \in \rho \wedge \\ & msg_i.start \le t \le msg_i.end \wedge \\ & msg_i.receiver.object = obj \\ 0 & ;\text{otherwise} \end{cases}$$

An Example

An example is given here to show how a distributed traffic function can be calculated. Let a DCCFP ρ=<CM_1,CM_2,RM_1,RM_2> and the messages of ρ are as the following:

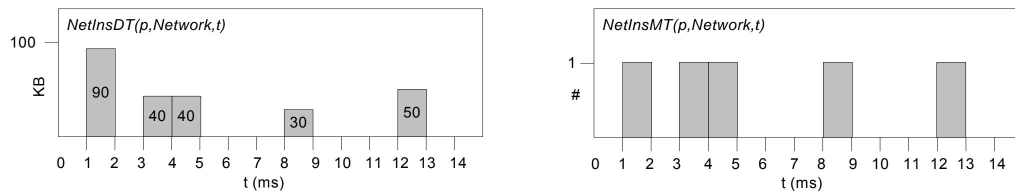
CM_1=((o_1,O_1,n_1),(o_2,O_2,n_2),t,<(p1:-),(p2:-)>,1,2)

CM_2=((o_2,O_2,n_2),(o_3,O_3,n_3),u,<(p3:-),(p4:-)>,3,5)

RM_1=((o_3,O_3,n_3),(o_2,O_2,n_2),<(x=u(-),-)>,8,9)

$$RM_2=((-,-,-,N),(o_1,O_1,n_1),<(y=t(-),-)>,12,13)$$

Let us suppose a SUT's NIG to be the one shown in Figure 36. Also suppose that the sizes of the four messages of DTCCFP ρ have been calculated using the RUF in Equation 1 and are 90 (CM_1), 80 (CM_2), 30 (RM_1), and 50 (RM_2) kilobytes. Using the above information, the following usage functions can be calculated.



$$\begin{aligned}
&NetIntDT(\rho, Network, (2,9)) \\
&= \sum_{t=2}^{8} NetInsDT(\rho, Network, t) \\
&= NetInsDT(\rho, Network, 2) + \cdots + NetInsDT(\rho, Network, 8) \\
&= 0 + 40 + 40 + \cdots + 30 = 110KB
\end{aligned}$$

Chapter 9

TIME-SHIFTING STRESS TEST TECHNIQUE

This section describes the excerpts from our first, simple heuristic-based stress test technique to stress test distributed traffic. The technique, referred to as *Time-Shifting Stress Test Technique*, is an optimization technique which is based on shifting DCCFPs along the time axis to find the time instant when maximum possible stress can occur.

The chapter is structured as follows. The problem statement is revisited in Section 9.1, where we express the problem using the formalism given in Chapter 5 to Chapter 8. Note that an initial problem statement was given in Section 2.2, where it was discussed in a general form, without prior knowledge of the modeling and formalism proposed in Chapter 5 to Chapter 8. Section 9.2 presents the heuristic of our stress test strategy. An example is presented in Section 9.3 to illustrate the heuristic.

Since we believe that GASTT is more interesting in terms of approach than TSSTT, due to space constraints, we do not report the details of TSSTT in this thesis. Interested readers can refer to [82] for extensive discussion on TSSTT. A shorter version of how

TSSTT works is also reported in [83]. We present in Section 9.4 only excerpts from TSSTT.

9.1 Problem Statement: Revisited

Having formalized the input and test model needs for our stress test technique in Chapter 5-Chapter 8, we re-state the problem in a more precise manner as follows:

> Suppose the UML 2.0 model of a distributed SUT is given. As we discussed earlier, the model should include at least the SUT's sequence diagrams, class diagrams, the network deployment diagram(s) showing interconnectivity and the network hierarchy of the system, and inter-SD constraints are specified using a MIOD. Suppose the CFA of system's SDs is done using the techniques in Chapter 6, that Inter-SD constraints are analyzed according to the techniques in Chapter 7, that SUT's Independent-SD Sets (ISDSs) and Set of SD Sequences (SSDS) are derived, and that the distributed traffic of the system is formalized as stated in Chapter 8. The problem is to automatically find and schedule a subset of system DCCFPs which will put a given set of networks or nodes under stress according to a given stress test strategy (defined in terms of location, direction, type or duration) in order to maximize the chance of exhibiting distributed traffic faults.

9.2 Stress Test Heuristic

Given a specific test objective for stress testing, for instance the data traffic over a specific network in the SUT, our heuristic is to first identify, for each DCCFP of SDs, a message (or a set of messages) which imposes maximum traffic. Let us refer to such messages as *maximum stressing messages*. Intuitively, if in a given SD's DCCFP, none of

the messages match the test objective (e.g., messages do not involve the selected network to be stress tested), then the SD's DCCFPs is discarded. If all the DCCFPs of a SD are discarded, then the SD is discarded. As a second step, using the start times of the maximum stress messages selected in each DCCFP, the selected DCCFPs are scheduled in a way that maximizes stress for the test objective. In the example above, selected DCCFPs are scheduled such that the maximum stress messages happen at the same time, thus resulting in maximum data traffic over the network under stress test.

A stress test requirement set will be generated by our technique. Assuming that the SUT has n SDs (SD_1, …, SD_n), a test requirement set will be a schedule of a selected set of SDs' DCCFPs and is an ordered set in the form of:

$$<(\rho_{1max}, \alpha\rho_{1max}), \ldots, (\rho_{nmax}, \alpha\rho_{nmax})>$$

where each scheduled DCCFP is represented as a tuple ($\rho_{imax}, \alpha\rho_{imax}$). ρ_{imax} is the DCCFP in the DCCFP set of SD_i, that maximizes the test objective. $\alpha\rho_{imax}$ is the start time of DCCFP ρ_{imax} in the schedule. Our test heuristic is illustrated next on a simple example.

9.3 An Example to Illustrate the Heuristic

Suppose a typical SUT whose NIG is shown in Figure 55: nodes n_1, n_2 and n_3 can communicate over a network (*SystemNetwork*).

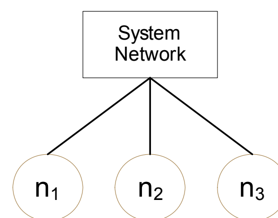


Figure 55-A simple system NIG.

For simplicity, let us assume that the CFA of the system's SDs yielded four DCCFPs ($DCCFP_1$,...,$DCCFP_4$). The timed inter-node representations (described in Section 6.5) of those DCCFPs are shown in Figure 56-(a), where each DCCFP includes several distributed messages. For example, among $DCCFP_1$'s messages, there is a call message starting in time t=1ms from node n_2 to node n_3 which lasts until time t=4ms and a return message from n_2 to n_1 from time=9 to time=10.

Let us further suppose that we want to derive test requirements for *network instant data traffic (NetInstDT)* stress for network *SystemNetwork*: i.e., we want to schedule the execution of DCCFPs and find a time instant when we can maximize data traffic over *SystemNetwork*. To visualize the data traffic incurred by $DCCFP_1$,...,$DCCFP_4$ over the network *SystemNetwork*, $NetInsDT(DCCFP_i,SystemNetwork,t)$ is depicted in Figure 56-(b) for each DCCFP. Again for simplicity, the calculation steps of those functions are not shown.

The next step of the heuristic is to find the *maximum stress messages* of each DCCFP. This is shown graphically in Figure 56-(b) by dashed lines around such messages. Recall that the criterion to find these messages is that the target of the stress test, in our case *SystemNetwork*, must be part of the path in a message's sender to receiver. Furthermore, the size of such messages has the maximum value among all messages of a DCCFP. In our example, since all nodes (n_1, ..., n_3) are members of the system network, therefore all distributed messages go through this network.

After finding maximum stress messages in each DCCFP, the next (and final) step in the derivation of stress test requirements is to use the start times of the maximum stress messages we have identified in each DCCFP to schedule the DCCFPs such that these

maximum stress messages all run concurrently. This is illustrated in Figure 56-(c). As shown, $DCCFP_1$, $DCCFP_2$, $DCCFP_3$, and $DCCFP_4$ will be scheduled to start running at times 0, 8, 6 and 9 (milliseconds) respectively. With this schedule, the highest data traffic stress in the system network will occur at time=9ms from the start of the test execution.

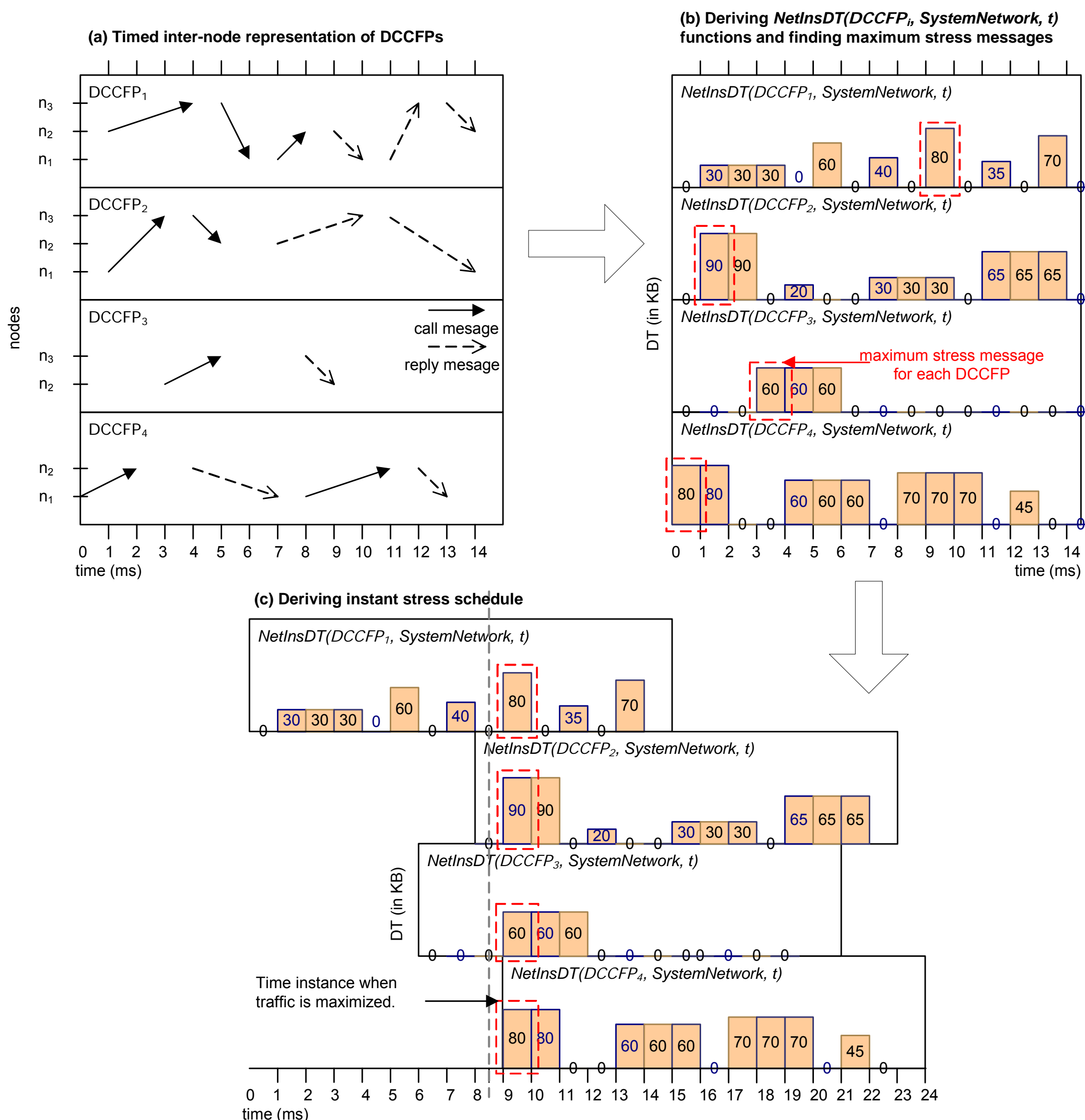


Figure 56-Heuristic to stress test instant data traffic on a network.

9.4 Excerpts

Since we believe that GASTT is more interesting in terms of approach than TSSTT, due to space constraints, we do not report the details of TSSTT in this thesis. Interested readers can refer to [82] for extensive discussion on TSSTT. A shorter version of how TSSTT works is also reported in [83]. We present next only excerpts from TSSTT.

One of TSSTT strategies is referred to as *StressNetInsDT(net)*, which derives stress test requirements for stress testing a *network* in an *instant* with *data traffic* type. To better understand this stress test strategy, the UML activity diagram in Figure 57 depicts the flow of activities to generate stress test requirements of *StressNetInsDT(net)*.

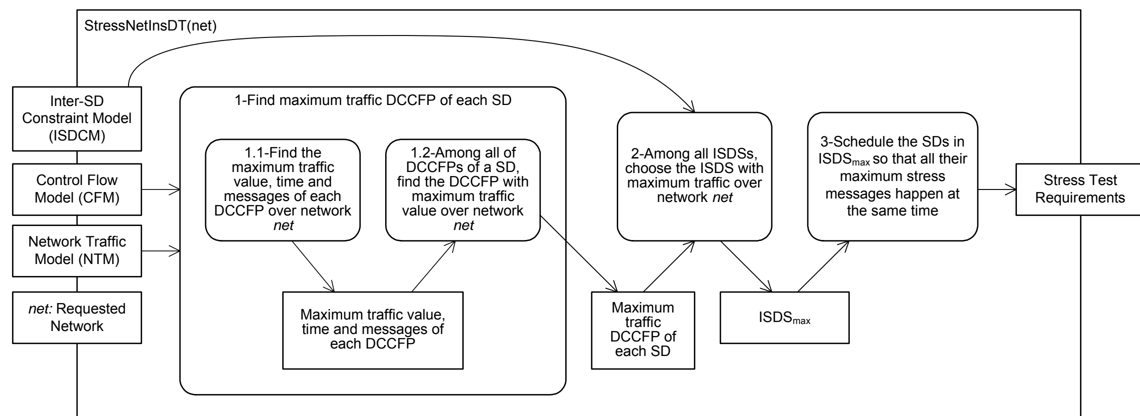


Figure 57-Activity diagram of stress test strategy *StressNetInsDT(net).*

To better clarify the idea, let us recall that a MIOD is composed of several ISDSs, an ISDS is a set of several SDs, a SD has several DCCFPs, and each DCCFP is composed of several messages.

To generate stress test requirements, the algorithm (activity 1.1 of Figure 57) first finds the maximum traffic messages of each DCCFP. Using the maximum traffic message, the maximum traffic DCCFP of each SD is then chosen (activity 1.2). Then we can

determine the maximum traffic of each ISDS (summing the maximum traffic for each selected DCCFPs of each SD in the ISDS). Then, among all ISDSs of a MIOD the ISDS with maximum traffic is chosen (activity 2 of Figure 57).

Chapter 10

GENETIC ALGORITHM-BASED STRESS TEST TECHNIQUE

As discussed in Section 5.3, we consider three types of SD constraints in the current work:

- Sequential constraints: Constraints which define a set of valid SD sequences.
- Conditional constraints: Conditional constraints are related to sequential constraints and indicate the condition(s) that have to be satisfied before a sequence of SDs can be executed. They also define valid SD sequences
- Arrival-pattern constraints*:* These constraints relate to timing of SDs, that is when a SD can start running. Considering each SD alone, it might only be allowed to be executed in some particular time instants.

Our approach in considering the above set of constraints when generating stress test requirements was as follows. We proposed a test requirement generation technique, as an optimization problem, in Chapter 9 which took into account the first two types of

constraints (sequential and conditional). The test technique was referred to as *Time-Shifting Stress Test Technique* (*TSSTT*). A more complex optimization algorithm, based on genetic algorithms, will be presented in this chapter which will consider *all* three types of constraints (sequential, conditional and arrival-pattern). The ideas of the optimization algorithm in this section, referred to as *Genetic Algorithm-based Stress Test Technique*, are built on the main concepts of the TSSTT.

We first discuss in Section 10.1 the types of arrival patterns presented by the UML-SPT profile and that we consider in this chapter. In order to study the arrival patterns and their impact on our test techniques, the timing characteristics of arrival patterns are analyzed in Section 10.2. Along with such timing characteristics, the concept of *Accepted Time Sets* is introduced in Section 10.3. Section 10.4 formulates the problem as an optimization problem, which accounts for arrival-pattern constraints. Section 10.5 describes the impacts of arrival patterns on various stress test strategies (Section 9.4).

Based on such impacts, instant and interval stress test strategies with arrival patterns are addressed separately. The derivation of instant stress test requirements while considering arrival patterns is presented in Sections 10.6-10.7. Our choice of the optimization methodology is described in Section 10.6. By optimization methodology, we mean the type of optimization technique used for the stress technique derivation technique presented in this chapter, such as traditional techniques including *Linear Programming (LP)*, *Dynamic Programming (DP)* and *Branch and Bound (BB)* or evolutionary algorithms such as *Genetic algorithms* and *Ant Colony*. For reasons explained in Section 10.6, genetic algorithms are selected and the optimization problem is formulated to be

solvable by a specific genetic algorithm in Section 10.7. Section 10.8 presents a variation of the TSSTT using genetic algorithms to derive interval stress test requirements.

10.1 Types of Arrival Patterns

Arrival-Pattern constraints (APC) relate to the timing of SDs, that is when a SD can start running. APCs can be modeled using the UML-SPT profile, as explained in Section 2.4.

As proposed in Section 4.2.2 of the UML-SPT profile [12], *RTarrivalPattern* tagged-values can be used to model the pattern in which a SD is triggered. Five arrival patterns are defined [12] in BNF (Backus-Nauer Form):

- <bounded> ::= 'bounded', <time-value1>, <time-value2>

  Describes a bounded inter-arrival pattern, where <time-value1> is the minimal interval between successive arrivals and the <time-value2> is the maximum. Both values are expressed using the *RTtimeValue* type. *RTtimeValue* is another tagged-value which is a general format in the UML-SPT profile [12] for expressing time value expressions, e.g., *(20, ms)*. Obviously, the maximum interval value between successive arrivals should be greater than the minimum value.

  For example, (*'bounded', (2, ms), (5, ms))* specifies a bounded pattern where the minimum and maximum time distances between successive arrivals are 2 ms and 5 ms, respectively. An event timing such as <0, 3, 7, 9, 14, 16>, where all times values are in ms, satisfies this arrival pattern.

- <bursty> ::= 'bursty', <time-value>, <integer>

This expression describes a bursty inter-arrival pattern, where `<time-value>` is the burst interval expressed using the *RTtimeValue* type and `<integer>` denotes the maximum number of events that can occur during that interval.

For example, (*'bursty', (5, ms), 2)* specifies a bursty inter-arrival pattern where there can be up to two arrivals in every 5 ms interval. The event timing <0, 4, 7, 12, 14>, where all times values are in ms, satisfies this arrival pattern.

- `<irregular> ::= 'irregular', <time-value> [, <time-value>]*`

  This expression describes an irregular inter-arrival pattern, where the ordered list of time values (expressed using the *RTtimeValue* type) represents successive inter-arrival times.

  For example, (*'irregular', (1, ms), (5, ms), (6, ms), (8, ms), (10, ms))* specifies a irregular pattern where the arrivals occur at specified time instants.

- `<periodic> ::= 'periodic', <time-value1> [, <time-value2>]`

  This expression describes periodic inter-arrival patterns, where `<time-value1>` defines the period and the optional `<time-value2>` denotes the maximal deviation (from the period value). Both values are expressed using the *RTtimeValue* type.

  For example, (*'periodic', (6, ms), (1, ms))* specifies a periodic inter-arrival pattern, where the period and the deviation values are 6 and 1 ms.

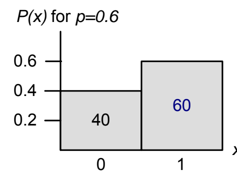
- `<unbounded> ::= 'unbounded', <PDF-string>`

  This expression describes a pattern specified by a *Probability Distribution Function (PDF)* defined in *RTtimeValue*'s BNF expression in Section 4.2.2 of [12]. The types of PDFs supported are: Bernoulli, binomial, exponential, gamma, geometric,

histogram, normal (Gaussian), Poisson, and uniform. Different PDF types are explained below using BNF, mathematical PDF formulas, and an example graph of the PDF.

- The Bernoulli distribution has one parameter, a probability p:

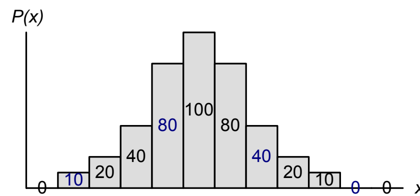
```
<bernoulliPDF> ::= 'bernoulli', <Real>
```



$$P(n)=\begin{cases}1-p & \text{for } n=0\\ p & \text{for } n=1\end{cases}$$

- The binomial distribution has two parameters: a probability p and the number of trials N (a positive integer):

```
<binomialPDF> ::= 'binomial', <Real>, <Integer>
```
[6]



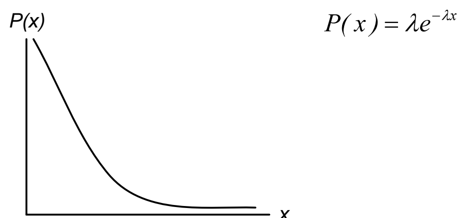
$$P_p(n\,|\,N)=\binom{N}{n}p^n(1-p)^{N-n}$$

- The exponential distribution has one parameter, the mean value λ:

```
<exponentialPDF> ::= 'exponential', <Real>
```

---

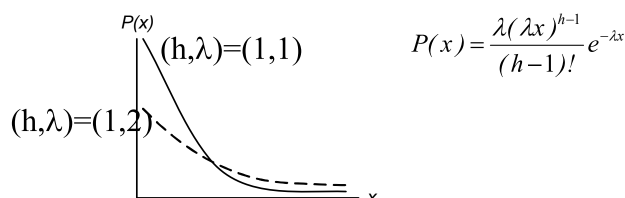
[6] This is written in the UML-SPT as <binomialPDF> ::= " 'binomial' ," <Integer>[6], in page 4-33 of [12] Object Management Group (OMG), "UML Profile for Schedulability, Performance, and Time (v1.0)," 2003.. We have altered the BNF to conform to the PDF's mathematical definition.



$$P(x) = \lambda e^{-\lambda x}$$

- The gamma distribution has two parameters (a positive integer h and a mean λ):

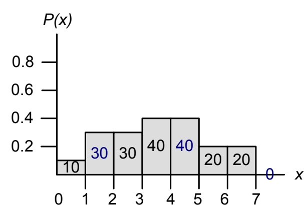
```
<gammaPDF> ::= 'gamma', <Integer>, <Real>
```



$$P(x) = \frac{\lambda(\lambda x)^{h-1}}{(h-1)!} e^{-\lambda x}$$

- The histogram distribution has an ordered collection of one or more pairs which identify the start of an interval and the probability that applies within that interval (starting from the leftmost interval) and one end-interval value for the upper boundary of the last interval:

```
<histogramPDF> ::= 'histogram', {<Real>, <Real>}*, <Real>
```

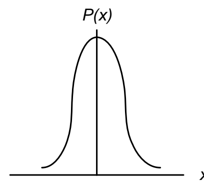


An example:

'histogram', {(0ms,0.1)}, {(1ms,0.3)}, {(3ms,0.4)), {(5ms,0.2)},7ms

- The normal (Gauss) distribution has a mean value μ and a standard deviation value σ (greater than 0):

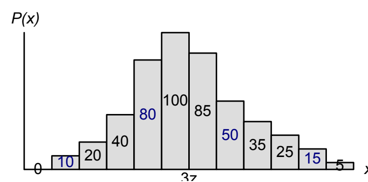
```
<normalPDF> ::= 'normal', <Real>, <Real>
```



$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

- The Poisson distribution has a mean value *v*:

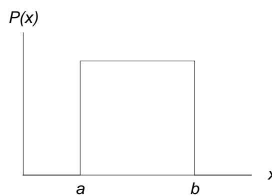
```
<poissonPDF> ::= 'poisson', <Real>
```



$$P_v(n) = \frac{v^n e^{-v}}{n!}$$

- The uniform distribution has two parameters designating the start *a* and end *b* of the sampling interval:

```
<uniformPDF> ::= 'uniform', <Real>, <Real>
```



$$P(x) = \begin{cases} 0 & \text{for } x < a \\ \frac{1}{b-a} & \text{for } a < x < b \\ 0 & \text{for } x > b \end{cases}$$

10.2 Analysis of Arrival Patterns

In order to study the arrival patterns and their impact on the test techniques presented in Chapter 9, the timing characteristics of arrival patterns as well as the test techniques should be analyzed. Furthermore, given an arrival time, we should be able to determine if it satisfies an arrival pattern (AP). Satisfying an AP, in this context, implies that an arrival time is legal given the AP.

The pseudo-code, shown in Figure 58, determines if a DCCFP arrival time satisfies an AP. The AP can be any of the following: {*'bounded', 'bursty', 'irregular', 'periodic', 'unbounded'*}. The pseudo-code is described in detail next.

```
Function IsAPCSatisfied(arrivalTime, AP)
AP∈{'bounded', 'bursty', 'irregular', 'periodic', 'unbounded'}
1 Switch AP {
2    'bounded':
3         If arrivalTime is in one of the intervals of the bounded pattern, then Return True
4         Else Return False
5    'bursty': Return True
6    'irregular':
7         If arrivalTime is one of the time values in the AP list, then Return True
8         Else Return False
9    'periodic':
10        If there exists an arbitrary integer k such that arrivalTime∈[kp-d… kp+d], where p and d are
          the period and the derivation values of the AP: then Return True
11        Else: Return False
12   'unbounded', i.e., AP has a Probability Distribution Function (PDF), (Section 10.1):
13        Return True
14} // end switch
```

Figure 58- Pseudo-code to check if the arrival pattern *AP* is satisfied by an arrival time.

If *AP* is *bounded*, *IsAPCSatisfied()* returns true if the arrival time is inside the time intervals specified by the bounded pattern. Such a pattern is identified by a *minimal and a maximal interval time* (*MinIAT*, *MaxIAT*). We assume that *MinIAT* and *MaxIAT* of a bounded arrival pattern can not be equal. If the two values are equal, the arrival pattern is equivalent to a periodic one. For example, a bounded AP where *MinIAT=MaxIAT=3ms*, is indeed a periodic arrival pattern with *period=3ms*. Consider a bounded arrival pattern

with *MinIAT=4ms* and *MaxIAT=5ms*. The gray eclipses in the timing diagram in Figure 59 depict the Accepted Time Intervals (ATI) of the arrival pattern. ATI here means the time intervals where an arrival pattern is satisfied.

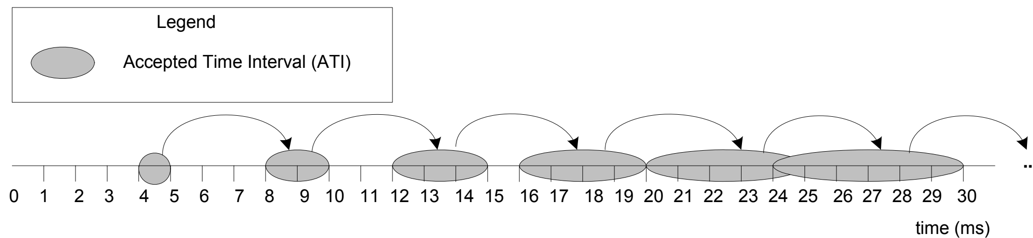


Figure 59-Accepted Time Intervals (ATI) of a *bounded* arrival pattern (*'bounded', (4, ms), (5, ms))*, i.e. *MinIAT=4ms, MaxIAT=5ms.*

Note that the ATIs of a bounded AP denote all *possible* arrival times, regardless of specific previous arrival times in a single scenario. The curved arrows in Figure 59 denote how an ATI is derived from the previous one. For the AP discussed above, assuming that the arrival pattern starts from time=0, the first ATI is [4..5ms]. If an event arrives in time=4ms, according to the fact that *MinIAT=4ms* and *MaxIAT=5ms*, the next event can arrive in interval [8…9ms]. Similarly, if an event arrives in time=5ms, according to the fact that *MinIAT=4ms* and *MaxIAT=5ms*, the next event can arrive in interval [9...10ms]. In a similar fashion, the value in between 4 and 5 ms will cause the next arrival time to be in the range [8…10ms]. Therefore, the second ATI is [8…10ms]. The next ATIs are [12…15ms], [16…20ms], [20…25ms], [24…30ms] and so on.

If the arrival pattern is *bursty*, the function in Figure 58 always returns true. This is because any arrival time satisfies a bursty arrival pattern. For example, consider the arrival pattern (*'bursty', (5, ms), 2)*, which indicates that there can be up to two arrivals in

every 5 ms interval. The gray eclipses in the timing diagram in Figure 60 depict the Accepted Time Intervals (ATI) of this arrival pattern.

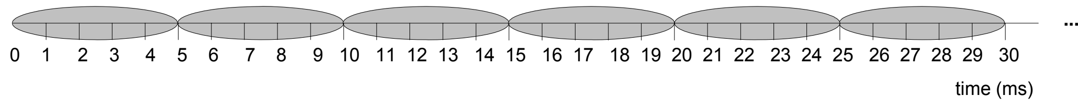


Figure 60-Accepted Time Intervals (ATI) of the bursty arrival pattern (*'bursty', (5, ms), 2)*.

As it can be seen, given a bursty pattern, a single arrival can happen at any time instant, with the constraint that number of arrival in the bursty interval is less than the specified number. For example, up to two arrivals can occur in any of the ATI's of the above pattern. Furthermore, since our aim is to schedule only one DCCFP of a SD execution in a specific time instant (to generate a stress test requirement), we can choose any time instant.

If the arrival pattern is *irregular*, the function returns true (indicating that arrival pattern constraints are satisfied), if the arrival time is one of the elements in the irregular pattern's set. For example, (*'irregular', (1, ms), (5, ms), (6, ms), (8, ms), (10, ms))* specifies a bursty pattern where the arrival occurs at specified time instants. In this case, if the arrival time is 5 ms, for example, the arrival pattern constraint is satisfied. Since the accepted arrival times for an irregular arrival pattern are not intervals, but rather time instants, we refer to them as *Accepted Time Points (ATP)*. An example is shown in Figure 61.

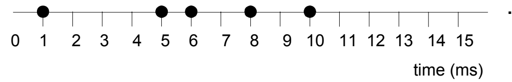


Figure 61-Accepted Time Points (ATP) of the irregular inter-arrival pattern (*'irregular', (1, ms), (5, ms), (6, ms), (8, ms), (10, ms)).*

For a *periodic* arrival pattern, the arrival pattern constraints are satisfied if the start time falls in an interval around periods within the given deviation interval. For example, Accepted Time Intervals (ATI) of the periodic inter-arrival pattern (*'periodic', (5, ms), (1, ms))* are shown in Figure 62. Only arrival times in any of the ATIs satisfy the arrival pattern.

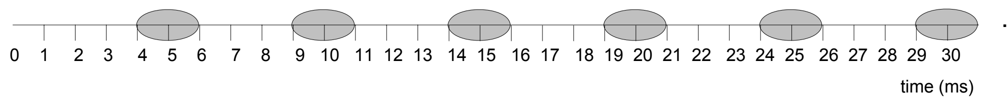


Figure 62-Accepted Time Intervals (ATI) of the periodic inter-arrival pattern (*'periodic', (5, ms), (1, ms)).*

If the arrival pattern is *unbounded*, the function *IsAPCSatisfied* in Figure 58 always returns true. Unbounded arrival patterns correspond to a Probability Distribution Function (PDF). As discussed in Section 10.1, such PDFs specify the probability which an arrival occurs in a specific time instant. For example, the PDF of (*'poisson', (5, ms))* arrival pattern is shown in Figure 63.

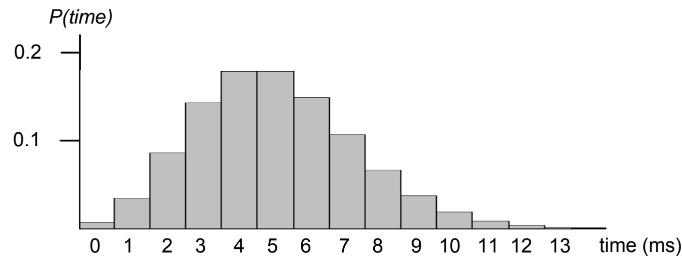


Figure 63-Probability Distribution Function (PDF) of (*'poisson', (5, ms))* arrival pattern.

Assuming that a first arrival occurs in 4 ms, the second arrival time is based on the above PDF, which can be *any* time after 4 ms, since the probability decreases as time goes by, but it never becomes zero. Other unbounded arrival patterns have similar behaviors to the poisson PDF, as discussed above. Therefore, any single arrival time satisfies an unbounded arrival patterns.

10.3 Accepted Time Sets

To facilitate our discussions in the next sections, we define the concept of *Accepted Time Set (ATS)* for each SD. An ATS is the set of time instants or time intervals when a SD is allowed to be triggered, according to its arrival pattern. An ATS can be derived from the arrival pattern of the corresponding SD. The ATS metamodel in Figure 64-(a) defines the fundamental concepts.

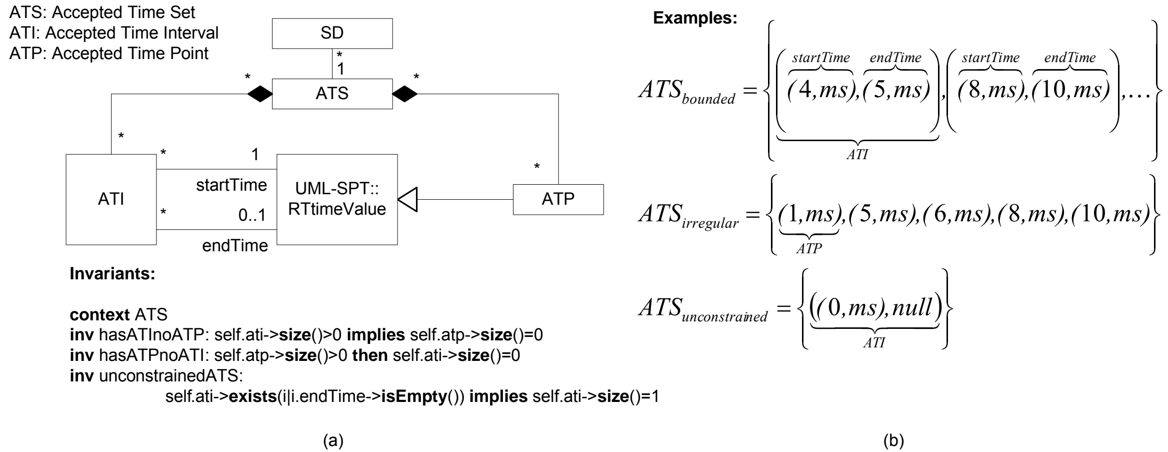


Figure 64-(a): Accepted Time Set (ATS) metamodel. (b): Three instances of the metamodel.

Each SD has an ATS. An ATS is made of several Accepted Time Points (ATP), for irregular and periodic (with no deviation) arrival patterns, *or* several Accepted Time Intervals (ATI), for the other arrival patterns. This is because irregular and periodic (with no deviation) arrival patterns specify the time instants when a SD can be triggered. On the other hand, all the other arrival patterns deal with time intervals. The mutual exclusion between ATIs and ATPs is shown by two OCL invariants (*hasATInoATP* and *hasATOnoATI*) in Figure 64-(a). Each ATI has a start time and an end time of type *RTtimeValue* (from the UML-SPT), denoting the start and end times of an interval. ATP is of type *RTtimeValue* too. The end time of an ATI can be null, which denotes an ATI which has no upper bound (this is further justified below).

Three examples of an ATS are illustrated in Figure 64-(b), which comply with the metamodel in Figure 64-(a). $ATS_{bounded}$ is the ATS corresponding to the arrival pattern whose timing diagram was shown in Figure 59. $ATS_{irregular}$ corresponds to the arrival

pattern in Figure 61. $ATS_{unconstrained}$ is an ATS for SDs which do not have any arrival pattern, i.e., can be triggered any time.

Our convention to represent an unconstrained ATS is to leave the end time of its only interval as *null*: it is unconstrained so no upper bound can be defined. Such an ATS has only one ATI from time 0 to ∞. This constraint has been formalized by the third OCL invariant (*unconstrainedATS*) in Figure 64-(a). Note that one could need to consider other kinds of constraints such as the following, that we refer to as *partly-constrained* ATS: $ATS_{partly\text{-}constrained} = \{((0,ms),(3,ms)),((5,ms),null)\}$, where the corresponding SD can be triggered in all times, except interval [3ms…5ms]. In such an ATS, there is at least one ATI where the end time is null. However, modeling arrival patterns which lead to partly-constrained ATSs is not currently possible using the UML-SPT. Since we assumed the UML-SPT as the modeling language to model arrival patterns in this work, we assume that there will not be any SD with a partly-constrained ATS.

Our GA-based algorithm in Section 10.7 will require computing the intersection of the ATSs of two SD. This will enable our algorithm to generate GA individuals (test requirements) with high stress values. Therefore, we define an intersection operator ($\cap$) for any pair of ATSs: Equation 6. For brevity, *startTime* and *endTime* have been replaced by *s* and *e*.

$$\forall \text{ATSs } ats_1, ats_2 :$$

$$ats_1 \cap ats_2 = \overbrace{\{atp \mid atp \in ATP \wedge atp \in ats_1 \wedge atp \in ats_2\}}^{\text{Common ATPs}}$$
$$\cup \overbrace{\{atp \mid atp \in ATP \wedge ((\exists ati_2 \in ats_2 : atp \in ats_1 \wedge atp \angle ati_2) \vee (\exists ati_1 \in ats_1 : atp \in ats_2 \wedge atp \angle ati_1))\}}^{\text{Common ATPs in ATIs}}$$
$$\cup \overbrace{\left\{ \begin{array}{l} ati \mid \exists ati_1 \in ats_1, ati_2 \in ats_2 : ((ati_2.s < ati_1.e \wedge ati_2.e > ati_1.s) \vee (ati_1.s < ati_2.e \wedge ati_1.e > ati_2.s)) \\ ati.startTime = \max(ati_1.s, ati_2.s) \wedge ati.e = \min(ati_1.e, ati_2.e) \end{array} \right\}}^{\text{Common ATIs}}$$

Equation 6-Intersection of two ATSs.

The membership operators ($\in$) between an ATI/ATP and an ATS denote if an ATI/ATP is a member of an ATS. For example, considering the ATP *(1,ms)* in Figure 64-(a), $(1,ms) \in ATS_{irregular}$.

The output of the formula is the union of three sets: (a) common ATPs (in the case the two ATSs contain only ATPs), (b) common ATPs in ATIs (in case one ATS contains only ATIs and the other contains only ATPs), and (c) common ATIs (in case the two ATSs contain only ATIs). In case (a), the result is the set of ATPs ($atp \in ATP$ means that atp is an ATP) that belong to both ATSs ats_1 and ats_2. The membership operators ($\in$) between an ATI/ATP and an ATS denote if an ATI/ATP is a member of an ATS. For example, considering the ATP *(1,ms)* in Figure 64-(b), $(1,ms) \in ATS_{irregular}$. In case (b), the result is the set of ATPs in one ATS (e.g., ats_1) for which there exists an ATI in the other ATS (e.g., ats_2), such that the (ATP) time point is inside the (ATI) time interval. The formula uses a (in-range) operator $\angle$ to compare a time point (i.e., an ATP) and a time interval (i.e., an ATI): $\forall\, atp \in ATP, ati \in ATI : ati.startTime \leq atp \leq ati.endTime \Leftrightarrow atp \angle ati$. In case (c), the result is the set of overlapping time intervals. The rationale for finding overlapping (common) intervals of two ATSs is illustrated in Figure 65.

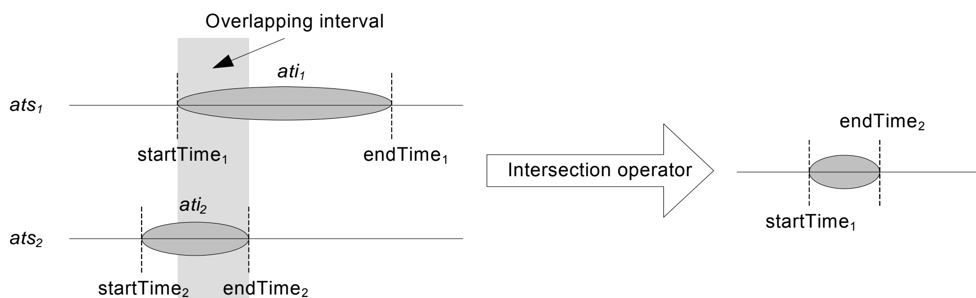


Figure 65- Illustrating the overlap of two ATSs' intervals.

Note that the union of the above three sets is allowed in the current context from the set theory point of view, since as the metamodel in Figure 64-(a) shows, ATS is a *hybrid* set of two element types: ATI and ATP. Therefore, a set of type ATIs together with another set of type ATP can be the operands of a union operator, yielding an ATS. Two examples, showing how intersections of two ATSs can be calculated using Equation 6, are illustrated in Figure 66: between an ATS made of ATIs and an ATS made of ATPs (upper part of the figure); between two ATSs made of ATIs (lower part of the diagram).

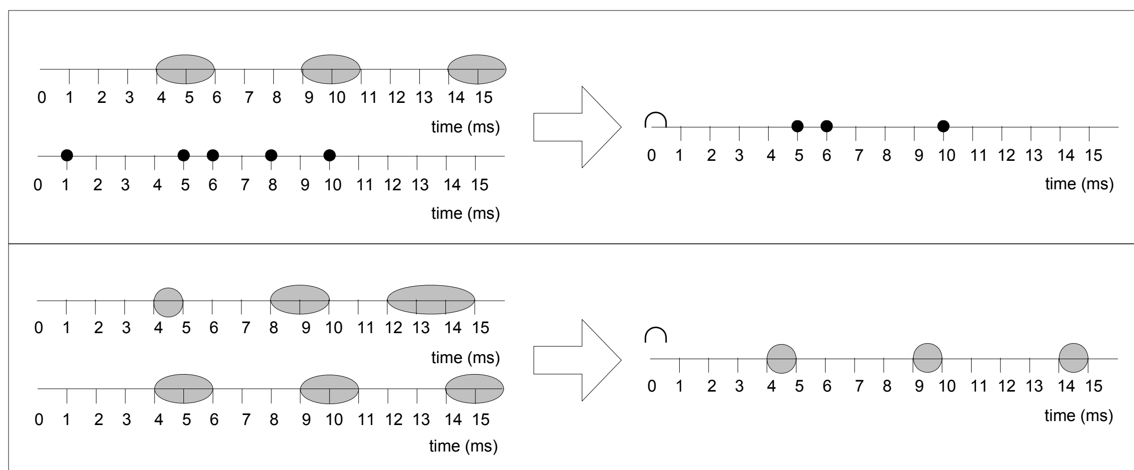


Figure 66-Example intersections of two ATSs.

Based on the above definition of intersection between two ATSs, the intersection of several ATSs can be defined as: $ats_1 \cap ats_2 \cap \ldots \cap ats_n = ((ats_1 \cap ats_2) \cap \ldots) \cap ats_n$

10.4 Formulating the Problem as an Optimization Problem

The problem of generating stress test requirements can be formulated as an optimization problem, as presented in Figure 67.

Objective: Maximize the traffic on a specified network or node (at a time instant or a period of time)

Variables:

- A subset of DCCFPs and the number of instances of each DCCFP with maximum traffic on a specified network or node
- Schedule to run the selected DCCFP instances

Constraints:

- Inter-SD sequential and conditional constraints
- SD arrival patterns

Figure 67-Formulating the problem of generating stress test requirements as an optimization problem.

10.5 Impact of Arrival Patterns on Stress Test Strategies

In Section 9.4, we discussed 32 Time-Shifting Stress Test Technique (TSSTT) such as: instant stress test towards a node with maximum data (*StressNodInInsDT*). We discuss here the impact of arrival patterns on those strategies and determine which strategies have to be tackled differently when considering arrival pattern constraints for a SUT.

Since arrival patterns enforce constraints on the start times of SDs (and hence DCCFPs), they will have an impact on TSSTT test strategies, which assume non-constrained start times for DCCFPs. Specifically, since TSSTT test strategies were grouped into two categories, namely instant and interval test strategies, we expect that arrival patterns will impact differently the two groups of strategies. This is the purpose of the following sub-sections.

10.5.1 Impact on Instant Stress Test Strategies

As we discussed in Section 9.4, instant stress test strategies search among all ISDSs and find the one with maximum instant stress. Then the SDs of the selected ISDS are scheduled to yield the maximum stress. As an example, consider Figure 68-(a), where an ISDS with three SDs (SD_1, SD_2, and SD_3) has been chosen and the SDs can be freely scheduled since none of them has an arrival pattern constraint. Conversely, consider Figure 68-(b) with the same SDs but with arrival pattern constraints, as shown by the ATIs on the time axis. SDs can not then be scheduled freely in any arbitrary time instants. The heuristics to find maximum possible stress, while respecting arrival patterns, will consist in searching among the ATS of every SD and find a time instant when the summation of traffic values entailed by DCCFPs from all the SDs is maximized. One of such possible schedules is shown in Figure 68-(b).

This means that deriving instant stress test requirements while considering arrival patterns requires a global search for an optimum result all across the ATSs of SDs with arrival patterns. SDs without arrival patterns (with unconstrained ATSs) do not need to be searched for a start time, since they can be scheduled anywhere on the time axis.

10.5.2 Impact on Interval Stress Test Strategies

Interval stress test strategies (Section 9.4) aim at increasing the chances of traffic faults by invoking a sequence of SDs, referred to as Concurrent SD Flow Path (CSDFP), which entails the maximum possible interval stress. A CSDFP is a path in a MIOD. It is assumed that each SD of a CSDFP is allowed to be invoked after all previous SDs in the sequence (a path in the MIOD). As to the scheduling of a SD with arrival pattern in a

CSDFP, the SD can start its first execution according to its arrival pattern as soon as all the previous SDs have finished executing (thus satisfying the sequential constraints of a SD). For example, consider Figure 68-(d), where a CSDFP with five SDs have been chosen and two of the SDs (SD_2 and SD_5) have arrival patterns. The flow of SDs in the CSDFP is as follows:

$$\rho = SD_1 \begin{pmatrix} SD_2 \\ SD_3 \end{pmatrix} SD_4 SD_5$$

As the CSDFP indicates, SD_5 can start as soon as SD_4 is finished. This is shown in Figure 68-(c), where no SD has an arrival pattern and SD_5 can start immediately as soon as SD_4 has finished. However, in the case when SD_5 has an arrival pattern, it cannot start until the first time instant in its ATS. Considering the fact that the goal of the interval stress test strategies is to maximize stress within a time interval (maximize possible stress in the shortest possible time of a CSDFP), the impact of arrival patterns on interval stress test strategies will constrain when the optimization technique can schedule each SD in its earliest ATS. SDs with arrival patterns can no longer start immediately after all their preceding SDs (in the MIOD) have been completed.

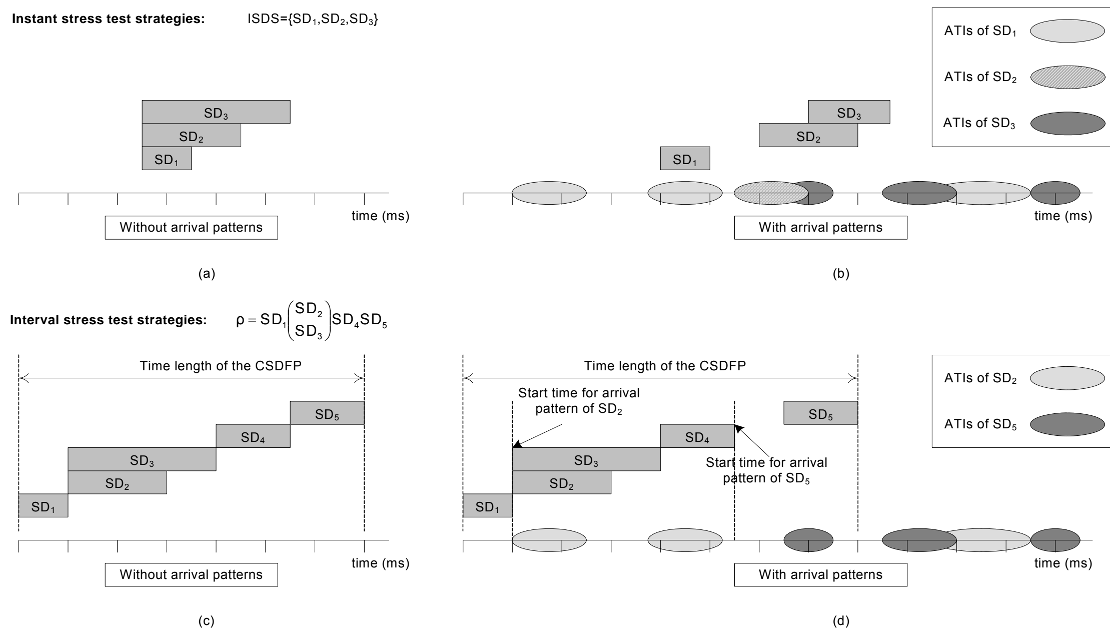


Figure 68-Impact of arrival patterns on instant (a)-(b) and interval (c)-(d) stress test strategies.

We now discuss the extent to which the impact of arrival patterns complicates the optimization technique of the interval stress test strategies. Interval stress test strategies need to account for the ATSs of SDs with arrival patterns. Recall that we want to maximize traffic over a period of time, and that this period of time is the overall duration of the execution of all the SDs (i.e., the selected CSDFP). Since the total traffic does not depend on the scheduling, but rather on the selected DCCFP of each SD, scheduling has only an impact on the overall duration of execution of all the SD. For each SD, the earliest time point in its ATS is considered as this reduces the overall duration of execution. We thus at the same time maximize traffic (selection of DCCFPs for SDs) and minimize the period of interest (scheduling), thus resulting in the highest stress. Therefore, no complicated global search is required in this case. Note however, that the

time length of CSDFPs will increase, compared to the case when none of the SDs of a CSDFP have an arrival pattern (refer to Figure 68-(c) and Figure 68-(d) as an example).

To provide more insights, we now discuss why and how the test requirements generated by the TSSTT (Chapter 9) might not comply with SD arrival pattern constraints. We consider an example to illustrate the idea. We described in Section 2.4 how SD arrival patterns can be modeled using the UML-SPT profile tagged-values. Figure 69-(a) depicts two (partial) SDs, each having an arrival pattern constraint. We described in Section 10.1 the types of arrival patterns defined by the UML-SPT profile that we consider in this chapter. The arrival pattern of SD_1 in Figure 69-(a) is *irregular*, and it has three arrival times (10, 25 and 70 ms). SD_2 is periodic, where period=15 ms and the maximal deviation of the period is 2 ms.

Based on the arrival pattern information of Figure 69-(a), and assuming that the maximum duration of DCCFPs of SD_1 and SD_2 are 15 ms and 10 ms, respectively, a timing diagram as the one in Figure 69-(b) can be drawn to show the effect of SD arrival pattern constraints on scheduling SDs. Arrival times of SD_1 are fixed, as specified by its arrival pattern. However, there can be up to a 2 ms deviation in the arrival time of SD_2. For example, assuming that SD_2 starts at time=0, its next arrival times can be 13-17 ms, 28-32 ms and so on.

Based on arrival pattern information, we define the concept of *Valid* and *Invalid SD Schedule (VSDS* and *IVSDS)*. Given a set of arrival patterns, a VSDS is a schedule of SDs (their start times) in which the start time of each SD satisfies its arrival pattern. For

example, if we show a schedule of SDs in a similar notation as output stress test requirements in Section 9.2, <(SD_1, 10 ms), (SD_2, 14ms)>[7] is a VSDS. On the other hand, <(SD_1, 0 ms), (SD_2, 0ms)> is an IVSDS, given the arrival patterns in Figure 69. These two schedules are depicted in Figure 69-(c).

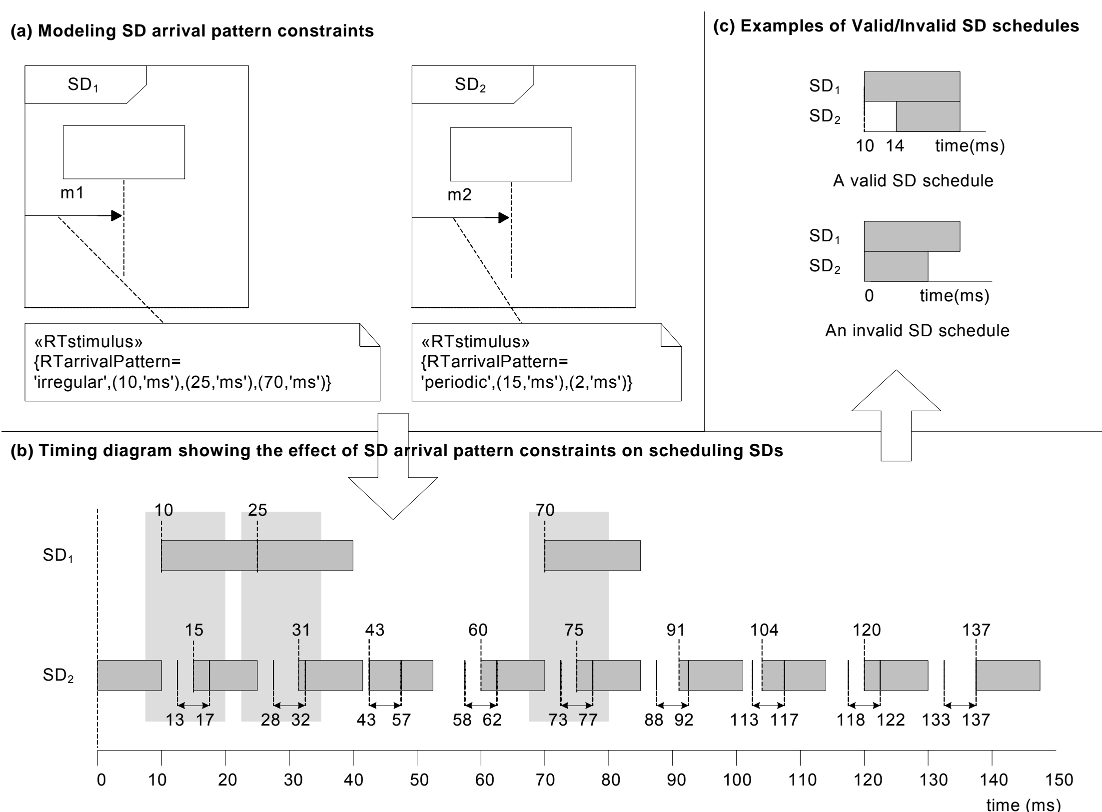


Figure 69-SD arrival pattern constraints.

10.5.3 How Arrival Patterns are Addressed by Stress Test Strategies

As discussed above, the impacts of arrival patterns on instant and interval stress test strategies are different. As we discussed, no complicated global search is required for the

[7] Meaning that SD_1 and SD_2 start at time=10 and 14 ms, respectively.

case of interval stress test strategies, while considering arrival patterns in instant stress test strategies needs a global search of an optimum result across all ATSs of SDs with arrival patterns.

We separate the two cases, i.e., instant and interval test requirements, and address them separately. Derivation of instant stress test requirements while considering arrival patterns is presented in Sections 10.6-10.7, based on a Genetic Algorithm. Section 10.8 presents a variation of the technique in Chapter 9 to derive interval stress test requirements while complying with arrival patterns.

10.6 Choice of the Optimization Methodology: Genetic Algorithms

A variety of methods exist for solving optimization problems. Perhaps the most common techniques are linear and global optimization techniques. In linear optimization, or linear programming (LP) as it is more commonly known, the objective/fitness function, as well as all constraints, are linear functions of the decision variables to be solved. Linear programming solutions are optimal as the search is performed on the intersections of the constraints [84]. Global optimization solutions, also known as meta-heuristic solutions, continually search for better solutions by altering a set of current solutions [85]. This is typically used when the solutions lie on an uneven solution space, characterized by multiple peaks and valleys. These peaks and valleys can result in locally optimal solutions; one where no other solution in the vicinity have better solutions. Global optimization solutions aim at avoiding local optima, reaching global ones instead. Simulated annealing, tabu search, ant colony and genetic algorithms are among the most common global optimization solutions.

For the test requirement generation problem at hand, which is actually a scheduling problem, the number of SDs and DCCFPs only depend on the SUT. As the number of SDs and DCCFPs increases and their arrival patterns get more complex, the different combinations representing solutions can grow exponentially. As a result, linear programming cannot be used, as they would lead to a combinatorial explosion problem [86]. Furthermore, any small change in the number of SDs and DCCFPs or the execution times may cause great changes in the solution. The solution space of the problem is thus uneven, characterized by multiple peaks and valleys. Last, the AP of a single SD can potentially impose a set of complex time interval constraints which are disjoint and of different duration. Thus, we have a set of non-linear constraints in our optimization problem and a global optimization technique is needed.

Genetic Algorithms (GA) are based on concepts adopted from evolutionary theory [87]. GAs involve a search from a population of solutions rather than a single solution. With each iteration of a GA, solutions with the highest fitness are recombined and mutated, and solutions with the lowest scores are eliminated. Tabu search (TS) is another global optimization technique which avoids cycles by penalizing moves that take the solutions to points previously visited in the solution space. In the Simulated Annealing (SA) method, each point of the search space is compared to a state of some physical system, and a so called *energy* function (to be minimized) is interpreted as the internal energy of the system in that state. Therefore the goal is to bring the system, from an arbitrary initial state, to a state with minimum possible energy. At each step, the SA heuristic considers some neighbors of the current state s, and probabilistically decides between moving the system to state s' or staying put in state s. The probabilities are chosen so that the system

ultimately tends to move to states of lower energy. Typically this step is repeated until the system reaches a state which is good enough for the application, or until a given computation budget has been exhausted [85].

According to the global optimization literature, GAs and SA are very similar. Some studies, such as [88] indicate that SA outperforms GAs, while others, such as Chardaire et al. [89] claim that GAs produce solutions equivalent or superior to SA. Most researchers, however, seem to agree that because GAs maintain a population of possible solutions, they have a better chance of locating the global optimum compared to SA and TS which proceed one solution at a time [90, 91]. Furthermore, because SAs maintain only one solution at a time, good solutions may be discarded and never regained if cooling occurs too quickly. Similarly, TS may miss the optimum solutions. Alternatively, steady state GAs, one of the variations of GAs, accept newly generated solutions only if they are fitter than previous solutions. Furthermore, GAs are usually more flexible and more scalable than other non-linear optimization methodologies, and lend themselves to parallelism, as they manipulate whole populations: computations for different parts of the population can be dispatched to different processors. SA, on the other hand, cannot easily run on multiple processors because only one solution is constantly manipulated [90]. Hence, we adopt GA as our optimization technique methodology. An overview on Genetic Algorithms is provided in Appendix A.

10.7 Tailoring Genetic Algorithm to Derive Instant Stress Test Requirements

A GA is used to solve the optimization problem of finding DCCFPs and their seeding times such that the maximum instant traffic on a network or a node increases. To solve the optimization algorithm for deriving instant stress test requirements, this section

describes the different components of the GA, tailoring them to our problem. We define a chromosome representation in Section 10.7.1. Constraints defining legal chromosomes are formulated in Section 10.7.2. Derivation of the initial GA population is discussed in Section 10.7.3. The concept of a time search range which is needed in our GA for the initialization process as well as the operators is discussed in Section 10.7.4. The objective (fitness) function is described in Section 10.7.5. GA operators (crossover and mutation) are finally presented in Section 10.7.6.

10.7.1 Chromosome

Chromosomes define a group of solutions to be optimized. The representation of chromosomes and their length have to be defined in a GA algorithm [87]. We discuss the chromosome representation of our application in Section 10.7.1.1. Chromosome length is described in Section 10.7.1.2.

10.7.1.1 Representation

In our application, we need to optimize the selection of SDs' DCCFPs and their schedule, i.e., their start times. Thus, we need to encode both DCCFP identifiers and their arrival times in a chromosome.

A gene can be depicted as a pair ($\rho_{i,selected}$, $\alpha\rho_{i,selected}$), where $\rho_{i,selected}$ is a selected DCCFP of SD_i, and $\alpha\rho_{i,selected}$ is the start time of $\rho_{i,selected}$. Together, the pair represents a schedule of a specific DCCFP. If no DCCFP is selected from a SD (because the SD does not have a traffic over a particular network, for example), the gene is denoted as *null*: this is to ensure that the number of genes in each chromosome remains constant as this facilitates the definition of mutation/cross-over operators and fitness function. This representation is

the same as the general form of a stress test requirement (the output of the technique in Chapter 9).

To formalize the concepts we employ, a metamodel is depicted in Figure 70-(a). *Chromosome* is composed of a sequence of *Gene* ordered in the same order as SDs (Recall that we assume SDs are indexed). The *Initialization*, *Crossover* and *Mutation* operators are all defined in *Chromosome*, as well as the objective function, *Evaluate*. These functions will be defined in Section 10.7.6.

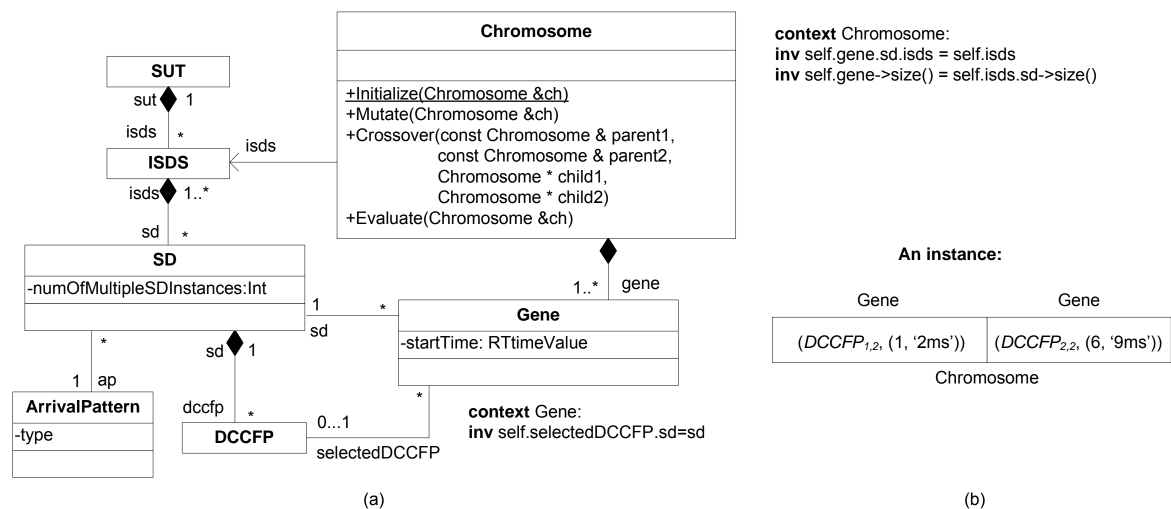


Figure 70-(a): Metamodel of chromosomes and genes in our GA algorithm. (b): Part of an instance of the metamodel.

Each *Gene* is associated with a SD. Furthermore, it has an association (*selectedDCCFP*) to zero (if no DCCFP is chosen) or one DCCFP, and has an attribute: *startTime*, which is the time value to trigger *dccfp,* and is of type *RTtimeValue* (defined in the UML-SPT). Each DCCFP belongs to a SD, whereas each SD can have several DCCFPs and has an attribute: *numOfMultipleSDInstances,* which is the number of multiple SD instances which are allowed to be triggered concurrently (Section 5.4). Furthermore, Each SD can

be a member of several ISDSs. Arrival pattern information of SDs is stored in instances of a class *ArrivalPattern* (attributes of such a class can be easily defined based on the discussions in Section 10.1, such as type and AP parameters), and are accessible by the *ap* association. Each ISDS can have one or more SDs. Finally, a SUT (model) has one or more ISDSs. Recall that we are optimizing traffic at a given instant and what matters is thus the number of SDs that can be triggered concurrently. We therefore do not need to model sequential and conditional constraints.

An example of a chromosome and a gene is illustrated in Figure 70-(b), which complies with the metamodel in Figure 70-(a). The chromosome is composed of two genes, since it is assumed that the SUT has two SDs: SD_1 and SD_2. $DCCFP_{1,2}$ and $DCCFP_{2,2}$ are selected DCCFPs of SD_1 and SD_2, respectively. The genes indicate that the DCCFPs' start times are 2 ms and 9 ms, respectively.

10.7.1.2 Length

The length of chromosomes, i.e., the number of genes in the chromosomes, is fixed and is equal to the number of SDs in a SUT. This is due to the fact that each gene of a chromosome corresponds to a SD, and we have a fixed number of SDs per SUT.

Furthermore, as discussed in Section 10.7.1.1, if no DCCFP is selected from a SD (because the SD does not have traffic over a particular network, for example), the selected gene of a *Gene* is represented as null (the reasons for have 0..1 multiplicity on the *selectedDCCFP* association). Therefore, the chromosome length remains the same at all times.

10.7.2 Constraints

Inter-SD and arrival pattern constraints should be satisfied when generating new chromosomes from parents, otherwise, GA *backtracking* procedures [87] should be used. Backtracking, however, has its drawbacks: it is deemed expensive because time consuming. Some GA tools incorporate backtracking while others do not. To allow for generality, we assume no backtracking methodology is available. Therefore, we have to ensure that the GA operators always produce chromosomes which satisfy the GA's constraints. In order to do so, we formally express inter-SD and arrival pattern constraints in the context of our GA.

10.7.2.1 Constraint #1: Inter-SD constraints

We incorporated inter-SD constraints in ISDSs (Chapter 7). A set of DCCFPs are allowed to execute concurrently in a SUT only if their corresponding SDs are members of an ISDS. As discussed in Section 10.7.1.1, each chromosome is a sequence of genes, where each gene is associated with zero or one DCCFP. Therefore, a chromosome satisfies Constraint #1 only if the SDs of DCCFPs corresponding to its genes are members of a same ISDS. In other words, each chromosome corresponds to one ISDS. We can formulate the above constraint as a class invariant on class *Chromosome* (Figure 70-(a)) as presented in Figure 71.

```
context Chromosome

inv: self.gene.selectedDCCFP.sd.isds->asSet->size=1
```

Figure 71- Constraint #1 of the GA (an OCL expression).

10.7.2.2 Constraint #2: Arrival pattern constraints

Given a chromosome, the OCL function in Figure 72 can be used to determine if the chromosome (the scheduling of its genes) satisfies the Arrival Pattern Constraints (APC) of SDs. The function *IsAPCSatisfiedByAChromosome(c :Chromosome)* returns true if all genes of the chromosome satisfy the APCs. The OCL function makes use of function *IsAPCSatisfied(startTime, AP)*, defined in Section 10.2.

```
1  IsAPCSatisfiedByAChromosome(c:Chromosome)
2     post: result=
3        if c.gene->exits(g| g.selectedDCCFP.notEmpty and
                            not IsAPCSatisfied(g.startTime, g.sd.ap) then
4           false
5        else
6           true
7        endif
```

Figure 72-Constraint #2 of the GA (an OCL function).

10.7.3 Initial Population

Determining the population size of a GA is challenging [85]. A small population size will cause the GA to quickly converge on a local minimum because it insufficiently samples the search space. A large population, on the other hand, causes the GA to run longer in search for an optimal solution. Haupt and Haupt in [87] list a variety of works that suggests adequate population sizes. The authors reveal that the work of De Jong [92] suggests a population size ranging from 50 to 100 chromosomes. Grefenstette et al. [93] recommend a range between 30 and 80, while Schaffer and his colleagues [94] suggest a smaller population size, between 20 and 30.

We choose 80 as the population size as it is consistent with most of experimental results. The GA initial population generation process should ensure that the two constraints of Section 10.7.2 are met. The pseudo-code to generate the initial set of chromosomes is presented in Figure 73. As indicated by the constraint #1 (Section 10.7.2.1), each chromosome corresponds to an ISDS. Therefore, line 1 of the pseudo-code chooses a random ISDS and the initialization algorithm continues with the selected ISDS to create an initial chromosome. Note that to generate our GA's initial population, *CreateAChromosome()* is 80 times.

```
Function CreateAChromosome(): Chromosome
c: Chromosome
1    ISDS=a random ISDS in the set of ISDSs
     // selecting genes (DCCFPs)
2    For all SD_i∈ISDS do
3         c. gene_i.selectedDCCFP= a random DCCFP from SD_i
4         c. gene_i.sd = SD_i
5    For all SD_i∉ISDS do
6         c. gene_i=null
7         c. gene_i.sd = SD_i
     // initial scheduling of genes (DCCFPs)
8    Intersection=ATS(SD_1) ∩ ATS(SD_2) ∩… ∩ATS(SD_i), for all SD_i∈ISDS
9    If Intersection≠ϕ then
10        Choose a random time instant t_schedule in Intersection
          // schedule all genes' start time to t_schedule
11        For all c. gene_i ≠null
12             c. gene_i.startTime= t_schedule
13   Else // Intersection=ϕ, SDs of ISDS do not have overlapping start times
          // schedule each gene with a random time in the ATS of its SD
14        For all c. gene_i ≠null do
15             c. gene_i.startTime= A random time instant t_i in ATS(SD_i)
16   End If
17   Return c
```

Figure 73-Pseudo-code to generate a chromosome for the GA's initial population.

For each SD in the ISDS selected in line 1, lines 2-4 choose a random DCCFP and assign it to the corresponding gene (i.e. $gene_i$ corresponds to SD_i). Other genes of the chromosome (those not belonging to the selected ISDS) are set to null (lines 5-7). An initial scheduling is done on genes in lines 8-16. The idea is to schedule the DCCFPs in such a way that the chances that DCCFPs' schedules overlap are maximized. This is done by first calculating the intersection of ATSs for SDs in the selected ISDS (line 8), using the intersection operator described in Section 10.2. If the intersection set is not null (meaning that the ATSs have at least one overlapping time instant), a random time instant is selected from the intersection set (line 10). All DCCFPs of the genes are then scheduled to this time instant (lines 11-12).

If the intersection set is null, it means that the ATSs do not have any overlapping time instant. In such a case, the DCCFP of every gene is scheduled differently, by scheduling it to a random time instant in the ATS corresponding to its SD (lines 14-15).

Following the algorithm in Figure 73, we ensure the initial population of chromosomes complies with both constraints of Section 10.7.2. In the case when the intersection of SD ATSs is null, one might wonder whether there are still any possibilities to run SDs concurrently to have maximum stress. The answer to this question is twofold:

- Although the ATS intersection of all SDs in the selected ISDS is null, a subset of SDs might still have a non-null ATS intersection. Triggering these SD concurrently can lead to traffic faults. For example, consider the timing diagram in Figure 74, where the ATS intersection of three SDs (SD_1 …SD_3) is null. Although there is no single time instant, when the three SDs can be triggered concurrently, a subset of them (SD_1 and SD_2) have a non-empty ATS intersection, which allow them to be triggered

concurrently. This situation can be made possible in a chromosome by our mutation operator (Section 10.7.6.2), since as we will discuss, our mutation operator will shift each SD in its ATSs to create a new offspring.

- Another situation when the high stressing messages of a set of SDs with empty ATS intersection might be triggered is when the execution of a SD is long enough such that it spans over the ATS intersection of other SDs. For example, SD_3 in Figure 74 has been triggered in one of its allowed times and continues to the ATS intersection of SD_1 and SD_2. In such a case, messages from all three SDs overlap (in the time domain) and thus may trigger high stress scenarios. Such a situation can also be made possible by our mutation operator (Section 10.7.6.2).

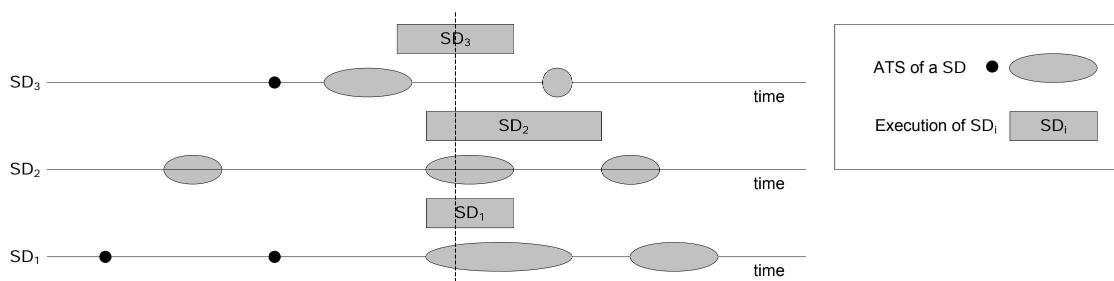


Figure 74-An example where the ATS intersection of all SDs is null, but they can overlap.

10.7.4 Determining a Maximum Search Time

One important issue in our GA design is the range of the random numbers chosen from the ATS of a SD with an arrival pattern. As discussed in Section 10.3, the number of ATIs or ATPs in some types of APs (e.g. periodic, bounded) can be infinite. Therefore, choosing a random value from such an ATS can yield very large values, thus creating implementation problems.

Another direct impact of such unboundedness on our GA is that it would significantly decrease the probability that all (or a subset) of start times of DCCFPs (corresponding to the genes of a chromosome) overlap or be close to each other. If the maximum range when generating a set of random numbers is infinity, the probability that all (or a subset) of the generated numbers are relatively close to each other is very small. Thus, to eliminate such problems, we introduce a solution: *GA's Maximum Search Time*. This maximum search time is essentially an integer value (in time units) which enforces an upper bound on the selection of random values for start times of DCCFPs, chosen from an ATS. The GA maximum search time will be used in our GA operators (Section 10.7.6) to limit the maximum ranges of generated random time values.

Different values of Maximum Search Time (*MST*) for a specific run of our GA might produce different results. For example, if the search range is too limited (small maximum search time), not all ATIs and ATPs in all ATSs will be exercised. On the contrary, if the range is too large (compared to maximum values in ATSs), it will take a longer time for the GA to converge to a maximum plateau, since the selection of random start times for DCCFPs will be sparse and the GA will have to iterate through more generations to settle on a stable maximum plateau (in which start times are relatively close to each other).

The impact of MST on exercising the time domain is illustrated in Figure 75 using an example, where the ATSs of three APs (a periodic, a bounded and a bursty one) are depicted. Four maximum search times (MST_i) have been arbitrarily chosen. The search range specified by MST_1 (*Search Range*$_1$) is not a *suitable* one since only time values in the first ATI of the bounded ATS will be chosen thus preventing the GA from searching all possible start times in the ATS range of the depicted bounded AP. This will limit the

search space, thus reducing the chances of finding the most stressful situations. Following a similar reasoning, the search range specified by MST_2 (*Search Range*$_2$) is not a suitable one either. MST_3 and MST_4 specify ranges in which a complete search over the possible ATS values can be performed. Comparing the last two, the latter does not provide any advantage in terms of completeness of the search range over the former, while at the same time causing a slower convergence of the GA. Therefore, MST_3 is a preferable maximum search time over MST_4. Note that the ATS of the bursty AP in Figure 75 does not play any role in determining a suitable maximum search time, since by having an unrestricted ATS, regardless of the choice of such a MST, any start time can be chosen for a bursty AP.

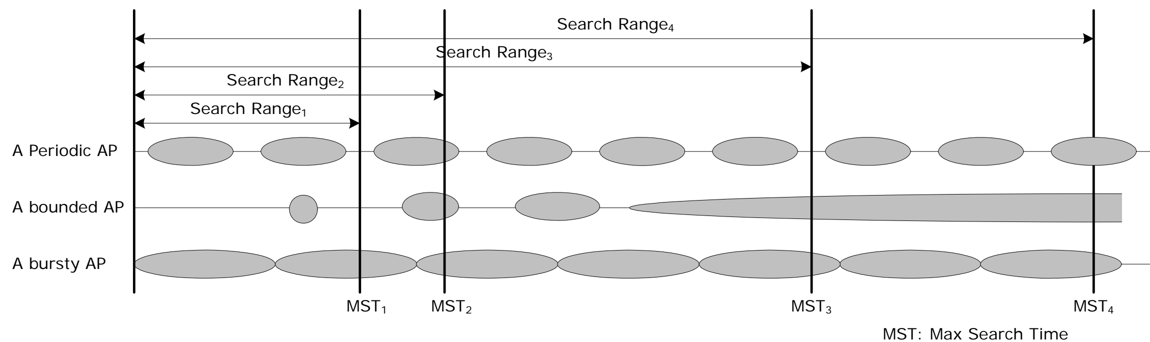


Figure 75-Impact of maximum search time on exercising the time domain.

As we saw in the above example, a suitable maximum search time depends on the occurrence and intersections of different ATSs. We discuss below how a suitable maximum search time can be estimated for a set of ATSs based on a set of heuristics. In order to do this, we group the types of arrival patterns (AP) into two groups:

- *Bounded Arrival Patterns*: APs which result in ATSs where the number of ATIs or ATPs is finite. Only irregular APs match this description.

- *Unbounded Arrival Patterns*: APs which result in ATSs where the number of ATIs or ATPs is infinite. With this definition, periodic, bounded, unbounded, and bursty APs are unbounded.

If all APs are irregular, then a suitable MST ($MST_{suitable}$) will be the maximum of all latest irregular arrival times in all ATSs. For example, the ATSs of three irregular APs are depicted in Figure 76. A $MST_{suitable}$ will be the last arrival time of the third AP (as depicted), which has the maximum time value. This maximum search time will allow the GA to effectively search in the time domain, considering all possible start times from all APs.

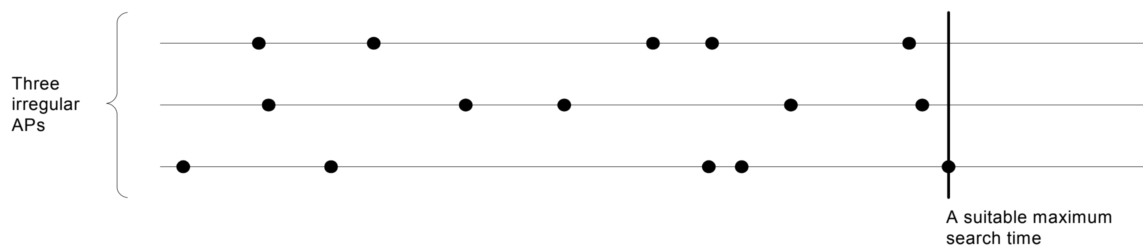


Figure 76- The ATSs of three irregular APs.

If APs are infinite, the occurrence and intersection of different ATIs in the APs should be taken into account. Since only periodic, irregular, and bounded APs have discrete ATSs (Section 10.3), we only consider them in finding a suitable maximum search time. Unrestricted APs (bursty and unbounded) do not impose any restrictions on the selection of a suitable maximum search time, since any time value is acceptable by a bursty or an unbounded AP.

We present in Table 6 a set of heuristics to identify a $MST_{suitable}$ based on a given set of periodic, irregular, and bounded APs. In the heuristics, *ap.type* denotes the type of an AP, e.g., *'bounded'*, *'periodic'*.

#	**Heuristics**	**Rationale**
1	$MST_{suitable} \geq \max_{\forall ap_i \mid ap_i.type=\text{'irregular'}} (ap_1.maxATP, \cdots, ap_n.maxATP)$	This heuristic will allow the GA to effectively search in the time domain, considering all possible start times from all irregular APs.
2	$MST_{suitable} \geq \max_{\forall ap_i \mid ap_i.type=\text{'bounded'}} (ap_1.URSP, \cdots, ap_n.URSP)$	This heuristic will provide a full search coverage on all bounded APs simultaneously.
3	$MST_{suitable} \geq LCD_{\forall ap_i \mid ap_i.type=\text{'periodic'}} (ap_1.period, \cdots, ap_n.period) + \max_{\forall ap_i \mid ap_i.type=\text{'periodic'}} (ap_1.deviation, \cdots, ap_n.deviation)$	The time range around this LCD value can yield schedules when all the periodic SDs can start simultaneously. This heuristic can also be used when generating the initial GA population to set start times close to this LCD value which, in turn, can potentially yield stress test schedules with high ISTOF values.

Table 6-A set of heuristics to identify a suitable MST (*$MST_{suitable}$*).

Heuristic #1 denotes that a $MST_{suitable}$ should be greater than the maximum value among all maximum ATPs of irregular APs. The rationale beyond this heuristic is the same as the case when all APs of a TM are irregular (discussed above). *ap.maxATP* denotes the maximum ATP of an irregular AP and can be calculated using the formula in Equation 7.

$$\forall ap \in AP : ap.maxATP = \begin{cases} atp_{max} \mid atp_{max} \in ap.ATS \wedge \forall atp \in ap.ATS : atp_{max} > atp & ;\text{if } ap.type = \text{'irregular'} \\ \text{undefined} & ;\text{else} \end{cases}$$

Equation 7-A formula to calculate the maximum ATP (*maxATP*) of an irregular AP.

Heuristic #2 is meant to provide a full search coverage on all bounded APs simultaneously. Recall from Section 10.3 that every bounded ATS has an ATI whose end time is infinity. Furthermore, all time instants after the start time of such an ATI are accepted arrival times. If the MST value is chosen to be greater than all such start times

among all bounded APs, the GA will be able to have a full search coverage on all of the APs, and thus, maximizing the chances of finding a test schedule with high stress value. To better formalize heuristic #2, we refer to such a start time as the bounded AP's *Unbounded Range Starting Point (URSP)*. The URSP of a bounded AP can be calculated using the formula in Equation 8, given the ATIs of the AP.

$$\forall ap \in AP : ap.URSP = \begin{cases} lastATIstart \mid (lastATIstart, 'null') \in ap.ATS & ;\text{if } ap.type = \text{'bounded'} \\ \text{undefined} & ;\text{else} \end{cases}$$

Equation 8-A formula to calculate the *Unbounded Range Starting Point (URSP)* of a bounded AP, given the ATIs of the AP.

For example, the URSP of the bounded APs in Figure 77 are denoted as $URSP_i$. If the minimum and maximum inter-arrival times (*minIAT* and *maxIAT*) of a bounded AP are given, the formula in Equation 9 can be used to calculate the value of the URSP. The proof of this formula is given in Appendix B.

$$\forall ap \in AP : ap.URSP = \begin{cases} \left\lceil \dfrac{minIAT}{maxIAT - minIAT} \right\rceil .minIAT & ;\text{if } ap.type = \text{'bounded'} \\ \text{undefined} & ;\text{else} \end{cases}$$

Equation 9-A formula to calculate the *Unbounded Range Starting Point (URSP)* of a bounded AP, given the minimum and maximum inter-arrival times (*minIAT* and *maxIAT*) of the AP.

Heuristic #3 is meant to provide a time range when all the periodic SDs can be triggered simultaneously or close-enough to each other. The Least Common Denominator (LCD) value of all the period values of the periodic APs provides one such a time range. The time range around this LCD value can yield schedules with potential high stress values.

The maximum value of the periodic APs' deviations is also included in the heuristic #3 to increase the chances of finding a potential schedule with a high stress value.

A $MST_{suitable}$ value should be calculated by considering all three heuristics in Table 6, i.e., a $MST_{suitable}$ is equal to the maximum value among the three right-hand side in the three ≥ inequalities. To better explain the above set of heuristics, an example with three irregular, three periodic, and three bounded APs is shown in Figure 77 and the process of deriving a $MST_{suitable}$ for this particular example is described next.

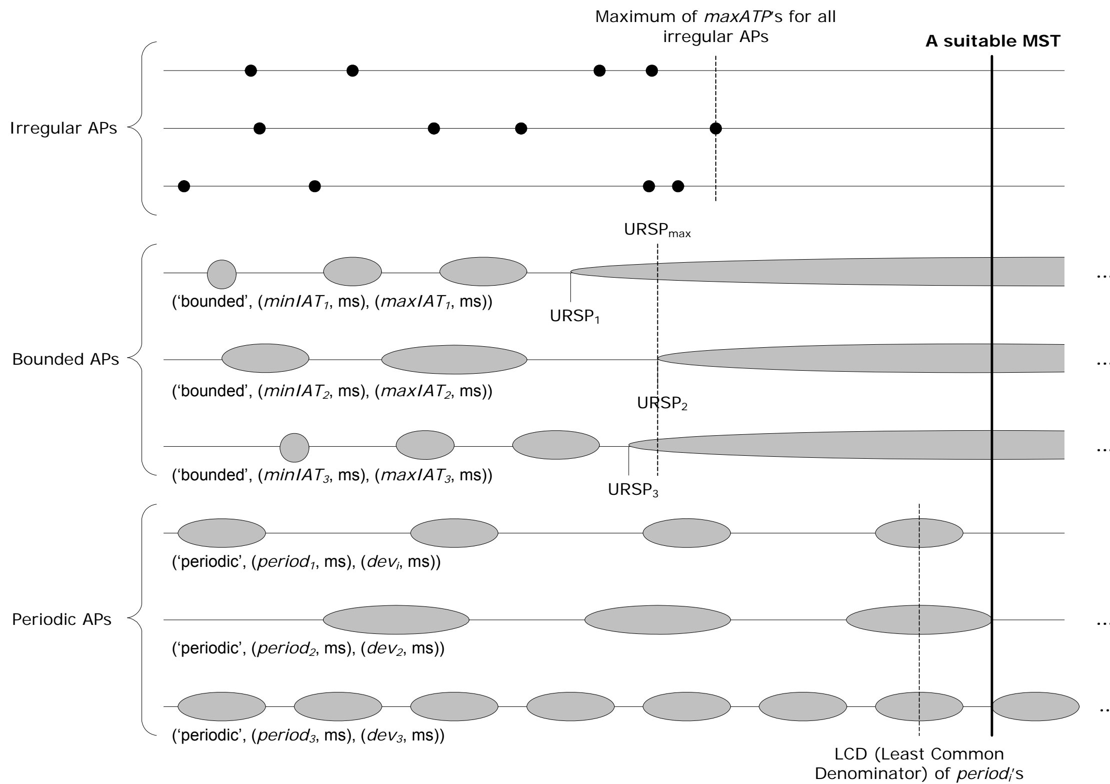


Figure 77-Illustration showing the heuristic of choosing a suitable maximum search time.

The maximum value of *maxATP*'s for the three irregular APs is shown. Heuristic #1 denotes that a $MST_{suitable}$ should be greater than this value.

The URSP of each bounded AP has been calculated using the formula in Equation 9 based on the $minIAT_i$ and $maxIAT_i$ of each AP, and is denoted as $URSP_i$. The maximum value among all $URSP_i$'s is referred to as $URSP_{max}$. Heuristic #2 denotes that a $MST_{suitable}$ should be greater than this value.

Finally, the LCD of the period values of the periodic APs is calculated based on the values of $period_i$ and is shown. Heuristic #3 denotes that a $MST_{suitable}$ should be greater than the sum of this value and the greatest deviation value among all periodic APs. A $MST_{suitable}$ (shown by a bold line) is the smallest time value which satisfies the above three heuristics.

10.7.5 Objective (Fitness) Function

Optimization problems aim at searching for a solution within the search space of the problem such that an objective function is minimized or maximized [85]. In other words, the objective function can aim at either minimizing the value of chromosomes or maximizing them. The objective function of a GA measures the fitness of a chromosome. Recall from Section 10.4 that our optimization problem is defined as follows: *What selection and what schedule of DCCFPs maximize the traffic on a specified network or node (at a specified time instant)?*

Recall from Section 10.2 that we only apply our GA-based technique to instant test objectives. Therefore, let us refer to the objective function in this section as *Instant Stress Test Objective Function (ISTOF).* The *ISTOF* should measure the maximum instant traffic entailed by a schedule of DCCFPs, specified by a chromosome. Using the network formalism in Chapter 8, we define ISTOF in Equation 10.

Note that what we define below as the ISTOF formula is only for the stress test objective: location=*network*, direction=*none*, and type=*data traffic*. Other values for these parameters would lead to a different ISTOF measure by simply using other distributed traffic usage functions from the set of functions defined in Section 8.5.

$$ISTOF : Chromosome \rightarrow Real$$
$$\forall c \in Chromosome : ISTOF(c) = \max_{\forall t \in SearchRange} \sum_{\forall g \in c.gene} NetInstDT(g.dccfp, net, t) \times g.numOfMultipleSDInstances$$
$$SearchRange = [\min_{\forall g \in c,gene}(g.startTime), \max_{\forall g \in c.gene}(g.startTime + Length(g.dccfp))]$$

Equation 10- Instant Stress Test Objective Function (ISTOF).

The first line of Equation 10 indicates that the input and output domains of ISTOF are chromosomes and real numbers. *c.Length(dccfp)* is a function to calculate the time duration of a DCCFP (modeled in the corresponding SD using the UML-SPT tagged-values). Such a calculation can be done as follows:

$$Length(\rho_{ij}) = \max_{\forall m \in CCFP(\rho_{ij})}(m.end)$$

net is the given network to stress test. *NetInsDT* is the distributed traffic usage function to measure the instant data traffic in a network (Section 8.5.2.1). The value of *NetInsDT* is multiplied by the gene's *numOfMultipleSDInstances* value. When multiple instances of a DCCFP are triggered at the same time, the entailed traffic at each time instant is proportional to the number of instances.

The heuristic underlying the ISTOF formula is that it tries to find the maximum instant data traffic considering all genes in a chromosome. The search is done in a predetermined time range. The starting point of the search is the minimum *startTime* (the start time of the earliest DCCFP), and the ending point of the range is the end time of the latest

DCCFP, which is calculated by taking maximum values among start times plus DCCFP lengths.

To better illustrate the idea behind ISTOF, let us discuss how ISTOF for the chromosome in Figure 70-(b) is calculated. The calculation process is shown in Figure 78. The chromosome contains two genes, which correspond to $DCCFP_{1,2}$ and $DCCFP_{2,2}$. The search range is [2ms, 20ms]. ISTOF sums the *NetInsDT* values in this range and finds the maximum value. The output value of ISTOF is 110 KB.

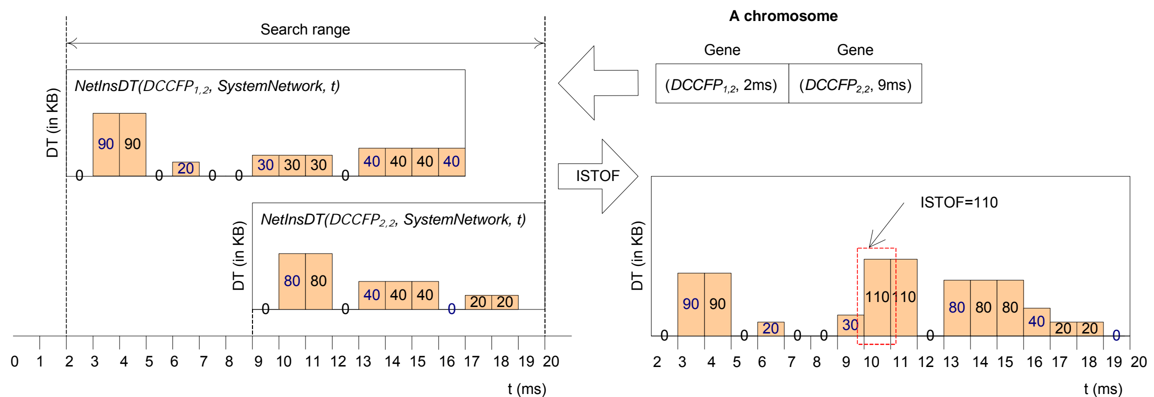


Figure 78-Computing the Instant Stress Test Objective Function (ISTOF) value of a chromosome.

10.7.6 Operators

Operators enable GAs to explore a solution space [87] and must therefore be formulated in such a way that they efficiently and exhaustively explore it. If the application of an operator yields a chromosome which violates at least one of the GA's constraints, the operation is repeated to generate another chromosome. This is an alternative to GA backtracking and is done inside each operator, i.e., each operator generates temporary children first and checks if they do not violate any constraints (Section 10.7.2). If the

temporary children satisfy all the constraints, they are returned as the results of the operator. Otherwise, the operation is repeated. Furthermore, operators should be formulated such that they explore the whole solution space. We define the crossover and mutation operators next.

10.7.6.1 Crossover Operator

Crossover operators aim at passing on desirable traits or genes from generation to generation [87]. Varieties of crossover operators exist, such as sexual, asexual and multi-parent. The former uses two parents to pass traits to two resulting children. Asexual crossover involves only one parent. Multi-parent crossover combines the genetic makeup of three or more parents when producing offsprings. Different GA applications call for different types of crossover operators. We employ the most common of these operators: sexual crossover.

The general idea behind sexual crossover is to divide both parent chromosomes into two or more fragments and create two new children by mixing the fragments [87]. Pawlowsky dubs this n-point crossover [95]. In n-point crossover, the two parent chromosomes are aligned and cut into n+1 fragments at the same places. Once the division points are identified in the parents, two new children are created by alternating the genes of the parents [95].

In our application, since each gene corresponds to a SD, we consider the fragmentation policy to be on each gene, making the size of each fragment to be one gene. Therefore, assuming n is the number of genes, the resulting crossover operator (using Pawlosky's terminology [95]) is (n-1)-point, and is denoted *nPointCrossover*. In our application, the

mixing of the fragments is additionally subject to a number of constraints (Section 10.7.2): A newly generated chromosome should satisfy the inter-SD and arrival pattern constraints. We ensure this by designing the GA operators in a way that they would never generate an offspring violating a constraint.

Whether the alternation process of the *nPointCrossover* operator starts from the first gene of one parent or the other is determined by a 50% probability. To further introduce an element of randomness, we alternate the genes of the parents with a 50% probability, hence implementing a second crossover operator, *nPointProbCrossover*. In *nPointCrossover*, the resulting children have genes that alternate between the parents. In *nPointProbCrossover*, the same alternation pattern occurs as *nPointCrossover*, but instead of always inheriting a fragment from a parent, children inherit fragments with a probability of 50%.

It is important to note that, for both crossover versions, if the set of genes (their corresponding SDs) do not belong to an ISDS, constraint #1 (Section 10.7.2.1) will be violated. In such a case, we do not commit the changes and search for different parent chromosomes (by applying the operator again). Regarding constraint #2 (Section 10.7.2.2), note that since the parents are assumed to satisfy the arrival pattern constraint, and the crossover operators do not change the start times of genes' DCCFPs, the child chromosomes are certain to satisfy such constraint. The start times of DCCFPs will be changed (mutated) by our mutation operator (described in the next section) and the arrival pattern constraint will be checked when applying that operator.

An activity diagram for depicting the crossover operators is shown in Figure 79. Note that the *crossover operator function* in the diagram can be any of the two

nPointCrossover or *nPointProbCrossover* operators (specified by the *operator type*, given as a parameter to the activity diagram).

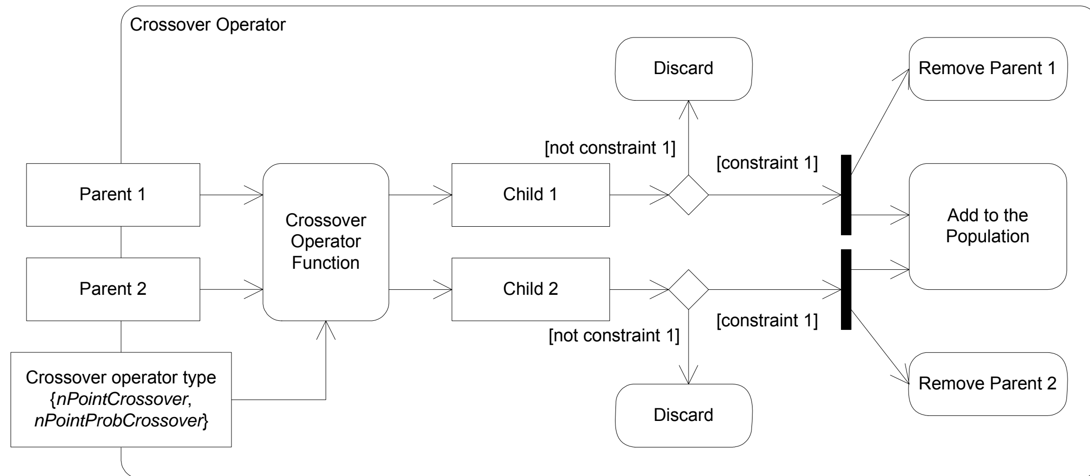


Figure 79-Crossover operators.

Let us consider the example in Figure 80 to see how our two crossover operators work. The number of genes in each parent chromosome is five (assuming that there are five SDs in the SUT). Assume that SD numbering is the same as gene numbering and $ISDS_1$={SD_1, SD_3, SD_4, SD_5}. Parent 1 has genes corresponding to DCCFPs in {SD_1, SD_4, SD_5}$\subset$ $ISDS_1$. Parent 2's genes are DCCFPs in {SD_1, SD_3, SD_4}$\subset$$ISDS_1$. The results of applying *nPointProbCrossover* and *nPointCrossover* are shown in Figure 80.

In *nPointCrossover*, the fragments of Parent 1 and Parent 2 are alternately interchanged. Using the same example for *nPointProbCrossover*, one possible outcome appears in Figure 80-(c). Bold genes indicate the fragments interchanged by *nPointProbCrossover*. All four generated children conform to constraint #1, i.e., the SD corresponding to their genes belong to one ISDS ($ISDS_1$), as well as constraint #2..

The advantages of *nPointProbCrossover* are twofold. It introduces further randomness in the crossover operation. By doing so, it allows further exploration of the solution space. However, *nPointProbCrossover* has its disadvantages: the resulting children may be replicas of the parents, with no alteration occurring. This is never the case with *nPointCrossover*; resulting children are always genetically distinct from their parents.

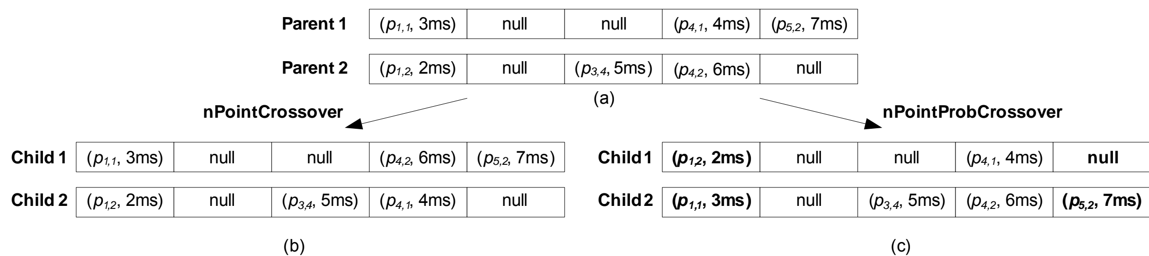


Figure 80-Two example uses of the crossover operators.

Crossover rates are critical. A crossover rate is the percentage of chromosomes in a population being selected for a crossover operation. If the crossover rates are too high, desirable genes will not be able to accumulate within a single chromosome whereas if the rates are too low, the search space will not be fully explored [87]. De Jong [92] concluded that a desirable crossover rate should be about 60%. Grefenstette et al. [93] built on De Jong’s work and found that crossover rates should range between 45% and 95%. Consistent with the findings of De Jong and Grefenstette, we apply a crossover rate of 70%.

10.7.6.2 Mutation Operator

Mutation aims at altering the population to ensure that the GA avoids being caught in local optima. The process of mutation proceeds as follows: a gene is randomly chosen for

mutation, the gene is mutated, and the resulting chromosome is evaluated for its new fitness. We define three mutation operators that (1) mutate a non-null gene (a gene with an already assigned DCCFP) in a chromosome by altering either its DCCFP, (2) mutate the start time of a non-null gene, or (3) mutate the entire chromosome by assigning another ISDS to it (i.e., assign to each gene of the chromosome to mutate a randomly selected DCCFP from the corresponding SD of a randomly selected ISDS and a start time from the ATS of that SD in a range up to GA's maximum search time). The mutation operators are referred to as *DCCFPMutation*, *startTimeMutation*, and *ISDSMutation,* respectively.

The idea behind the *DCCFPMutation* operator is to choose different DCCFPs of the SD, corresponding to a gene. The idea behind the *startTimeMutation* operator is to move DCCFP executions along the time axis. The aim of the operators is to find the optimal DCCFPs and start times at which instant traffic of the selected genes (DCCFPs) is maximized. This is done in such a way that the constraints we defined on the chromosomes are met (Section 10.7.2).

Since the mutation operators alter non-null genes only, they do not change the set of SDs corresponding to a chromosome, thus ensuring that constraint #1 is satisfied (the set of SDs will still belong to the same ISDS). However, start times are changed by the mutation operator *startTimeMutation*, resulting in a possible violation of constraint #2. The output of the *DCCFPMutation* operator will always adhere to constraint #2, since the start times are unchanged by the operator. One way of making sure that a generated chromosome by the *startTimeMutation* operator satisfies the arrival pattern constraints is to set the new start times to a random value in the range of accepted arrival time values of

a SD, i.e., Accepted Time Sets (ATS) – (Section 10.2). Therefore, we design the *startTimeMutation* operator in such a way that the altered start times are always among the accepted one. In other words, there will be no need to backtrack in this case.

The above descriptions of the three mutation operators can be illustrated as two activity diagrams in Figure 81 to Figure 83, respectively. Note that the manipulations used in these two figures are in the chromosome and genes metamodel domain (Figure 70). For example, *g.selectedDCCFP* denotes the DCCFP assigned to a gene. *ATS(sd)* return the ATS of SD *sd*.

Mutation rates are critical. Mutation rate is the percentage of chromosomes in a population being selected for mutation. Throughout the GA literature, various mutation rates have been used. If the rates are too high, too many good genes of a chromosome are mutated and the GA will stall in converging [87]. Back [96] enumerates some of the more common mutation rates used. The author states that De Jong [92] suggests a mutation rate of 0.001, Grefenstette [93] suggests a rate of 0.01, while Schaffer et al. [94] formulated the expression $1.75/\lambda\sqrt{length}$ (where λ denotes the population size and *length* is the length of chromosomes) for the mutation rate. Mühlenbein [97] suggests a mutation rate defined by 1/*length*. Smith and Fogarty [98] show that, of the common mutation rates, those that take the length of the chromosome and population size into consideration perform significantly better than those that do not. Based on these findings, we apply the mutation rate suggested by Schaffer et al.: $1.75/\lambda\sqrt{length}$.

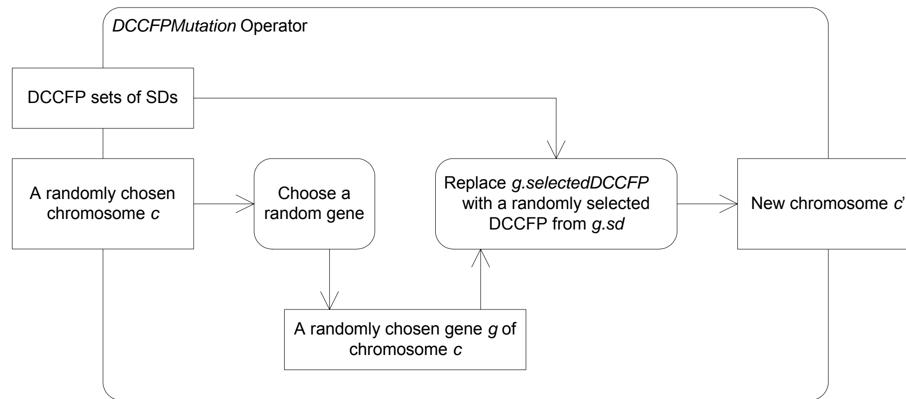


Figure 81- *DCCFPMutation* operator.

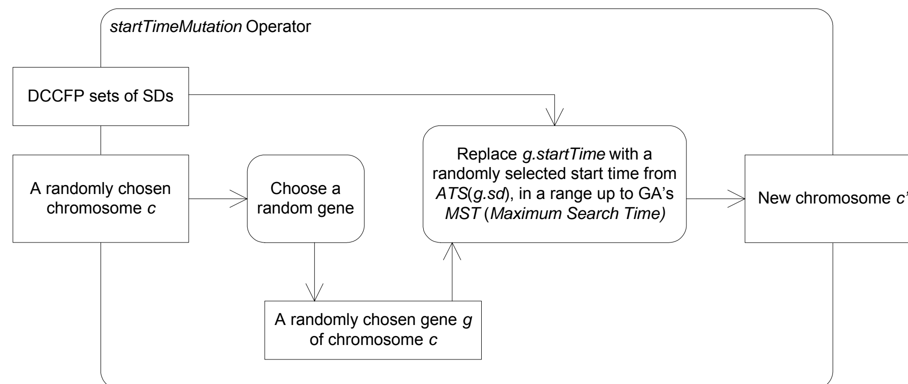


Figure 82- *startTimeMutation* operator.

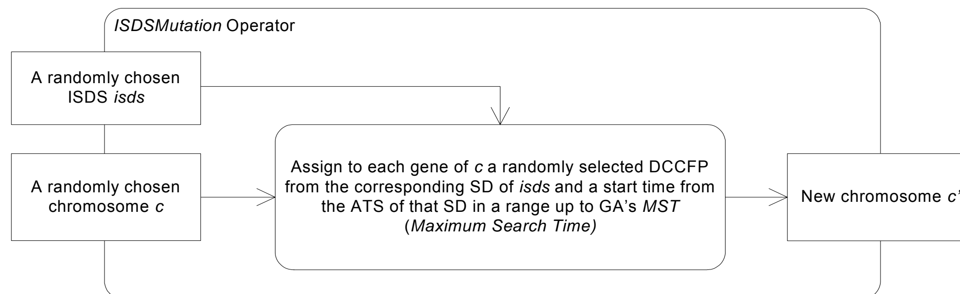


Figure 83- *ISDSMutation* operator.

10.8 Interval Stress Test Strategies accounting for Arrival Patterns

As discussed in Section 10.5, interval stress test strategies need to account for the ATSs of SDs with arrival patterns. For such SDs, the earliest time points in their ATSs are

considered (to cause the most stressful situation, that is the shortest overall SDs execution). Therefore, no complicated global search (such as the GA used for the instant stress strategies) is required in this case. The time length of CSDFPs will increase in such a case, compared to the case when there is no SD in a CSDFP with arrival patterns (refer to Figure 68-(c) and Figure 68-(d) as an example). We present an pseudo-code in Algorithm 2, referred to as *APStressNetIntDT,* which takes into account arrival patterns.

1. Find the DCCFP of each SD with maximum unit data traffic

1.1. For each SD_i

1.1.1. For each DCCFP ρ_{ij} of SD_i // Finding maximum stress message of each DCCFP

Calculate Unit Data Traffic (UDT) of ρ_{ij}, using:

$$NetUDT(\rho_{ij}, net) = \frac{\sum_t \left(NetInsDT(\rho_{ij}, net, t)\right)}{Duration(\rho_{ij})}$$

where $Duration(\rho_{ij})$ is the time length of DCCFP ρ_{ij} and can be calculated as:

$$Duration(\rho_{ij}) = \max_{\forall m \in CCFP(\rho_{ij})} (m.end)$$

where $CCFP(\rho_{ij})$ is the CCFP corresponding to DCCFP ρ_{ij}.

1.1.1. Among all DCCFPs ρ_{ij} of SD_i, find the one with maximum unit data traffic

$$MaxNetPerDTDCCFP(SD_i, net) = \rho_{i\max} \left| \begin{array}{l} \forall \rho_{i\max}, \rho_{ij} \in DCCFP(SD_i): \\ NetUDT(\rho_{i\max}, net) \geq NetUDT(\rho_{i\max}, net) \end{array} \right.$$

If no DCCFP in SD_i is found with the above criteria, the function returns null.

2. Choose a CSDFP (Concurrent SD Flow Path) with maximum stress: // Inter-SD constraints are considered here

2.2 For each $CSDFP_i$ // Calculate each CSDFP's Unit Data Traffic (UDT)

$$NetUDT(CSDFP_i, net) = \frac{\sum_{\forall SD \in CSDFP_i} \sum_{\forall t} NetInsDT(MaxNetPerDTDCCFP(SD, net), net, t)}{minAPDuration\left(BuildDCCFPS\left(CSDFP_i, MaxNetIntDT, net\right)\right)}$$

where *minAPDuration* is an extended version of the function *Duration* (presented in Section 7.2.3) that calculates the minimum time length of a DCCFPS (DCCFP Sequence) given the arrival pattern of its SDs. Arrival pattern constraints are considered in this step, affecting the length of DCCFPSs, and hence helping the algorithm to find the DCCFPS with highest stress per time unit. *BuildDCCFPS* is function that builds a DCCFPS from the given $CSDFP_i$ using the given criteria: $MaxNetIntDTDCCFP, net$.

2.2 Among all CSDFPs, find the sequences with maximum $NetUDT(CSDFP_i, net)$ and return it as output ($CSDFP_{max}$)

Algorithm 2-Derivation of interval stress test requirements for data traffic on a given network, considering arrival patterns (*APStressNetIntDT*).

In the algorithm, *minAPDuration(aDCCFPS),* in Step 2.1, is a variation of function *Duration* (presented in Section 7.2.3) that calculates the minimum time length of a DCCFPS (DCCFP Sequence) given the arrival pattern of its SDs. Arrival pattern constraints are accounted for in this step, affecting the length of DCCFPSs, and hence helping the algorithm to find the DCCFPS with highest stress per time unit. *BuildDCCFPS* is a function that builds a DCCFPS from the given $CSDFP_i$ using the given criteria $MaxNetIntDTDCCFP_{,net}$. The pseudo-code of *minAPDuration()* is shown in Algorithm 3 which is very similar to that of *Duration(),* presented in Section 7.2.3. The only difference is how the duration of an atomic CCFPS is calculated. The illustration in Figure 84 shows the impact of arrival patterns in the actual duration of a CCFP. On the left-hand side of this figure, the duration of a CCFP has been calculated using *Duration*, since the CCFP's corresponding SD does not have an arrival pattern. Conversely, the right-hand side of the figure shows the case when the corresponding SD of a CCFP has an arrival pattern. The ATIs of the arrival pattern are depicted. In this case, the actual duration of the CCFP has been calculated using *minAPDuration*, which is the summation of the CCFP's duration plus the minimum arrival time of the corresponding SD, based on its arrival pattern.

The idea is formalized in the calculation of the function *earliestAT* (arrival time) in Equation 11 which calculates the earliest arrival time of a SD given its arrival pattern. If a SD does not have an arrival pattern, *earliestAT* returns 0, meaning that the SD can start immediately, given that its sequential/conditional SD constraints are satisfied.

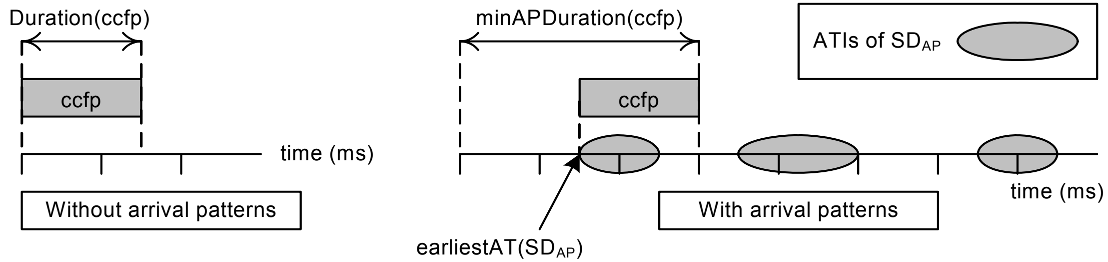


Figure 84-Impact of arrival patterns on the duration of a CCFP.

1. Function minAPDuration(*ccfps*: CCFPS): integer
2. if *ccfps* is atomic (only made of one CCFP)
3. return $\max_{\forall m \in ccfps}(m.endTime) + earliestAT\,(SD_{ccfps}) \mid ccfp = \text{the only CCFP of } ccfps$
4. else if *ccfps* is the serial concatenation of several CCFPSs (i.e., $ccfps = ccfps_1 \cdots ccfps_n$)
5. return minAPDuration ($ccfps_1$)+…+ minAPDuration ($ccfps_n$)
6. else if *ccfps* is the concurrent combination of several CCFPSs (i.e., $ccfps = \begin{pmatrix} ccfps_1 \\ \cdots \\ ccfps_n \end{pmatrix}$)
7. return max(minAPDuration ($ccfps_1$),…, minAPDuration ($ccfps_n$))
8. End Function

Algorithm 3-Calculating the minimum duration of a Concurrent Control Flow Path Sequence (CCFPS), accounting for arrival patterns.

$$earliestAT(SD) = \begin{cases} \min\limits_{\forall atp \in \mathrm{ATS}(SD)}(atp) & ;\text{if } SD \text{ has an arrival pattern} \\ 0 & ;\text{else} \end{cases}$$

Equation 11- Function returning the earliest arrival time of a SD based on its arrival pattern.

For example, let us calculate the duration of the following CCFPS:

$$CCFPS = \rho_1 \begin{pmatrix} \rho_2 \\ \rho_3 \end{pmatrix} \rho_4$$

where each ρ_i is a CCFP of SD_i. Assume the duration of each of the individual CCFPs and arrival patterns of their corresponding SDs are given as in Table 7. Following the analysis in Section 10.2 and 10.3, we first compute the Accepted Time Sets (ATS) of the SDs. The result of *earliestAT(SD)* for each SD is also shown. For example, since the AP of SD_1 is bursty, its earliest arrival time can be 0ms.

CCFP	Duration
$CCFP_1$	2800 ms
$CCFP_2$	1300 ms
$CCFP_3$	1000 ms
$CCFP_4$	1000 ms

(a)

SD	Arrival Pattern	*earliestAT(SD)*
SD_1	('bursty', (500, ms), 2)	0ms
SD_2	No arrival pattern	0ms
SD_3	('periodic', (500, ms), (100, ms))	400ms
SD_4	('bounded', (500, ms), (600, ms))	500ms

(b)

Table 7-(a): Durations of several CCFPs. (b): Arrival patterns of several SDs.

The call tree of the recursive algorithm *minAPDuration* applied to *CCFPS* is shown in Figure 85. Since $CCFPS_1$ is a serial concatenation of three CCFPSs itself, three recursive calls are made, whose results will be summed upon return. One of these CCFPSs ($\begin{pmatrix}\rho_2\\ \rho_3\end{pmatrix}$), is the concurrent combination of two CCFPs, therefore the maximum value of their durations are returned as the durations of this CCFPS. For example *minAPDuration*(ρ_3)=1000+400=1400ms.

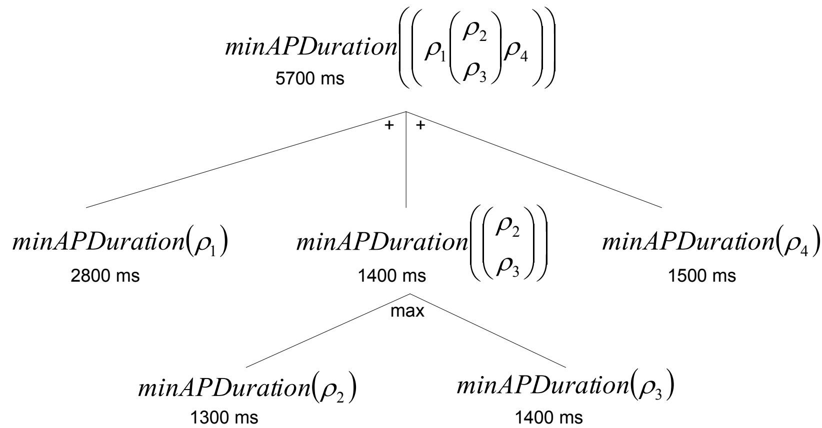


Figure 85-Call tree of the recursive algorithm *minAPDuration* applied to a CCFPS.

10.9 Impacts of Uncertainty in Timing Information on our Stress Test Methodology

We discussed the possible uncertainty in timing information (execution times of messages) in Section 5.1.1.2. Now that we already presented the simpler version of our stress test methodology (Time-Shifting Stress Test Technique) in the earlier sections of the current chapter, we can now discuss the impacts of uncertainty in timing information on TSSTT. Since each of our instant and interval stress test strategies consider time differently (the former focuses on time instants and the latter on time intervals), the impacts of time uncertainty are likely to be different on each of the two strategies, as described next.

Recall the heuristics of our instant stress test strategy (Section 9.2), where the shifting (scheduling) of CCFPs along the time axis is performed such that the instant traffic (on a specific network) is maximized. Referring to Figure 56 as an example showing our stress test heuristics, it can be observed that if the start and/or end times of any message are changed (an effect of uncertainty in timing information), the output stress test requirements might change dramatically. Such an impact (in an optimal output) is

specially great if there are uncertainties in timing information of maximum stressing message(s) of DCCFPs. Such uncertainties will most probably change the output test requirement by shifting an involved DCCFP along time axis or by entirely changing the ISDS associated with the test requirement.

Investigating the impacts of uncertainty in timing information on outputs of our stress test methodology can actually be performed by adopting *sensitivity analysis*[8] theories from the optimization field. Based on the above discussions, we expect that our optimization problem (stress test methodology) is quite sensitive to uncertainty in timing information. Informally, it can be stated as the average variance (as a measure of uncertainty) in timing information in a SUT increases, the preciseness of the output stress test requirements generated by our methodology decreases. The preciseness of a test requirement in this context corresponds to the certainty by which executing test cases corresponding to that test requirement will maximize traffic on a given network (or a node).

The formal (mathematical) sensitivity analysis of the impacts of uncertainty in timing information on our stress test methodology would require more discussions that, due to space shortage and our project boundaries, we leave to future works.

---

[8] A procedure to determine the sensitivity of the optimal result of an optimization problem to changes in its parameters (e.g. constraints and coefficients). If a small change in a parameter results in relatively large changes in the optimal result, the problem is said to be sensitive to that parameter.

One possible ad-hoc idea to help our stress test methodology cope with uncertainty in timing information is to focus on the middle time-unit portion of maximum stressing messages (whenever it exists) when attempting to build a stress test schedule. To better express this idea, consider Figure 56-(c) where the first portions of four maximum stressing messages (in the four DCCFPs) are considered to build a stress test schedule. Furthermore, assume that there is uncertainty on the start and end times of the only call message of $DCCFP_3$. The above idea suggests to slightly change the stress test schedule so that the middle portion of $DCCFP_3$ (from 4 to 5 ms in Figure 56-(b)) is triggered on the maximum stress time instant. The heuristic behind the above ad-hoc idea is that: if the start or the end time of the only call message of $DCCFP_3$ slightly differs (less than a 1 ms) at test runtime, the hope is that the stress test can still maximize the amount of traffic. Of course, the success of such slight changes in DCCFP schedules of a stress test requirement in coping with uncertainty in timing information is dependent on several factors, such as: (1) the levels of timing uncertainty, i.e. low chances of highest possible traffic if there are high levels of uncertainty in the durations of the involved messages, (2) the time durations of the involved messages, i.e., low chances of highest possible traffic if the durations of the involved messages are relatively short.

10.10 Wait-Notify Stress Test Technique

Recall from Section 5.1.1.3 where we discussed messages with unpredictable execution times (or start/end times). After using different schedulability analysis techniques to estimate execution times of messages in a SUT, there might still be messages whose execution times are unpredictable, or no WCET/BCET can be found for them. Examples

of such messages are those with data-intensive parameters whose data sizes can not be estimated by any means in advance.

Estimating the execution times of such messages will lead to great amounts of uncertainty in such time values, which might lead to great deals of indeterminism in our stress test methodology and the output test requirements it will generate (Section 10.9). Thus, we will present in the current section a different version of our stress test methodology, referred to as *Wait-Notify Stress Test Technique (WNSTT)*, which can be used to stress test systems with at least one message with unpredictable execution time.

The underlying ideas of WNSTT is taken from the *barrier* scheduling heuristics [21] for finding concurrent bugs in Java programs. To find concurrent bugs, the authors of [21] created interesting interleavings (interesting here means effective at finding bugs) related to a given shared variable by installing thread barriers before and after accessing the variable. The barrier is implemented by using a counting semaphore. The semaphore causes threads to wait just before the shared variable is accessed. When more than one thread is waiting, then the Java statement *notifyAll()* is used to simultaneously advance the waiting threads. Thus, threads access the variable simultaneously. As a result, the probability of a data race occurring increases and with it the probability of a concurrent bug manifesting.

We adopt the barrier scheduling heuristics to devise WNSTT as follows. To increase the chances of manifesting real-time faults, WNSTT intends to generate stress test requirements by installing barriers before messages which entail maximum traffic (among all messages of a DCCFP) on a given network (or a node). The barrier can be implemented by using a counting semaphore. The semaphore causes all DCCFPs (of an

ISDS, chosen among all ISDSs such that total instant traffic is maximal) to wait just before each of the involved messages are triggered. When all the DCCFPs (of the chosen ISDS) is waiting, then all the DCCFPs are notified to simultaneously trigger the waiting messages. Thus, all the maximum stressing messages send their instant traffic simultaneously. As a result, the probability of a traffic fault occurring increases and with it the probability of a real-time bug manifesting.

The above heuristics of the WNSTT are illustrated with an example in Figure 86. Barriers are installed before messages which entail maximum traffic (among all messages of a DCCFP).

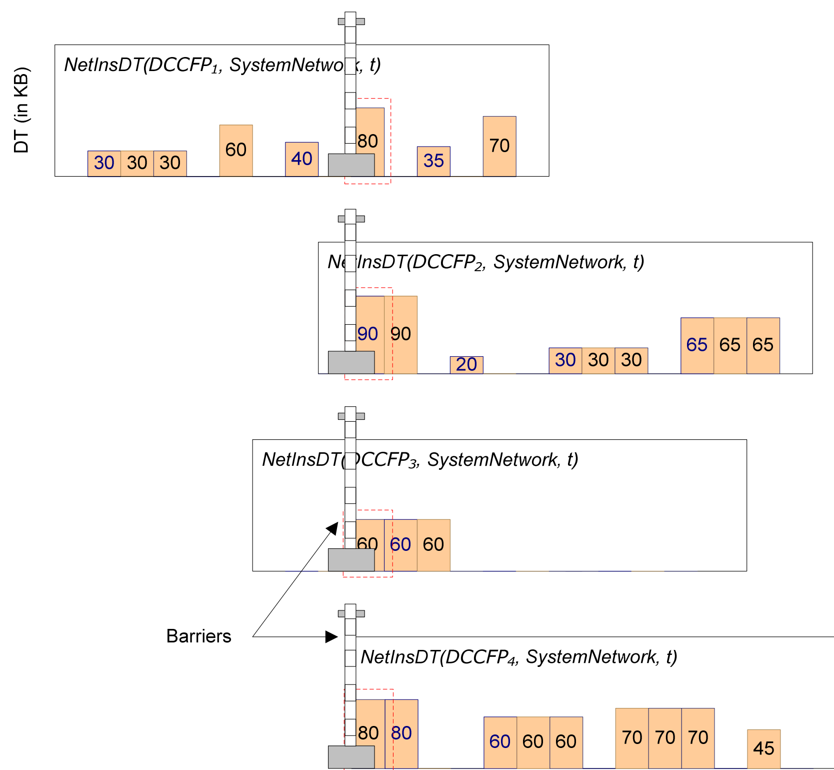


Figure 86-Heuristics of the Wait-Notify Stress Test Technique (WNSTT).

Chapter 11

AUTOMATION AND ITS EMPIRICAL ANALYSIS

To improve automation for the two stress test techniques, namely Time-Shifting Stress Test Technique (*TSSTT*) in Chapter 9, and *Genetic Algorithm-based Stress Test Technique (GASTT) in* Chapter 10, we implemented a prototype tool, referred to as *GARUS (GA-based test Requirement tool for real-time distribUted Systems)*. Note that GARUS supports both GASTT and TSSTT. Although it is primarily implemented for GASTT, it can be used for TSSTT as well. This is done by simply specifying that none of the SDs of a SUT have arrival patterns. This will be discussed in detail in Section 11.2.

We used GAlib [99], an open source C++ library for GAs. An overview of GAlib is presented in Section 11.1. Section 11.2 describes our tool. Section 11.3 reports how we validated the test requirements generated by GARUS and the efficiency and effectiveness of our GA through empirical means.

11.1 GAlib

The library used to implement our GA-based tool was GAlib [99]. GAlib was developed by Matthew Wall at the Massachusetts Institute of Technology. GAlib is a library of C++

objects. The library includes tools for implementing genetic algorithms to do optimization in any C++ program using any chromosome representation and any genetic operators. The library has been tested on multiple platforms, specifically DOS/Windows, MacOS and UNIX. It can also be used with parallel virtual machines to evolve populations in parallel on multiple CPUs.

Figure 87 illustrates the basic GAlib class hierarchies. Only the major classes of the library are shown. For complete class listing, the reader is referred to [99].

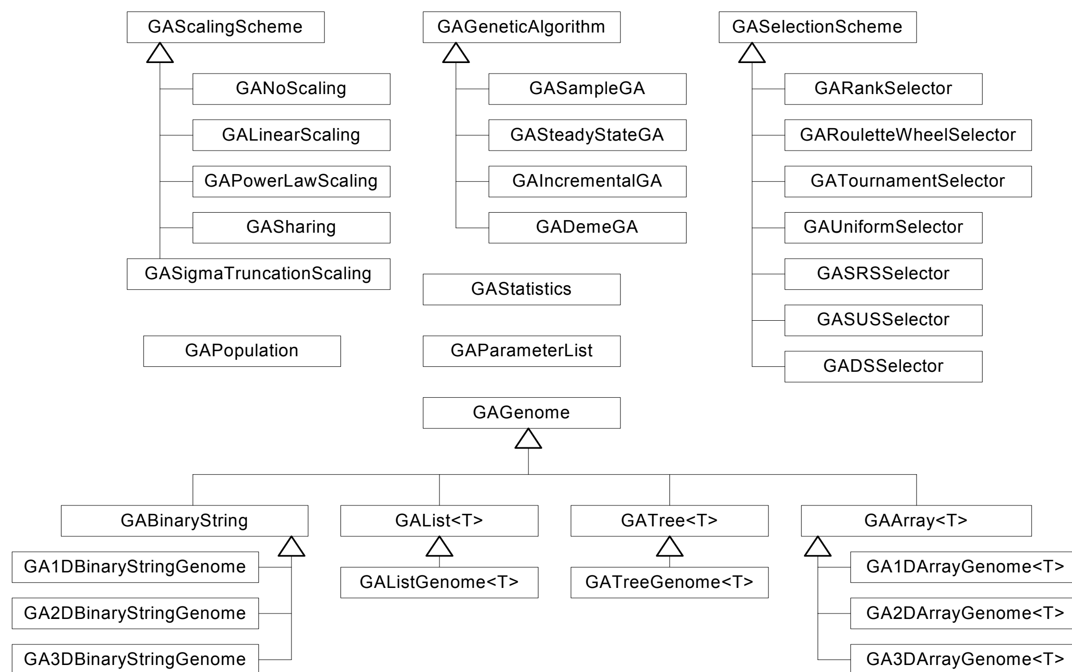


Figure 87-Basic GAlib class hierarchy (adopted from [99]).

GAlib defines many options. It supports four types of genetic algorithms: simple, steady state, incremental and deme. The former three types are described in Appendix A. The deme genetic algorithm evolves multiple populations in parallel using a steady state algorithm. During each population, some individuals are migrated between the populations [99]. GAlib also supports various selection methods for choosing an

individual for mutation and crossover. These include rank selection, roulette wheel, tournament, stochastic remainder sampling (SRS), stochastic uniform sampling (SUS) and deterministic sampling (DS).

11.2 GARUS[9]

GARUS (GA-based test Requirement tool for real-time distribUted Systems) is our prototype tool for deriving stress test requirements. Section 11.2.1 presents the class diagram of GARUS. The overview activity diagram of GARUS is described in Section 11.2.2. The input/output file formats are presented in Section 11.2.3 and Section 11.2.4, respectively.

11.2.1 Class Diagram

The simplified class diagram of GARUS is shown in Figure 88. The classes in the class diagram are grouped in three packages: *TestModelGenerator*, *TestModel* and *GA*.

To simplify the implementation of GARUS, we assume that a TM has already been built from a given UML model and a set of test parameters by a test model generator (the *TestModelGenerator* package). The TM is also assumed to be filtered by the given set of test parameters. For example, if test parameters are for a *StressTestNetInsDT* test strategy

---

[9] The source code of the tool, as well as sample input/output files for several systems under test, a random input model generator, and the data files for empirical GA validations of the tool are available from the World Wide Web [100] V. Garousi, "GARUS (Genetic Algorithm-based test Requirement tool for real-time distribUted Systems)," in *http://squall.sce.carleton.ca/tools/GARUS*, 2006..

over a network *net*, all DCCFPs in the CFM and network usage pattern parts of a TM are assumed to have been filtered by that particular network. Thus, we would ideally have a package in GARUS that handles this. However, to simplify the implementation of GARUS, the package is currently bypassed (the filtered TM is built manually from a UML model).

The classes in the *TestModel* package store information about the test model of a SUT. The *GA* package includes the GA domain-specific classes, which solve the optimization problem and derive stress test requirements.

One object of class *TestModel* and one object of class *GASteadyState GA* are instantiated at runtime for a SUT. The connection between the two packages is achieved via class DCCFP (in the *TestModel* package) and class *GARUSGene* (in the *GA* package).

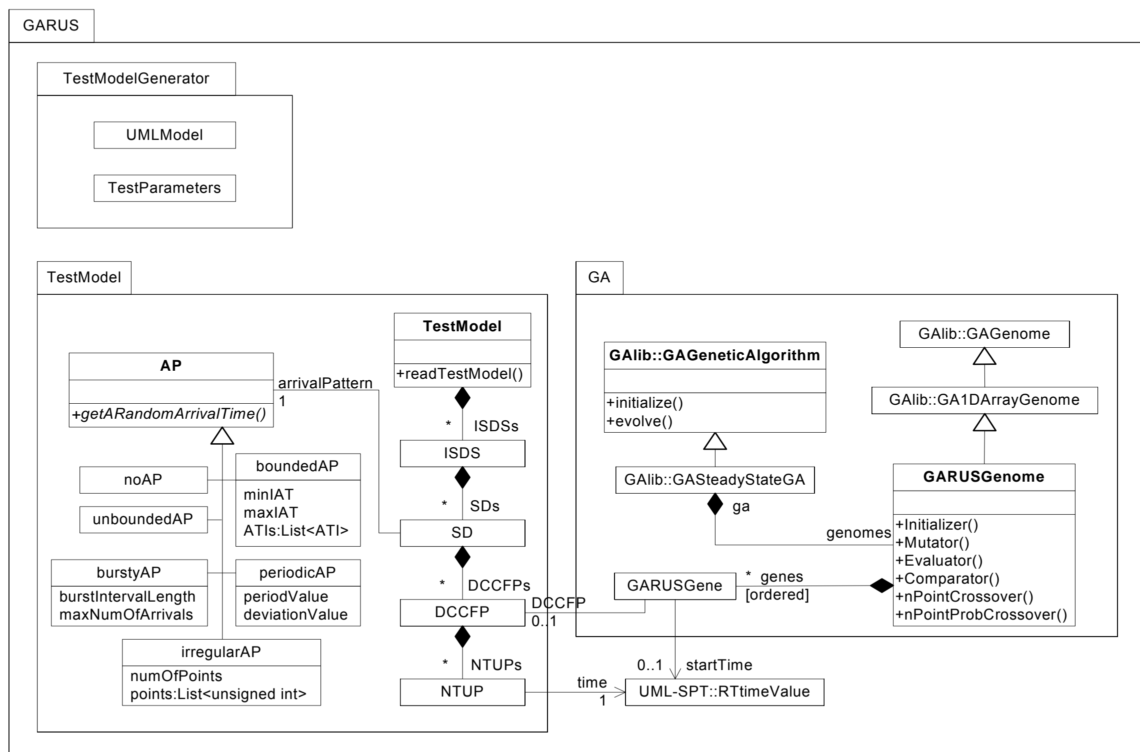


Figure 88-Simplified class diagram of GARUS.

Abstract class *AP* in the *TestModel* package realizes the implementation of arrival patterns. Six subclasses are inherited from class *AP*, five of which correspond to the five types of arrival patterns (Section 10.1). Objects of type class *noAP* are associated with SDs which have no arrival patterns. Due to the implementation details, this choice was selected instead of setting the *arrivalPattern* association of such SDs to *null*. Function *getARandomArrivalTime()* is used in the mutation operator of GARUS (*Mutation()* in class *GARUSGenome*) and, for each subclass of *AP*, it returns a random arrival time in the corresponding ATS (Section 10.3) according to the type of arrival pattern.

11.2.2 Activity Diagram

The overview activity diagram of GARUS is presented in Figure 89. The test model of a SUT is given in an input file. GARUS reads the test model from the input file and creates an object named *tm* of type *TestModel*, initialized with the values from the input test model. Then, an object named *ga* of type *GAlib::SteadyStateGA* is created, such that *tm* is used in the creation of *ga*'s initial population (Section 10.7.3). Note that object *ga* has a collection of chromosomes of type *GARUSGenome,* and each object of type *GARUSGenome* has an ordered set of genes of type *GARUSGene* (refer to the class diagram in Figure 88). Furthermore, *ga*'s parameters (e.g. mutation rate) are set to the values as discussed in Section 10.7.

GARUS then evolves *ga* using the overloaded GA mutator and crossover operators (Section 10.7.6). When the evolution of *ga* finishes, the best individual (accessible by `ga.statistics().bestIndividual()`) is saved in the output file (see Section 11.2.4).

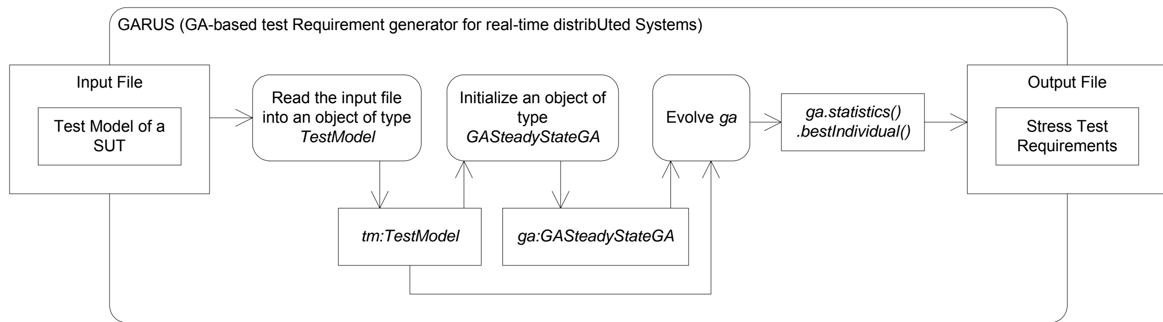


Figure 89-Overview activity diagram of GARUS.

11.2.3 Input File Format

The input file provided to GARUS contains the test model (TM) of a SUT. As it was shown in Figure 10, a TM consists a CFM (including DCCFPs), inter-SD constraint (ISDSs) and distributed traffic usage patterns.

Referring to Figure 10, stress test parameters are also part of the input. As discussed in Chapter 9, stress test parameters are in fact the type of stress test technique (e.g. *StressTestNetInsDT* and *StressTestNodInIntMT*) and a set of parameters specific to the technique (e.g. a node name and a period's start/end times for the *StressTestNodInIntMT* stress test technique). Furthermore, as it was discussed in the algorithms and equations in Chapter 8, a test model can be filtered based on different attributes discussed in distributed traffic usage analysis (e.g. location, direction, and period).

The input file is in a format to accommodate a *filtered* TM. For example, if test parameters are for a *StressTestNetInsDT* test strategy over a network *net*, all DCCFPs in the CFM and network usage pattern parts of a TM are assumed to have been filtered by that particular network. The input file format consists of several blocks, each specifying

different elements of a TM. GARUS input file format is shown using the BNF in Figure 90.

The input file format can be best described using an example. An example input file is shown in Figure 91. Different blocks are separated with a gray highlight. The TM starts with a block of two ISDSs *ISDS0* and *ISDS1* (*ISDSsBlock* in Figure 90). For example, *ISDS0* consists of three SDs: *SD0*, *SD1*, and *SD2*.

The second block of the input file shows SDs (*SDsBlock* in Figure 90). There are five SDs: *SD0,…, SD4*. Each SD line consists of a SD name, number of concurrent multiple instances allowed, followed by the number of its DCCFPs and their names. For example *SD2* has two DCCFPs named *p21* and *p22*.

$$inputFileFormat ::= ISDSsBlock\ \ SDsBlock\ \ SDAPsBlock\ \ DCCFPsBlock$$
$$ISDSsBlock ::= nISDSs\ \ ISDS_1 \ldots ISDS_{nISDSs}$$
$$ISDS_i ::= ISDSName_i\ \ nSDsInISDS_i\ \ SDName_1 \ldots SDName_{nSDsInISDS_i}$$
$$SDsBlock ::= nSDs\ \ SD_1 \ldots SD_{nSDs}$$
$$SD_i ::= SDName_i\ nMultipleInstances_i\ \ \ nDCCFPsInSD_i\ \ DCCFPName_1 \ldots DCCFPName_{nDCCFPsInSD_i}$$
$$SDAPsBlock ::= SDAP_1 \ldots SDAP_{nSDs}$$
$$SDAP_i ::= SDName_i\ \ APType_i\ \ APParameters_i$$
$$APType_i ::= no_arrival_pattern \mid periodic \mid bounded \mid irregular \mid bursty \mid unbounded$$
$$APParameters_i ::= \begin{cases} \in & ;\text{if } APType_i \in \{no_arrival_pattern, bursty, unbounded\} \\ periodValue_i\ \ deviationValue_i & ;\text{if } APType_i = periodic \\ minIAT_i\ \ maxIAT_i & ;\text{if } APType_i = bounded \\ nArrivalPointsInAP_i\ \ APoint_1 \ldots APoint_{nArrivalPointsInAP_i} & ;\text{if } APType_i = irregular \end{cases}$$
$$DCCFPsBlock ::= \underbrace{DCCFP_1 \ldots DCCFP_{nDCCFPs}}_{nDCCFPs = \sum_{\forall SD_i} nDCCFPsInSD_i}$$
$$DCCFP_i ::= DCCFPName_i\ \ nDTUPPsInDCCFP_i\ \ DTUPP_1 \ldots DTUPP_{nNTUPPsInDCCFP_i}$$
$$DTUPP_i ::= (\ time_i\ \ value_i\ )$$
$$GAMaxSearchTime$$

Figure 90-GARUS input file format.

The third block shows SD Arrival Pattern (AP) - (*SDAPsBlock* in Figure 90). Each line in this block consists of a SD name, followed by its AP type and a set of parameters specific to that AP type. For example, *SD1* has a periodic arrival pattern. The period and deviation

values of this periodic arrival pattern are 4 and 2 units of time. Note that units for all time values in an input file are assumed to be the same, and hence they are not specified. It is up to a user to interpret the unit of time. If the AP of a SD is bounded, the minimum and maximum inter-arrival time (*minIAT*, *maxIAT*) are specified. In case when a SD has no arrival pattern (`no_arrival_pattern` keyword), or it is bursty or unbounded, no additional parameters need to be specified. This is because such APs do not impose any timing constraints in our stress test requirement generation technique. Refer to Sections 10.2 and 10.5 for further details.

The next block in an input file is the *DCCFPsBlock*. The number of DCCFPs in a *DCCFPsBlock*, is equal to the sum of DCCFPs of all SDs, specified in the *SDsBlock*. For example, in the example input file in Figure 91, this total is equal to: 5 (*SD0*) + 3 (*SD1*) + 2 (*SD2*) + 1 (*SD3*) + 4 (*SD4*)=15. All 15 DCCFPs have been listed, each following by its DTUP (Distributed Traffic Usage Pattern). The format for specifying DTUP of a DCCFP is described next. As discussed in Section 8.5, the DTUP of a DCCFP (with a fixed traffic location, direction and type) is a 2D function where the Y-axis is the traffic value and the X-axis is time. The non-zero values of a DTUP are specified in an input file. Each such value is specified by a pair consisting of the corresponding time and traffic values, and is referred to as a *DTUPP (Distributed Traffic Usage Pattern Point)*. For example, DTUPPs of the DTUP in Figure 92 are: (1, 90), (3, 40), (4, 40), (8, 30), and (12, 50). For example, in the input file in Figure 91, p41 has two DTUPPs: (4, 20) and (7, 4). The “,” symbol between time and traffic values is eliminated in the input file to ease the parsing process.

```
2
ISDS0 3 SD0 SD1 SD2
ISDS1 4 SD0 SD2 SD3 SD4
5
SD0 1 5 p01 p02 p03 p04 p05
SD1 1 3 p11 p12 p13
SD2 1 2 p21 p22
SD3 1 1 p31
SD4 1 4 p41 p42 p43 p44
SD0 periodic 5 0
SD1 periodic 4 2
SD2 bounded 4 5
SD3 no_arrival_pattern
SD4 irregular 5 2 3 6 8 9
p01 5 ( 2 10 ) ( 3 5 ) ( 6 7 ) ( 12 20 ) ( 15 9 )
p02 2 ( 1 5 ) ( 4 20 )
p03 3 ( 3 5 ) ( 5 10 ) ( 6 7 )
p04 2 ( 3 9 ) ( 6 35 )
p05 1 ( 5 40 )
p11 2 ( 4 4 ) ( 7 3.4 )
p12 3 ( 1 1 ) ( 2 9 ) ( 5 6 )
p13 5 ( 2 3 ) ( 5 4 ) ( 7 1 ) ( 9 6 ) ( 11 20 )
p21 1 ( 4 30 )
p22 4 ( 2 20 ) ( 3 10 ) ( 7 15 ) ( 9 30 )
p31 3 ( 3 3 ) ( 5 9 ) ( 7 20 )
p41 2 ( 4 20 ) ( 7 4 )
p42 6 ( 2 3 ) ( 5 6 ) ( 8 8 ) ( 10 1 ) ( 12 9 ) ( 15 10 )
p43 5 ( 4 2 ) ( 6 7 ) ( 10 5 ) ( 12 3 ) ( 15 2 )
p44 2 ( 4 32 ) ( 6 10 )
25
```

Figure 91-An example input file of GARUS.

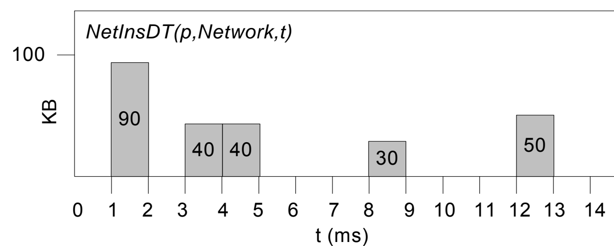


Figure 92-An example DTUP of a DCCFP.

The last piece of information in the input file is a Real value, referred to as *GAMaxSearchTime* (Section 10.7.4). This value (in time units) specifies the range (from zero) in which the GA tries to search for a best result. The initialization and the mutation operators use this maximum search value to select a random start time for the DCCFP of

a gene. For example, the *GAMaxSearchTime* in the input file in Figure 91 is 25 time units. Therefore, the GA operators in GARUS will choose random seeding times for DCCFPs in the range of [0…25] time units. The higher the *GAMaxSearchTime* value, the more broad the GA's search range. However, we expect that higher *GAMaxSearchTime* values deteriorates our GA's performance in converging, since the higher the *GATimeSearchRange* value, the less probable that multiple DCCFPs overlap with each other. A suitable *GAMaxSearchTime* can be calculated using the two heuristics we presented in Section 10.7.4. However, to allow variability of choices for *GAMaxSearchTime* in our experimentation, we assume that such a time instant has been calculated by a tester and is given in the input file.

11.2.4 Output File Format

GARUS exports the stress test requirements to an output file, whose name is specified in the command line. If no output file name is given by the user, the output is simply printed on the screen. Furthermore, the output file also contains standard GAlib statistics, including the numbers of selections, crossovers, mutations, replacements and genome evaluations since initialization, as well as min, max, mean, and standard deviation of each generation. The main output is the stress test requirements, while GA statistics are just informative values for debugging purposes. The format of stress test requirements in an output file is shown in Figure 93-(a). An example set of stress test requirements is presented in Figure 93-(b), which is generated by GARUS for the input file in Figure 91.

SD	DCCFP	start time
$SDName_1$	$SelectedDCCFPName_1$	$startTime_1$
…	…	…
$SDName_{nSDs}$	$SelectedDCCFPName_{nSDs}$	$startTime_{nSDs}$

ISTOF = a float value
Max. stress time = an integer value

(a)

SD	DCCFP	start time
SD0	p04	10
SD1	p12	14
SD2	p21	12
SD3	none	
SD4	none	

ISTOF=74
Max stress time=16

A stress test schedule

(b)

Figure 93-(a): Stress test requirements format in GARUS output file. (b): An example.

The first block of the output file is a *stress test schedule* which, if executed, entails maximum traffic. Each line in the first block of the output file corresponds to a SD of the SUT, and specifies a selected DCCFP with a start time to trigger. For example, Figure 93-(b) indicates that *p04* of *SD0*, *p12* of *SD1*, and *p21* of *SD2* should be triggered at start times 10, 14 and 12 unit of time, respectively. No DCCFP has been specified to be triggered for *SD3* and *SD4*. This is because a set of stress test requirements corresponds to an ISDS in a SUT, and as shown in Figure 91, the SUT we used for these results has two ISDSs and *SD0*, *SD1* and *SD2* are members of one of them. In other words, triggering all SDs *SD0 …SD4* is not allowed in this SUT. Note that GARUS never schedules a DCCFP in a start time which is not allowed according to SDs' arrival patterns.

11.3 Validation of Test Requirements Generated by GARUS

Along with a stress test requirement, GARUS also generates a maximum traffic value and a maximum traffic time. The maximum traffic value is in fact the objective function value of the GA's best individual at the completion of the evolution process. The

objective function was described in Section 10.7.4, and was referred to as *Instant Stress Test Objective Function (ISTOF)*. The maximum traffic time is the time instant when the maximum traffic happens. For example the ISTOF value and maximum traffic time for the SUT specified by the input file in Figure 91 are 74 (unit of traffic, e.g. KB) and 16 (unit of time, e.g. ms), respectively.

Test requirements generated by GARUS can be validated according to at least six criteria:

1. *Satisfaction of ATSs by start times of DCCFPs in the generated stress test requirements* (Section 11.3.1): As explained in Section 10.7, each chromosome (including the final best chromosome) should satisfy this constraint, i.e., the start times of each DCCFP in the final best chromosome of the GA should be inside the Accepted Time Set (ATS) of its corresponding SD.

2. *Checking ISTOF values* (Section 11.3.2): As a heuristic, GAs do not guarantee to yield optimum results, and checking that the ISTOF value of the final best chromosome is the maximum possible traffic value among all interleavings is a NP-hard problem. It is, therefore, not possible to fully check how optimal GA results are. However, simple checks can be done to determine if, for example, GARUS has been able to choose the DCCFP with maximum traffic value among all DCCFPs in a SD.

3. *Repeatability of GA results across multiple runs* (Section 11.3.3)*:* It is important to assess how stable and reliable the results of the GA will be. To do so, the GA is executed a large number of times and we assess the variability of the average or best chromosome's fitness value.

4. *Convergence efficiency across generations towards a maximum* (Section 11.3.4): In order to assess the design of the selected mutation and cross-over operators, as well as the chosen chromosome representation, it is useful to look at the speed of convergence towards a maximum fitness plateau [101]. This can be measured, for example, in terms of number of generations required to reach the plateau. This can be easily computed as, for each generation, GAlib statistics provide min, max, mean, and standard deviation values.

5. *Impacts of variations in test model size (scalability of the GA)* - (Section 11.3.6): Assessing how The GA performance and its repeatability are affected with different test model sizes.

6. *Impacts of variations in parameters other than test model size* - (Sections 11.3.7-11.3.9): Assessing how the GA performance and its repeatability are affected when it is applied to different test models with different properties. In the current work, we investigate the impacts of variations in arrival pattern types (Section 11.3.7), arrival pattern parameters (such as periodic arrival pattern period and deviation, and bounded arrival pattern minimum and maximum inter-arrival time values) (Section 11.3.8), and GA maximum search time (Section 11.3.9) on the GA results and on its repeatability aspect, respectively.

Using the above six criteria, we analyze the stress test requirements generated by running GARUS on a set of *experimental* test models, which were designed to test the repeatability and scalability aspects of our GA. We discuss in Section 11.3.5 how we designed the set of the experimental test models, which will be used in the rest of this chapter as a test-bed for our experiments and validations.

11.3.1 Satisfaction of ATSs by Start Times of DCCFPs in the Generated Stress Test Requirements

We check whether the start times of the DCCFPs in the generated stress test requirements satisfy the ATSs of the corresponding SDs. In order to investigate this, we first derive the ATSs of the SDs in the test model corresponding to the input file in Figure 91. Consistent with discussions in Section 10.3, they are shown in Figure 94.

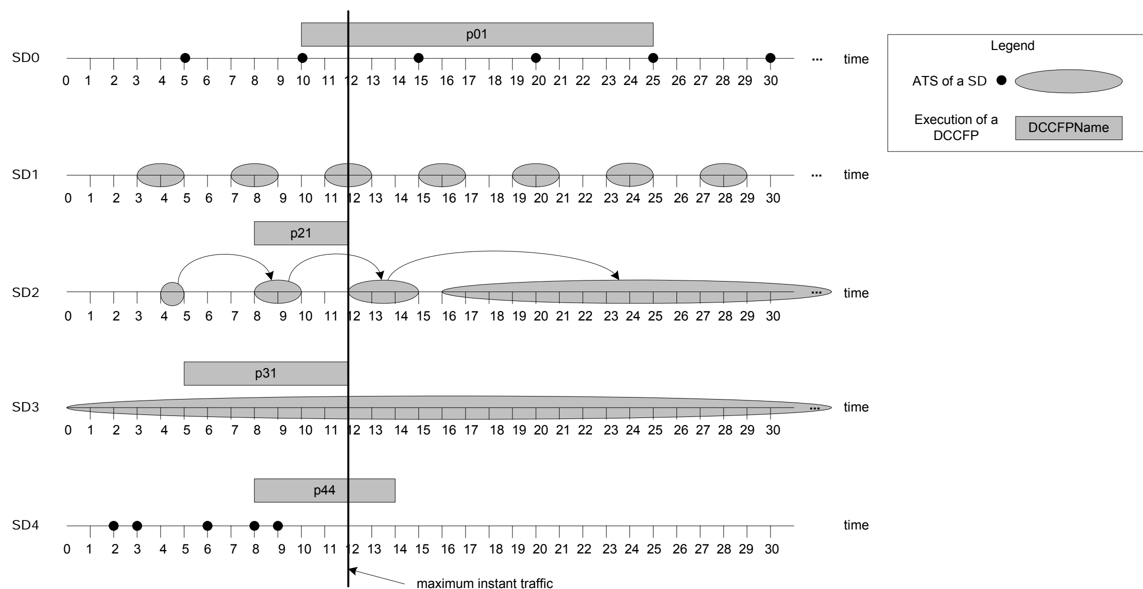


Figure 94-ATSs of the SDs in the TM in Figure 91, and a stress test schedule generated by GARUS.

For example, as *SD0* has a periodic AP with period value=5 and zero deviation, its ATS comprises time instants 5, 10, 15 and so on. Since *SD3* has no AP, therefore its ATS includes all the time instants from zero to infinity. As an example, the stress test schedule generated by run number 2 in Table 10 has been depicted in Figure 94. This stress test schedule includes *p01* from *SD0*, no DCCFPs from *SD1*, *p21* from *SD2*, *p31* from *SD3*, and *p44* from *SD4* to be triggered on time instants 10, 8, 5, and 8, respectively. The time

instant when the maximum traffic occurs (time=12) is depicted with a vertical bold line. The ISTOF value at this time is 92 units of network traffic.

As it can be seen in Figure 94, the start times of all selected DCCFPs in the stress test schedule reside in the ATSs of the respective SDs. This is explained by the way the initial population of chromosomes is created (Section 10.7.3) and the allowability property of our mutation operator (Section 10.7.6.2). The start time of each DCCFP is always chosen from the ATS of its corresponding SD. This is achieved by building the ATS of each SD according to its type of AP when GARUS initializes a test model. Then, when a random start time is to be chosen for a DCCFP, method *getARandomArrivalTime()*, which is associated with a SD is invoked on an object from a subclass of the abstract class AP. Refer to Figure 88 for details.

11.3.2 Checking the extent to which ISTOF is maximized

As a test to check if GARUS is able to choose the DCCFP with maximum traffic value among all DCCFPs of a SD, we artificially modify DTUPs of the DCCFPs in the test model of Figure 91 such that one DCCFP of each SD gets a much higher peak value in its DTUP. The modified values are shown in bold in Figure 95.

For example, the DTUP value of *p03* at time=5 was 10, whereas its new value is 500. This value is an order of magnitude larger than all other DTUP values of other DCCFPs in *SD0*. We then run GARUS with this modified TM for a large number of times and see if the DCCFPs with high DTUP values are part of the output stress test schedule generated by GARUS.

We executed GARUS 10 times with this TM, and the 10 schedules generated by GARUS had the format described in Table 8, where *x* stands for values which changed across different runs. As expected, DCCFPs *p21*, *p31*, and *p42* were present in all 10 stress test schedules, thus suggesting that GARUS selects the correct DCFFPs. On the other hand, different DCCFPs from *SD0* were reported in the output schedules. This can be explained as *SD0*'s ATS contains specific time points (5, 10, 15, and so on) and *p03* (the modified DCCFP) will therefore not be able to have an effect on the maximum possible instant traffic (at time=16 or 17) since its modified DTUP point is at time=5.

```
--DCCFPs
p01 5 ( 2 10 ) ( 3 5 ) ( 6 7 ) ( 12 20 ) ( 15 9 )
p02 2 ( 1 5 )( 4 20 )
p03 3 ( 3 5 )( 5 500 ) ( 6 7 )
p04 2 ( 3 9 )( 6 35 )
p05 1 ( 5 40 )
p11 2 ( 4 4 )( 7 3.4 )
p12 3 ( 1 1 )( 2 900 ) ( 5 6 )
p13 5 ( 2 3 )( 5 4 ) ( 7 1 ) ( 9 6 ) ( 11 20 )
p21 1 ( 4 300 )
p22 4 ( 2 20 ) ( 3 10 ) ( 7 15 ) ( 9 30 )
p31 3 ( 3 3 )( 5 9 ) ( 7 700 )
p41 2 ( 4 20 ) ( 7 4 )
p42 6 ( 2 3 )( 5 6 ) ( 8 800 ) ( 10 1 ) ( 12 9 ) ( 15 10 )
p43 5 ( 4 2 )( 6 7 ) ( 10 5 ) ( 12 3 ) ( 15 2 )
p44 2 ( 4 32 ) ( 6 10 )
```

Figure 95-Modified DCCFPs of the test model in Figure 91.

SD	DCCFP	Start Time
SD0	x	x
SD1	none	
SD2	p21	x
SD3	p31	x
SD4	p42	x

ISTOF=1500 or 1520
Max stress time=16 or 17

Table 8-Output format of 10 schedules generated by GARUS.

The reason why *p12* (from *SD1*) is not selected in any of the outputs across different runs is that a set of DCCFPs are generated by GARUS as a stress test schedule only if the SDs corresponding to the DCCFPs belong to one ISDS. The set of SDs {*SD0, SD1, SD2, SD3, SD4*} does not belong to an ISDS. Furthermore, among all ISDSs (*ISDS0*={*SD0*, *SD1*, *SD2*} and *ISDS1*={*SD0*, *SD2*, *SD3*, *SD4*}) of the test model, the maximum instant traffic of *ISDS1* has a larger value than that of *ISDS0,* thus not letting *SD1* (and all of its DCCFPs) play a role in the output stress test schedules.

11.3.3 Repeatability of GA Results across Multiple Runs

Since GAs are heuristics, their performance and outputs can vary across multiple runs. We refer to such a GA property as *repeatability* of GA results. This property is investigated by analyzing maximum ISTOF values as they are the fitness values of chromosomes in our GA. We furthermore analyze maximum stress time values as such a time value denotes the time instant when a stress situation actually occurs.

Figure 96-(a) depicts the distributions of maximum ISTOF and stress time values for 1000 runs of test model corresponding to the input file in Figure 91. From the ISTOF distribution, we can see that the maximum fitness values for most of the runs are between 60 and 72 units of traffic. Descriptive statistics of the fitness values are shown in Table 9. Average and median values are very close, thus indicating that the distribution is almost symmetric.

Min	Max	Average	Median	Standard Deviation
50	92	66.672	65	6.4

Table 9-Descriptive statistics of the maximum ISTOF values over 1000 runs. Values are in units of data traffic (e.g. KB).

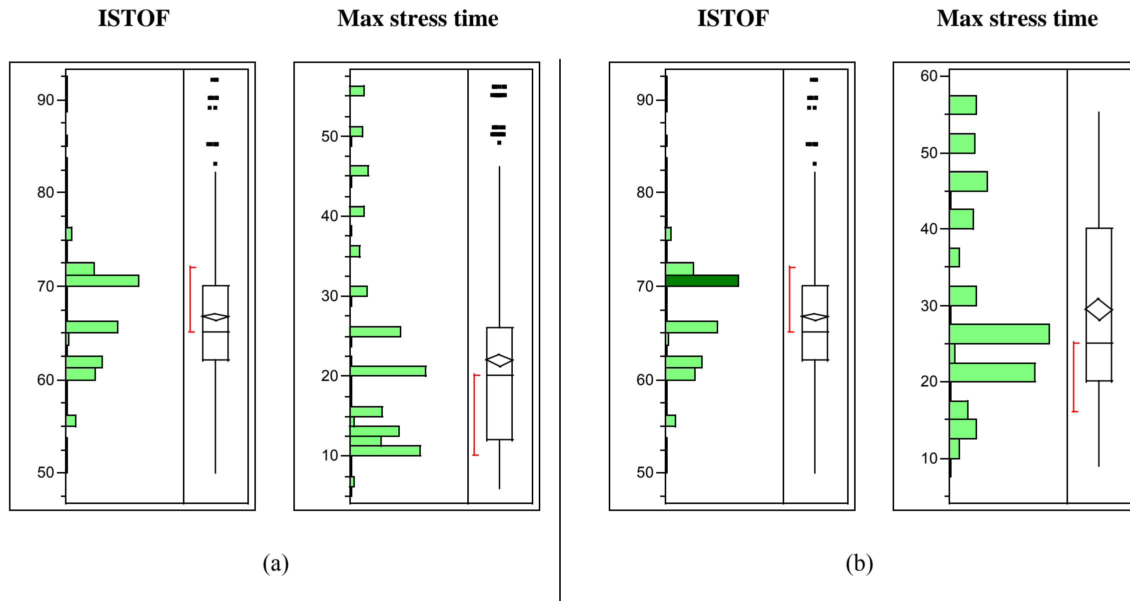


Figure 96-(a): Histogram of maximum ISTOF and stress time values for 1000 runs of test model corresponding to the input file in Figure 91. (b): Corresponding max stress time values for one of the frequent maximum ISTOF values (72 units of traffic).

Such a variation in fitness values across runs is expected when using genetic algorithms on complex optimization problems. However, though the variation above is not negligible, one would expect based on Figure 96-(a) that with a few runs a chromosome with a fitness value close to the maximum would likely be identified. Since each run lasts a few seconds, perhaps a few minutes for very large examples, relying on multiple runs to generate a stress test requirement should not be a practical problem.

Corresponding portions of max stress time values for the most frequent maximum ISTOF value (72 units of traffic) have been highlighted in black in Figure 96-(b). As we can see, these maximum stress time values are scattered across the time scale (e.g., from 10 to 60 units of time). This highlights that a single ISTOF value (maximum stress traffic) can happen in different time instants, thus suggesting the search landscape for the GA is

rather complex for this type of problem. Thus, a strategy to further explore would be for testing to cover all (or a subset of) such test requirements with maximum ISTOF values in different time instants. Indeed, although their ISTOF value are the same, a SUT's reaction to different test requirements might vary, since a different DCCFP (and hence set of messages) in a different stress time instant may be triggered. This might in turn lead to uncovering different RT faults in the SUT.

11.3.4 Convergence Efficiency across Generations

Another interesting property of the GA is the number of generations required to reach a stable maximum fitness plateau. The distribution of these generation numbers over 1000 runs of test model corresponding to the input file in Figure 91 is shown in Figure 97, where the x-axis is the generation number and the y-axis is the probability of achieving such plateau in a generation number. The minimum, maximum and average values are 20, 91, and 52, respectively. Therefore, we can state that, on the average, 52 generations of the GA are required to converge to the final result (stress test requirement). The variation around this average is limited and no more 100 generations will be required. This number is in line with the experiments reported in the GA literature [87] but is however likely to be dependent on the number and complexity of SDs as well as their ATSs.

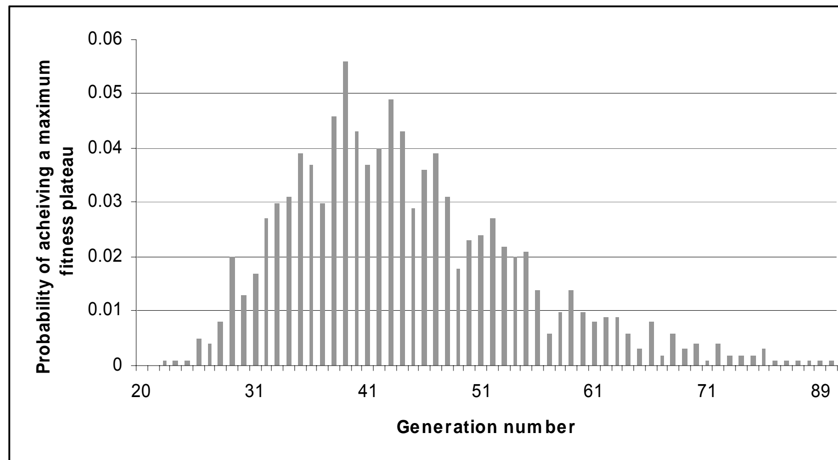


Figure 97-Histogram of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of the test model corresponding to the input file in Figure 91 by GARUS.

From the initial to the final populations in the test model corresponding to the input file in Figure 91, the maximum fitness values typically increase by about 80%, which can be considered a large improvement. So, though we cannot guarantee that a GA found the global maximum, we clearly generate test requirements that will significantly stress the system.

The variability in the objective function and start times as well as detailed information for the first five runs of the test model corresponding to the input file in Figure 91 are reported in Table 10. We can clearly see from the table that in a single run when the generation number increases, the population converges (i.e. deviation value decreases).

For example, in the outputs reports in Table 10, the value of the *deviation*[10] column decreases continuously from 8.95 at generation #0 to 0.00 in generation #80. Also notice the convergence of minimum and mean values towards the maximum (ISTOF) values when the generation number increases.

Run #	**Generation #**	**Mean**	**Max (ISTOF)**	**Min**	**Deviation**	**Best individual**		
1	0	36.74	55	30	8.95	SD	DCCFP	start time
	10	44.47	58	38	7.04	----	----	----------
	20	52.46	61	41	6.44	SD0	p05	25
	30	61.14	66	55	5.86	SD1	none	
	40	67.23	71	61	4.90	SD2	p22	21
	50	71.43	79	70	4.21	SD3	p31	23
	60	74.62	85	70	3.01	SD4	p42	2
	70	82.03	88	72	2.95			
	80	90.00	90	90	0.00	ISTOF=90		
	90	90.00	90	90	0.00	Max stress time=30		
	100	90.00	90	90	0.00			
2	0	36.45	58	30	8.82	SD	DCCFP	start time
	10	43.84	60	36	7.13	----	----	----------
	20	51.23	65	41	6.97	SD0	p01	10
	30	57.82	66	50	5.45	SD1	none	
	40	64.70	73	59	5.27	SD2	p21	8
	50	72.52	76	62	5.04	SD3	p31	5
	60	80.50	82	80	3.39	SD4	p44	8
	70	81.34	84	80	3.78			
	80	83.78	86	80	2.58	ISTOF=92		
	90	91.64	92	80	2.05	Max stress time=12		
	100	92.00	92	92	0.00			

[10] *Deviation* of a population in GAlib is defined as the standard deviation of individuals' objective scores [87] R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms*: Wiley-Interscience, 1998..

3	0	36.93	49	30	7.98	SD	DCCFP	start time
	10	45.36	50	39	7.94	----	----	----------
	20	54.05	58	44	7.63	SD0	p04	15
	30	62.35	68	52	6.93	SD1	none	
	40	70.01	72	65	3.46	SD2	p22	19
	50	73.63	74	72	1.49	SD3	p31	14
	60	75.00	75	75	0.00	SD4	p44	9
	70	75.00	75	75	0.00			
	80	75.00	75	75	0.00	ISTOF=75		
	90	75.00	75	75	0.00	Max stress time=21		
	100	75.00	75	75	0.00			
4	0	37.03	53	30	8.94	SD	DCCFP	start time
	10	45.37	58	37	8.14	----	----	----------
	20	55.14	60	43	7.21	SD0	p05	15
	30	66.63	69	52	7.08	SD1	none	
	40	73.29	78	70	6.22	SD2	p22	18
	50	79.02	80	72	2.62	SD3	p31	13
	60	80.00	80	80	0.00	SD4	p43	9
	70	80.00	80	80	0.00			
	80	80.00	80	80	0.00	ISTOF=80		
	90	80.00	80	80	0.00	Max stress time=20		
	100	80.00	80	80	0.00			
5	0	37.54	55	30	8.44	SD	DCCFP	start time
	10	45.60	58	39	7.50	----	----	----------
	20	54.09	64	48	6.93	SD0	p05	5
	30	61.67	66	52	6.32	SD1	none	
	40	68.42	69	65	2.52	SD2	p21	12
	50	70.37	71	70	0.78	SD3	p31	11
	60	71.14	72	70	0.99	SD4	p44	6
	70	72.00	72	72	0.00			
	80	72.00	72	72	0.00	ISTOF=72		
	90	72.00	72	72	0.00	Max stress time=10		
	100	72.00	72	72	0.00			

Table 10-Summary of GARUS results for five runs.

11.3.5 Our Strategy for Investigating Variability/Scalability

To assess and validate test requirements generated by GARUS, we design a set of different test models (referred to as *experimental* test models), and execute GARUS with each of them as input. Note that these test models are in the input file format, described in Section 11.2.3. To ensure variability in the experimental test models, a set of experimental test models were designed based on the following criteria:

- Test models with variations in test model sizes (Section 11.3.5.3)
- Test models with variations in SD arrival patterns (Section 11.3.5.4)
- Test models with variations in the GA's maximum search time (Section 11.3.5.5)

Since most of the input files containing the test models are large in size, we do not list them in this article. As an example, the contents of one input file corresponding to one of the test model in this section are shown in Appendix B. We discuss next a set of *variability parameters*, we used in our experiments to incorporate variability in different test models based on the above criteria.

11.3.5.1 Variability Parameters

The variability parameters used in our experiments are listed in Table 11, along with their explanations. The parameters are grouped into three groups corresponding to the above three criteria: (1) Test model size, (2) SD arrival patterns, and (3) Maximum search time.

We have defined eight parameters under the group of 'test model size' to incorporate variability in different sizes perspectives in experimental TMs. Each parameters in this group correspond to a sizes perspective, e.g., number of ISDSs, number of SDs, and

minimum number of DCCFPs per SD. For example, a large TM might have many ISDSs (by setting large values for *nISDSs*), while another large TM can have many DCCFPs per SD (by setting large values for *minnDCCFPs* and *maxnDCCFPs*).

Parameter Group	Parameter	Parameter Explanation
Test model size	*nISDSs*	# of ISDSs
	nSDs	# of SDs in TM
	minISDSsize	min. # of SDs per ISDS
	maxISDSsize	max. # of SDs per ISDS
	minnDCCFPs	min. # of DCCFPs per SD
	maxnDCCFPs	max. # of DCCFPs per SD
	minnDTUPPs	min. # of DTUPPs per DCCFP
	maxnDTUPPs	min. # of DTUPPs per DCCFP
SD arrival patterns (all parameter values are in time units, except *minnAPirregularPoints* and *maxnAPirregularPoints*)	*APtype*	type of AP
	minAPperiodicPeriod	min. period value
	maxAPperiodicPeriod	max. period value
	minAPperiodicDeviation	min. deviation value
	maxAPperiodicDeviation	max. deviation value
	minAPboundedMinIAT	min. value for min. inter-arrival time
	maxAPboundedMinIAT	max. value for min. inter-arrival time
	minAPboundedMaxIATafterMin	min. distance between *maxIAT* and *minIAT*
	maxAPboundedMaxIATafterMin	max. distance between *maxIAT* and *minIAT*
	minnAPirregularPoints	min. # for irregular points
	maxnAPirregularPoints	max. # for irregular points
	minAPirregularPointsValue	min. time value for irregular points
	maxAPirregularPointsValue	max. time value for irregular points
Maximum search time	*MaximumSearchTime*	GA maximum search time

Table 11- Variability parameters for experimental test models.

Parameters prefixed with *min* and *max* serve as statistical means, which enable us to incorporate statistically-controlled randomness into the sizes of our experimental TMs. For example, we can control the minimum and maximum number of DCCFPs per SD in a TM by *minnDCCFPs* and *maxnDCCFPs* parameters. Such a statistical range for number of SDs per ISDSs, DCCFPs per SD, and DTUPPs per DCCFP also conforms to real-

world models, where for example, there are not variant number of DCCFPs per SDs in a TM. Our experimental TMs with variations in TM sizes based on the variability parameters are presented in Section 11.3.5.3.

The second group of the variability parameters (SD arrival patterns) is designed to incorporate variability in different SD arrival pattern properties of our experimental TMs. All parameter values in this group are in time units, except *minnAPirregularPoints* and *maxnAPirregularPoints*. The first parameter in this group (*APtype*) determines the type of APs to be assigned for the SD of an experimental TM. *APtype* is of type enumeration with possible values of:

- *no_arrival_pattern*: All SDs of the TM will have no APs, i.e., any arrival time in test schedules is accepted for all SDs.
- *periodic*: All SDs of the TM will have periodic APs. Each AP's parameters are set to the four **APperiodic** variability parameters.
- *bounded*: All SDs of the TM will have bounded APs. Each AP's parameters are set to the four **APbounded** variability parameters.
- *irregular*: All SDs of the TM will have irregular APs. Each AP's parameters are set to the four **APirregular** variability parameters.
- *mixed*: Different SDs of a TM have different arrival patterns. The type of AP for a SD is chosen from (*no_arrival_pattern*, *periodic*, *bounded*, or *irregular*) with equal probabilities

Parameters with *APperiodic, APbounded* and *APirregular* substring are specific to periodic, bounded and irregular APs, respectively, and specify the range of values to be

set for specific parameters of these AP. For example, *minAPperiodicPeriod*, *maxAPperiodicPeriod*, *minAPperiodicDeviation*, and *maxAPperiodicDeviation* specify min./max. period values, and min./max. deviation values for periodic APs of SDs. If they are set to 5ms, 10 ms, 2ms, and 3ms respectively, the following APs might be generated in an experimental TMs: (*'periodic', (6, ms), (2, ms)),* (*'periodic', (8, ms), (3, ms)),* (*'periodic', (10, ms), (3, ms)),* and (*'periodic', (6, ms), (3, ms))*. Our experimental TMs with variations in SD arrival patterns time based on the parameters in the second group are presented in Section 11.3.5.4.

The third group of the variability parameters (maximum search time) has only one parameter (*MaximumSearchTime*) which is designed to incorporate variability in GA maximum search time (Section 10.7.4) of our experimental TMs. Our experimental TMs with variations in maximum search time based on the *MaximumSearchTime* parameters are presented in Section 11.3.5.5.

11.3.5.2 Random Test Model Generator

To facilitate the generation of experimental TMs based on the variability parameters (Section 11.3.5.1), we developed a random test model generator (*RandTMGen*) in C++. A simplified activity diagram of *RandTMGen* is shown in Figure 98, where the value for variability parameters are provided as input, and an experimental TM is generated (in input file format of Figure 90).

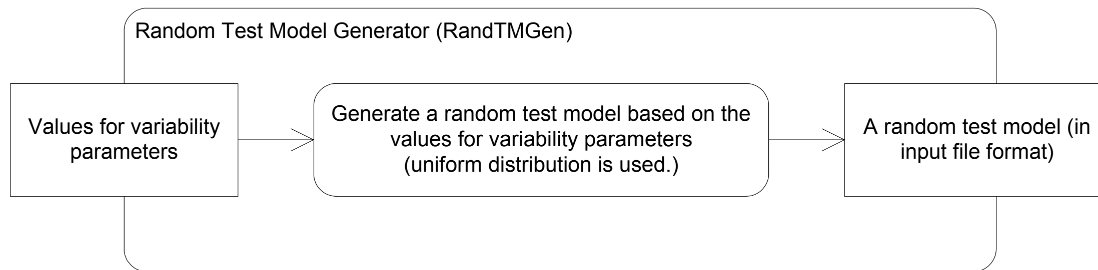


Figure 98-Simplified activity diagram of our random test model generator.

For the pair of parameters specifying min./max. of a feature, e.g. *minnDCCFPs* and *maxnDCCFPs*, *RandTMGen* uses uniform distribution to generate a value in the range of [*minnDCCFPs, maxnDCCFPs*]. This is a design decision, which can be modified easily, i.e., any other distribution can be used to generate a random value in [*minnDCCFPs, maxnDCCFPs*]. As an example on how *RandTMGen* generate a random value in a range, assume *minnDCCFPs*=3 and *maxnDCCFPs*=8. In such a case, all the integer values in the range of [3, 8] are chosen by an equal probability to be set for the number of DCCFPs per a SD in a TM.

In the following three subsections, we describe the parameters provided to *RandTMGen* to generate a set of experimental test models to ensure variability in test models as well as the scalability of GARUS.

11.3.5.3 Test Models with Variations in Sizes

In order to investigate the performance and size scalability of our GA (implemented in GARUS), six test models with different sizes were generated using *RandTMGen* as reported in Table 12.

The SDs of the each of the above test models were assigned *arbitrary (mixed)* arrival patterns (*no_arrival_pattern*, *periodic*, *bounded*, or *irregular*) with equal probabilities.

Depending on the selected arrival pattern type for a SD, the AP variability parameters were set as follows.

Test Models / Parameters	*tm1* (small) Figure 91	*tm2* (many ISDSs)	*tm3* (many SDs in TM)	*tm4* (many SDs per ISDS)	*tm5* (many DCCFPs per SD)	*tm6* (many DTUPs per DCCFP)
nISDSs	2	**100**	10	10	10	2
nSDs	5	50	**200**	50	10	5
minISDSsize	3	2	2	**20**	2	2
maxISDSsize	4	5	5	**30**	5	5
minnDCCFPs	1	1	2	1	**10**	1
maxnDCCFPs	5	3	5	3	**50**	5
minnDTUPPs	2	1	1	1	1	**50**
maxnDTUPPs	6	10	10	10	10	**100**

Table 12-Experimental test models with different sizes.

- Periodic arrival pattern
  - *minAPperiodicPeriod*=5
  - *maxAPperiodicPeriod*=10
  - *minAPperiodicDeviation*=0
  - *maxAPperiodicDeviation*=3
- Bounded arrival pattern
  - *minAPboundedMinIAT*=2
  - *maxAPboundedMinIAT*=4
  - *minAPboundedMaxIATafterMin*=2
  - *maxAPboundedMaxIATafterMin*=5

- Irregular arrival pattern
    - *minnAPirregularPoints*=5
    - *maxnAPirregularPoints*=15
    - *minAPirregularPointsValue*=1
    - *maxAPirregularPointsValue*=30

Note that the above value for the AP variability parameters are chosen to be typical values, as our main intention in designing TMs *tm1*,…, *tm6* is to experiment our GA's behavior and scalability aspect with reaction to different TM sizes.

11.3.5.4 Test Models with Variations in SD Arrival Patterns

To investigate the effect of variations in SD arrival patterns in generated test requirements by GARUS, 12 test models were generated using *RandTMGen* based on two variation strategies as reported in Table 13. The two AP-related variation strategies we followed when generating TMs in this section were:

1. *Different-AP-Types*: Comparing generated test requirements for TMs with different AP types
2. *Same-AP-Different-Parameters:* Comparing generated test requirements for TMs with a same AP type, but different AP variability parameters (e.g. *maxAPperiodicPeriod*, *maxAPboundedMinIAT* and *minnAPirregularPoints*).

A dash "-" in a cell of Table 13 indicates that the parameter (corresponding to the cell) does not apply to the corresponding TM. For example, all SDs in *tm8* are supposed to be periodic. Therefore, only periodic AP parameters apply to this TM. The following TM-size parameters were used for the test models in Table 13.

	AP-related variation strategies											
	Different-AP-Types					*Same-AP-Different-Parameters*						
Parameters \ **Test Models**	*tm7*	*tm8*	*tm9*	*tm10*	*tm11*	*tm12*	*tm13*	*tm14*	*tm15*	*tm16*	*tm17*	*tm18*
APtype	*no_arrival_pattern*	*periodic*	*bounded*	*irregular*	*mixed*	*periodic*	*periodic*	*periodic*	*bounded*	*bounded*	*irregular*	*irregular*
minAPperiodicPeriod	-	5	-	-	5	**5**	5	**20**	-	-	-	-
maxAPperiodicPeriod	-	10	-	-	10	**5**	5	**25**	-	-	-	-
minAPperiodicDeviation	-	0	-	-	0	1	**5**	1	-	-	-	-
maxAPperiodicDeviation	-	3	-	-	3	5	**5**	2	-	-	-	-
minAPboundedMinIAT	-	-	2	-	2	-	-	-	**50**	2	-	-
maxAPboundedMinIAT	-	-	4	-	4	-	-	-	**60**	4	-	-
minAPboundedMaxIATafterMin	-	-	2	-	2	-	-	-	2	**60**	-	-
maxAPboundedMaxIATafterMin	-	-	5	-	5	-	-	-	5	**70**	-	-
minnAPirregularPoints	-	-	-	5	5	-	-	-	-	-	**50**	5
maxnAPirregularPoints	-	-	-	15	15	-	-	-	-	-	**100**	15
minAPirregularPointsValue	-	-	-	1	1	-	-	-	-	-	1	**100**
maxAPirregularPointsValue	-	-	-	30	30	-	-	-	-	-	30	**200**

Table 13-Experimental test models with variations in SD arrival patterns.

As these parameters denote, the size of TMs *tm7, ..., tm18* have been chosen to be relatively medium (a typical setting) as our main intention in designing these TMs is to experiment our GA's behavior and output results with reaction to variations in SD arrival patterns.

- nISDSs=10

- minISDSsize=5
- maxISDSsize=10
- nSDs=20
- minnDCCFPs=2
- maxnDCCFPs=5
- minnDTUPPs=1
- maxnDTUPPs=10

We discuss next our rationale of designing each TM based on the two above variation strategies (*Different-APs* and *Same-AP-Different-Parameters*).

- *Different-AP-Types*: The four TMs *tm7… tm11* are intended to investigate the impacts of variations in AP types on repeatability and convergence efficiency of the GA outputs. Each of these TMs have different criteria to assign APs to SDs, as discussed in Section 11.3.5.1.
- *Same-AP-Different-Parameters:* Sets of two or more TMs where SDs in each TM have the same AP, but different AP parameters. For example, all SDs in *tm12*, *tm13* and *tm14* have periodic APs, but have different values for *minAPperiodicPeriod*, *maxAPperiodicPeriod*, *minAPperiodicDeviation* or *maxAPperiodicDeviation*. *tm15* and *tm16* are our pair of experimental test models with bounded APs. TMs *tm17* and *tm18* have SDs with irregular APs. Such a variation strategy will enable us to study the impacts of variations in AP parameters on repeatability and convergence efficiency of the GA outputs.

11.3.5.5 Test Models with Variations in the GA Maximum Search Time

To investigate the effect of variations in maximum search time (Section 10.7.4) on GARUS test requirements by GARUS, we created the following two test models using our random test model generator. As an example, the contents of the input file corresponding to *tm20* is shown in Appendix B.

- *tm19*: SUT components (SDs, DCCFPs, ISDSs, etc.) are the same as *tm1*, but the *MaximumSearchTime* is 5 time units instead of 50 in *tm1*.
- *tm20*: SUT components (SDs, DCCFPs, ISDSs, etc.) are the same as *tm1*, but the *MaximumSearchTime* is 150 time units instead of 50 in *tm1*.

11.3.6 Impacts of Test Model Size (Scalability of the GA)

We investigate how the GA performance and its repeatability are affected when it is applied to test models with different sizes. We study scalability by analyzing the impact of variations in test model size on the following metrics:

- Execution time
- Repeatability of maximum ISTOF values
- Repeatability of maximum stress time values
- Convergence efficiency across generations

11.3.6.1 Impact on Execution Time

To investigate the impact of test model size on the execution time of our GA, the average, minimum and maximum execution times over all the 1000 runs, by running GARUS with *tm1*…*tm6* on an 863MHz Intel Pentium III processor with 512MB DRAM memory are

reported in Table 14. As we can see in the table, minimums and maximums of the statistics in Table 14 for each test model are relatively close to the corresponding average value. Therefore, we use the average values to discuss the impacts of test model size on execution time of our GA. To better illustrate the differences, the average values are depicted in Figure 99.

Test Model	Min	Max	Average	Median	Standard Deviation
tm1 (small)	46	125	58	62	11.34
tm2 (many ISDSs)	743	1,150	1,089	375	44.79
tm3 (many SDs in TM)	1,015	2,219	1,240	1,171	199.10
tm4 (many SDs per ISDS)	734	1,641	809	782	97.61
tm5 (many DCCFPs per SD)	141	597	263	250	62.83
tm6 (many DTUPs per DCCFP)	734	1,375	820	797	74.74

Table 14-Execution time statistics of 1000 runs of *tm1…tm6*.

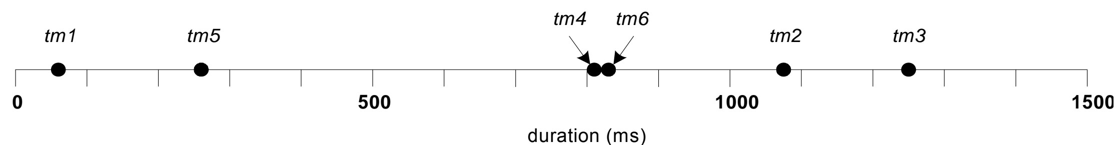


Figure 99-Visualization of the average values in Table 14.

Average duration of the GA run of test model *tm1* (58 ms) is the smallest among all. This is expected since TM *tm1* has the smallest size in terms of test model components (ISDSs, SDs, and DCFFPs). *tm3* has the highest average execution time among the six TM runs. Durations of *tm2*, *tm6*, *tm4*, and *tm5* are next in decreasing order. Based on the above order of execution values, we can make the following observations:

- The execution time of the GA is strongly sensitive to an increase in number of SDs in a TM. The more SDs in a TM, the longer a single run of our GA takes

(e.g. *tm3*). This can be explained as the number of genes per chromosome in our GA is the same as the number of SDs in a TM. Thus, as the execution results indicate, the execution time of our GA sharply increases when the number of genes per chromosome increases. Such an increase impacts all functional components of the GA, the two operators (Section 10.7.6) and the fitness evaluator (Section 10.7.5).

- As expected, the execution time of our GA is also highly dependent on the number of ISDSs (e.g. *tm2*). As the number of ISDSs increases, the size of initial population grows, and so does the number of the mutations and crossovers applied in each generation. The number of times the operators are applied is determined by the mutation and crossover rates and the size of initial population.
- The execution time of our GA is also dependent on an increase in number of SDs per ISDS (e.g. *tm4*), as well as an increase in number of DTUPs per DCCFP (e.g. *tm6*). As the number of SDs per ISDS increases, the number of non-null genes per chromosome will increase. This will, in turn, lead to more mutations and crossovers and an increase in computation for the fitness evaluator. Similarly, an increase in number of DTUPs per DCCFP will lead to an increase in fitness computation time (Section 10.7.5).
- The execution time of our GA is not as dependent on an increase in number of DCCFPs per SD (e.g. *tm5*), when compared to other TM components. This can be explained as there will not be any change in chromosome size, nor in the initial population in that case. Even the frequency of mutations and crossovers will not change. For example, as the mutation operator chooses a random DCCFP among

all DCCFPs of a SD, there will be no effect in terms of execution time if the number of DCCFPs per SD increases. The small difference between average durations of *tm5* and *tm1* in Figure 99 is due to the fact that *tm5's* number of SDs is slightly more than that of *tm1*.

11.3.6.2 Impact on Repeatability of Maximum ISTOF Values

To investigate the impact of test model size on the repeatability of maximum ISTOF values generated by our GA, Figure 100 depicts the histograms of maximum ISTOF values for 1000 runs on each of the test models *tm1,…,tm6*. The corresponding descriptive statistics are shown in Table 15. Average and median values of all distributions are very close, thus indicating that the distributions are almost symmetric.

We can see from the ISTOF distributions that the maximum fitness values for most of *tm1* runs, for example, are between 60 and 74 units of traffic. Referring to Figure 100, the variations in fitness values across runs is expected when using genetic algorithms on complex optimization problems. However, though the variation above is not negligible, one would expect based on Figure 100 that with a few runs, a chromosome with a fitness value close to the maximum would likely be identified. To discuss the practical implications of multiple runs of GARUS to get a sub-maximum result (stress test requirement) for *tm1*, we can perform an analysis by using the probability distributions of maximum ISTOF values in the histogram of Figure 100-(a).

In the distributions of maximum ISTOF values for , as it can be easily seen in Figure 100-(a), 1000 runs of GARUS has generated mainly three groups of outputs: values between 70 and 80 units of traffic ($group_{70}$ in Figure 100-(a)), [60,70] and [50,60]. Obviously, the

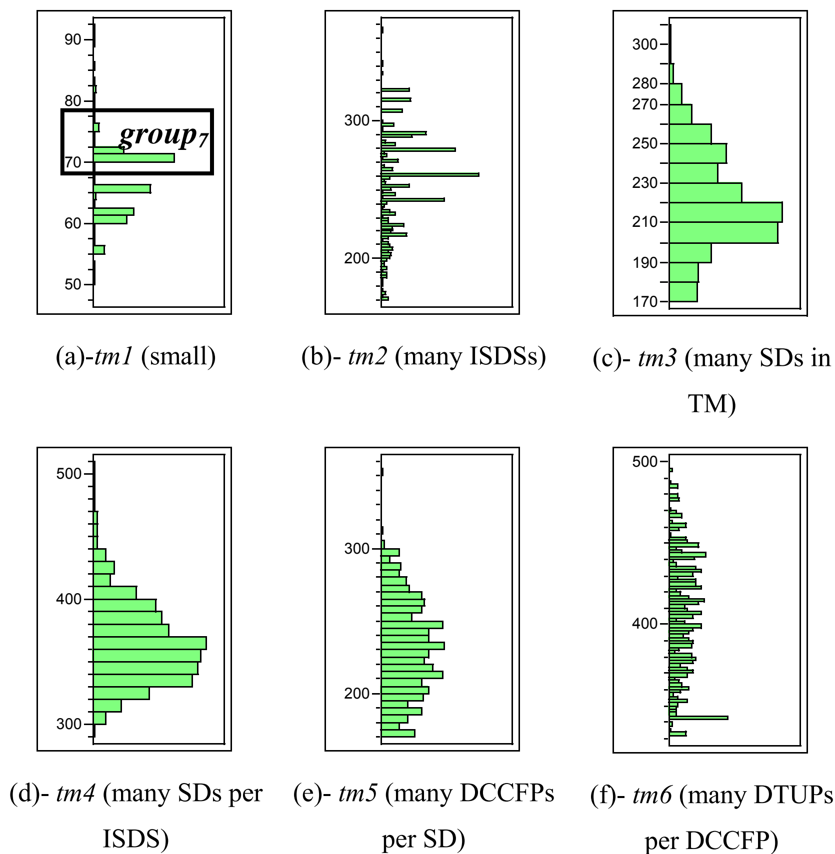
goal of using GARUS is to find stress test requirements which have the highest possible ISTOF values. Thus, the strategy is to run GARUS for multiple times and choose a test requirement with the highest ISTOF value across all runs.

The practical implication of multiple runs to achieve a test requirement with the highest ISTOF value is to predict the minimum number of times GARUS should be executed to yield an output with an ISTOF value in $group_{70}$ in Figure 100-(a). By looking into the raw data of the distribution (a), we found out that 425 (of 1000) values in the histogram belong to $group_{70}$. Thus, in a sample population of 1000 GARUS outputs, the probabilities that an output belongs to $group_{70}$ is $p(group_{70})$=0.425. Thus, to predict the minimum number of times GARUS should be executed to yield an output with an ISTOF value in $group_{70}$, we can use the following probability formula:

p(a test requirement with an ISTOF value in group$_{70}$ *is yielded in a series of n runs of GARUS)=*

$$1-(1-p(group_{70}))^{n-1}\,p(group_{70}) = 1-(0.575)^{n-1}(0.425)$$

The above probability function is depicted in Figure 101, for n=1…15. Figure 101-(b) depicts a zoom-out of the curve in Figure 101-(a) for n=0…40. For values of n higher than 15, the value of the above function is very close to 1, meaning that one can get a stress test requirement with a ISTOF value in the range [70, 80] in 15 runs. Since each run lasts a few seconds (Section 11.3.6.1), perhaps a few minutes for very large examples, relying on multiple runs to generate a stress test requirement should not be a practical problem.



(a)-*tm1* (small) (b)- *tm2* (many ISDSs) (c)- *tm3* (many SDs in TM)

(d)- *tm4* (many SDs per ISDS) (e)- *tm5* (many DCCFPs per SD) (f)- *tm6* (many DTUPs per DCCFP)

Figure 100-Histograms of maximum ISTOF values (y-axis) for 1000 runs of each test model. The y-axis values are in traffic units.

Test Model	Min	Max	Average	Median	Standard Deviation
tm1 (small)	65	112	81.66	81	7.0
tm2 (many ISDSs)	171	367	255	260	36.9
tm3 (many SDs in TM)	171	306	220	217	25.2
tm4 (many SDs per ISDS)	299	502	364	360	32.9
tm5 (many DCCFPs per SD)	171	352	230	231	32.2
tm6 (many DTUPs per DCCFP)	333	494	404	406	36.8

Table 15-Descriptive statistics of the maximum ISTOF values for each test model over 1000 runs. Values are in units of data traffic.

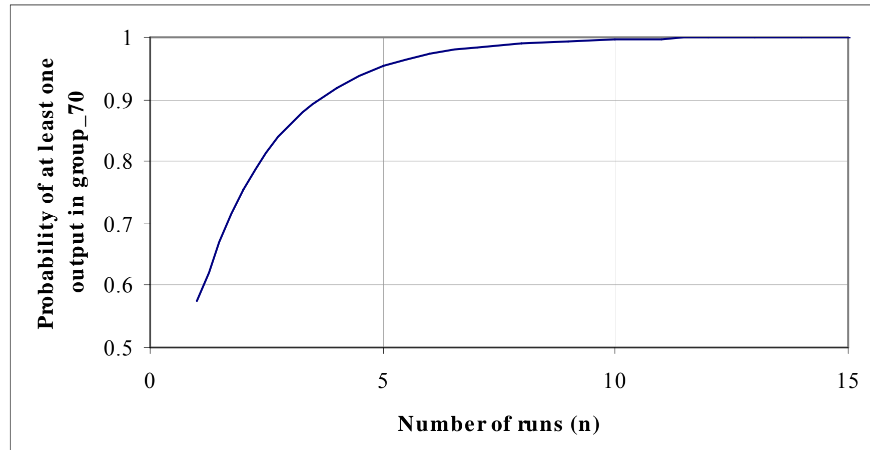


Figure 101- Probability of the event that at least one test requirement with an ISTOF value in $group_{70}$ is yielded in a series of n runs of GARUS.

We discuss two main observations based on the results shown in Figure 100.

- In all of the histograms, despite the fact that the results correspond to 1000 runs of different test models which were designed to test the scalability and repeatability properties of our GA, the maximum ISTOF values of the outputs produced by the GA lie in rather small intervals. For example as shown in Figure 100-(f), the maximum ISTOF values generated for *tm6* range in [330…500] units of traffic.
- In terms of scalability, histograms in (b), (c), (d), (e), and (f) suggest that the GA can reach a maximum plateau when the size of a specific component (SD, ISDS, DCCFP, etc) of a given TM is very large.

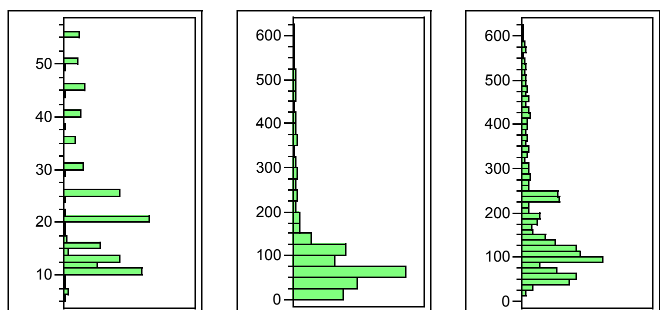
11.3.6.3 Impact on Repeatability of Maximum Stress Time Values

To investigate the impact of test model size on the repeatability of maximum stress time values generated by our GA, Figure 102 depicts the histograms of maximum stress time values for 1000 runs on each of test models *tm1, …, tm6*. The corresponding descriptive

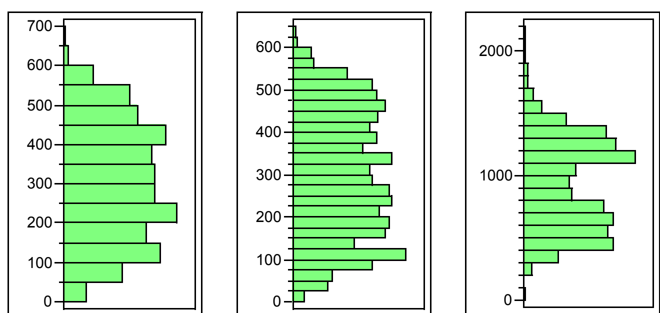
statistics are shown in Table 16. We discuss four main observations based on the results shown in Figure 102.

- Average and median values of distributions in (d), (e), and (f) are quite close, thus indicating that the distributions are almost symmetric. Conversely, distributions in (a), (b), and (c) are not symmetric. This reveals that for *tm4* (d), *tm5* (e), and *tm6* (f), the GA might produce maximum stress time values with peak values only at some points. For *tm1* (a), *tm2* (b), and *tm3* (c), the GA generated stress time values with low standard deviation. Furthermore, these distributions are skewed towards the minimums. Thus, one has to run the GA many number of times to get a value close to the maximums.
- Distributions in (d) and (e) are quite flat. This can be explained as for *tm4* (d) *tm5* (e), the number of alternatives to yield a best chromosome by the GA is higher than the others. Furthermore, large sets of best chromosomes can yield different maximum stress time values due to the random start times selected from the legal start times of DCCFPs.
- The standard deviation value for *tm1* (a) is relatively smaller than the other distributions because of the smaller range of maximum ISTOF values in *tm1* results.
- The standard deviation of distribution (f), *tm6*, is much higher than the five others. This can be explained by the higher number of DTUPs per DCCFP (50-100) in *tm6*, compared to the other TMs (1-10). This will, in turn, result in a wider range in the time domain for the GA to search in and find best individuals. As it can be seen in Figure 102, the range of minimum and maximum values in distribution

(f), [87, 2128], is much larger than the other five distributions, which is also resulted from the aforementioned property of *tm6*.



(a)-*tm1* (small) (b)-*tm2* (many ISDSs) (c)-*tm3* (many SDs in TM)



(d)-*tm4* (many SDs per ISDS) (e)-*tm5* (many DCCFPs per SD) (f)- *tm6* (many DTUPs per DCCFP)

Figure 102-Histograms of maximum stress time values for 1000 runs of each test model. The y-axis values are in time units.

Test Model	Min	Max	Average	Median	Standard Deviation
tm1 (small)	6	56	22	20	12
tm2 (many ISDSs)	11	602	103	61	108.8
tm3 (many SDs in TM)	16	618	180	123	135.2
tm4 (many SDs per ISDS)	7	655	301	296	147.0
tm5 (many DCCFPs per SD)	19	626	298	289	149.0
tm6 (many DTUPs per DCCFP)	87	2128	937	933	373

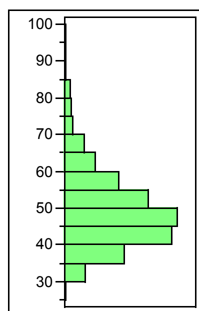
Table 16-Descriptive statistics of the maximum stress time values for each test model over 1000 runs. Values are in time units.

11.3.6.4 Impact on Convergence Efficiency across Generations

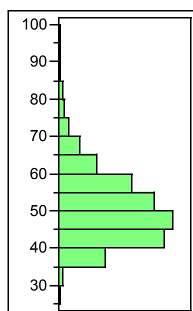
Another interesting property of the GA to look at is the number of generations required to reach a stable maximum fitness plateau. To investigate the impact of test model size on convergence efficiency across generations in the GA, Figure 103 depicts the histograms of the generation numbers required to reach a stable maximum fitness plateau over 1000 runs of test models *tm1, ..., tm6*. The corresponding descriptive statistics are shown in Table 17. On the average, 49-50 generations of the GA were required to converge to the final result (a stress test requirement) for *tm1, ..., tm6*. We therefore see that the TM size does not affect the convergence efficiency across generations in our GA.

The variations around this average in different TMs are limited and no more 100 generations will be required. This number is in line with the experiments reported in the GA literature [87]. As we can see, test model size does not have an impact on

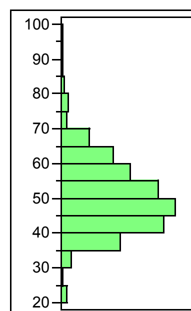
convergence efficiency across generations, and the GA is able to reach a stable maximum fitness plateau after about 50 generations on average, independent of test model size.



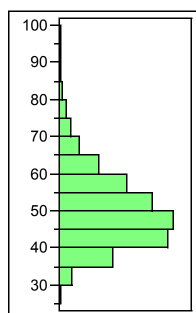
(a)- *tm 1* (small)



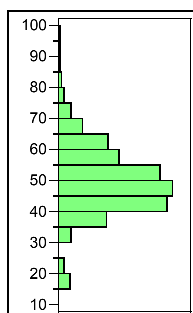
(b)- *tm 2* (many ISDSs)



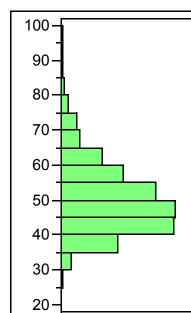
(c)- *tm 3* (many SDs in TM)



(d)- *tm 4* (many SDs per ISDS)



(e)- *tm 5* (many DCCFPs per SD)



(f)- *tm 6* (many DTUPs per DCCFP)

Figure 103- Histograms of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of each test model.

Test Model	Min	Max	Average	Median	Standard Deviation
tm1 (small)	27	98	49	47	10.78
tm2 (many ISDSs)	28	96	50	49	10.44
tm3 (many SDs in TM)	23	97	50	49	10.84
tm4 (many SDs per ISDS)	25	98	49	49	10.63
tm5 (many DCCFPs per SD)	17	96	49	49	11.77
tm6 (many DTUPs per DCCFP)	27	98	50	48	10.74

Table 17-Minimum, maximum and average values of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of each test model.

11.3.7 Impacts of Arrival Pattern Types

The impact of variations in arrival pattern types are investigated by running GARUS with test models *tm7…tm11*. The results are reported in the following four subsections.

- Impact on Execution Time
- Impact on Repeatability of Maximum ISTOF Values
- Impact on Repeatability of Maximum Stress Time Values
- Impact on Convergence Efficiency across Generations

11.3.7.1 Impact on Execution Time

We computed the average, minimum and maximum execution times over all the 1000 runs, by running GARUS with test models *tm7…tm11* on an 863MHz Intel Pentium III processor with 512MB DRAM memory (Table 18). Minimums and maximums of the statistics in Table 18 for each test model run are relatively close to the corresponding

average value. Therefore, we use the average values to discuss the impacts of variations in arrival patterns on execution time. To better illustrate the differences, the average values are depicted in Figure 104.

Recall from Section 11.3.5.4 that test models *tm7…tm11* have been designed such that they all have the same number of SDs, CCFPs, and DTUPPs (same TM size). Based on the values depicted in Figure 104, the average execution times of the test models *tm7…tm11* with the same test model size, but different arrival patterns for SDs, are relatively close to each other (within 100 ms). This indicates that execution time is not strongly dependent on SD arrival patterns in a test model. Furthermore, as we discuss below, the difference in execution times are mainly due to the implementation details of a method of class *AP* in GARUS.

Test Model	Min	Max	Average	Median	Standard Deviation
tm7 (all SDs with no arrival patterns)	234	593	296	281	51.27
tm8 (all SDs with periodic arrival patterns)	234	625	290	266	50.47
tm9 (all SDs with bounded arrival patterns)	250	594	295	281	53.78
tm10 (all SDs with irregular arrival patterns)	156	344	186	172	29.12
tm11 (SDs with arbitrary arrival patterns)	217	245	231	224	61.23

Table 18-Execution time statistics of 1000 runs of *tm7…tm11*.

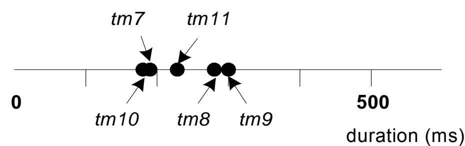


Figure 104-Visualization of the average values in Table 18.

The execution times of two of these test models (*tm8* and *tm9*), are slightly higher than those of *tm7* and *tm10*. The difference between the two TM groups (*tm8* and *tm9* versus *tm7* and *tm10*) can be explained by an implementation detail of GARUS. Function `getARandomArrivalTime`, a member function of class *AP* (Figure 88), is overridden in each of *AP*'s subclasses. The time complexity of this function in *noAP* and *irregularAP* classes is *O(1)*, i.e., choosing a random value from a range or an array, respectively. However, the implementation of the function in *periodicAP* and *boundedAP* classes required some extra considerations (related to the ATSs of periodic and bounded APs), and thus the time complexities of the function are not constant anymore, but dependent on the specific arrival pattern parameters.

The execution time of *tm11*, in which each SD can have an arbitrary arrival pattern, is placed somehow close to the average value of the other four TMs (*tm7*, *tm8, tm9,* and *tm10*). This is as predicted since the APs of SDs in *tm11* are a mix of APs in the other four, thus leading to such an impact in its average execution time.

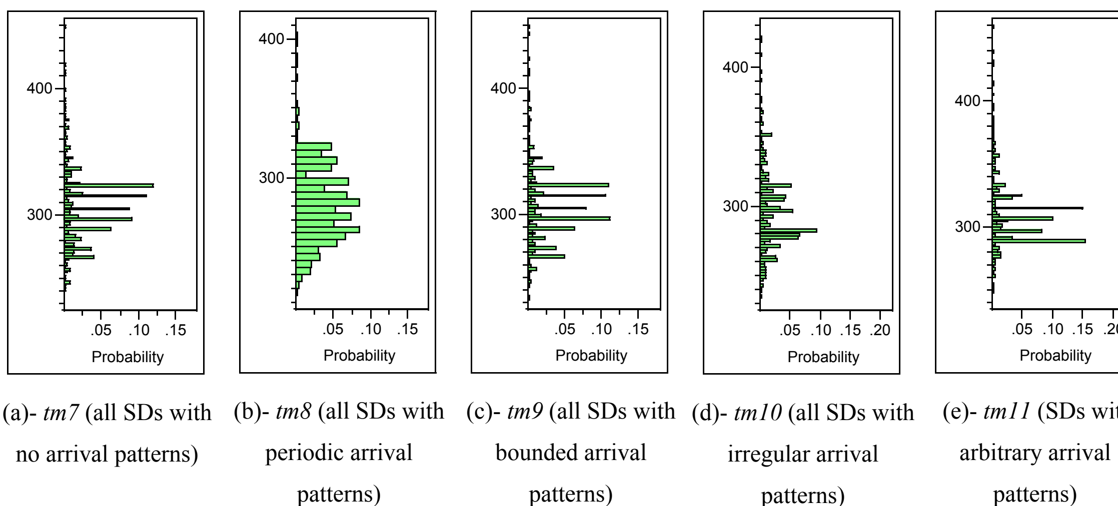
11.3.7.2 Impact on Repeatability of Maximum ISTOF Values

To investigate the impact of variations in arrival pattern types on the repeatability of maximum ISTOF values, Figure 105 depicts the histograms of maximum ISTOF values for 1000 runs on each of the test models *tm7,…, tm11*. The corresponding descriptive statistics are shown in Table 19.

As we can see, there are no big differences among the five distributions. The histogram of maximum ISTOF values for *tm8* is the only noticeable difference, in which the distribution is flatter than the four others (with more peaks and valleys). This is perhaps

due to the specific ATS properties of periodic arrival patterns (the arrival pattern type of SDs in *tm8*).

Another observation is that the histograms in Figure 105-(a) and Figure 105-(c) are quite similar. Recall from Sections 10.2-10.3 that the ATS of a bounded arrival pattern covers the entire time domain except few time intervals close to zero. Therefore, if the common unconstrained time intervals of a set of bounded arrival patterns are considered, they resemble a set of SD with no arrival patterns. The effect of such a property, thus, shows itself in the two histograms.



(a)- *tm7* (all SDs with no arrival patterns)

(b)- *tm8* (all SDs with periodic arrival patterns)

(c)- *tm9* (all SDs with bounded arrival patterns)

(d)- *tm10* (all SDs with irregular arrival patterns)

(e)- *tm11* (SDs with arbitrary arrival patterns)

Figure 105-Histograms of maximum ISTOF values for 1000 runs of each test model. The y-axis values are in traffic units.

Test Model	Min	Max	Average	Median	Standard Deviation
tm7 (all SDs with no arrival patterns)	240	448	305	304	27.7
tm8 (all SDs with periodic arrival patterns)	216	404	279	279	27.0
tm9 (all SDs with bounded arrival patterns)	232	448	306	304	28.8
tm10 (all SDs with irregular arrival patterns)	234	420	294	289	28.41
tm11 (SDs with arbitrary arrival patterns)	248	459	309	307	25.72

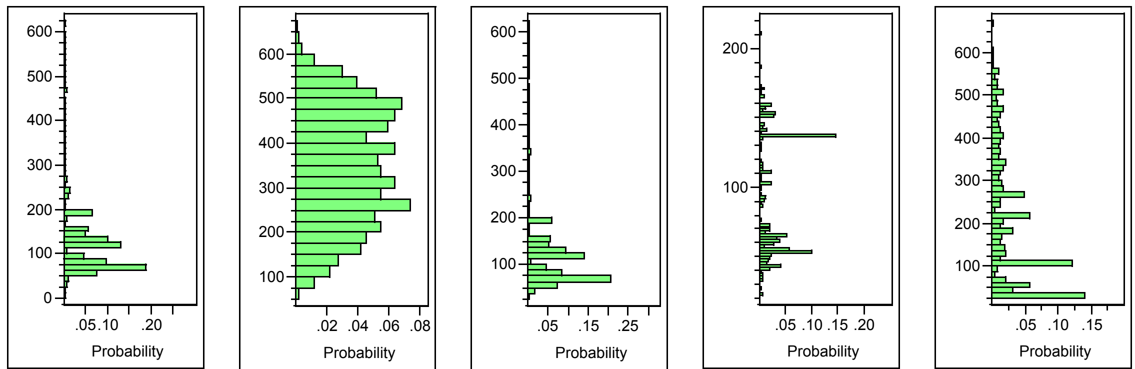
Table 19-Descriptive statistics of the maximum ISTOF values for each test model over 1000 runs. Values are in units of data traffic.

11.3.7.3 Impact on Repeatability of Maximum Stress Time Values

To investigate the impact of variations in arrival pattern types on the repeatability of maximum stress time values, Figure 106 depicts the histograms of maximum stress time values for 1000 runs on each of test models *tm7,…*, *tm11*. The corresponding descriptive statistics are shown in Table 20.

We attempt below to interpret the distributions shown in Figure 106.

- Distributions (a) and (c) are skewed towards their minimum values. For example, the mode of (a) is around 70 units of traffic which is closer to 0 (the minimum) than 620 (the maximum). This can be explained based on the early start times of DTUPs in DCCFPs of the fittest GA individual generated for *tm7* and *tm9*. Since the ATS of *tm7* is unconstrained and the one for *tm9* has only few constrained ATIs, the GA chooses the common start times of maximum stressing DCCFPs as early as possible.



(a)- *tm7* (all SDs with no arrival patterns) (b)- *tm8* (all SDs with periodic arrival patterns) (c)- *tm9* (all SDs with bounded arrival patterns) (d)- *tm10* (all SDs with irregular arrival patterns) (e)- *tm11* (SDs with arbitrary arrival patterns)

Figure 106-Histograms of maximum stress time values for 1000 runs of each test model. The y-axis values are in time units.

Test Model	Min	Max	Average	Median	Standard Deviation
tm7 (all SDs with no arrival patterns)	12	622	140	119	105.7
tm8 (all SDs with periodic arrival patterns)	58	655	347	346	129.8
tm9 (all SDs with bounded arrival patterns)	33	618	137	118	105.0
tm10 (all SDs with irregular arrival patterns)	22	211	89	66	43.0
tm11 (SDs with arbitrary arrival patterns)	29	669	214	184	157.04

Table 20-Descriptive statistics of the maximum stress time values for each test model over 1000 runs. Values are in time units.

- The distribution in (b), corresponding to *tm8,* is flatter than the others. This can be explained based on the specific ATS properties of periodic arrival patterns (the arrival pattern type of SDs in *tm8*). The intersection of several periodic ATSs will be a *discrete* unbounded ATS (refer to Figure 107 for an example). Therefore, given a TM with periodic SDs, the GA might converge to any of the common ATPs in the intersection of all periodic ATSs.

- Another observation is that the histograms in (a) and (c) are quite similar. Recall from Sections 10.2 and 10.3 that the ATS of a bounded arrival pattern covers the entire time domain except few time intervals close to zero. Therefore, if the common unconstrained time intervals of a set of bounded arrival patterns are considered, they resemble a set of SD with no arrival patterns. Such similarity is visible in the two histograms.

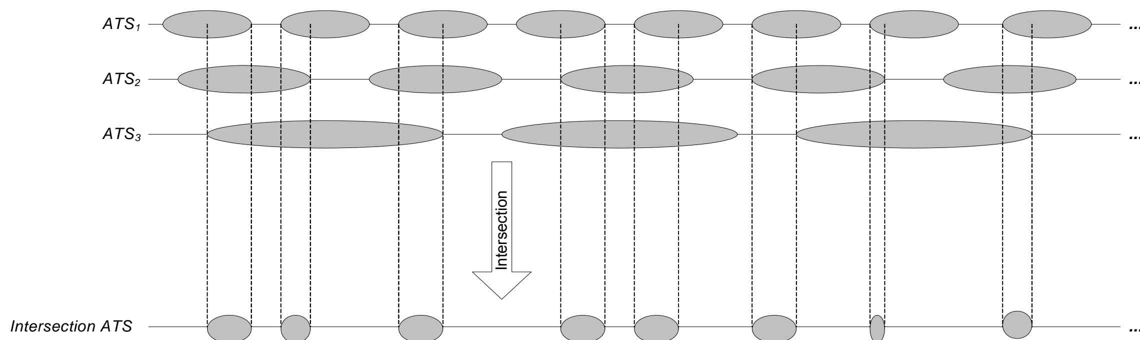


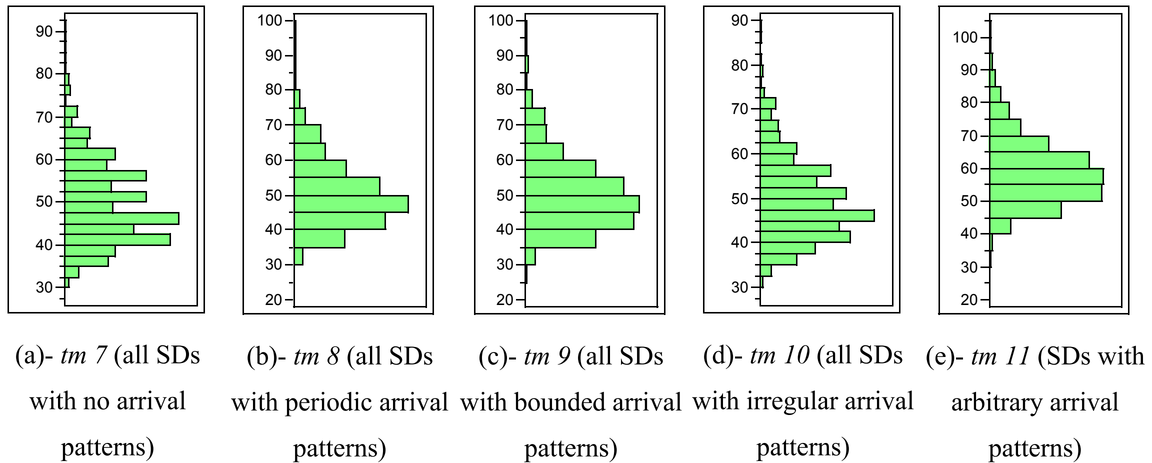
Figure 107- The intersection of several periodic ATSs is a *discrete* unbounded ATS.

11.3.7.4 Impact on Convergence Efficiency across Generations

Regarding the impact of arrival patterns on convergence efficiency across generations in the GA, Figure 108 depicts the histograms of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of test models *tm7,…, tm11*. The corresponding descriptive statistics are shown in Table 21. It is interesting to see that, on average, 49-50 generations were required to converge to the final result (a stress test requirement) across all TMs: *tm7,…, tm11*.

The standard deviations variations of the distributions are limited and no more 100 generations are required in all cases. Therefore, variations in arrival pattern types do not have a significant impact on convergence efficiency across generations, and the GA is

able to reach a stable maximum fitness plateau after about 50 generations on average, independent of any arrival pattern types.



(a)- *tm 7* (all SDs with no arrival patterns) (b)- *tm 8* (all SDs with periodic arrival patterns) (c)- *tm 9* (all SDs with bounded arrival patterns) (d)- *tm 10* (all SDs with irregular arrival patterns) (e)- *tm 11* (SDs with arbitrary arrival patterns)

Figure 108- Histograms of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of each test model.

Test Model	Min	Max	Average	Median	Standard Deviation
tm7 (all SDs with no arrival patterns)	30	90	50	48	9.97
tm8 (all SDs with periodic arrival patterns)	28	97	50	49	10.36
tm9 (all SDs with bounded arrival patterns)	30	98	50	49	10.86
tm10 (all SDs with irregular arrival patterns)	27	88	49	48	9.35
tm11 (SDs with arbitrary arrival patterns)	28	96	58	57	11.58

Table 21-Minimum, maximum and average values of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of each test model.

11.3.8 Impacts of Arrival Pattern Parameters

The impact of variations in arrival pattern parameters were investigated by running GARUS on experimental test models in which all SDs were periodic (*tm8, tm12, tm13,* and *tm14*), bounded (*tm9, tm15* and *tm16,* or irregular (*tm10, tm17* and *tm18*) (Table 13). The combinations of TMs to investigate in this section were chosen to study the impacts

of arrival pattern *parameters* such the period and deviations values of a periodic AP, and minimum/maximum inter-arrival values of a bounded AP. The results are reported in the following four subsections.

- Impact on Execution Time
- Impact on Repeatability of Maximum ISTOF Values
- Impact on Repeatability of Maximum Stress Time Values
- Impact on Convergence Efficiency across Generations

11.3.8.1 Impact on Execution Time

We computed the average, minimum and maximum execution times over all the 1000 runs, by running GARUS on test models *tm8, tm9, tm10, tm12,…tm18* on an 863MHz Intel Pentium III processor with 512MB DRAM memory (Table 22).

Minimums and maximums of the statistics in Table 22 for each test model run are relatively close to the corresponding average value. Therefore, we use the average values to discuss the impacts of variations in arrival patterns on execution time. Based on the execution values in Table 22, we can make the following observations:

- The GA execution takes longer time for higher numbers of periodic ATIs in a specific maximum search time. The distribution mean and median values for execution times of *tm12* (AP period value=5) and *tm13* (AP period value=5) are higher than those of *tm8* (AP period values from 5 to 10) and *tm14* (AP period value from 20 to 25). This is explained by an implementation detail of GARUS. Function `getARandomArrivalTime`, a member function of class *AP* (Figure 88), is overridden in each of *AP*'s subclasses. The explanations can be made

similar to the discussions in Section 11.3.7.1. As the period value of an AP increases, the number of periodic ATIs in a specific maximum search time decreases. This, in turn, reduces the GA's execution time.

- Increasing the minimum inter-arrival time range (from [2,4] in *tm9* to [50,60] in *tm15*) in bounded APs has increased the average (from 590 in *tm9* to 667 in *tm15*) and median execution times (from 562 in *tm9* to 657 in *tm15*). This is explained by formula $k = \lceil minIAT / (maxIAT - minIAT) \rceil$ (proved in Appendix C), where k is the number of ATIs for a bounded AP. Due to the implementation detail in member function `getARandomArrivalTime` of class *AP* (Section 11.3.7.1), increasing the number of ATIs in a bounded AP will increase execution time. Note that the denominator value in the above formula is the same in both *tm9* and *tm15*, by using the *minAPboundedMaxIATafterMin* parameter (Table 11), while the nominator value is changed.
- Increasing the different between minimum and maximum inter-arrival time range (from [2,5] in *tm9* to [60,70] in *tm16*) in bounded APs has decreased the average (from 667 in *tm9*to 503 in *tm16*) and median execution times (from 657 in *tm9*to 494 in *tm16*). This is explained again by the above formula and the implementation detail in member function `getARandomArrivalTime` of class *AP* (Section 11.3.7.1), whereas decreasing the number of ATIs in a bounded AP will decrease execution time. Note that the nominator value in the above formula is the same in both *tm9* and *tm16*, while the denominator value is changed.

Test Model Group	Test Model	Min	Max	Average	Median	Standard Deviation
periodic	*tm8*	468	1,250	580	532	100.94
	tm12	625	1,359	746	703	123.13
	tm13	640	1,547	758	703	125.11
	tm14	393	1,109	519	556	101.55
bounded	*tm9*	500	1,188	590	562	50.56
	tm15	625	890	667	657	22.92
	tm16	447	853	503	494	35.50
irregular	*tm10*	312	688	372	344	58.24
	tm17	453	1235	582	531	97.77
	tm18	500	984	557	532	64.4

Table 22-Execution time statistics of 1000 runs of *tm8, tm9, tm10, tm12,…tm18*.

11.3.8.2 Impact on Repeatability of Maximum ISTOF Values

To investigate the impact of variations in arrival pattern parameters on the repeatability of maximum ISTOF values, Figure 109 depicts the histograms of maximum ISTOF values for 1000 runs on each of the test models *tm8, tm9, tm10, tm12,…tm18*. The corresponding descriptive statistics are shown in Table 23. We discuss three main observations based on the results shown in Figure 109.

- Among TMs with all periodic APs (*tm8, tm12, tm13,* and *tm14*), *tm13* has the highest maximum ISTOF value (500). This is because the APs in *tm13* have both period and deviation values of 5, which means that the corresponding ATSs are virtually unconstrained, i.e., all time instants are accepted. This, in turn, lets the GA search the entire time domain and find the best possible

stress test schedule. The ATSs of other three TMs (*tm8, tm12,* and *tm14*) are constrained and do not include any time instant in the time domain.

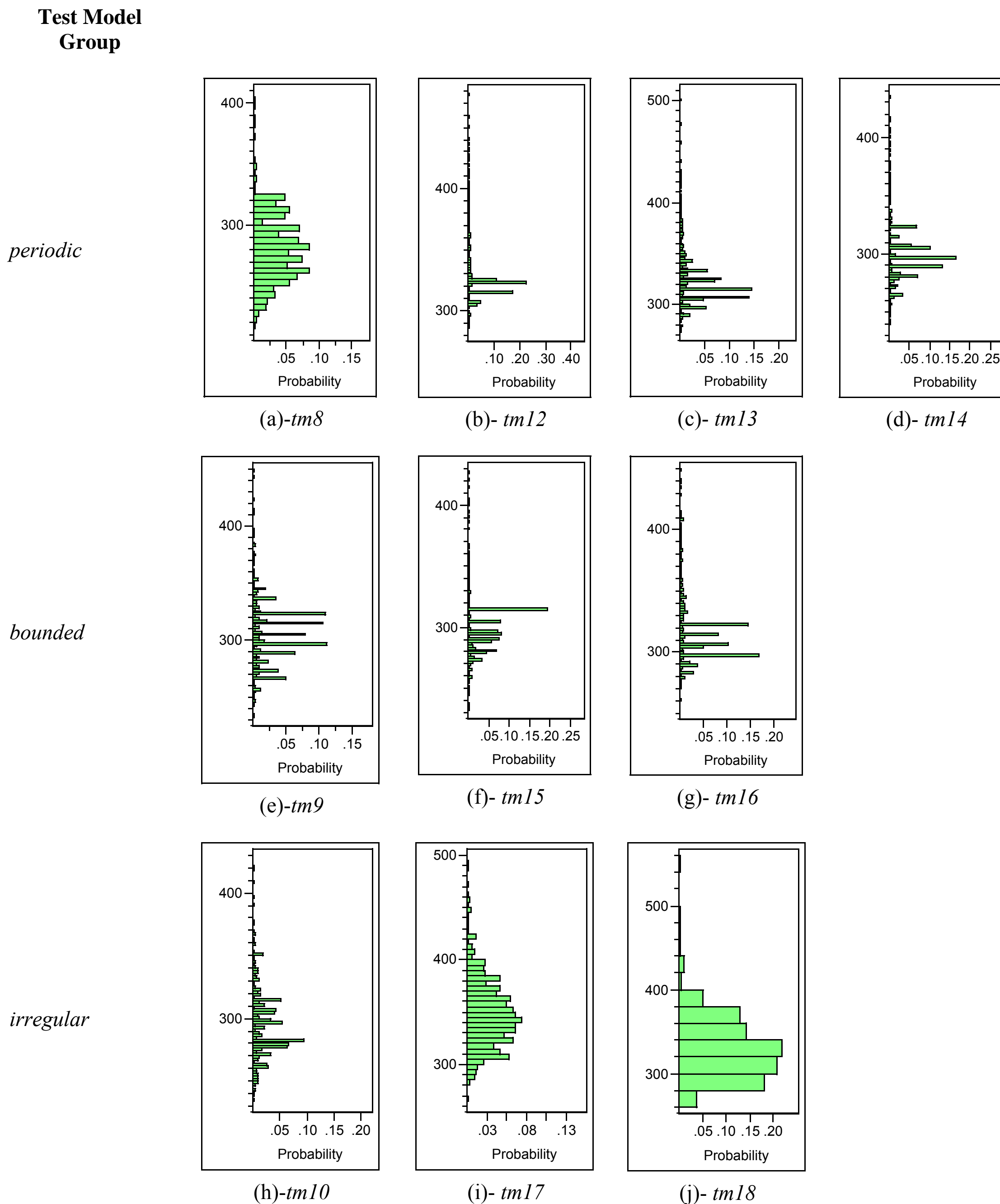


Figure 109-Histograms of maximum ISTOF values for 1000 runs of each test model. The y-axis values are in traffic units.

- Increasing the number of arrival points (from [5,15] in *tm10* to [50,100] in *tm17*) in irregular APs has increased the highest maximum ISTOF value (from 420 in *tm10* to 490 in *tm17*). This is because as the number of arrival points increases, the number of possible stress test requirements increases, and so does the chances of finding a stress test requirement with the highest stress value.
- Increasing the location of arrival points (from [1,30] in *tm10* to [100,200] in *tm18*) in irregular APs does not have a significant impact on the distribution of maximum ISTOF values. This is because such a change will only shift (in time domain) the time instant when stress traffic will be entailed (maximum stress time).

Test Model Group	**Test Model**	**Min**	**Max**	**Average**	**Median**	**Standard Deviation**
periodic	*tm8*	216	404	279	279	27.0
	tm12	273	476	323	317	26.65
	tm13	287	500	331	323	27.8
	tm14	240	435	298	296	25.21
bounded	*tm9*	232	448	306	304	28.8
	tm15	249	507	306	297	22.29
	tm16	260	449	317	310	28.34
irregular	*tm10*	234	420	294	289	28.41
	tm17	268	490	348	345	33.58
	tm18	262	556	329	330	35.01

Table 23-Descriptive statistics of the distributions in Figure 109.

11.3.8.3 Impact on Repeatability of Maximum Stress Time Values

To investigate the impact of variations in arrival pattern types on the repeatability of maximum stress time values, Figure 110 depicts the histograms of maximum stress time values for 1000 runs on each of test models *tm8, tm9, tm10, tm12,…tm18*. The corresponding descriptive statistics are shown in Table 24. We discuss three main observations based on the results shown in Figure 110.

- The difference between the mode and other values of the distribution (d) is slightly more than such a difference in distributions (a), (b) or (c). This can be explained as *tm14*, corresponding to the distribution (d), has relatively high period values ([20-25]) compared to the other three TMs. This, in turn, affects the GA by providing less chances of finding overlaps between different ATSs in a TM. Because of this, the number of such overlaps are maximized in fewer instants in the time domain in *tm14* compared to the other three.
- Among TMs with bounded AP, distributions (e) and (g) are skewed towards their minimum values, while the values in (f) are distributed almost evenly across the distribution's range, i.e., the distribution is not skewed neither towards its minimum nor it maximum value. This can be explained by the different values of the *Unbounded Range Starting Point (URSP),* Section 10.7.4., in the bounded APs of the above three TMs. As shown in Appendix C, the URSP of a bounded AP can be calculated by: $URSP = \lceil minIAT / (maxIAT - minIAT) \rceil . minIAT$. Therefore, the URSP value of *tm15* is higher than those of *tm9* and *tm16*, since the range of *minIAT* in *tm15* is higher than those in *tm9* and *tm16*, while the denominator of the above formula are in the same range. For example, considering a bounded AP in *tm15* with *minIAT=55*

and *maxIAT=58*, the URSP will be 1045 units of time. This value is even higher than the maximum search time (500 units of time) set in all the above three TMs. Therefore, there will not be any unconstrained ATI in the ATSs of *tm15*, and thus the GA will not be able to do an unconstrained search for stress test requirements in time domain. Such a search is possible in *tm9* and *tm16* because their URSPs will be inside the specified maximum search time. For example, one example URSP for a SD AP in *tm9* with *minIAT=3* and *maxIAT=5* is *6* units of time, which is < 500.

- As discussed in the previous subsection, increasing the location of arrival points (from [1,30] in *tm10* to [100,200] in *tm18*) in irregular APs does not have a significant impact on the distribution of maximum ISTOF values. However, as we can see by comparing distributions (h), (i) and (j), such an increase shifts (in time domain) the time instant when stress traffic will be entailed (maximum stress time). As we can see, both average and median values in distribution (j), corresponding to *tm18*, are higher than those in distribution (h), corresponding to *tm10*.

Test Model Group	Test Model	Min	Max	Average	Median	Standard Deviation
periodic	*tm8*	58	655	347	346	129.8
	tm12	18	613	332	337	131.54
	tm13	49	664	334	334	131.15
	tm14	41	631	333	337	125.3
bounded	*tm9*	33	618	137	118	105.0
	tm15	72	657	366	362	139.32
	tm16	26	615	151	115	109.11
irregular	*tm10*	22	211	89	66	43.0
	tm17	14	235	67	59	32.54
	tm18	136	372	239	242	40.39

Table 24-Descriptive statistics of the distributions in Figure 110.

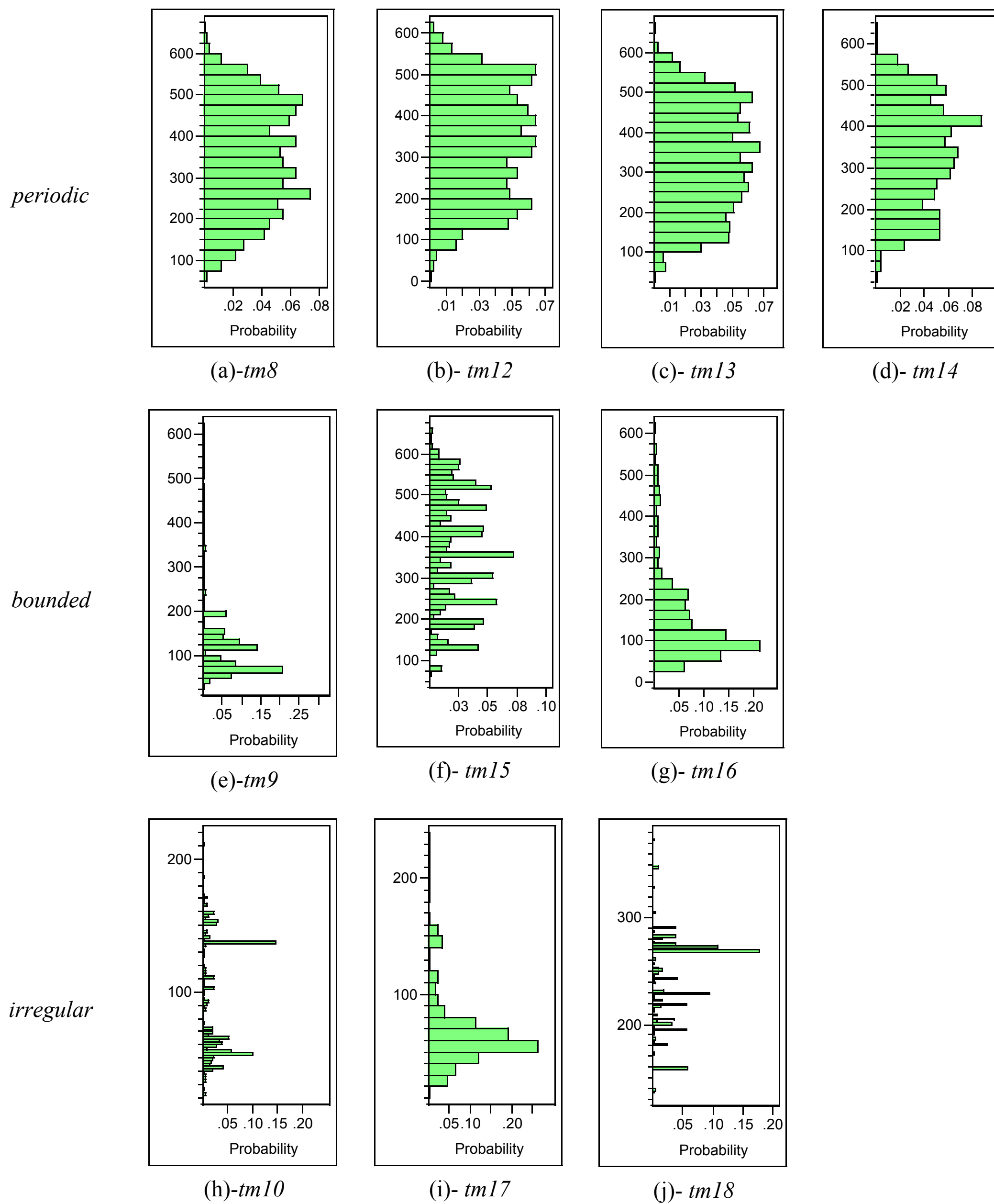


Figure 110- Histograms of maximum stress time values for 1000 runs of each test model. The y-axis values are in time units

11.3.8.4 Impact on Convergence Efficiency across Generations

Regarding the impact of arrival patterns on convergence efficiency across generations in the GA, Figure 111 depicts the histograms of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of test models *tm8, tm9, tm10, tm12,…tm18*. The corresponding descriptive statistics are shown in Table 25. On average, 49-59 generations were required to converge to the final result (a stress test requirement) across all TMs: *tm8, tm9, tm10, tm12,…tm18*. No more 100 generations are required in all cases.

As we can see, variations in arrival pattern parameters has a slight impact on convergence efficiency across generations. We discuss such impacts individually for each of the three test model groups:

- TMs with periodic APs (*tm8, tm12, tm13,* and *tm14*): The results for *tm13* converge relatively faster than the other three (almost the same), as the minimum, average and medians denote. This can be explained as the APs in *tm13* are periodic with both period and deviation values of 5. This AP resembles to a unconstrained ATS, in which all time instant are accepted. Therefore, the GA will have better chances to find "fit" individuals earlier. In the other three, more applications of the GA operators are required to converge the population.
- TMs with bounded APs (*tm9, tm15* and *tm16*): As the corresponding minimum, average and medians denote, the results for *tm16* converge relatively faster than the other two (almost the same). This can be explained by the URSP formula for bounded APs ($URSP = \lceil minIAT/(maxIAT - minIAT) \rceil . minIAT$). By calculating the range

of URSPs based on the given ranges of *minIAT* and *maxIAT*, *tm16* has the lowest value of URSP among the above set of TMs with bounded APs (*tm9, tm15* and *tm16*). By having a smaller value for URSP, our GA can search the time domain (up to maximum search time) to a greater extent, thus, yielding a faster convergence.

- TMs with irregular APs (*tm10, tm17* and *tm18*): As the corresponding minimum, average and medians denote, the results for *tm17* converge relatively faster than the other two. This can be explained by the higher number of irregular arrival points in *tm17,* i.e., the range of [50,100] compared to [5,10] in *tm10*and *tm18*. By having a higher number of irregular arrival points, our GA can search the time domain (up to maximum search time) to a greater extent, thus, yielding a faster convergence.

Test Model Group	**Test Model**	**Min**	**Max**	**Average**	**Median**	**Standard Deviation**
Periodic	*tm8*	42	99	57	56	10.45
	tm12	41	99	58	57	10.94
	tm13	28	97	50	49	10.81
	tm14	43	99	58	57	11.70
Bounded	*tm9*	39	97	58	57	11.0
	tm15	40	99	58	57	10.86
	tm16	30	98	50	49	10.48
Irregular	*tm10*	44	99	59	58	11.27
	tm17	27	88	49	48	9.35
	tm18	36	96	59	57	11.09

Table 25-Descriptive statistics of the distributions in Figure 111.

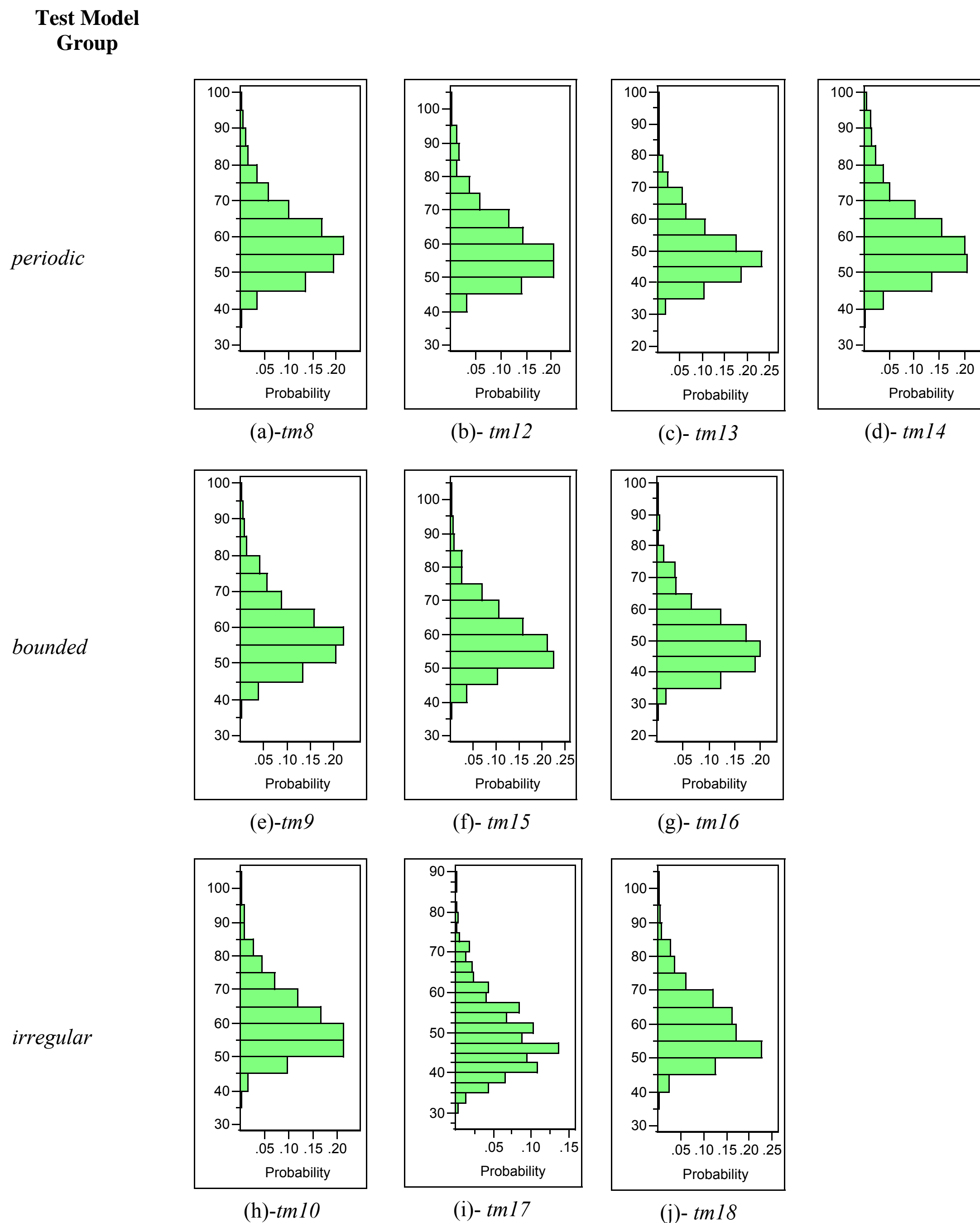


Figure 111- Histograms of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of each test model.

11.3.9 Impact of Maximum Search Time

We report in this section the impact of variations in GA maximum search time on execution time, repeatability of outputs (maximum ISTOF and stress time values), and also maximum plateau generation numbers. Maximum search time is the range of the random numbers chosen from the ATS of a SD with arrival pattern (Section 10.7.4).

We first compare GA results for TMs *tm1*, *tm19* and *tm20* in Figure 112. As described in Section 11.3.5.5, *tm19* and *tm20* have the same SUT components (SDs, DCCFPs, ISDSs, etc.) as *tm1*, but the *GATimeSearchRange* value for *tm19* and *tm20* are 5 and 150 time units, respectively, instead of 50 in *tm1*. Therefore, comparing GA results for *tm1*, *tm19* and *tm20* should reveal the impact of maximum search time on all variables of interest. The corresponding descriptive statistics are shown in Table 26.

There are 12 graphs (3 rows in 4 columns) in Figure 112. Three rows correspond to different maximum search time, while columns relate to GA variables (execution time, maximum ISTOF values, maximum stress time values, and maximum plateau generation number).

In terms of execution time, variations in maximum search time do not have an impact. Across 1000 runs, all three TMs (*tm1*, *tm19* and *tm20*) show execution times in the range [45 ms, 130 ms]. Since a change in maximum search time only changes the range in which a random time from an ATS is selected, it is not surprising that there is no effect on the workload of different GA components.

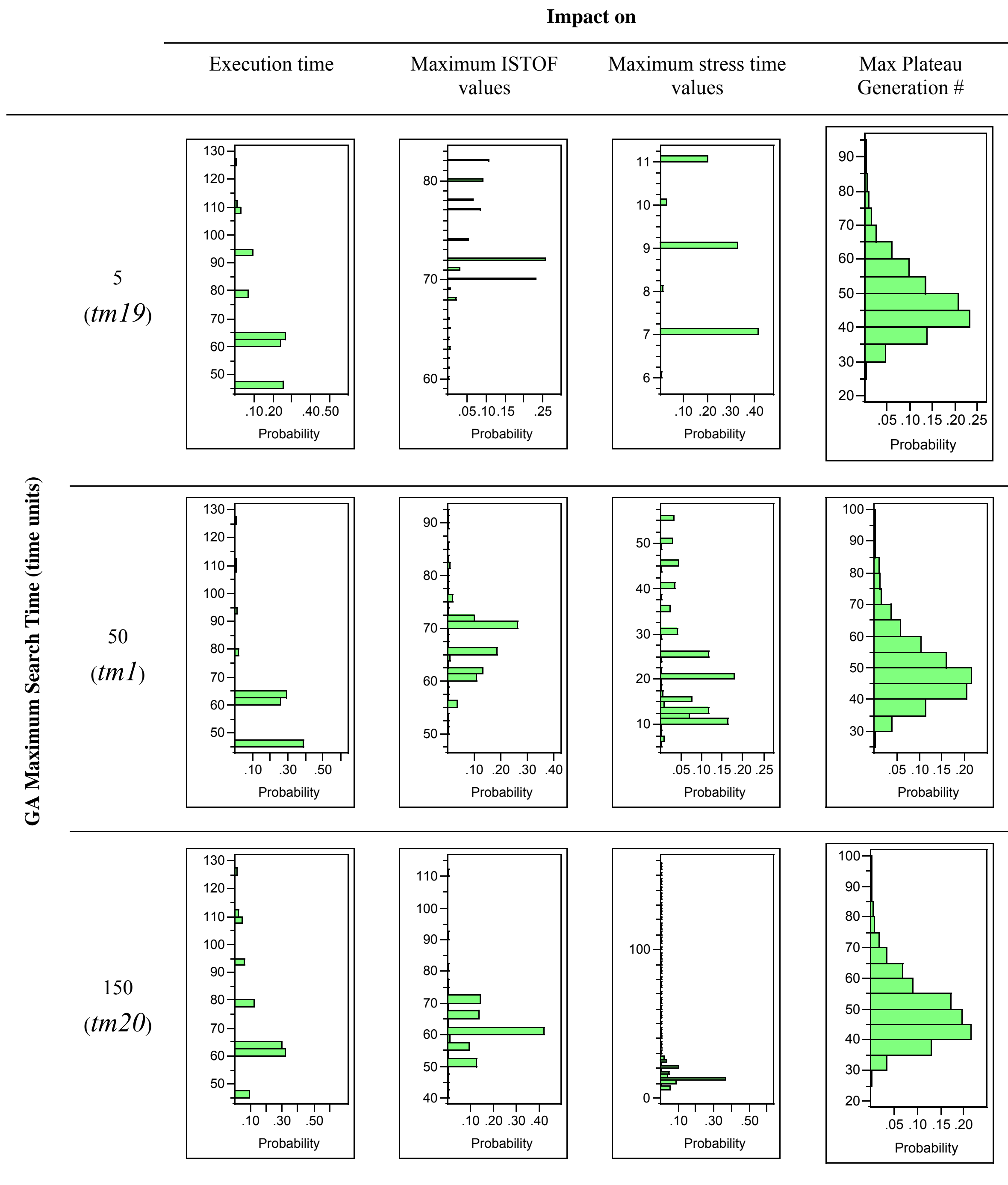


Figure 112- Impact of variations in maximum search time on the GA's behavior and outputs.

As the maximum search time increases across the three test models (5 in *tm19* to 50 in *tm1* and 150 in *tm20*), the maximum of maximum ISTOF values across 1000 runs of a TM increases, i.e. 82 traffic units for *tm19*, 91 traffic units for *tm1* and 110 traffic units for *tm20.* This can be explained by an increase in the size of GA's time search range (in ATSs) from *tm19* to *tm1* and *tm20*. From another perspective, the difference between the maximum and minimum of maximum ISTOF values also increases with the maximum search time. The differences between the maximum and minimum of maximum ISTOF values for *tm19*, *tm1* and *tm20* are 20 (82-62), 41 (91-50), and 69 (110-41) respectively. This can also be explained by the increase in the size of GA's time search range (in ATSs) from *tm19* to *tm1* and *tm20*.

In terms of maximum stress time values, similar patterns to maximum ISTOF values can be observed among the three distributions in column 'maximum stress time values' of Figure 112. In terms of maximum plateau generation number, we can see that the increase in maximum search time slightly *delays* convergence across generations, i.e., the maximum plateau generation number in *tm19* runs is reached at 91, while it is 100 for both *tm1* and *tm20* runs.

Distribution Group	Test Model	Min	Max	Average	Median	Standard Deviation
Execution time	*tm1*	46	125	58	62	11.34
	tm19	46	125	65	62	16.86
	tm20	46	125	69	63	17.50
Maximum ISTOF values	*tm1*	65	112	81	81	7.0
	tm19	60	82	73	72	4.67
	tm20	42	112	61	62	6.79
Maximum stress time values	*tm1*	6	56	22	20	12
	tm19	6	11	8	9	1.54
	tm20	9	156	31	13	37.56
Max Plateau Generation #	*tm1*	27	98	49	47	10.78
	tm19	26	94	48	46	10.52
	tm20	27	99	48	47	10.69

Table 26-Descriptive statistics of the distributions in Figure 112.

Chapter 12

CASE STUDY

A comprehensive case study is presented in this chapter. An overview of the types of targeted systems by our stress test technique is provided in Section 12.1. Section 12.2 discusses the requirements for a suitable case study. As discussed in Section 12.2, none of the systems in our survey meets the requirements. Therefore, we developed a prototype system introduced in Section 12.3, based on actual specifications. The system is referred to as *SCAPS (A SCADA-based Power System)*. The UML design model of SCAPS is also given in Section 12.3. Section 12.4 presents the stress test architecture used in our case study. Some descriptions of the stress test execution environment are given in Section 12.5. Using the UML design model of SCAPS, the building process of a corresponding stress test model (required by our test technique) is described in Section 12.6. Stress testing SCAPS by time-shifting stress test technique is described Section 12.7. In each of the last two sections, stress test results are also reported, which are used to assess the effectiveness of our stress test techniques at triggering network traffic-related failures. Stress Testing SCAPS by Genetic Algorithm-based stress test technique is presented Section 12.8.

12.1 An Overview of Target Systems

Our stress test technique can be used to stress test systems which are distributed, hard real-time, and safety-critical. We present a brief introduction here on two important groups of such systems.

1. Distributed Control Systems (DCS)
2. Supervisory Control and Data Acquisition (SCADA) Systems

Although some systems can fall in both categories, it is more convenient to discuss them separately.

A Distributed Control Systems (DCS) [4] is a computer-based control system where sections of a plant have their own processors, linked together to provide both information dissemination and manufacturing coordination. A DCS system is used in industrial and civil engineering applications to monitor and control distributed equipment with remote human intervention. A DCS system is generally, since the 1990s, digital, and normally consist of field instruments, connected via wiring to computer buses or electrical buses to multiplexer/demultiplexers, analog to digital converters, and Human-Machine Interface or control consoles. A DCS is a very broad umbrella that describes solutions across a large variety of industries, including: Electrical power distribution grids and generation plants; Environmental control systems; Traffic signals; Water management systems; Refining and chemical plants.

SCADA stands for Supervisory Control And Data Acquisition. As the name indicates, SCADA systems are not full control systems (like DCS), but they rather focus on supervisory aspects of a system. As such, it is a software package that is executed on top

of hardware to which it is interfaced, in general via Programmable Logic Controllers (PLCs), or other commercial hardware modules [102]. SCADA systems interact with their controlled environment via input/output (I/O) channels.

SCADA systems are used not only in industrial processes, e.g., steel making [5], power generation (conventional and nuclear) and distribution [103-107], chemistry and oil [108], but also in facilities such as nuclear fusion [109, 110]. The size of such plants ranges from a few to several thousands I/O channels. SCADA systems evolve rapidly and are now penetrating the market of plants with a number of I/O channels of several 100 K.

SCADA and DCS are related but they are different in important ways. A DCS is process-oriented as it focuses on the control process (such as a chemical plant), and presents data to operators. On the other hand, a SCADA system is data-gathering oriented, where the control centre and operators are the main focus points. The remote equipment is merely there to collect the data--though it may also do some very complex control process.

A DCS operator station is normally intimately connected with its I/O (through local wiring, field bus, networks, etc.). When the DCS operator wants to see information he usually makes a request directly to the field I/O and gets a response. Field events can directly interrupt the system and advise the operator.

A SCADA system must operate reasonably when field communications fail. The quality of the data shown to the operator is an important facet of a SCADA system operation. SCADA systems often provide special event processing mechanisms to handle conditions that occur between data acquisition periods. A typical architecture of SCADA systems is shown in Figure 113.

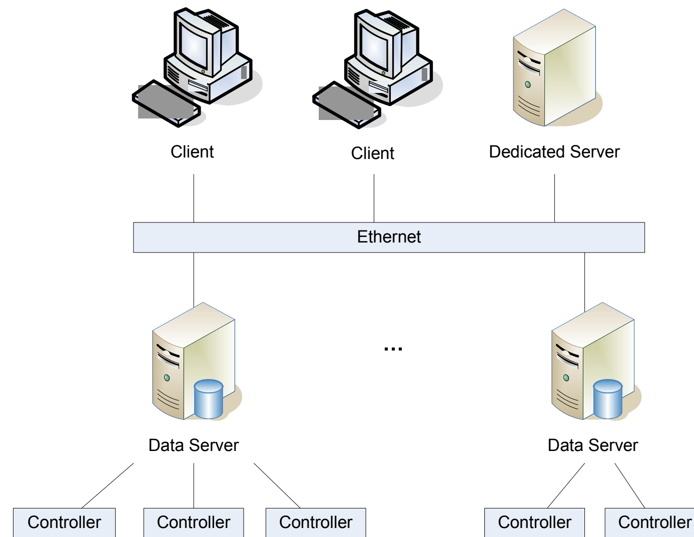


Figure 113-A typical architecture of SCADA systems.

12.2 Choosing a Target System as Case Study

There are various distributed, real-time prototype systems in academia (e.g. [111], [112], [104]) and also real systems in industry (e.g. [113], [114], [115], [116]), which are currently in use.

12.2.1 Requirements for a Suitable Case Study

We group the requirements for a suitable case study into two groups: (1) system's functional features and behaviors, and (2) modeling technique. A suitable case study should have the following functional features and behaviors:

- *Requirement 1*: It should be distributed, hard real-time, and preferably safety-critical, in which deadline misses can lead to catastrophic results, as our stress test technique tries to force the system to exhibit distributed traffic faults which will, in turn, lead to (hard) real-time faults.

- *Requirement 2*: The system should be preferably data-intensive, i.e., a distributed system in which most (or at least some) of the messages exchanged among distributed nodes have large data sizes. Again, our stress test technique tries to find the most data intensive distributed messages and produces schedules so that such messages run concurrently.
- *Requirement 3*: It should be possible to run the system on a typical hardware/software platform in a research institute. We can replace the embedded components and special hardware with test stubs or component simulators, if necessary.

Since our stress test technique needs a SUT's design model, a suitable case study should also meet the following requirements in terms of modeling technique:

- *Requirement 4*: The design model or source code of the system should be available. The design model can be built by reverse engineering the source code. However reverse engineering of UML models of a system from its source code is usually costly for large systems.
- *Requirement 5*: The design model should be in UML 2.0, since our test technique needs it to be so. Since UML 2.0 has enhanced compared to its previous versions, models based on UML 1.x are also suitable.

12.2.2 None of the Systems in our survey Meets the Requirements

None of the existing systems we are aware of meets all of the above requirements. We provide a brief, structured summary below:

- *Requirement 1*: Not all distributed systems we surveyed are hard RT and safety-critical. For example QADPZ (Quite Advanced Distributed Parallel Zystem) [111] is a distributed system, but does not have safety-critical constraints.
- *Requirement 2*: None of the systems we surveyed which meets other requirements also meets this one, such as the RT distributed factory automation system [112] which was RT, but not data intensive.
- *Requirement 3*: Most systems need special software/hardware platforms to run on, which can not easily be deployed and executed in an academic setting, like ours. We are even flexible in replacing the embedded components and special hardware with test stubs or component simulators, if possible. However, doing this for a complex system is not easy, for example COACH (Component Based Open Source Architecture for Distributed Telecom Applications) [113] seemed to satisfy the other requirements, but it did not have a well-document design. Considering the huge size of the system (in the order of million lines of code), we estimated that we can not have a good understanding of the system architecture and carefully find the dependant components of each component and embed test stubs for specific hardware components that we did not have and were required to run the system.
- *Requirement 4*: The systems model/source code are not freely available or even not available at all. This is either due to the fact that this information is sensitive and/or classified, such as for JITC (The Joint Interoperability Test Command)

[114] and [104], or to the fact that the systems are very expensive, such as CitectSCADA [115] and ElipseSCADA [116].

- *Requirement 5*: As a corollary of our discussion on requirement 4, no UML 2.0 model of the systems in our selection pool was available.

12.3 Our Prototype System: A SCADA-based Power System

Because none of the systems we surveyed meets the requirements, we decided to analyze, design, and build our own prototype system by using the ideas and concepts from existing distributed system technologies.

The Specifications of our *SCAda-based Power System* (SCAPS) is described in Section 12.3.1. In Section 12.3.2, we discuss how and why SCAPS meets our case study requirements. We present the SCAPS' UML design model in Section 12.3.3. Our stress test objective is presented in Section 12.3.4. Relevant implementation issues are presented in Section 12.3.5. Section 12.3.6 provides a brief description of SCAPS' hardware and configuration.

12.3.1 SCAPS Specifications

We intend to design a SCADA power system which controls the power distribution grid across a nation consisting of several provinces. Each province has several cities and regions. Each city and region has several local power distribution grids. There is one central server in each province which gathers the SCADA data from Tele-Control units (TCs) from all over the province, installed in local grids, and perform the following real-time data-intensive safety-critical functions as part of the *Power Application Software* installed on the SCAPS servers:

- *Overload monitoring and control (OM and OC)*: Using the data received from local TCs, each provincial server identifies the overload conditions on a local grid and cooperates with other provinces' servers to reduce the load on overloaded local grids. If the grid stays overloaded for several seconds and the load does not get decreased, a system malfunction is to occur, such as hardware damage and regional black-out. Due to the safety-critical nature of the system, overload monitoring and overload control should be performed in less than 1,300 and 1,000 ms, respectively. Note that these values are in the range of real deadline values for the OM and OC operations in operational SCADA-based power systems [103-107].
- *Detection of separated power system*: Any separated (disconnected) grid should be identified immediately by the central server, and proper precautions should be made to balance the regional/provincial/national load due to this black-out so that the rest of the system stays stable. Due to the safety-critical nature of the system, detection of separated power system should be performed in less than 1,300 ms.
- *Power restoration after network failure*: This functionality provides emergency strategies to prevent network disruption just after a network fault and later presents strategies and switching operation of breakers and disconnectors to restore power while keeping network's reliability. Due to the safety-critical nature of the system, this functionality should be performed in less than 1,000 ms.

It should be noted that we only focus on the real-time data-intensive safety-critical functions of the SCAPS here. Therefore, our stress test technique will be more effective in revealing faults if it is applied to such functions (use-cases) of a SUT. The above three

functions are typical functions performed by SCADA power systems [104, 117], and will be shown in a use case diagram (Section 12.3.3.1), where we present the partial UML model of SCAPS. Some of the non real-time, non safety-critical functions of these systems, which we do not consider in our system, are [104, 117]:

- *State estimation*: Estimates most likely numerical data set to represent current network
- *Load forecasting*: Anticipates hourly total loads (24 points) for 1-7 days ahead based on the weather forecast, type of day, etc. utilizing historical data about weather and load.
- *Power flow control*: Supports operators activities by providing effective power flow control by evaluating network reliability for each several-minute time period for the next several hours, considering anticipated total load, network configuration, load flow, and contingencies.
- *Economical load dispatching*: Controls generator outputs economically according to demand considering the dynamic characteristics of boiler controller of thermal power generators while keeping ability to respond quickly to sudden load changes.
- *Unit commitment of generator*: A suitable schedule for starting/stopping the generators for the next 1-7 days is made using dynamic programming.

12.3.2 SCAPS Meets the Case-Study Requirements

To justify our decision, we discuss below how SCAPS meets all the requirements in Section 12.2.1:

- *Requirement 1*: SCAPS is distributed, hard real-time, and safety-critical, as discussed in Section 12.2.1.
- *Requirement 2*: TCs send large amounts of information about the status and load of each component in their distribution grid to the provincial servers. SCAPS is therefore data-intensive.
- *Requirement 3*: We design and build a SCAPS prototype, using the architecture of existing similar systems. We had, however, to account for the limitation of our research center's hardware/software platforms when designing and implementing the system in such a way to preserve the realism of our case study. For example, we did not have access to dedicated power distribution hardware such as load meters and sensors and we used stubs to emulate their behavior.
- *Requirement 4*: We develop the SCAPS UML model and source code, hence ensuring we have a complete set of development artifacts.
- *Requirement 5*: Our SCAPS models make use of UML 2.0.

12.3.3 SCAPS UML Design Model

Consistent with the SCAPS specification in Section 12.3.1, its partial UML model is provided below. What we mean by a partial model is one which mostly includes the model elements required by our stress test approach, as discussed in Chapter 5. The UML model, presented in this section, consists of the following artifacts:

- Use-Case diagram: Although this diagram is not needed by our testing technique, we present it to provide the reader with a better understanding on the overall functionality of the system.
- Network deployment diagram

- Class diagram
- Sequence diagrams
- Modified Interaction Overview Diagram (MIOD)

12.3.3.1 Use-Case Diagram

We design SCAPS to be used in Canada. To simplify the design and implementation, we consider only two Canadian provinces in the system, Ontario (ON) and Quebec (QC). For example, OM_ON stands for overload monitoring for the province of Ontario; and DSPS_QC stands for Detection of Separated Power System (DSPS) for the province of Quebec.

The use-case diagram is shown in Figure 114. A timer actor triggers Overload Monitoring (OM) and Detection of Separated Power System (DSPS) use-cases according to specified arrival patterns.

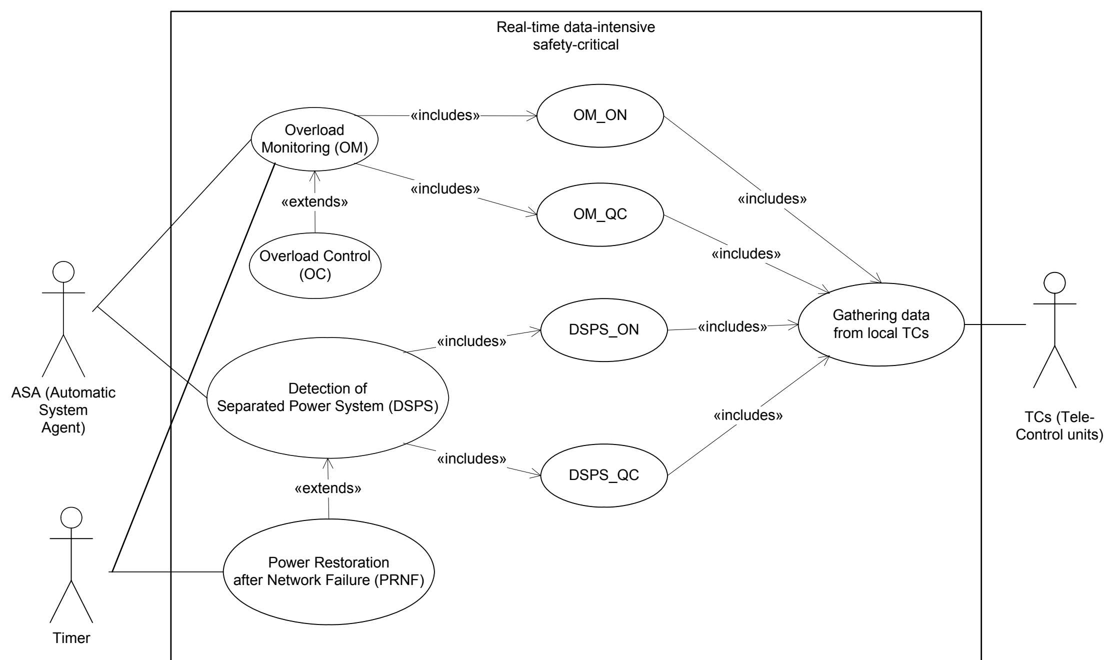


Figure 114- Modified SCAPS Use-Case Diagram including a Timer actor.

12.3.3.2 Network Deployment Diagram

The Network Deployment Diagram (NDD) of SCAPS is shown in Figure 115.

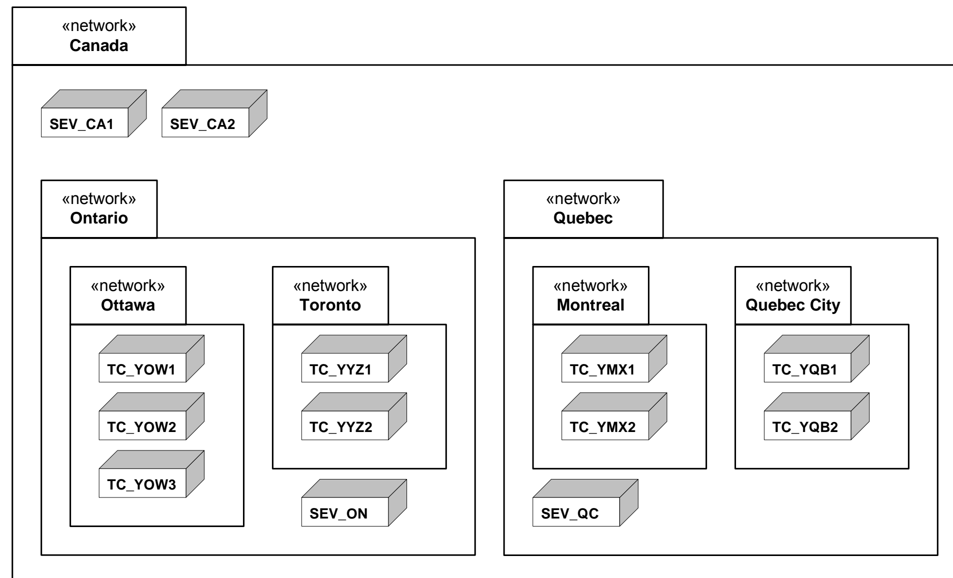


Figure 115- SCAPS network deployment diagram.

The networks for the provinces of Ontario and Quebec are shown in the NDD. Only two cities are considered in each of these two provinces. Three TCs (Tele-Control units) are considered for the city of Ottawa, while other cities have two TCs. There is one server (*SEV_ON* and *SEV_QC*) in each of the provinces. There are two servers at the national level: *SEV_CA1* is the main server. *SEV_CA2* is the backup server, i.e., it starts to operate whenever the main server fails.

12.3.3.3 Class Diagram

Part of the SCAPS class diagram which is required to illustrate the case study is shown in Figure 116. The classes are grouped in two groups: entity and control classes [49, 118]. Entity classes are those which are used either as parameters (by inheriting from *SetFuncParameter*) or return values (by inheriting from *QueryFuncResult*) of methods of

control classes. Control classes are those from which active control objects will be instantiated and are the participating objects in SDs. All entity classes are data-intensive, since grid and load data of power systems for each region (or city) usually contain huge amounts of data [106, 119].

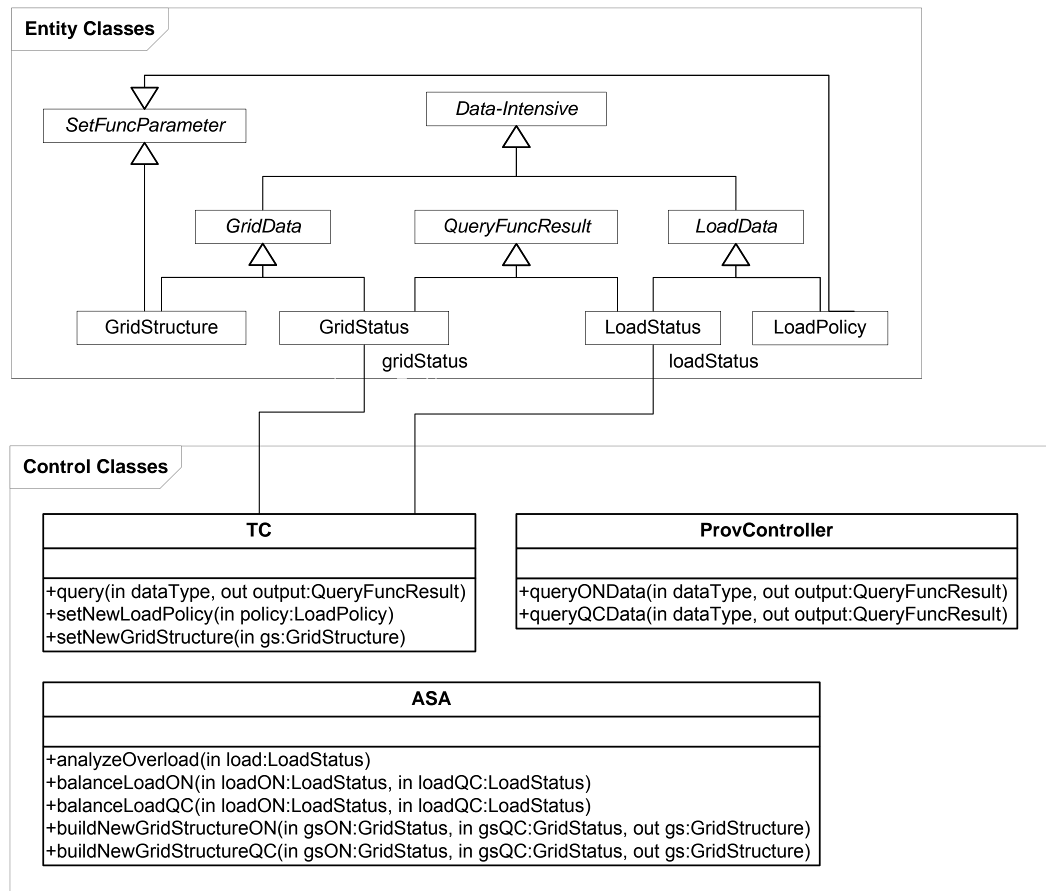


Figure 116-SCAPS partial class diagram.

Furthermore, since there are two main groups of use-cases (overload and separated grid handlers), we group entity classes by two abstract classes *GridData* and *LoadData*. *LoadStatus* and *GridStatus* are the results of function *query* in class *TC* and *queryONData* and *queryQCData* in class *ProvController*. *LoadPolicy* and *GridStructure* are the parameters of set functions *setNewLoadPolicy* and *setNewGridStructure* in class

TC, respectively. For brevity, usage dependencies among classes have not been shown in the class diagram, e.g. from *ProvController* to *QueryFuncResult*.

Tele-Control (TC) unit objects will be instantiated from class *TC*. Objects of class *ProvController* and *ASA* will be deployed on provincial (*SEV_ON* and *SEV_QC*) and national servers (the main server *SEV_CA1* and the backup *SEV_CA2*), respectively.

12.3.3.4 Sequence Diagrams

To render the effort involved in our case study manageable, we simplified the design model and implementation of SCAPS by only accounting for a subset of use cases and by implementing stubs simulating some of the functionalities of the system. In doing so, we tried to emulate as closely as possible the behavior of real SCADA-based power systems.

More precisely, we designed the SDs in ways that the simplifications did not impact the types of faults (e.g., RT faults) targeted by our stress test technique. We incorporated enough messages and alternatives in SDs to allow the generation of non-trivial stress test requirements. Since we designed SCAPS as a hard RT system, we therefore modeled the RT constraint using the UML SPT profile [12].

Eight SDs are presented in Figure 117-Figure 122. They correspond to use-cases in the SCAPS use-case diagram (Figure 114). SDs *OM_ON* and *OM_QC* in Figure 117 correspond to the overload monitoring use case. For example, an object of type *ASA* (Automatic System Agent) sends a message to an object of type *ProvController* (provincial controller) in SD *OM_ON* to query Ontario's load data. The result is returned and is stored in *ASAloadON*. The object of type *ASA* then analyzes the overload situation by analyzing the *ASAloadON*.

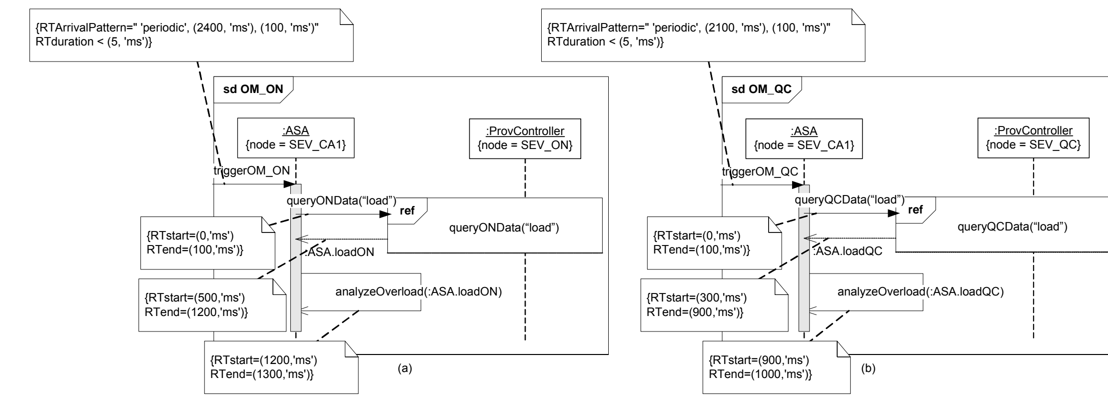


Figure 117- SDs *OM_ON* and *OM_QC* (Overload Monitoring).

To find out how the timing information of the SDs in this section have been devised, refer to Section 12.3.3.5.

The two SDs in Figure 118 (*queryONData(dataType)*) and Figure 119 (*queryQCData(dataType)*) are utility SDs which are used by the other SDs using the *InteractionOccurrence* construct. As it was shown in the Network Deployment Diagram (NDD) of SCAPS (Figure 115), five TCs (Tele-Control units) were considered for the province of Ontario. Therefore, there is a parallel construct made up of five interactions in the SD of Figure 118 which queries the load data from each of the five TCs. Reply messages in *queryONData(dataType)* and *queryQCData(dataType)* have been labeled based on the name of the sender object. For example, the reply message *YOW1* is a reply to the load query from the TC deployed on the node *YOW1* (one of the TCs in the city of Ottawa). The entire load data of each province is finally returned by an object of type *ProvController* to the caller.

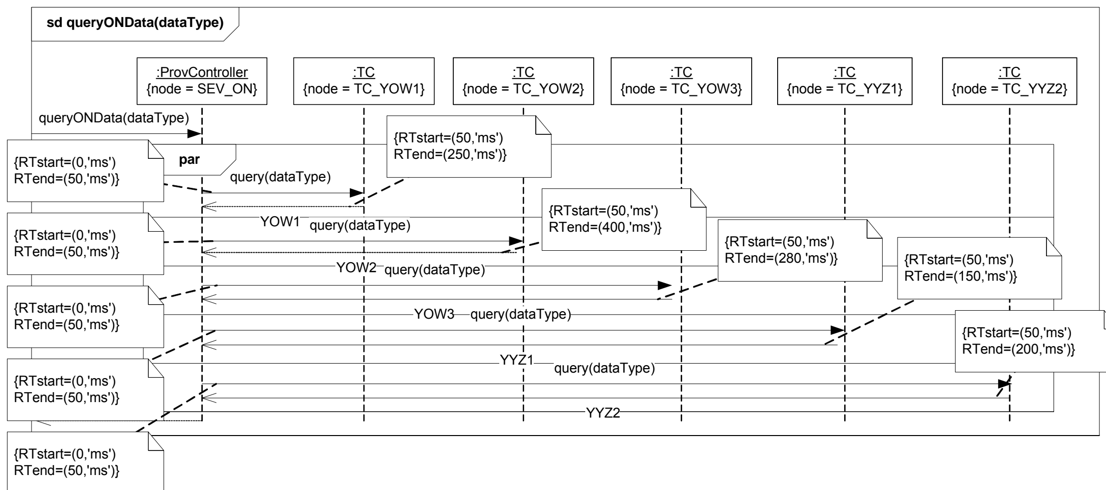


Figure 118-SD *queryONData(dataType).*

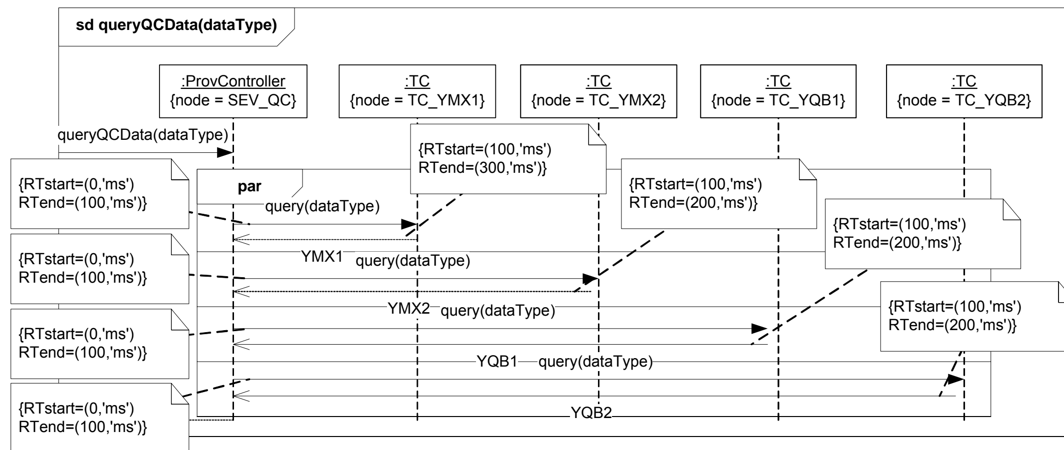


Figure 119-SD *queryQCData(dataType).*

OC (Overload Control) SD (Figure 120) checks if there is an overload situation in any of the two provinces (using *overloadIn()* as a condition). If this is the case in any of the two provinces, a new power distribution load policy is generated by an object of type *ASA* and it is sent to the respective provincial controller (using *setNewLoadPolicy()*).

Similar to the *OM_ON* and *OM_QC* SDs, *DSPS_ON* and *DSPS_QC* SDs (Figure 121) fetch grid connectivity data from the provincial controllers and check whether there is any separated power system (using *detectSeparatedPS()*).

Similar to the *OC* SD (Figure 120), *PRNF* (Power Restoration after Network Failure) SD (Figure 122) checks if there is any separated power system in any of the two provinces (using *anySeparationIn()* as a condition). If this is the case in any of the two provinces, a new power grid structure is generated by an object of type *ASA* and it is sent to the respective provincial controller (*setNewGridStructure()*).

We assign to SCAPS SDs realistic arrival pattern constraints. A realistic periodic arrival pattern value must be larger than the execution duration of the SD it is assigned to. This is because an invocation of the SD should complete execution before it is re-executed (due to a new event according to its arrival pattern). For example, recall from Section 12.3.3.4 that SD *OM_ON*'s duration is 1300 ms. We assume a periodic arrival pattern value of, say, 2400 ms for it. Similarly, since the durations of *DSPS_ON* and *DSPS_QC* are 1300 and 1100 ms, periodic arrival patterns with period and deviation values of 1700, 200 ms, and 1400, 200 ms are assigned to them, respectively. To account for time delays in real-world, a deviation of 100 ms is considered for this periodic arrival pattern. Since SCAPS is a reactive system, we assign periodic arrival pattern to its SDs. Reactive systems usually check for environment changes periodically and take appropriate actions.

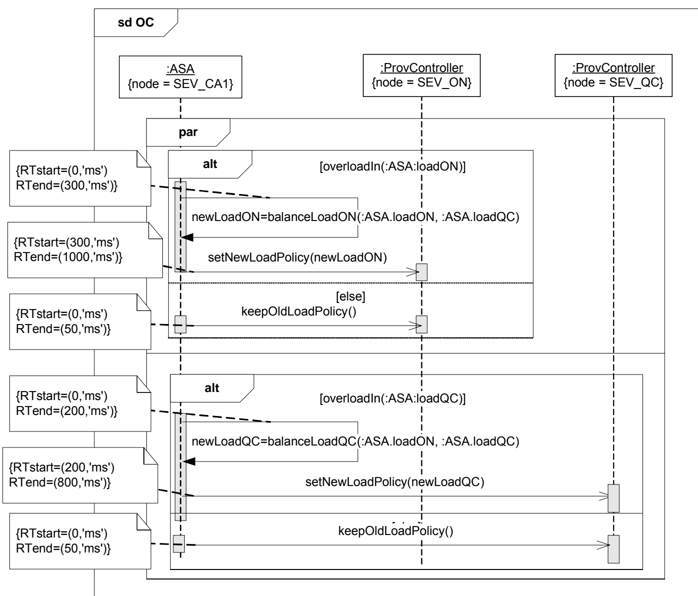


Figure 120- SD *OC* (Overload Control).

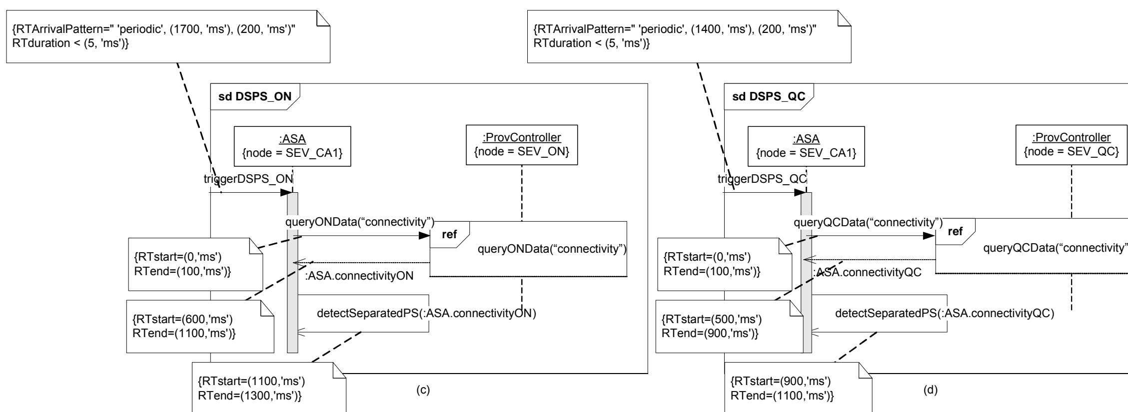


Figure 121-SD *DSPS_ON* and *DSPS_QC* (Detection of Separated Power System).

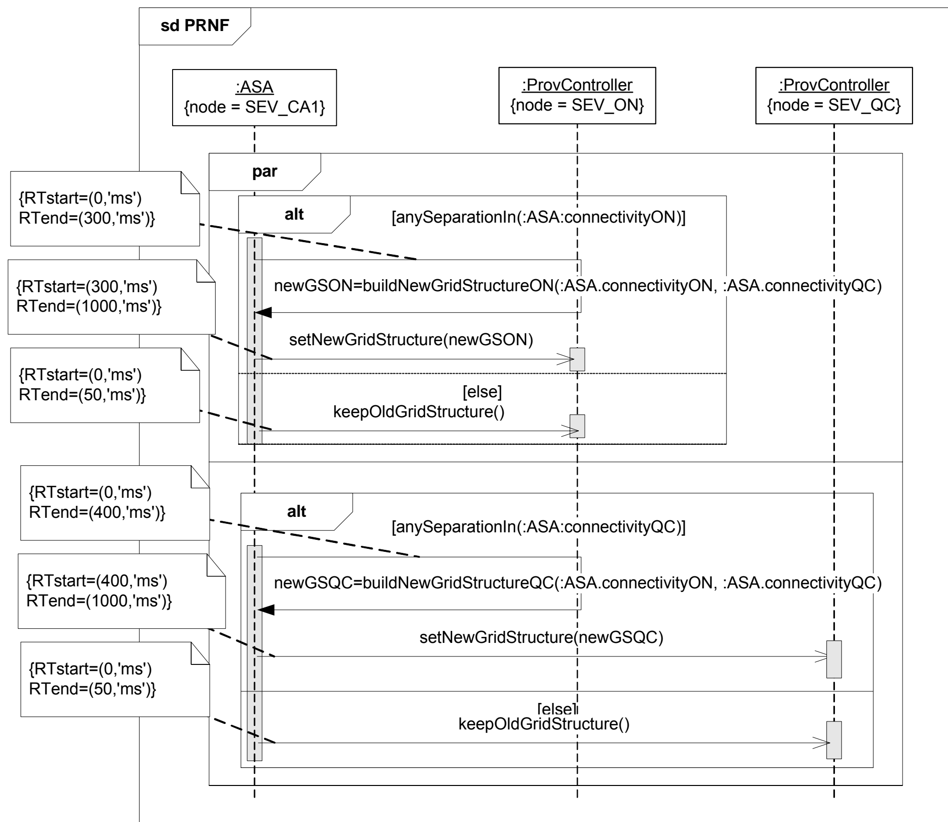


Figure 122-SD *PRNF* (Power Restoration after Network Failure).

12.3.3.5 Timing Information of Messages

Recall from Section 5.1.1.1 that three of the approaches used in the literature to estimate timing information of messages in DRTSs are: (1) Static analysis and manual estimations (such as [55]), (2) Runtime monitoring (such as [56]), and (3) Benchmarks (such as [49]). In a real development environment, one of those techniques will necessarily be used.

Since our case study did not take place in a real development environment, the timing information of the SDs in the previous section was derived according to a specific procedure (a type of runtime monitoring) that is described next. Once a running system was developed, the code was instrumented to log message durations. Such a runtime monitoring technique was used to get a statistical overview of the time length of

messages at runtime prior to the testing phase. Statistical distributions of start and end times of messages were derived by running the system and were used to set the message timing information as specified in the SDs in the previous section. One important objective in our data collecton process was to reduce the probe effects (due to monitoring) as much as possible. We followed ideas from [1] to do so, e.g., having simple, short log statements. Our measurements of the durations of those log statements confirmed that their execution durations were negligible compared to the durations of the measured messages, thus providing a minimal probe effect. Furthermore, since we were using dedicated PC/OS configurations for our measurements (i.e., no other major application was running on the machines), negligible side effects from the OS and other applications affected our process. Refer to Sections 12.3.5 and 12.3.6 for further details on SCAPS implementation details and runtime configurations. Furthermore, to simplify the scheduling calculations in Sections 12.7 and 12.8, message durations were rounded to the closest 100 ms, e.g., 86 ms for message *queryONData()* was rounded to 100 ms.

One other objective in estimating timing information was to obtain realistic values. We considered two aspects: (1) system executions must be based on an operational profile, and (2) real-time constraints from real SCADA systems should be accounted for. SCAPS was executed according to an operational profile (Section 12.8.2.1), which will also be used as a baseline for the comparison of our test results. Furthermore, recall from Section 12.3.1 that, due to the safety-critial nature of the system, overload monitoring and overload control was specified to be performed in less than 1,300 and 1,000 ms, respectively. Note that these deadlines are realistic ranges according to the SCADA literature [103-107]. To have values in realistic ranges, the following control procedure

was followed. Since the system used as a case study was a prototype developed for the specific purpose of our analysis, the design variables (e.g,, the data size of message parameters transmitted over the network) were controlled in a way that the summation of average message durations (derived from the above runtime monitoring technique) be within the real-time deadline for SDs with real-time constraints. Such a control was done in part by manipulating data sizes of function parameters and return values, e.g., parameter *ASALoadON* in Figure 117. For example, if a certain setting for data sizes of function parameters led to a deadline miss at runtime, the data size of the function parameter(s) leading to the deadline miss were reduced. Thus, we made sure that the no deadline misses occured in the system execution based on the operational profile, e.g., the total duration of messages in SD *OM* was within 1,300 ms. In other words, having set a SD execution deadline, we derived its message durations by controlling system parameters (design variables). Refactoring was performed to repeatedly change system design parameters after runtime monitoring to make sure that all specified real-time constraints were met.

In the context of real system modeling and testing, the above time estimation procedure will typically be done in the reverse order: after estimating message durations with a certain degree of uncertainty, a conservative SD execution deadline may be set and documented in the system specification. If such a deadline does not satisfy the business logic of a system (i.e., a long conservative duration for a short critical deadline), the relevant system resources (e.g., network bandwidth) have to be increased to satisfy real-time constraints. To study the impacts of uncertainty in estimating timing information,

one would need need to perform sensitivity analysis on our stress test generation technique (Section 10.9).

12.3.3.6 Modified Interaction Overview Diagram

The MIOD of SCAPS is shown in Figure 123. The four dashed edges in Figure 123 are due to the system's reactive nature and indicate that the system will continue repeating the two main functionality (overload monitoring and detection of separated power systems) until it is stopped. Thus, those four edges will not be analyzed when deriving the Independent-SD Sets (ISDSs) of SCAPS in Section 12.6.4.

As denoted in the SCADA-based power systems literature (e.g. [103-105, 107, 120]), such systems have both soft and hard RT constraints. As discussed in Section 5.6, RT constraints can be either specified at the SD level (on messages execution times) or at the MIOD level (on SDs execution times). MIOD-level RT constraints are dependent on SD-level constraints, since a SD's actual execution time is the sum of the messages execution times in one of its CCFPs, which executes in a particular run.

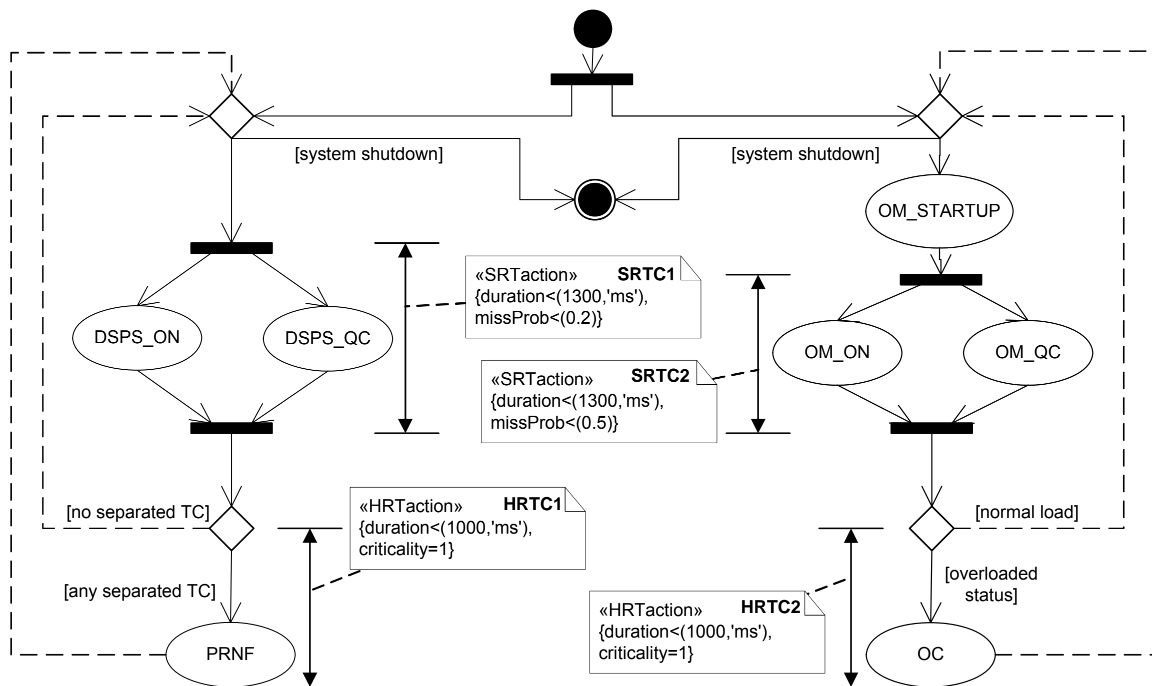


Figure 123-SCAPS Modified Interaction Overview Diagram (MIOD).

We consider four MIOD-level RT constraints for SCAPS. Figure 123 shows two MIOD-level Soft RT (SRT) and two Hard RT (HRT) constraints for SCAPS. We model them using the extended stereotypes (*«SRTaction»* and *«HRTaction»*) from the UML-SPT profile, as proposed in Section 5.6. The constraints are labeled (bold face text) to make it easier to refer to them later, and are explained below.

1. SRT constraints

    a. $SRTC_1$: Detection of separated power systems (concurrent runs of *DSPS_ON* and *DSPS_QC*) should be done in less than 1300 ms, with an acceptable missing probability of 0.2 (20%). In other words, this constraint must not be missed in more than 20% of the runs.

b. $SRTC_2$: Overload monitoring (concurrent runs of *OM_ON* and *OM_QC*) should complete within less than 1300 ms from its start time. We set the acceptable missing probability of this SRT constraint to 0.5.

2. HRT constraints

   a. $HRTC_1$: As soon as a separated power system is detected, the power restoration policy (*PRNF* SD) should be executed in less than 1000 ms. We assign criticality=1 to this constraint.

   b. $HRTC_2$: As soon as an overload situation is detected, overload control policy (*OC* SD) should be executed in less than 1000 ms. We assign criticality[11]=1 to this constraint. As discussed in Section 5.6, criticality of a HRT constraint ranges between 0 (for a HRT constraint with no critical consequences) to 1 (for a constraint with highly critical consequences).

12.3.4 Stress Test Objective

In order to derive test requirements, recall that our stress test technique requires the definition of test objectives according to the following template:

- *Stress location*: either a network or a node name
- *Stress direction (only for nodes)*: in, out or bidirectional. Recall, only a bidirectional stress direction is applicable to a network stress location. Since

[11] As defined by UML SPT profile [12], criticality determines the extent to which the consequences of missing a hard deadline are unacceptable.

networks are not end points of communication, "in" and "out" directions do not apply to them.

- *Stress type*: data or number of messages
- *Stress duration*: instant or interval (with period value)

To stress test SCAPS using GASTT, we consider one of the test objectives chosen in [82]:

Stress Test Objective: *(Canada, -, data, instant)*

12.3.5 Implementation

SCAPS was developed using Borland Delphi[12], which is a well-known IDE (Integrated Development Environment) for RAD (Rapid Application Development). Delphi is an Object-Oriented (OO) graphical toolset for developing Windows applications in Pascal programming language. Delphi was selected as it enables rapid development of prototype applications without spending extensive time on programming details.

We developed a Delphi application for SCAPS. The application asks the user for the node on which it is to run, e.g., *SEV_CA1*, *SEV_ON*, and *TC_YOW1*. Afterwards, the business logic of the application changes accordingly. For example, if *SEV_CA1* is chosen, the application switches to the national server node, waiting for connections from provincial nodes. When different copies of the application on different nodes have been deployed and all nodes connections are ready, the system then starts functioning. A

---

[12] www.borland.com/delphi

screenshot of the main screen of SCAPS is shown in Figure 124, where the application is running as a *SEV_CA1* node and has just accepted a connection from the *TC_YOW1* node.

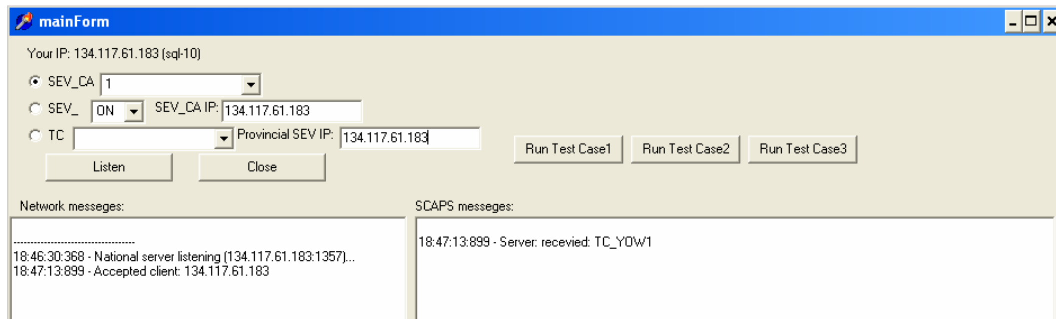


Figure 124-A screenshot of the main screen of SCAPS.

We had to account for the limitation of our research center's hardware/software platforms when implementing the system in such a way to preserve the realism of our case study. The parts of the system for which we had to incorporate stubs to emulate behavior were: (1) dedicated power distribution hardware such as load and connectivity meters and sensors, which are parts of the TC actors (refer to the SCAPS use-case diagram in Figure 114), and (2) complex functionalities of the power application software, such as the *analyzeOverload* function in the *ASA* class to decide whether a load overload situation has occurred, given an instance of the *LoadPolicy* class (refer to the SCAPS class diagram in Figure 116).

As to the design of stubs for the dedicated power distribution hardware, there was no need to try to emulate similar data to what is done in real systems, because as we will see in Section 12.8.1, testing SCAPS in this work is based on triggering specific DCCFPs in specific time instants. To enforce SCAPS to execute specific DCCFPs, we found it easier, in terms of implementation and controllability, to embed a test driver component inside

SCAPS than manipulating data values so that specific edges of decision nodes are taken. The test driver was responsible for guiding the control flow in each conditional statement to follow the edges specified by a test case. In terms of returned values by stubs for the dedicated power distribution hardware, for example function *query()* of class *TC*, they only return a random large data object.

The implementation of stubs for complex functionalities of the power application software was similar to that of the dedicated power distribution hardware. The results generated by such functions were not really needed in our context to execute test cases. However, we had to make sure the durations of such functions were as close as possible to real world situations. We made realistic assumptions in such cases using the power systems literature [103-105, 107, 120], e.g., we assumed that function *analyzeOverload* of class *ASA* takes 100 ms to run (refer to the SDs *OM_ON* in Figure 117). As we had embedded a test driver component inside SCAPS, we could easily use it to make the control flow take specific paths inside each stubbed function.

12.3.6 Hardware and Network Specifications

The *SEV_CA1* server application was deployed on a PC with Windows XP, Pentium 4 2.80 GHz CPU, with 2 GB of RAM and a *3COM Gigabit LOM* network card. The Quebec server *SEV_QC* and its regional tele-control units were deployed on a PC with Windows 2000, 2 GHz CPU, 1 GB of RAM, and a *3COM Fast Ethernet Controller* network card. The Ontario server *SEV_ON* and its regional tele-control units were executed as different applications on a Dell PowerEdge 2600 server with Windows 2000, two Pentium 4 2.8GHz CPUs, and an Intel PRO/1000 *XT* network card. The LAN was a 100 Mbps network.

12.4 Stress Test Architecture

An overview of the SCAPS stress test architecture is shown in Figure 125. The sequence of high-level steps to be performed by a tester to run a complete stress test procedure is shown.

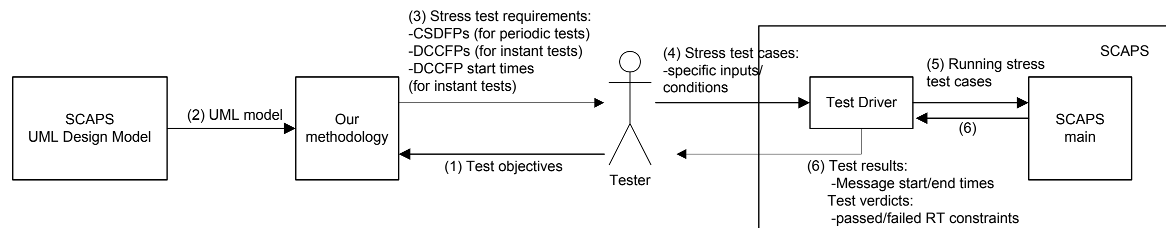


Figure 125-Overview of SCAPS Stress Test Architecture.

The steps are briefly explained below.

(1) The tester feeds the test objectives to the methodology. For example, we considered three test objectives in our case study.

(2) The methodology uses the SCAPS UML model as input.

(3) The methodology uses the SCAPS UML model to generate test requirements for the given test objectives and returns the test requirements to the tester. Note that this step is completely automated.

(4) The tester devises appropriate test case for the test requirements. Note that this step is currently done manually by the tester. The tester feeds the test cases into a test driver which is responsible for running the test cases.

(5) The test driver runs the generated test cases by feeding them into the SUT. Note that we have made the test driver a component of the SCAPS system in our current implementation. Embedding the test driver inside SCAPS helped us simplify the actual test environment and test executions. It also enabled us to reduce the probe

effects (due to monitoring) as much as possible. The probe effects resulting from the test driver were negligible since the test driver only feeds specific test cases and monitors the system. Feeding test cases consisted in setting the attributes of an instance of a test class (in the test driver) to specific values and starting the system. The resulting probe effect in this case was then the time to set specific variables to specific values, which is in the range of several milliseconds, which is negligible when compared to the SCAPS message durations (several hundreds of milliseconds, as it can be seen in the SCAPS SDs in Figure 117-Figure 122). Monitoring SCAPS consisted in exporting the time duration of statements into a log file, which again had very negligible probe effects when compared to executing the statements of SCAPS' main functionalities. Similar to the case when feeding test cases, the statements responsible for monitoring SCAPS have short execution times. We furthermore designed SCAPS to support a high level of controllability[13]. This included features such as flexibility in scheduling DCCFPs (via a scheduler in the test driver).

(6) Test results are gathered from the SUT. They include: start/end times of distributed messages and test verdicts on real-time constraints, which indicate whether each real-time constraint has been adhered to in a particular run. Test results are both logged in

---

[13] Controllability is an important property of a control system and plays a crucial role in many control problems, such as stabilization of unstable systems by feedback, or optimal control [121] Wikipedia, "Definition of Controllability," in *http://en.wikipedia.org/wiki/Controllability*, 2005..

files and also displayed live in a text box to the tester. A high level of observability[14] has been designed in the output interface of SCAPS to better assess the behavior of the system. For example, in order to make it more convenient for the tester to notice real-time faults due to network-aware stress testing, we have incorporated a built-in functionality in the SCAPS main module to monitor the time duration of each message and SD, and report any real-time constraint violation.

12.5 Running Stress Test Cases

As shown in the SCAPS stress test architecture in Figure 125, we developed a test driver module inside SCAPS to run test cases. In running the stress test cases, we adhered to the following general principles to make our test environment as real as possible:

- Since we did not have access to a dedicated network infrastructure[15] to run our prototype tool (SCAPS), we ran all the test cases in late day hours (after 8 PM) and on the weekends in order to mimic a dedicated network and minimize the effects of unpredictable network delays in our test results. In public networks

---

[14] Observability is a measure of how well the internal state of a system can be inferred by knowledge of its external outputs [122] Wikipedia, "Definition of Observability," in *http://en.wikipedia.org/wiki/Observability*, Last accessed: Feb. 2006..

[15] What we mean by a dedicated network is a network which has been designed and devoted to a particular safety-critical system (such as SCAPS) so that no other system is using the network. This is usually done to avoid unpredictable network delays due to indeterministic network traffic and also for security reasons.

(such as our institution's network), the durations of different runs of a distributed data intensive function may be different, due to variable network traffic triggered by other activities in the network.

- Since any distributed system behavior is to some extent indeterministic (multiple runs might exhibit different behavior), we run each test case several times and calculate the mean values of the data collected to account for random variation.

12.6 Building the Stress Test Model for SCAPS

Using the given UML design model in Section 12.3.3, we first build the test model required by our test technique.

12.6.1 Network Interconnectivity Tree

The Network Interconnectivity Graph (NIG) of SCAPS can be derived from the Network Deployment Diagram (NDD) in Figure 115. The NIG is shown in Figure 126.

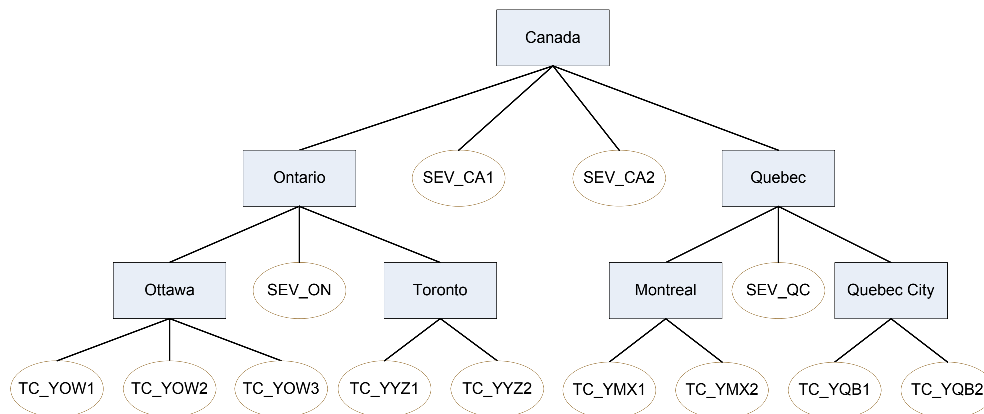


Figure 126- SCAPS Network Interconnectivity Graph (NIG).

12.6.2 Control Flow Analysis of SDs

We presented a technique in Chapter 6 to perform control flow analysis on UML 2.0 SDs. We presented the concept of CCFG (Concurrent Control Flow Graph) as a CFM (Control Flow Model) for SDs. We apply the technique on the SDs of Section 12.3.3.4. CCFGs shown in Figure 127 to Figure 132 correspond to SDs in Figure 117 to Figure 122. CCFGs have been labeled by following the convention: *CCFG(SD_name)*.

Since SD *OM_STARTUP* does not have any distributed message and has only one CCFP, it will not be relevant to our stress testing technique. Hence, there is no need to derive its control flow information.

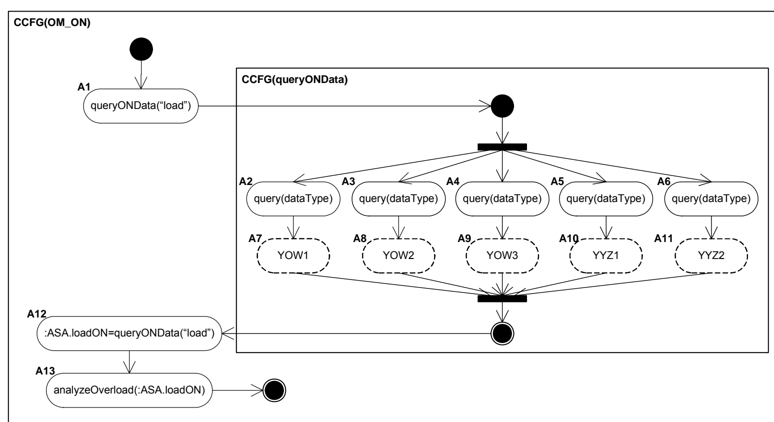


Figure 127-CCFG(OM_ON).

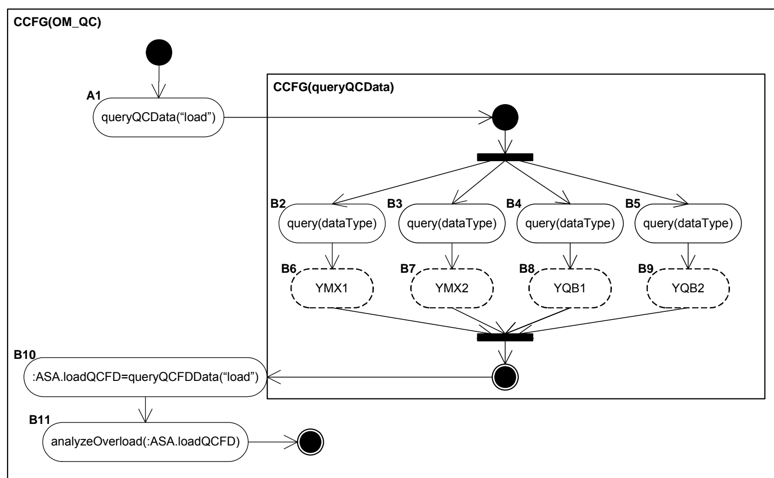


Figure 128-CCFG(OM_QC).

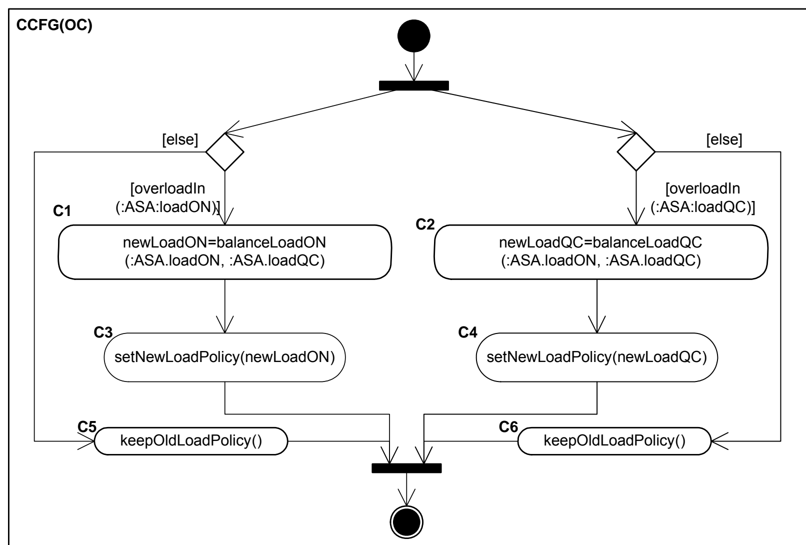


Figure 129-CCFG(OC).

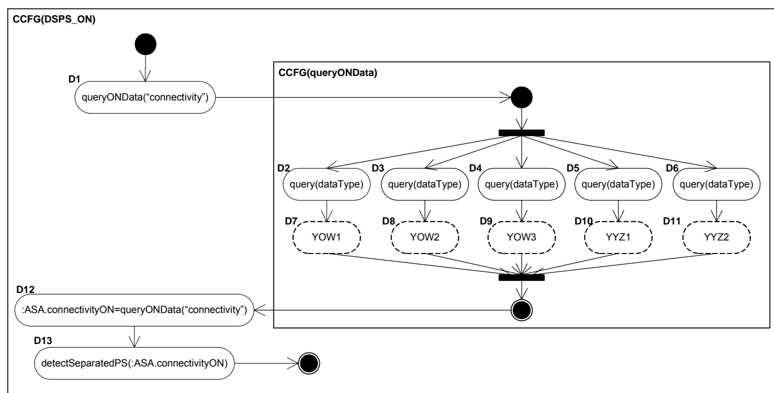


Figure 130-CCFG(DSPS_ON).

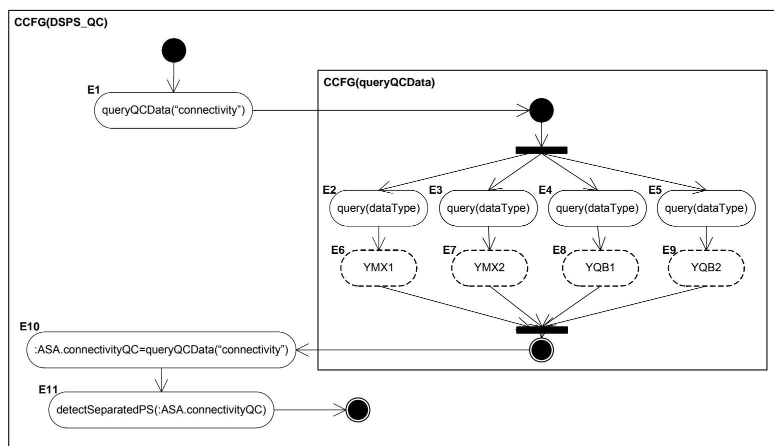


Figure 131-CCFG(DSPS_QC).

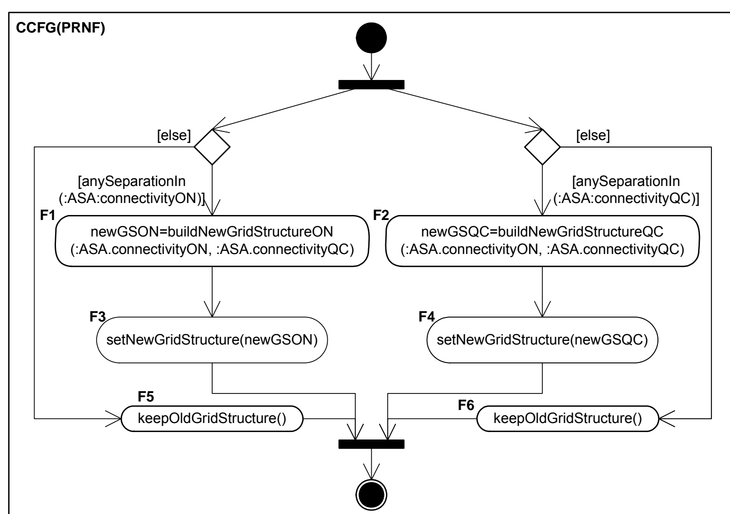


Figure 132-CCFG(PRNF).

12.6.3 Derivation of Distributed Concurrent Control Flow Paths

Using the technique presented in Chapter 6, the CCFPs and DCCFPs are derived from the CCFGs shown in Figure 127 to Figure 132, and are shown in Figure 133. To ease future references, we assign SD_i and $\rho_{i,j}$ indices to SDs and the DCCFPs of each SD, respectively. Let us assign $\rho_{0,0}$ to the only CCFP of SD *OM_STARTUP*, which does not contain any distributed message.

$$CCFP(\underbrace{OM_ON}_{SD_1}) = \left\{ A_1 \begin{pmatrix} A_2A_7 \\ A_3A_8 \\ A_4A_9 \\ A_5A_{10} \\ A_6A_{11} \end{pmatrix} A_{12}A_{13} \right\} \Rightarrow DCCFP(OM_ON) = \left\{ \underbrace{A_1 \begin{pmatrix} A_2A_7 \\ A_3A_8 \\ A_4A_9 \\ A_5A_{10} \\ A_6A_{11} \end{pmatrix} A_{12}}_{\rho_{1,1}} \right\}$$

$$CCFP(\underbrace{OM_QC}_{SD_2}) = \left\{ B_1 \begin{pmatrix} B_2B_6 \\ B_3B_7 \\ B_4B_8 \\ B_5B_9 \end{pmatrix} B_{10}B_{11} \right\} \Rightarrow DCCFP(OM_QC) = \left\{ \underbrace{B_1 \begin{pmatrix} B_2B_6 \\ B_3B_7 \\ B_4B_8 \\ B_5B_9 \end{pmatrix} B_{10}}_{\rho_{2,1}} \right\}$$

$$CCFP(\underbrace{QC}_{SD_3}) = \left\{ \begin{pmatrix} C_1C_3 \\ C_2C_4 \end{pmatrix}, \begin{pmatrix} C_5 \\ C_6 \end{pmatrix} \right\} \Rightarrow DCCFP(OC) = \left\{ \underbrace{\begin{pmatrix} C_3 \\ C_4 \end{pmatrix}}_{\rho_{3,1}}, \underbrace{\begin{pmatrix} C_3 \\ C_6 \end{pmatrix}}_{\rho_{3,2}}, \underbrace{\begin{pmatrix} C_5 \\ C_4 \end{pmatrix}}_{\rho_{3,3}}, \underbrace{\begin{pmatrix} C_5 \\ C_6 \end{pmatrix}}_{\rho_{3,4}} \right\}$$

$$CCFP(\underbrace{DSPS_ON}_{SD_4}) = \left\{ D_1 \begin{pmatrix} D_2D_7 \\ D_3D_8 \\ D_4D_9 \\ D_5D_{10} \\ D_6D_{11} \end{pmatrix} D_{12}D_{13} \right\} \Rightarrow DCCFP(DSPS_ON) = \left\{ \underbrace{D_1 \begin{pmatrix} D_2D_7 \\ D_3D_8 \\ D_4D_9 \\ D_5D_{10} \\ D_6D_{11} \end{pmatrix} D_{12}}_{\rho_{4,1}} \right\}$$

$$CCFP(\underbrace{DSPS_QC}_{SD_5}) = \left\{ E_1 \begin{pmatrix} E_2E_6 \\ E_3E_7 \\ E_4E_8 \\ E_5E_9 \end{pmatrix} E_{10}E_{11} \right\} \Rightarrow DCCFP(DSPS_QC) = \left\{ \underbrace{E_1 \begin{pmatrix} E_2E_6 \\ E_3E_7 \\ E_4E_8 \\ E_5E_9 \end{pmatrix} E_{10}}_{\rho_{5,1}} \right\}$$

$$CCFP(\underbrace{PRNF}_{SD_6}) = \left\{ \begin{pmatrix} F_1F_3 \\ F_2F_4 \end{pmatrix}, \begin{pmatrix} F_5 \\ F_6 \end{pmatrix} \right\} \Rightarrow DCCFP(PRNF) = \left\{ \underbrace{\begin{pmatrix} F_3 \\ F_4 \end{pmatrix}}_{\rho_{6,1}}, \underbrace{\begin{pmatrix} F_3 \\ F_6 \end{pmatrix}}_{\rho_{6,2}}, \underbrace{\begin{pmatrix} F_5 \\ F_4 \end{pmatrix}}_{\rho_{6,3}}, \underbrace{\begin{pmatrix} F_5 \\ F_6 \end{pmatrix}}_{\rho_{6,4}} \right\}$$

Figure 133-CCFP and DCCFP sets of SDs in SCAPS.

12.6.4 Derivation of Independent-SD Sets

Using the method in Section 7.1 and the SCAPS MIOD (Figure 123), we derive SCAPS Independent-SD Sets (ISDSs). We need to first derive the Independent-SDs Graph (ISDG) corresponding to the MIOD. Using the algorithm presented in Section 7.1.2, the ISDG shown in Figure 134 is derived from the MIOD of Figure 123. Note that we do not include SD *OM_STARTUP* in this ISDG, since it does not have any distributed messages. Furthermore, as discussed in Section 12.3.3.6, the four loop-edges (shown as dashed lines in Figure 123) incorporated in SCAPS MIOD due to the system's reactive nature are discarded when deriving the ISDG in Figure 134.

As discussed in Section 7.1.2, this amounts to finding maximal-complete subgraph in a graph. By finding the maximal-complete subgraph of the ISDG in Figure 134, the Independent SD Sets of SCAPS can be derived. SCAPS has seven ISDSs:

$$SDS_1 = \{OM_ON, OM_QC, DSPS_ON, DSPS_QC\} \quad SDS_2 = \{OM_ON, OM_QC, PRNF\}$$
$$SDS_3 = \{DSPS_ON, DSPS_QC, OC\} \quad SDS_4 = \{OC, PRNF\}$$

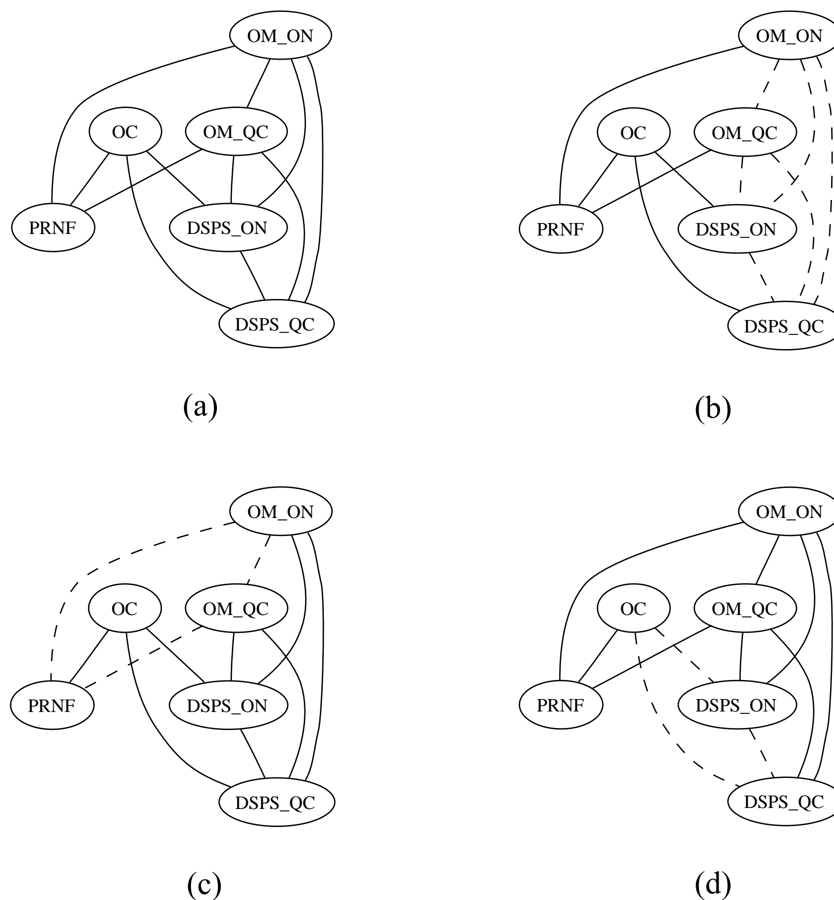


Figure 134-(a):Independent-SDs Graph (ISDG) corresponding to the MIOD of Figure 123. (b), (c) and (d): Three of the maximal-complete subgraphs of the ISDG (shown with dashed edges), yielding three ISDSs.

12.6.5 Derivation of Concurrent SD Flow Paths

Using the method in Section 7.2 and the SCAPS MIOD (Figure 123), we derive SCAPS' Concurrent SD Flow Paths (CSDFPs). As discussed in Section 7.2, in order to derive CSDFPs from a MIOD, we can have an approach similar to the one used in the CFA of SDs (Chapter 6) to derive the CCFPs of a CCFG. Any path from the start node to the final node of the SCAPS MIOD yields a CSDFP.

Since there are loops in the SCAPS MIOD, the number of CSDFPs is infinite. The rationale for having loops in this MIOD is to execute overload monitoring and separated grid detection use cases repeatedly as long as the system is up and running. Referring to the SCAPS MIOD (Figure 123), the control flow may take different paths across multiple *operation cycles* of SCAPS. An operation cycle here denotes when SCAPS revisits the two decision nodes just after the start node in its MIOD and repeats the overload monitoring and separated grid detection scenarios. Therefore, depending on which path is taken in each cycle, different CSDFPs can be derived as modeled by the grammar in Figure 135.

$$CSDFP = \begin{pmatrix} OM_STARTUP\begin{pmatrix} OM_ON \\ OM_QC \end{pmatrix} \\ \begin{pmatrix} DSPS_ON \\ DSPS_QC \end{pmatrix} \end{pmatrix} CSDFP \mid \begin{pmatrix} OM_STARTUP\begin{pmatrix} OM_ON \\ OM_QC \end{pmatrix} OC \\ \begin{pmatrix} DSPS_ON \\ DSPS_QC \end{pmatrix} \end{pmatrix} CSDFP \mid$$
$$\begin{pmatrix} OM_STARTUP\begin{pmatrix} OM_ON \\ OM_QC \end{pmatrix} \\ \begin{pmatrix} DSPS_ON \\ DSPS_QC \end{pmatrix} PRNF \end{pmatrix} CSDFP \mid \begin{pmatrix} OM_STARTUP\begin{pmatrix} OM_ON \\ OM_QC \end{pmatrix} OC \\ \begin{pmatrix} DSPS_ON \\ DSPS_QC \end{pmatrix} PRNF \end{pmatrix} CSDFP \mid \varepsilon$$

Figure 135-A grammar to derive CSDFPs from SCAPS' MIOD.

In order to limit the number of CSDFPs for the purpose of deriving stress test requirements, we limited the number of cycles (in the MIOD) to derive CSDFPs. Some of the CSDFPs which can be derived from the grammar in Figure 135 are shown in Figure 136.

$$CSDFP_1 = \begin{pmatrix} OM_STARTUP \begin{pmatrix} OM_ON \\ OM_QC \end{pmatrix} \\ \begin{pmatrix} DSPS_ON \\ DSPS_QC \end{pmatrix} \end{pmatrix} \quad CSDFP_2 = \begin{pmatrix} OM_STARTUP \begin{pmatrix} OM_ON \\ OM_QC \end{pmatrix} OC \\ \begin{pmatrix} DSPS_ON \\ DSPS_QC \end{pmatrix} \end{pmatrix}$$

$$CSDFP_3 = \begin{pmatrix} OM_STARTUP \begin{pmatrix} OM_ON \\ OM_QC \end{pmatrix} \\ \begin{pmatrix} DSPS_ON \\ DSPS_QC \end{pmatrix} PRNF \end{pmatrix} \quad CSDFP_4 = \begin{pmatrix} OM_STARTUP \begin{pmatrix} OM_ON \\ OM_QC \end{pmatrix} OC \\ \begin{pmatrix} DSPS_ON \\ DSPS_QC \end{pmatrix} PRNF \end{pmatrix}$$

$$CSDFP_5 = \begin{pmatrix} OM_STARTUP \begin{pmatrix} OM_ON \\ OM_QC \end{pmatrix} OC \\ \begin{pmatrix} DSPS_ON \\ DSPS_QC \end{pmatrix} PRNF \end{pmatrix} \begin{pmatrix} OM_STARTUP \begin{pmatrix} OM_ON \\ OM_QC \end{pmatrix} OC \\ \begin{pmatrix} DSPS_ON \\ DSPS_QC \end{pmatrix} \end{pmatrix}$$

$$CSDFP_6 = \begin{pmatrix} OM_STARTUP \begin{pmatrix} OM_ON \\ OM_QC \end{pmatrix} OC \\ \begin{pmatrix} DSPS_ON \\ DSPS_QC \end{pmatrix} \end{pmatrix} \begin{pmatrix} OM_STARTUP \begin{pmatrix} OM_ON \\ OM_QC \end{pmatrix} \\ \begin{pmatrix} DSPS_ON \\ DSPS_QC \end{pmatrix} PRNF \end{pmatrix}$$

Figure 136-Some of the CSDFPs of SCAPS derived from the grammar in Figure 135.

There is only one cycle in the basic CSDFPs: $CSDFP_1$, …, $CSDFP_4$. $CSDFP_5$ and $CSDFP_6$ are two of the possible CSDFPs which can be derived assuming two cycles. Other CSDFPs can be derived by arbitrary concatenations of the basic CSDFPs.

12.6.6 Data Size of Messages

Note that, for brevity, we do not discuss the data structure of the entity data classes in SCAPS (Figure 116). But according to the literature on SCADA-based power systems [103-105, 107, 120], data items such as load status/policy and grid status/structure are usually data-intensive and can be implemented using large data structures such as arrays. As the exact (or statistical average) sizes of this data classes is needed by our stress test technique, we assume the values given in Table 27 as the mean data sizes of the entity data classes in Figure 116. These values are realistic size estimates of real grid and load values according to the literature on SCADA-based power systems [123]. For example, an

instance of the load object of the power distribution grid of a city includes the load values of the different hubs and components of the grid. This value can vary depending on the size of the city as well as the complexity of the distribution grid. We assume the data size to be in the order of several mega-bytes, which is reasonable assumption based on what is reported in the literature.

Note that we assume the data sizes in Table 27 to be representative for instances of all TCs. However, as different TCs are deployed in different cities/regions, the load or grid status data can vary to a large extent. This can be easily accounted for by extending data sub-classes and calculating the corresponding data sizes. For example, sub-classes like *OttawaLoadStatus* and *TorontoLoadStatus* (with different data fields sizes) can be derived from the class *LoadStatus* in Figure 116.

Data Class	**Mean Data Size**
LoadStatus	4 MB
LoadPolicy	2 MB
GridStatus	3 MB
GridStructure	1 MB

Table 27-Mean data sizes of the entity data classes of SCAPS.

12.7 Stress Testing SCAPS by Time-Shifting Stress Test Technique

Since none of SCAPS SDs have arrival pattern constraints, we can use the simpler version of our stress test technique (TSSTT, Chapter 9) to stress test SCAPS. Since we believe that GASTT is more interesting in terms of approach than TSSTT, due to space constraints, we do not report the details and results of our case study with TSSTT in this thesis. Interested readers can refer to [82] for extensive discussion on this topic. Shorter results are also reported in [83]. In that case study, we consider three stress test objectives

and describe how the stress test requirements and test cases corresponding to the chosen test objectives are derived, respectively. We finally present our stress test results.

12.8 Stress Testing SCAPS by Genetic Algorithm-based Stress Test Technique

As a case study for the Genetic Algorithm-based Stress Test Technique (GASTT)-(Chapter 10), we use it to stress test SCAPS. As it can be seen in the SCAPS UML design model (Section 12.3), none of its SDs has arrival pattern constraints. Therefore, to investigate the feasibility of results of stress testing SCAPS with GASTT, we first modify SCAPS SDs by assigning to them realistic arrival pattern constraints.

The SCAPS UML design model (where some SDs are assigned pattern constraints) is presented in Section 12.3.3. A stress test objective is described in Section 12.3.4. The use of GARUS to derive stress test requirements corresponding to the stress test objective is described in Section 12.8.1. Execution results for the derived stress test cases are reported in Section 12.8.2. Finally, Section 12.8.3 draws a set of conclusions.

12.8.1 Using GARUS to Derive Stress Test Requirements

In this section, we use GARUS to derive stress test requirements for the above test objective. The derivation of GARUS input files for the test objective is described in Section 12.8.1.1. The GA execution and the repeatability of its results are discussed in Section 12.8.1.2. The stress test requirements generated by GARUS are presented in Section 12.8.1.3.

12.8.1.1 Input File

Recall from Section 11.2.3 that an input file passed to GARUS contains the test model of a SUT. We furthermore assumed that the test model in an input file has already been

filtered according to the test parameters of a test objective (e.g. stress location, stress direction). 'Filtered' in a sense that, for example, given a set of test parameters, only those SDs messages are included in a TM, which comply with the criteria specified in the test parameters. For example, SD messages going through a specified network are included. We use the SCAPS test model (Section 12.6) to build an input file (Figure 137) corresponding to the above test objective. This input file is based on the format presented in Section 11.2.3.

```
--ISDSs
4
ISDS1 4 OM_ON OM_QC DSPS_ON DSPS_QC
ISDS2 3 OM_ON OM_QC PRNF
ISDS3 3 DSPS_ON DSPS_QC OC
ISDS4 2 PRNF OC
--SDs
6
OM_ON 1 1 p11
OM_QC 1 1 p21
OC 1 4 p31 p32 p33 p34
DSPS_ON 1 1 p41
DSPS_QC 1 1 p51
PRNF 1 4 p61 p62 p63 p64
--SD_Arrival_Patterns
OM_ON periodic 24 1
OM_QC periodic 21 1
OC no_arrival_pattern
DSPS_ON periodic 17 2
DSPS_QC periodic 14 2
PRNF no_arrival_pattern
--DCCFPs
p11 7 ( 5 2.8 ) ( 6 2.8 ) ( 7 2.8 ) ( 8 2.8 ) ( 9 2.8 ) ( 10 2.8 ) ( 11 2.8 )
p21 6 ( 3 2.66 ) ( 4 2.66 ) ( 5 2.66 ) ( 6 2.66 ) ( 7 2.66 ) ( 8 2.66 )
p31 8 ( 2 0.33 ) ( 3 0.61 ) ( 4 0.61 ) ( 5 0.61 ) ( 6 0.61 ) ( 7 0.61 ) ( 8 0.33 ) ( 9
0.33 )
p32 7  ( 3 0.28 ) ( 4 0.28 ) ( 5 0.28 ) ( 6 0.28 ) ( 7 0.28 ) ( 8 0.28 ) ( 9 0.28 )
p33 6 ( 2 0.33 ) ( 3 0.33 ) ( 4 0.33 ) ( 5 0.33 ) ( 6 0.33 ) ( 7 0.33 )
p34 0
p41 5 ( 6 3 ) ( 7 3 ) ( 8 3 ) ( 9 3 ) ( 10 3 )
p51 4 ( 5 3 ) ( 6 3 ) ( 7 3 ) ( 8 3 )
```

```
p61 7 ( 3 0.14 ) ( 4 0.3 ) ( 5 0.3 ) ( 6 0.3 ) ( 7 0.3 ) ( 8 0.3 ) ( 9 0.3 )
p62 7 ( 3 0.14 ) ( 4 0.14 ) ( 5 0.14 ) ( 6 0.14 ) ( 7 0.14 ) ( 8 0.14 ) ( 9 0.14 )
p63 6 ( 4 0.16 ) ( 5 0.16 ) ( 6 0.16 ) ( 7 0.16 ) ( 8 0.16 ) ( 9 0.16 )
p64 0
--GASearchTimeRange
200
```

Figure 137- Input File containing SCAPS Test Model for a GASTT Test Objective.

The first block (--ISDSs) of the input file lists SCAPS' four ISDSs. SD data descriptions (--SDs) then follow. We explained earlier that only one copy of each SD will be triggered in SCAPS at a single time instant (as denoted the parameters set to '1' after SDs' names in *--SDs* block). Numbers and names of DCCFPs for each SD (e.g. DCCFP *p11* for *OM_ON*) are taken from Section 12.6.3. Arrival pattern data are extracted from SDs in Section 12.3.3.4. Note that, for brevity, the time unit is assumed to be 100 ms. For example, the first DTUPP of DCCFP *p11*, states that it entails 2.8 units of traffic in time instant 500 ms (5x100ms). Finally, the GA time search range has been set to 200 time units (20,000 ms). This value was selected using the heuristics in Section 10.7.4.

Our experimentation of different values for the GA time search range also confirmed that 200 time units is a suitable value. To explain how we come to this conclusion, a timing diagram showing the relationship between ATSs of SCAPS SDs and their possible execution durations, shown as data series *d(sd_name)*, in 50 time units is shown in Figure 138.

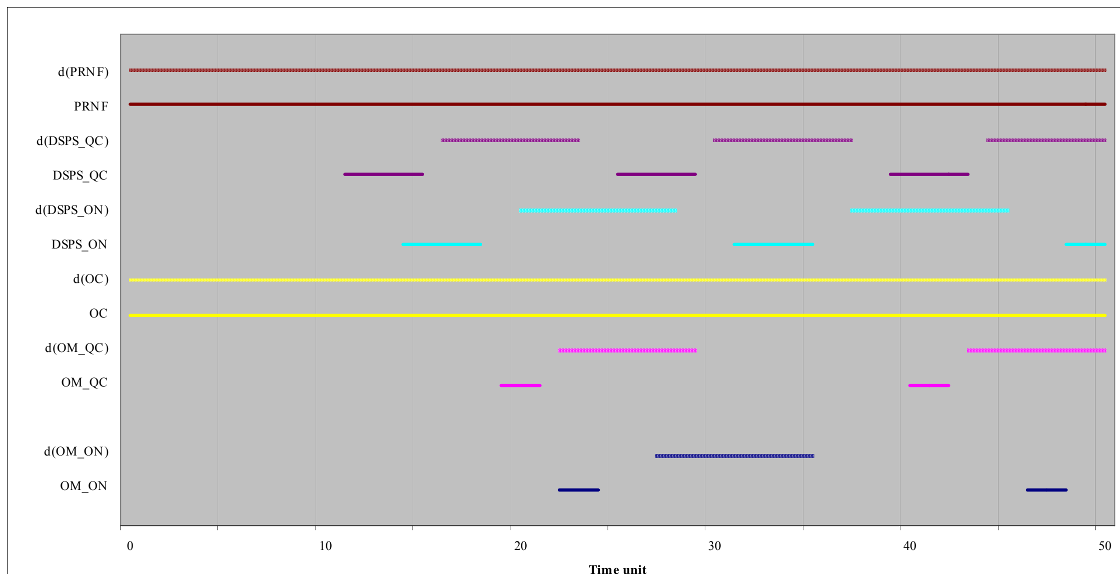


Figure 138-Relationship between ATSs of SCAPS SDs and their execution durations, *d(sd_name)*, to each other in 50 time units.

For example, as the period and deviation values of the periodic AP for SD *OM_ON* are 24 and 1 time units, its ATS has been depicted as ATIs: [23, 25], [47, 49] and etc. The execution duration of a SD is the longest interval made of the difference between the minimum and maximum timing values of a DCCFP of the SD. For example, referring to the input file in Figure 137, the minimum and maximum timing values in the DCCFP *p11* of SD *OM_ON* are 5 and 11 time units. Therefore, considering an ATI such as [23, 25] in which an execution of SD *OM_ON* can be started, the distributed messages of such an execution can take place in interval of [23+5=28, 25+11=36], as depicted in Figure 138. The rationale of analyzing the ATSs and execution durations of SDs is to find a suitable GA time search range. Since SDs *OC* and *PRNF* have no arrival patterns, they can be triggered in any time instant.

As we can easily see in Figure 138, there will not be any chance for our GA algorithm (GASTT) to find a time instant when the execution durations of all six SDs overlap, and thus generating a stress test schedule to entail maximum traffic on the network. This is because the intersection of all execution durations in Figure 138 is simply null. This is why 50 time units is not a suitable GA time search range. We now discuss how the overlapping between ATSs and SD execution durations change by increasing the search range from 50 to, say, 200 time units (Figure 139).

When the search range is increased to 200 time units, the situation changes and as it can be calculated (and seen in Figure 139, the six execution durations will overlap in time instants: 106-107, 128-130, and 173-176. Therefore, there will be chances of overlapped executions of SDs in the random schedules generated during GASTT's operation. This will, in turn, enable GASTT to generate GA individuals which have higher fitness values (entailed traffic values). Increasing the search range to values more than 200 will not increase GASTT's chances in generating better individuals (better stress test requirements), since as long as there is overlapping time among the six execution durations, there are chances to find such test requirements. However, an increase in the search range will deteriorate our GA's performance, since it will take longer time for the GA to converge to a maximum plateau. This is because the selection of random start times for DCCFPs will be sparser (compared to when the range is 200) and GA has to iterate through more generations to settle on a stable maximum plateau (Section 10.7.4). Refer to Sections 11.3.9 for the impacts of variations in GA maximum search time on our GA's performance.

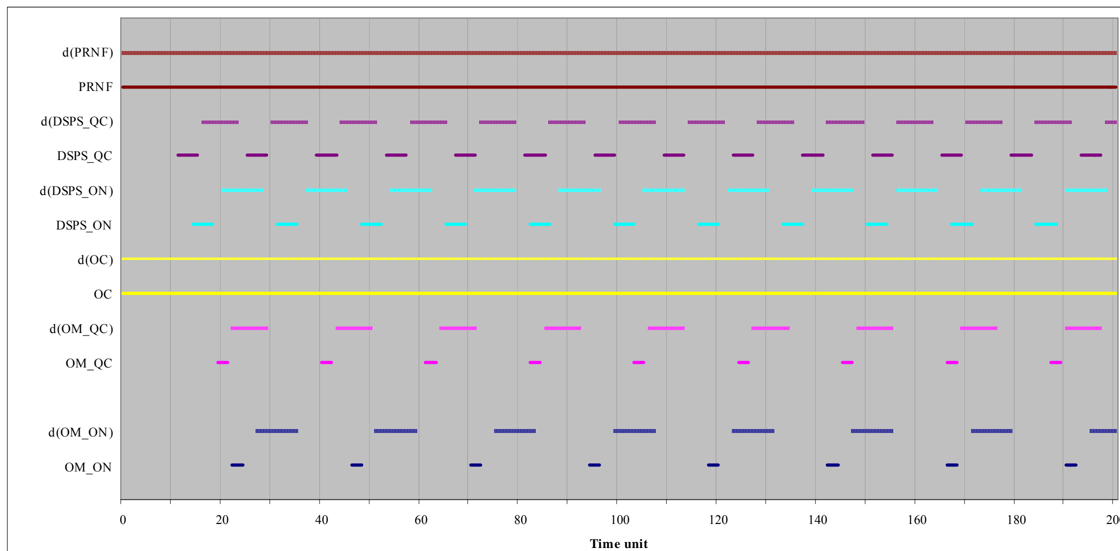
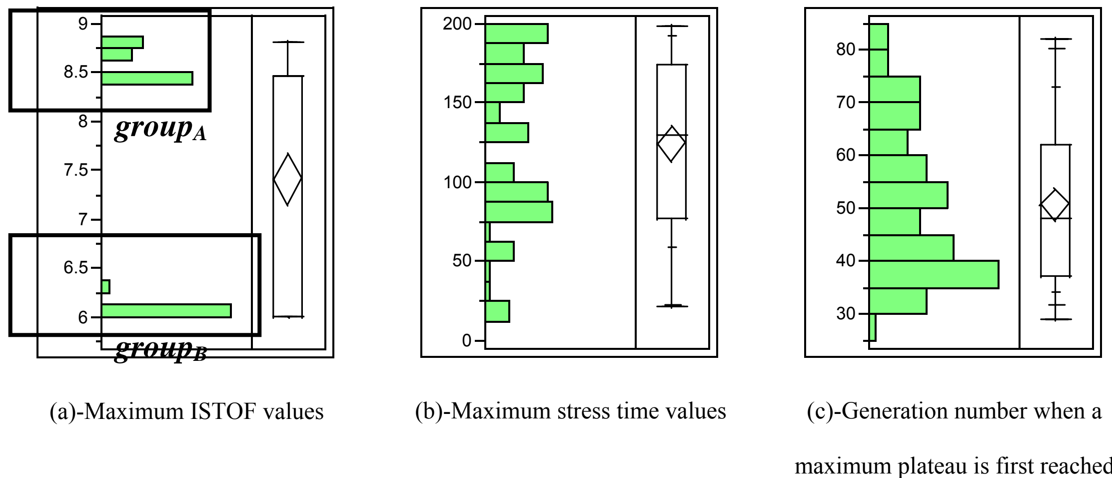


Figure 139--Relationship between ATSs of SCAPS SDs and their execution durations, *d(sd_name)*, to each other in 200 time units.

12.8.1.2 GA Execution and the Repeatability of Results

Similar to discussions in Section 11.3, since GARUS is based on GAs, the stability and repeatability of results across multiple runs need to be investigated as GAs are by definition a heuristic. Therefore, GARUS was executed 100 times with the above input test file. Histograms of maximum ISTOF values, maximum stress time values, and maximum plateau generation number are depicted in Figure 140.

Each execution had the duration of 326 ms on average. Therefore, running GARUS for 100 times was not a practical problem from a time standpoint. As we can see in Figure 140-(a), 100 runs of the test model generated five different ISTOF values. The corresponding maximum stress time values are spanned in the range of [21…198] time units (each time unit=100 ms). As the histogram in Figure 140-(c) shows, GARUS was able to converge to a maximum plateau in 29 to 82 generations (48 on average) across the 100 runs.



(a)-Maximum ISTOF values (b)-Maximum stress time values (c)-Generation number when a maximum plateau is first reached

Figure 140-Histograms of 100 GARUS Outputs for a SCAPS Test Objective.

We now discuss the practical implications of multiple runs of GARUS to get stable results. For the particular case study in this chapter, as it can be easily seen in Figure 140-(a), 100 runs of GARUS has generated mainly two groups of outputs: a group with maximum ISTOF values of between 8.25 and 9 units of traffic ($group_A$ in Figure 140-(a)), and the one with values between 6 and 6.5 ($group_B$ in Figure 140-(a)). Obviously, the goal of using GARUS is to find stress test requirements which have the highest possible ISTOF values. Thus, the strategy is to run GARUS for multiple times and choose a test requirement with the highest ISTOF value across all runs.

The practical implication of multiple runs to achieve a test requirement with the highest ISTOF value is to predict the minimum number of times GARUS should be executed to yield an output with an ISTOF value in $group_A$ in Figure 140-(a). Such an analysis can be performed by using the probability distributions of the above two groups of maximum ISTOF values in the histogram of Figure 140-(a). 54 and 46 (of 100) values in the histogram belong to $group_A$ and $group_B$, respectively. Thus, in a sample population of 100

GARUS outputs, the probabilities that an output belongs to $group_A$ or $group_B$ are $p(group_A)$=0.54 and $p(group_B)$=0.46, respectively.

The two outcomes of an output being in $group_A$ or $group_B$ are two *mutually-exclusive* events[16]. Thus, to predict the minimum number of times GARUS should be executed to yield an output with an ISTOF value in $group_A$, we can use the following probability formula for two independent events:

> For two mutually-exclusive events *A* and *B* with probabilities p_A and p_B, the probability that *A* occurs at least once in a series of *n* samples (runs) is:

$$p(\text{event } A \text{ occurs at least once in a series of } n \text{ samples})=1- {p_B}^{n-1} p_A$$

Substituting $group_A$ and $group_B$ probabilities in the above formula will yield us:

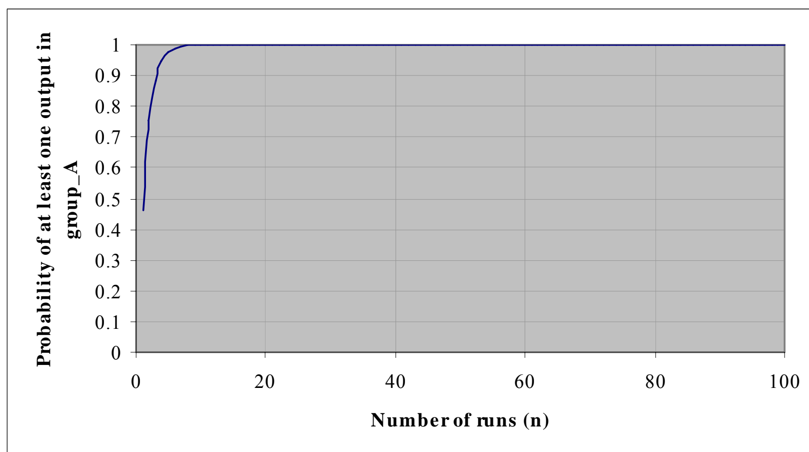
$$p(\text{a test requirement with an ISTOF value in } group_A \text{ is yielded in a series of } n \text{ runs of GARUS})=$$

$$1-p(group_B)^{n-1} p(group_A)=1-(0.46)^{n-1}(0.54)$$

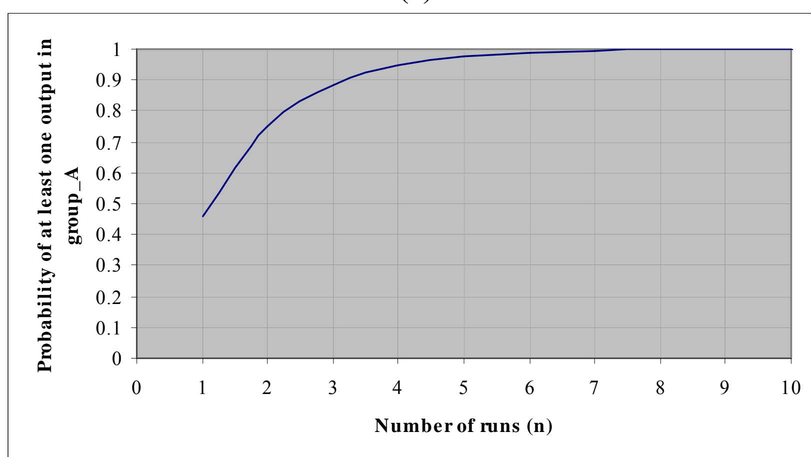
The above probability function is drawn in Figure 141. The probability values (y-axis) are shown in linear scale in Figure 141-(a). Figure 141-(b) depicts a zoom-out of the curve in Figure 141-(a) for n=0…10.

---

[16] In probability theory, two events A and B are said to be mutually-exclusive if event A happens, then event B cannot, or vice-versa.



(a)



(b)

Figure 141- Probability of the event that at least one test requirement with an ISTOF value in $group_A$ is yielded in a series of n runs of GARUS for SCAPS.

As it can be easily seen in Figure 141, the probability values increases exponentially by increasing number of runs (n) and converges to 1 very quickly in values above n=6. Therefore, it can be said that one can achieve a good stress test requirement (with a value in $group_A$) by running GARUS with SCAPS test model for about 6 times and selecting the best output (with highest ISTOF value). Similar probabilistic analysis can be done when using GARUS to generate test requirements for other SUTs.

12.8.1.3 Stress Test Requirements

As discussed above, 100 runs of the test model by GARUS generated five different ISTOF values (Figure 140-(a)). Five of the test requirements reported by GARUS for those five different ISTOF values are shown in Figure 142.

```
SD        DCCFP   start time
----      ----    ----------
OM_ON     none
OM_QC     none
OC        p34     148
DSPS_ON   p41     155
DSPS_QC   p51     156
PRNF      none
```

(a)- *TR1* (ISTOF=6)

```
SD        DCCFP   start time
----      ----    ----------
OM_ON     none
OM_QC     none
OC        p32     149
DSPS_ON   p41     151
DSPS_QC   p51     152
PRNF      none
```

(b)- *TR2* (ISTOF=6.28)

```
SD        DCCFP   start time
----      ----    ----------
OM_ON     p11     23
OM_QC     p21     21
OC        none
DSPS_ON   p41     19
DSPS_QC   p51     12
PRNF      none
```

(c)- *TR3* (ISTOF=8.46)

```
SD        DCCFP   start time
----      ----    ----------
OM_ON     p11     119
OM_QC     p21     188
OC        none
DSPS_ON   p41     185
DSPS_QC   p51     183
PRNF      none
```

(a)- *TR4* (ISTOF=8.66)

```
SD        DCCFP   start time
----      ----    ----------
OM_ON     p11     71
OM_QC     p21     85
OC        none
DSPS_ON   p41     70
DSPS_QC   p51     68
PRNF      none
```

(b)- *TR5* (ISTOF=8.8)

Figure 142-Five Different Test Requirements (TR) generated by GARUS for a SCAPS Test Objective.

As we can see, some of the test requirements (TR) generated by GARUS have different combinations of DCCFPs (e.g. *TR1*, *TR2* and *TR3*), while some have the same combinations of DCCFPs, but with different schedules. For example, in *TR3*, *TR4* and *TR5,* DCCFPs *p11*, *p21*, *p41* and *p51* have been selected with different start times.

Since our stress test goal is to maximize the amount of traffic, we choose *TR5*, and stress test SCAPS with its corresponding test case. The test requirement is to trigger DCCFPs

p11, *p21*, *p41* and *p51* from SDs *OM_ON*, *OM_QC*, *DSPS_ON* and *DSPS_QC* in time units 71, 85, 70 and 68 respectively.

12.8.2 Test Results

Our fundamental approach to show the usefulness of our stress test technique in this work is to observe the system and analyze the RT-constraint violations due to specific schedules and subsets of DCCFPs, as generated by our technique. Results are then compared with what we refer to as *Operation Profile-based Test Cases (OPTC)*, which act as the baseline of comparison described in Section 12.8.2.2. We discuss in Section 12.8.2.1 how we derive OPTCs.

In the presentation of the test results, we compare the statistical start and end times of distributed messages (recall that we run test cases several times) and also determine if a stress test case causes a RT-constraint violation. This will help us assess whether our methodology is useful in terms of increasing the chances of exhibiting network traffic faults which lead to RT failures.

Note that, in the test results we report, we analyze and discuss the MIOD-level soft and hard RT constraints described in Section 12.3.3.6. SD-level constraints can be defined in a similar way and the corresponding test results can also be analyzed.

12.8.2.1 Baseline of Comparisons

We define here the baseline of comparison we use to assess the effectiveness of our stress test case. We consider Operation Profile-based Test Cases (OPTC) which are derived from the operational profile [77] of a SUT. The operational profile of a system is defined

as the expected workload of the system once it is operational in the field. In other words, OPTCs actually test a SUT in terms of its expected behavior in the field.

To derive OPTCs for SCAPS, we present an operational profile, which takes into account the system's business logic in the context of SCADA-based power systems. Using the SCAPS MIOD (Figure 123) and CCFGs (Figure 127 to Figure 132), we model the operational profile to be the probabilities for the true and false edges of conditions to be taken. More precisely, we focus on the decision nodes in the CCFGs of SDs *OC* and *PRNF*, Figure 129 and Figure 132, respectively. These two CCFGs are the only CCFGs of SCAPS where there are alternatives in control flow.

The two decision nodes in *CCFG(OC)* check for overload status in load data for the provinces of Ontario and Quebec. In case of overload in any of the provinces, a new load policy is generated and is sent to the respective provincial server. Otherwise, a message is sent to keep the old policy. As power systems are designed in such a way to minimize the chances of overload, we assume the probabilities that the Ontario and Quebec grids experience overload are %1 and %2, respectively. Thus the probabilities that the control flow in decision nodes *CCFG(OC)* will follow the overload paths will be the same.

The above control flow path probabilities can be expressed as the operational profile of SCAPS shown in Table 28, where probabilities of SD per province are shown and have been mapped to paths after decision nodes.

SD	CCFG	Function-Province	Path after Decision Node in CCFG	Probability
OC	CCFG(OC)	Overload monitoring-Ontario	Ontario overload	%1
			Ontario normal load	%99
		Overload monitoring-Quebec	Quebec overload	%2
			Quebec normal load	%98
PRNF	CCFG(PRNF)	Detecting separated power system -Ontario	Separated power system (SPS) in Ontario	%0.5
			No SPS in Ontario	%99.5
		Detecting separated power system - Quebec	Separated power system in Quebec	%0.25
			No SPS in Quebec	%99.75

Table 28-An operational profile for SCAPS.

For example, Figure 143 shows a part of *CCFG(OC)* with edges outgoing from decision nodes annotated with probabilities. The probability of an edge after a decision node in a CCFG denotes the probability with which the control flow takes one of the subpaths started with this edge.

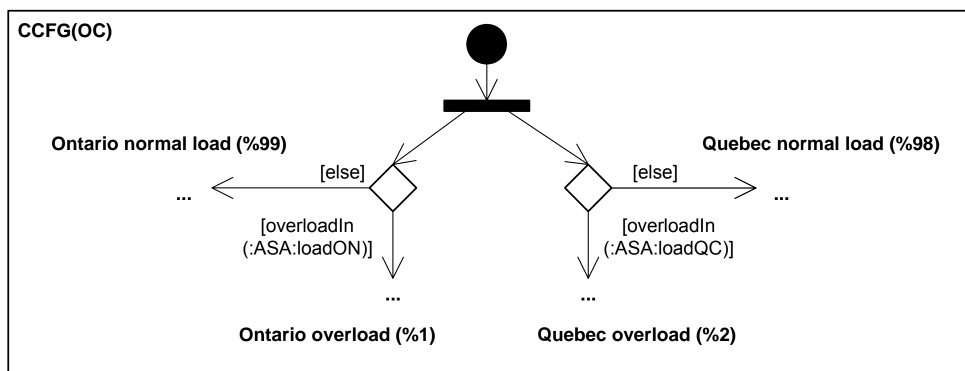


Figure 143-Part of *CCFG(OC),* annotated with probabilities of paths after decision nodes.

Using the operational profile in Table 28, we can derive the probabilities of different DCCFPs in *OC* and *PRNF*. When the probabilities of taking edges after decision nodes

are given, the probability of any DCCFP can be calculated. For example DCCFP $\rho_{3,1}$ of SD *OC* corresponds to taking "Ontario overload" and "Quebec overload" edges of the decision node in *CCFG(OC),* Figure 143. Using the operational profile in Table 28, the probability to choose this DCCFP will be then %1x%2=%0.02. Using a similar approach, the probabilities of taking other DCCFPs of SDs *OC* and *PRNF* have been calculated and are shown in Table 29.

SD	DCCFP	Probability
OC	$\rho_{3,1}$	%0.02
	$\rho_{3,2}$	%0.98
	$\rho_{3,3}$	%1.98
	$\rho_{3,4}$	%9702
PRNF	$\rho_{6,1}$	%0.00125
	$\rho_{6,2}$	~%0.0049
	$\rho_{6,3}$	~%0.0024
	$\rho_{6,4}$	%0.9925

Table 29-Probabilities of taking DCCFPs of SDs *OC* and *PRNF* according to the operational profile given in Table 28.

Now we discuss how a set of OPTCs can be derived from the SCAPS operational profile. To derive OPTCs, we first derive *Operation Profile-based Test Requirements (OPTR)*. An *OPTC* is the set of inputs/conditions to a SUT that trigger an OPTR. An OPTR here means any concurrent SD flow path (CSDFP) in the SCAPS MIOD and any corresponding control flow path (CCFP) for each SD in the chosen CSDFP. In other words, an OPTR corresponds to a DCCFPS (Section 7.2.2). The main constraint in choosing an OPTR is to take into account the probabilities given in the operational profile. The higher the probability of a flow path after a decision node, the more likely the CCFPs containing that path will be selected. For example, assume that the following CSDFP of SCAPS is selected.

$$\begin{pmatrix} OM_STARTUP\begin{pmatrix} OM_ON \\ OM_QC \end{pmatrix}OC \\ \begin{pmatrix} DSPS_ON \\ DSPS_QC \end{pmatrix}PRNF \end{pmatrix}$$

The set of probabilities given in the operational profile (Table 28) can be used to select CCFPs for each of the SD in the above CSDFP. DCCFPs probabilities (Table 29) can then be used to randomly select a DCCFP for each SD in a selected CSDFP.

12.8.2.2 Stress Test Results

Considering the test requirement *TR5* in Figure 142, and by referring to the SCAPS MIOD (Figure 123), we can see that *SRTC1* and *SRTC2* are visited by triggering *TR5*. Therefore, we can say that the test objective in Section 12.3.4 is associated with *SRTC1* and *SRTC2*, and thus we report here how the time difference between the start and end events of the soft RT constraint *SRTC1* (Figure 123) is affected when running test cases corresponding to the test requirement *TR5*.

In order to determine if stress testing makes a difference in the duration of time between the start and end events of the soft RT constraint *SRTC1* when compared the results with test cases based on operational profiles, we measured the executions of 500 randomly selected OPT test cases. Note that all OPT test cases (their start times to be precise) were derived in such a way that they did not violate the SDs' arrival patterns (Section 12.3.3). We then ran 500 test cases corresponding to *TR5* and collected the duration of *SRTC1* across all these runs. The comparison between Operational-Profile-based Test cases (OPT) (Section 12.8.2.1) and stress test cases from GASTT (ST) is depicted by the two execution time distributions in Figure 144. The 1,300 ms deadline of *SRTC1* is shown with a bold horizontal line. The x-axis is the test type and the y-axis is execution time.

The quantiles and the histograms of the three distributions are depicted. Quantiles of the distribution are shown in Table 30.

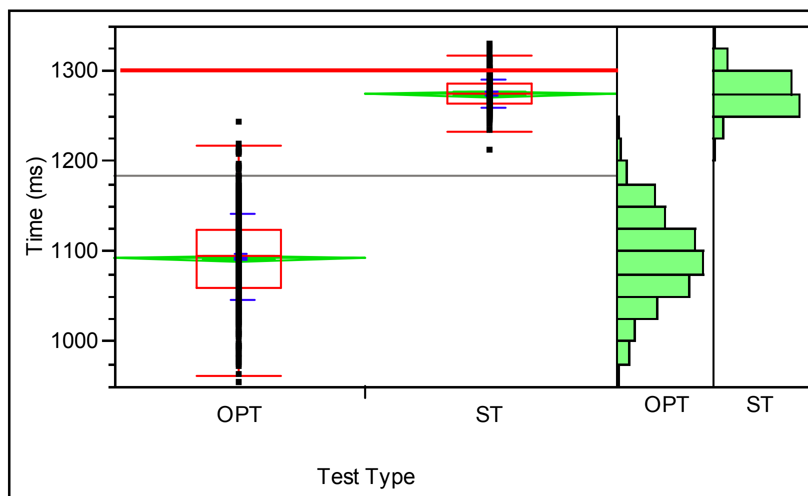


Figure 144-Execution time distributions of test suites corresponding to SRT constraint *SRTC1* by running operational profile test (OPT) and stress test cases from GASTT (ST).

Level	Min.	10%	25%	Median	75%	90%	Max.
OPT	953	1029	1059	1094	1125	1156	1241
ST	1211	1254	1263	1274	1285	1295	1327

Table 30-Quantiles of the distributions in Figure 144.

Due to the indeterminism of distributed environments, the duration of distributed messages can be different across different executions, hence the variance in the distributions of Figure 144. However, all OPT test executions satisfy *SRTC1* whereas *SRTC1* is violated in almost 7.8% (39/500) of ST stress test cases. Our experiments showed that test results corresponding to *SRTC2* had a similar behavior when comparing OPT and ST observations. Furthermore, the difference in average and median value

between OPT and ST distributions are large too, denoting the ability of ST test cases in stressing the system.

12.8.3 Conclusions

In Section 12.3.3, to demonstrate the results of stress testing SCAPS with GASTT, we modified SCAPS SDs by assigning to them realistic arrival pattern constraints. A stress test objective to stress test SCAPS using GASTT technique was described in Section 12.3.4. The use of GARUS tool to derive stress test requirements was described in Section 12.8.1. Results of executing the derived stress test cases were reported in Section 12.8.2. The results are promising as they suggest that stress test cases derived by GASTT technique (using GARUS tool) can help increase the probability of exhibiting network traffic-related faults in distributed systems, while respecting arrival pattern constraints of a SUT's SDs.

Chapter 13

GENERALIZATION OF OUR STRESS TEST METHODOLOGY TO TARGET OTHER TYPES OF FAULTS

We discuss in this chapter how our stress test methodology can be generalized to target other types of resources (e.g. CPU or memory) than network traffic, or other types of faults, such as distributed or resource unavailability faults (refer to our fault taxonomy in Chapter 3). Note that although we discuss the fundamental heuristics of such generalizations here, detailed stress test processes for targeting such faults need further work based on the ideas we present. Considering the tree of generalized fault classes for DRTSs in Figure 7, our stress test methodology can be generalized in at least two ways:

- Targeting other types of resources (Section 13.1)
- Targeting other types of faults (Section 13.2)

13.1 Targeting other Types of Resources

As discussed in Section 3.3, the type of faults targeted by our methodology in terms of resource and distribution were overload usage and distributed traffic faults, respectively. Network bandwidth was considered as the type of resource in targeted faults. As we designed the stress test process in a structured and modular way (see Figure 10), to

generalize our methodology to target other types of resources, we only need to change the parts of the methodology which are dependent on the type of the resource targeted. By a careful look at the overview of the methodology (Figure 10), we can determine that only the test model (TM) must be tailored to adapt the stress test methodology for other resource types.

As discussed in Section 4.2.2, the test model of our methodology consists of four sub-models: (1) network interconnectivity tree, (2) control flow analysis model, (3) distributed traffic usage model, and (4) inter-SD constraints. Of these four sub-models, only the distributed traffic usage model was specific to our particular resource type (i.e. network bandwidth). This model was used to analyze the resource usage patterns of UML SDs. Therefore, only this part needs to be changed if the methodology is to be used for other types of resources. We propose in Section 13.1.1 a generalization of the traffic usage analysis described in Chapter 8, under the form of a general resource usage analysis framework. This framework is then used in Sections 13.1.2 and 13.1.3 to present two other types of resource analyses for CPU and memory, respectively.

13.1.1 Resource Usage Analysis of other Types of Resources

The resource usage analysis of Chapter 8 is generalized into a Model-Based Resource Usage Analysis framework (MBRUA), which we describe in Figure 145 by means of an activity diagram.

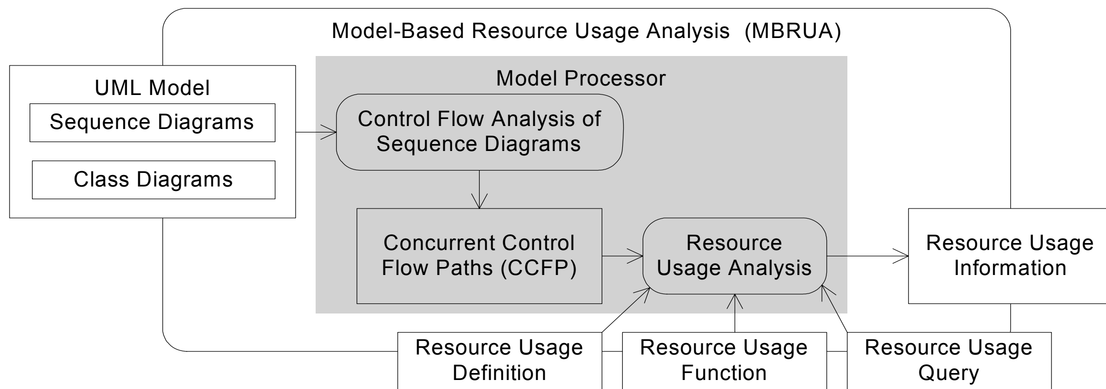


Figure 145-Overview activity diagram of the Model-Based Resource Usage Analysis (MBRUA) technique.

The activity diagram conforms to the *general model-processing framework*, proposed by the *UML Profile for Schedulability, Performance, and Time* (*UML-SPT*) [12], where the model analysis technique acts as the *model processor*. As defined by the UML-SPT, "*a model processor takes in a UML model, analyzes it and generates the analysis results*".

The MBRUA technique takes in the UML model of a system and a set of parameters (for resource usage analysis) as input. The behavior models (SD, augmented with timing information using UML-SPT [12]) are used to predict the behavior of a *System Under Analysis (SUA)*. The structure models (class diagrams) are used to find out about the generalization (inheritance) relationships among classes and therefore be able to appropriately handle polymorphic behaviors of objects in SD lifelines when analyzing control flow [50, 59]. Furthermore, as it was discussed in Chapter 8, resource usage analysis requires the structure of classes in a SUA to estimate data sizes of messages.

The technique then analyzes the control flow in the given model and generates *Concurrent Control Flow Paths (CCFP)*. CCFPs are then used by the resource usage analysis activity to generate resource usage information. Resource usage definition,

function and query are specific input parameters for the resource usage analysis activity, and are discussed next.

Resource Usage Definition (RUD) is a set of criteria which define how the usage of resource usage should be quantified from behavior models. For example, as discussed in Chapter 8, the RUD of network traffic usage is to filter CCFPs by selecting only messages which are sent across different nodes. Such a CCFP was referred to as a Distributed CCFP (DCCFP). A DCCFP is built from a given CCFP by removing all local messages (sent between two objects on the same node) and keeping the distributed ones. The formal definition of network traffic RUD was presented in Section 8.1.

A Resource Usage Measure (RUM) defines how the usage of a specific resource by model constructs should be quantified. A RUM can be considered as a function from a set of model elements to a Real value. We defined the RUM of network traffic usage in Section 8.1.

A Resource Usage Query (RUQ) is a query to filter the Resource Usage Analysis (RUA) results. For the network traffic usage analysis, we defined four query fields (attributes): Traffic location: nodes, objects or networks.; Traffic direction (for nodes only): into, from, or bidirectional; Traffic attribute (data traffic or number of messages); Traffic duration: instant or interval (Section 8.3).

In the next two sections, we present the RUD and RUM for two additional resource types: CPU and memory. Using the following definitions of RUD and RUM, our stress test methodology can be generalized to target these two resource types instead of network traffic

13.1.2 CPU

Our heuristics for calculating CPU usage using SD messages are as follows. Among call, reply and signal messages, only call and signal messages consume CPU power. This is a simplification of the fact that reply (return) messages only return values to the caller of a message and the CPU usage entailed by reply messages is considered negligible. As an example, let us consider a call message *cm* and its corresponding reply message *rm*. The receiver object starts to execute a method body upon receiving the message *cm* from the caller. Such an execution consumes CPU and finally the receiver returns a reply message (*rm*) to the caller. The CPU usage entailed by *rm* is negligible compared to the execution of *cm*, since it is basically copying the return results to a stack and returning back the control to the caller. The practical impact of such simplification and more accurate measures should however be investigated in future work

The CPU usage of each call message depends on the processing complexity of the operation of the message, which can be either (1) *predicted* (calculated by a performance tool), (2) *measured* (if an executable implementation of the SUA is available), (3) *required* (coming from the system requirements or from a performance budget based on a message, e.g., a required response time for a scenario), or (4) *assumed* (based on experience) by modelers. These four alternatives denote the *source* of the information and are defined in the Performance Modeling section of the UML–SPT[12]. Predicting the processing complexity of an operation is a challenging task in the early design phase. Existing works such as [124, 125] have proposed ways to predict CPU utilization using

different analysis models. The prediction in [124] is specific to rule-based systems[17] (for instance used in telecommunication applications) and is done using a model of the CPU cost per rule, referred to as a *capacity model*. The work in [125] uses fuzzy logic and the concept of stochastic processes to predict CPU utilization of mainframe systems based on a log file from the recent system behavior. Thus, the technique requires the complete implementation or at least an earlier version of the system to be available. The predicted values help system programmers tune a system's performance by moving out unimportant jobs during peak time. Another CPU utilization estimation heuristic was proposed by Gomaa [49] in which a developer tries to implement pieces of code that are representative enough of the foreseen implementation. The CPU utilization of the "early" code is then measured to estimate the final code's CPU utilization.

The Performance Modeling section of the UML–SPT [12] discusses ways to model CPU usage in behavioral models. For example consider the SD in Figure 146, where message *op()* is annotated with the *PAdemand* stereotype from the Performance Modeling section of the UML–SPT. *PAdemand* (PA for Performance Analysis) is used to model the resource demand of a *scenario step* (e.g., a message in a SD, or an activity in an activity diagram). The right hand side of a *PAdemand* equality should be of type *PAperfValue*

---

[17] Rule-based systems represent knowledge in terms of a set of rules that tell the system what it should do or what it could conclude in different situations. A rule-based system consists of a set of If-Then-Else rules, a set of facts, and some interpreter controlling the application of rules, given the facts as inputs[126] A. Ligeza, *Logical Foundations for Rule-Based Systems*: Springer, 2006..

(Performance Value), which is used to specify a complex performance value as defined below.

PAperfValue=(" <source-modifier> "," <type-modifier> "," <time-value> ")

where: *<source-modifier>::='req'|'assm'|'pred'|'msr'* is a string that defines the source of the value: *required*, *assumed*, *predicted*, or *measured*. *<type-modifier>::='mean'|'sigma'|'kth-mom', <Integer>|'max'|'percentile,'<real>|'dist'* is a specification of the type of value meaning: average, variance, kth-moment (integer identifies value of k), percentile range (real identifies percentage value), or probability distribution. *<time-value>* is a time value described by the *RTtimeValue* type (defined by in the General Time Modeling section of the UML–SPT). Thus, the *PAdemand* annotation of the message *op()* in Figure 146 means that it is *predicted* that this message will utilize 90% of the CPU (on n_2).

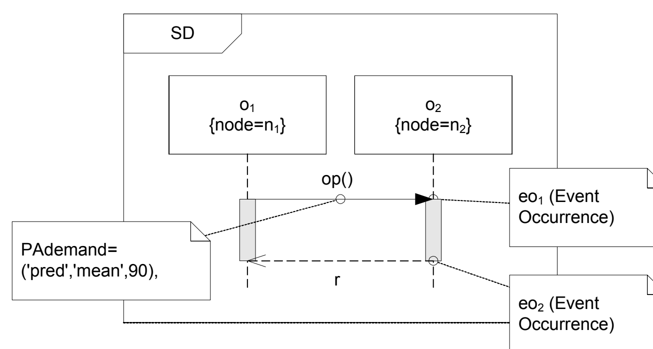


Figure 146-Modeling CPU usage example.

Based on our heuristics for CPU usage by SD messages, we present the RUD and RUM for CPU resource analysis in Equation 12, where function *CPUUsage(message)* returns the processing complexity value associated with a message. CPU usage is specific to messages and not operations involved in those messages. The reason is that the same

operation used in different sequence diagrams may entail different CPU usages as different computations may be triggered. As discussed above, for the sake of simplification, the RUD_{CPU} does not consider reply messages, but only call messages.

The RUM_{CPU} of a message is simply equal to the processing complexity value of the operation associated with the message. It is important when analyzing CPU usage to consider *event occurrences*[18] in SDs. For example consider the SD in Figure 146, and assume *op()* is the only message considered in the RUA according to RUD_{CPU}. Since the processing of *op()* starts on object o_2 at event occurrence eo_1 and finishes at eo_2, the message consumes CPU power only in the time period between two event occurrences. Note that this notion of resource usage over time is not incorporated in Equation 12, since RUD and RUM are by nature not time-based. The issue should be further investigated when designing a set of time-based RUA functions for CPU usage (similar to the ones in Chapter 8 for network usage).

$$RUD_{CPU} : TCCFP \rightarrow TCCFP$$
$$\forall \rho \in TCCFP : RUD_{CPU}(\rho) = \rho - \{msg \mid msg \in \rho \wedge msg.msgType = 'reply'\}$$

$$\forall msg \in Message : RUM_{CPU}(msg) = CPUUsage(msg)$$

Equation 12-RUD and RUM for CPU resource.

---

[18] "EventOccurrences represent moments in time to which Actions are associated. An EventOccurrence is the basic semantic unit of Interactions. EventOccurrences are ordered along a Lifeline." [127]

Another important consideration when analyzing CPU usage in distributed systems is the *locality* of the usage. Similar in concept to the location attribute of network traffic (Chapter 8), the CPU usage location denotes the particular CPU on which a message is processed. For example, as shown in Figure 146, o_1 and o_2 are deployed on nodes n_1 and n_2. Therefore, the actual execution of operation *op()* takes place on n_2 and leads to CPU usage on n_2 only. The locality aspect of CPU usage is important because it is crucial for engineers to determine the host CPU which must handle the processing load of a message in a DRTS. Furthermore, we made a simplification in this section that the CPU utilization of message during its execution is uniform. A more realistic but complex approach would be to define a time-based function, which would predict a message's CPU utilization at each time instant during its execution.

13.1.3 Memory

Our heuristics for calculating memory usage by SD messages is as follows. Memory is used by messages in two ways:

- Messages which associated method or signal name is *create* or *destroy*, or
- Temporary (heap) memory used by local variables as a result of message invocations.

For example, consider the SD in Figure 147. Object o_1 creates an object of class C_3 and destroys it after sending a message (m_2) to it and receiving a reply (r_2). Thus, temporary (heap) memory corresponding to the data size of C_3 is allocated and then de-allocated. Furthermore, assume the source-code implementation of messages m_1 and m_2 results in 10 and 20 integer local variables, respectively. Assuming that each integer variable

consumes four bytes of memory, invocation of m_1 and m_2 will consume 40 and 80 bytes of heap memory. Estimating such information may be possible in late design stages by using, for example, heuristics similar to the ones used in the COMET (Concurrent Object Modeling and Architectural Design with UML) [10] object-oriented life cycle, where Gomaa proposed a heuristic to estimate time durations of messages based on benchmarks made of previously-developed similar messages. Such an approach can be adapted to the estimation of number and types of local variables of a method by comparing the functionality/role of a method at hand to benchmarks of previously-developed methods local variables (in the same target programming language). It should be acknowledged that this is in general a complex task which would need substantial experience and skills from developers. Such information should then be provided by modelers in an appropriate way, for example by using specific tagged-values.

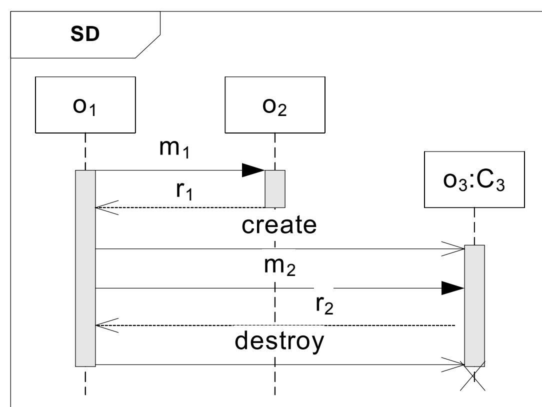


Figure 147-Memory usage analysis example.

Based on our heuristics for CPU usage by SD messages, we present the RUD and RUM for memory usage in Equation 13, where function *dataSize(class)* returns the data size of a class (Section 8.1). We consider the amount of memory allocated/de-allocated in RUM_{Memory}. A *create* message allocates memory space (denoted with +), while a *destroy*

message releases memory (denoted with -). Note that, for simplicity, the temporary (heap) memory used by local variables resulting from message invocations has not been incorporated into RUD or RUM. The temporary memory allocated in the beginning of an operation by its local variables will be de-allocated upon return from the operation. Based on this simplification, the granularity of the RUA which will use the RUD and RUM in Equation 13 is assumed to be at the message level, and thus the invocation of operations with local memory usage will not cause any change in the amount of memory space consumed. However if a time-based RUA is to be performed, time-based RUD and RUM should be defined where the intra-message-invocation memory usage should also be accounted for. Furthermore, a part of such a RUA technique should make sure that there is *enough* temporary memory space to invoke such messages. Similar to the *locality* aspect of CPU usage, memory usage analysis should also take *locality* into account in the context of distributed systems.

$$RUD_{Memory} : TCCFP \rightarrow TCCFP$$
$$\forall \rho \in TCCFP : RUD_{Memory}(\rho) = \rho - \{msg \mid msg \in \rho \wedge msg.methodOrSignalName \notin \{create, destroy\}\}$$

$$\forall msg \in Message : RUM_{Memory}(msg) = \begin{cases} + dataSize(msg.receiver.class) & ;\text{if } msg.methodOrSignalName = create \\ - dataSize(msg.sender.class) & ;\text{if } msg.methodOrSignalName = destroy \end{cases}$$

Equation 13-RUD and RUM for memory resource.

13.2 Targeting other Types of Faults

We discuss in this section how our stress test methodology can be generalized to target other types of faults, based on our fault taxonomy in Chapter 3. Since the main testing aspect of our methodology is stress testing, we investigate next how stress testing can be performed to increase chances of exhibiting the following three types of faults.

- Distributed unavailability faults (Section 13.2.1)
- Resource unavailability faults (Section 13.2.2)
- Concurrency faults (Section 13.2.3)

13.2.1 Distributed Unavailability Faults

As discussed in Section 3.1.2.1, Distributed Unavailability Faults (DUF) relate to the *availability* (readiness for correct service) and *reliability* (continuity of correct service) attributes of a system. The specification of most distributed systems usually dictates that the system's network links and nodes should be highly available and reliable. For example, in a safety-critical system like a distributed air traffic control, the flight and runway information should be updated frequently in the system's central database. Failing to do so, which might be caused for example by a DUF between a radar and the controller, might result in disastrous consequences.

A DUF is said to have happened when a system component (either a network link or a node) is no longer available and can not provide service to other components in the system. For example, a distributed message from a source node may not reach the destination node because one of the network links in the path from the source to the destination node is exhibiting a DUF.

As we discussed in our methodology, triggering distributed traffic faults in a DRTS is not straightforward, i.e., finding specific scenarios in behavior models of a SUT, and specific start times to trigger each of those scenarios are not easy. Our traffic-aware stress test methodology aimed at increasing the chances of exhibiting distributed traffic faults by finding such specific test requirements. On the other hand, as the definition of a DUF

indicates, such a fault can be easily triggered when a node or a network becomes unavailable, and hence, a simple test objective to test a SUT under DUFs is to simulate the unavailability of a network or a node in each distributed message and observe the reaction of the SUT. For example, consider a SUT identified by the SDs in Figure 148, where most of the messages are distributed. Each $p_{i,j}$ is a process object deployed on a specific node.

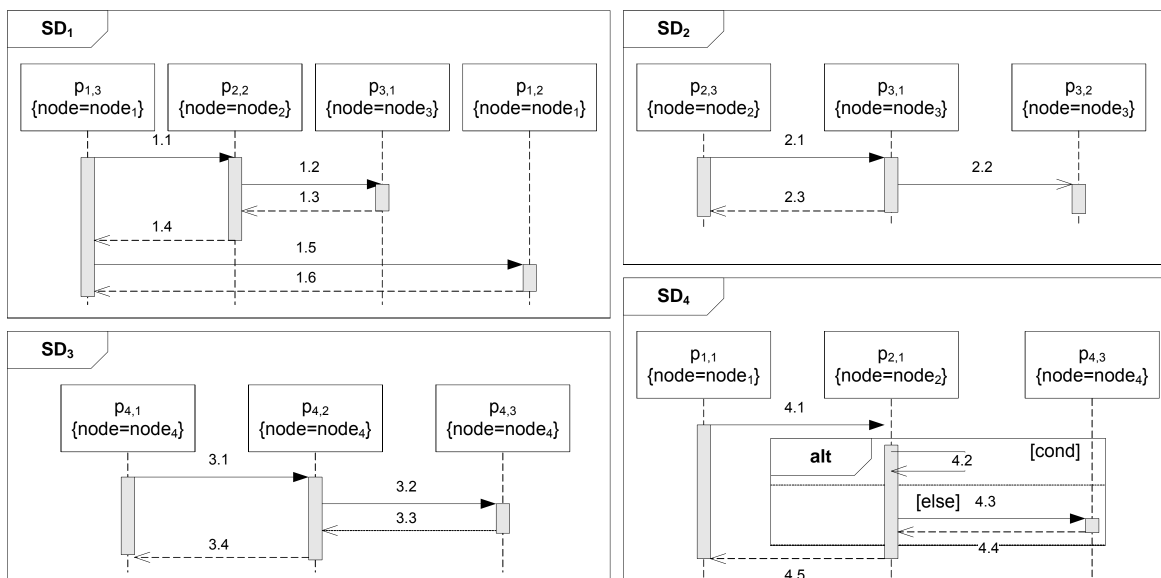


Figure 148-Four SDs with distributed messages.

In order to apply stress testing targeted at DUFs (i.e., generalize our stress test methodology to such faults), we use the following heuristics but acknowledge that other heuristics could be used. In order to maximize the impacts of a DUF (resulting failures), the stress test objectives are:

1. The messages, in which the component originating the DUF is involved as the receiver, are scheduled to be sent concurrently,
2. After such messages are sent, the unavailability of the component is emulated, and the SUT's reaction is assessed.

As an example of the above stress test objectives, assume that we want to maximize the impacts of a DUF when *$node_2$* becomes unavailable in Figure 148. According to the above heuristics, we choose the messages (each from the set of CCFPs for each SD) which are sent towards *$node_2$*, and schedule them to be sent concurrently. There are two messages (*1.1* and *4.1*) in *SD_1* and *SD_4*, which are sent towards *$node_2$*. When there are several messages in a SD towards a node, for which we are trying to stress test the impacts of a DUF, different criteria can be used to choose the one which is predicted to cause the worst impact due to a DUF, e.g., the strongest data dependency in the next messages. Messages *1.1* and *4.1* are then sent concurrently (to *$node_2$*). Then *$node_2$* is emulated to be unavailable, and the impact of the DUF in *$node_2$* on the SUT is assessed. Note that scheduling the chosen messages (and thus their corresponding CCFPs) to stress test a SUT with DUFs can present challenges similar to the ones we faced in our methodology, i.e., inter-SD constraints and SD arrival patterns.

13.2.2 Resource Unavailability Faults

Stress testing with respect to *Resource Unavailability Faults (RUnF)* can be done in a similar way to testing with respect to DUFs. For example, assume a sensor as a resource in a system. Suppose that $p_{2,2}$ and $p_{2,1}$ in *SD_1* and *SD_4* in Figure 148 are replaced with object *s:Sensor*. Messages *1.1* and *4.1* can be selected from *SD_1* and *SD_4* to be sent concurrently (to *s:Sensor*). Then *s:Sensor* is emulated to be unavailable, and the impact of the RUnFs in *s:Sensor* to the SUT is assessed.

Note that the RUnF is rephrased depending on the type of a resource it is associated with. For example, since the resource type was a sensor in the above example, a corresponding RUnF occurs when the sensor is not available (not functioning properly). For CPU as the

resource type, the associated RUnF might be that a CPU's utilization is already 100% and it can not offer any extra processing power at a time instant. For memory, the associated RUnF can be rephrased as: the memory is currently full and no extra memory space is available.

13.2.3 Concurrency Faults

As discussed in Section 3.1.4, a concurrency fault is said to have occurred if the root cause of a system failure is due to a fault in concurrency among processes. There might be, for example, a shared resource that is accessed by several processes in a system. The synchronization scheme and order in which a shared resource is accessed might lead to a concurrency fault. Some types of concurrency faults are: deadlock, livelock, starvation and data races (race conditions).

In order to generalize our stress test methodology to concurrency faults, we can merge the ideas of our work and the existing techniques which target such faults, such as [20-22, 49] which aim at finding data-race related faults. For example, Ben-Asher et al. [23] propose a set of heuristics to increase the probability of manifesting data-race related faults in Java programs. The goal is to increase the chance of exercising data-races in the program under test and thus increase the chance of manifesting concurrency faults that are data-race related. The proposed technique first orders global shared variables according to the number of times they are accessed by different processes. Then data-race based heuristics are used to change the runtime interleaving of threads so that the probability of fault manifestation increases. One of the proposed heuristics in [21] is called *barrier scheduling,* in which barriers are installed before and after accessing a particular shared variable. A barrier causes the processes accessing the variable to wait

just before accessing it. When a predefined number of processes are waiting, the heuristic then simultaneously resumes all the waiting processes to access the shared variable, for example using *notifyAll()* in Java.

We can use a similar approach to apply stress testing based on UML models toward exhibiting concurrency faults. Such an approach can find the *global shared objects* (corresponding to global shared variables in [21]) according to the number of times they are targeted by different messages in different SDs. The data-race based heuristics (such as barrier scheduling [21]) can then be used to derive test requirements from SDs such that specific interleavings of messages towards *global shared objects* are triggered concurrently so that the probability of concurrency faults increases.

The application of the barrier scheduling heuristics in the context of UML-based stress testing is illustrated using an example in Figure 149. Assume that object *O:C* has been identified as an global shared object to derive concurrency stress test requirements. Also assume that the three SDs in Figure 149-(a) are the SD in the UML model of a SUT. The concurrency stress test requirement is shown using a SD in Figure 149-(a), where the test requirements have been modeled based on the UML testing profile.

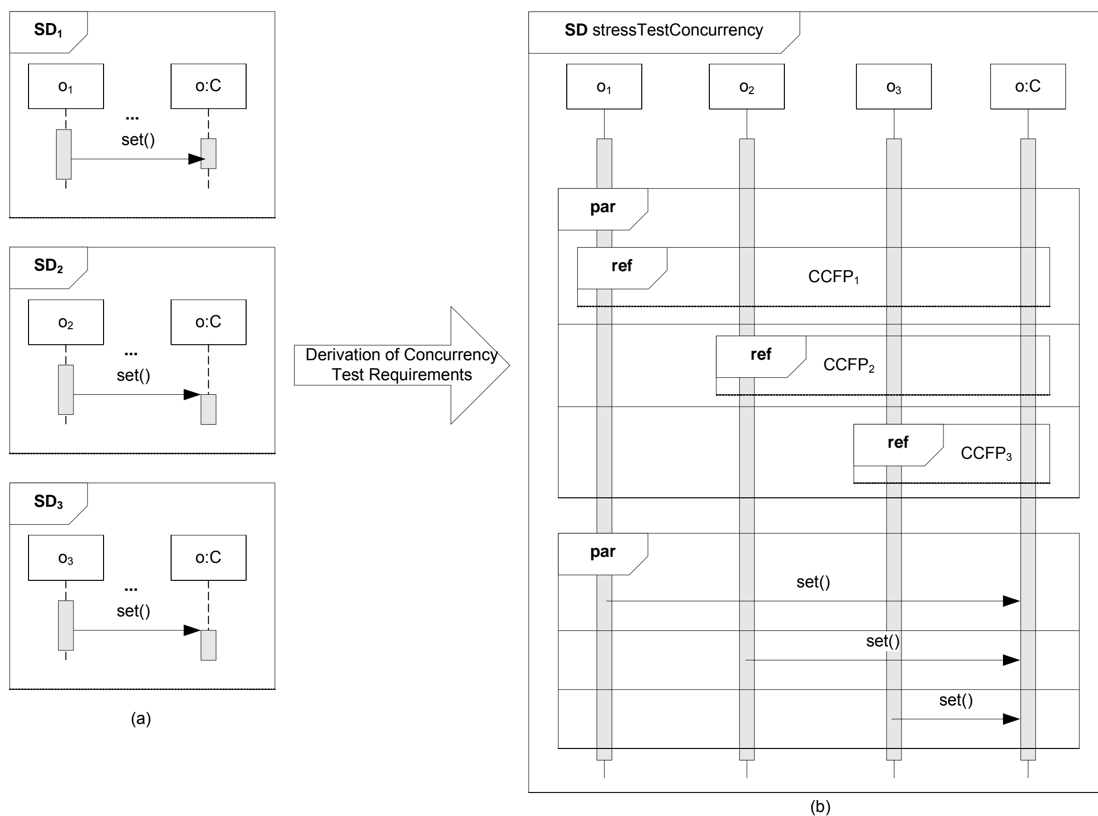


Figure 149-Heuristics for the application of the barrier scheduling heuristic in the context of UML-based stress testing.

Among all the messages of a SD where the global shared object *O* is one of the two participating objects, different criteria can be used to select the one which is expected to cause the maximum stress in terms of a concurrency fault (e.g., following data flow dependencies). Assume that the *set()* message has been identified in each of the three SDs as such a message. Based on the barrier scheduling heuristic, the concurrency stress test objective is to run the CCFP containing the *set()* message in each of the SDs and pause the control flow just before the *set()* messages. Assume that $CCFP_i$'s are the sub-CCFPs of those CCFPs including only the messages before the *set()* message. $CCFP_i$'s are triggered in parallel before the three *set()* messages. This scenario is equivalent to the barrier scheduling heuristic [21] in the context of UML-based stress testing. Although the

three *set()* messages are triggered concurrently, there might be very small differences between their start and end times at runtime, thus enabling chances for concurrency faults. This situation is inline with the Java-based concurrency testing discussions in [21].

Chapter 14

SUMMARY

The summary presented in this chapter includes the following:

- Conclusions (Section 14.1)
- Future research directions (Section 14.2)

14.1 Conclusions

A model-driven, stress test methodology aimed at increasing chances of discovering faults related to distributed traffic in distributed systems was presented. The technique uses as input a UML 2.0 model of a system, augmented with timing information. We specified an adequate and realistic input test model which includes (1) a Network Deployment Diagram (following the UML package notation) that describes the distributed architecture in terms of system nodes and networks and (2) a Modified Interaction Overview Diagram (following the UML 2.0 interaction overview diagram notation) that describes execution constraints between sequence diagrams. Our stress testing technique relies on a careful identification of control flow paths in UML 2.0 Sequence Diagrams and the distributed traffic they entail. This information is used to

generate stress test requirements composed of specific control flow paths (in Sequence Diagrams) along with time values indicating when those paths have to be triggered so as to stress the network to the maximum extent possible. To do so, we resort to optimization algorithms. In the most complex case, when external system events follow complex arrival patterns, we make use of a specifically tailored Genetic Algorithm, which has shown promising initial results. .

Using the specification of a real-world distributed system, we designed and implemented a system and described how the stress test cases were derived and executed using our methodology. We furthermore reported the results of applying our stress test methodology on this system and discussed its effectiveness in detecting violations of a hard real-time constraint when compared to test cases based on an operational profile. Our first results are promising as they suggest that our generated stress test cases significantly increase the probability of exhibiting distributed traffic-related faults in distributed systems.

Our test methodology, as presented in this thesis, has some limitations:

- Lack of automation regarding the creation of test models: Although we do not have an automated tool to generate test models, such a tool can be developed based on the discussions in Chapter 6-Chapter 8 and using technologies such as Rational Software Architect (RSA) and Eclipse Modeling Framework (EMF). This may lead, for example, to the refinement of the OCL consistency rules we presented in Chapter 6: although we have made the effort to write a complete set of correct rules, and validated them manually, only their use during the

implementation of a tool supporting our methodology will reveal whether they are complete and correct.

- Uncertainty of timing information: Our methodology is based on design diagrams, and relies in particular on timing information of SD messages. During design such information is usually estimated, with hopefully a reasonable accuracy. How such an estimation can be performed was outside of the scope of our work, and we reported on existing techniques, and used them in our case study. It is therefore clear that the results of our methodology depend on the quality of those estimates: the more accurate the estimates the better the stress tests (i.e., more likely to reveal problems). We have not performed any sensitivity analysis of our methodology to those estimates but we acknowledge (Section 10.9) that the methodology is likely to be sensitive. This will be further investigated in our future work.
- We furthermore assumed that the SDs of a SUT are given as an input to our stress test methodology. Thus, our methodology can not be applied to stress test systems whose SDs are not be available, e.g., legacy systems. As a solution to this limitation, SD reverse engineering techniques (e.g. [21]) can be used to build SDs of a SUT first and then our technique may be applied. However, the cost-effectiveness of such an approach has to be investigated

14.2 Future Research Directions

Our stress test methodology can be generalized to other distributed-type faults, such as distributed unavailability of networks and nodes, and other resources such as CPU,

memory, and database usage. Stress testing a distributed system with respect to distributed unavailability fault (Section 3.1.2.1) is to cause scenarios in which the maximum stress on a system occurs when a node (or a network) becomes unavailable. CPU or memory-aware stress testing will put a SUT under maximum possible usage of CPU or memory and will increase the chances of exhibiting resource usage faults related to CPU or memory.

In order to give a scheduled stress test requirement that will cause stress on network traffic on a predicted time instant (or period), we required that all SD messages have precise or statistical timing information (Section 5.1.1). We have thought of a heuristic to derive stress test requirements when it is impossible (or hard) to determine such timing information for messages. Such a heuristics will first determine the maximum stressing messages. The corresponding stress test requirement will then be to *wait* before such messages and advancing the execution when the execution is waiting before all such maximum stressing messages. Two other research directions are: (1) How can we account for data flow and parameters in the SD sequential constraint modeling?, and (2) How can we account for the variation in the data traffic value of a distributed message during its execution?.

As we discussed in Section 7.2.2, the current automatic procedure to derive CCFPS (and DCCFPS) may produce infeasible (one could say illegal) CCFPS (DCCFPs). Another future work is to do a form of data flow analysis on the set of derived CCFPS (DCCFPs) to eliminate the infeasible (illegal) ones.

The UML Testing Profile [128] defines a language for designing, visualizing, specifying, analyzing, constructing and documenting the artifacts of test systems. It is a test modeling

language that can be used with all major object and component technologies and applied to testing systems in various application domains. The UML Testing Profile (UML-TP) can be used in an integrated manner with UML to handle a system's test artifacts [129]. Specifying the generated stress test requirements and the stress test process of our methodology with the UML-TP would lead to having all software artifacts, from analysis and design to specified test suites, modeled with UML. This would facilitate traceability between analysis, design, and testing artifacts and since UML-TP has paved the way for possible tools to execute test cases modeled in the UML-TP, test automation could potentially be improved.

UML models can be statically verified to make sure that behavior models do not lead to RT faults by checking if there is any possible scenario in which a RT fault can occur under stress conditions in terms of different types of resources, e.g. network traffic, CPU and memory. The verification can be applied on a system’s design model before it is implemented. The overall procedure for the verification is to find the maximum possible stress conditions of behavior models and check if, for example, the maximum possible traffic exceeds the network bandwidth. Resource usage information can either be modeled by modelers using resource usage modeling constructs proposed by the UML-SPT, or can be predicted from models [129].

Performance bottlenecks of a DRTS can be pinpointed using PERT (Program Evaluation and Review Technique) technique. Given the time duration of each use case in a system and also their sequential constrains (using a MIOD), the PERT technique can be used to find the critical paths in a MIOD, i.e., performance bottlenecks.

We saw in Chapter 9 that test objectives are parts of the input to our stress testing technique, where the rest of the steps are done automatically. It is worth investigating if the test objectives can also be derived automatically in the order of importance to be stress tested first. More importance, in this context, means if the failure of a RT constraint has more severity than another. This automated process can reduce the workload done by testers.

Risk assessment/fault analysis of distributed-type faults, the investigation of *QoS faults* and the impacts of stress on QoS in a system as well as implementation of a test model generator from UML models are also worthwhile future research directions. A QoS fault is said to have occurred when a system component does not function according to its QoS requirement. We also intend to stress test more complex distributed systems using our methodology and perform more empirical investigations of its effectiveness.

Furthermore, in the cases when stress tests show that there can be scenarios in which one or more of the RT constraints can be violated in a SUT, such as the case in our case study (Chapter 12), performance engineering techniques should be applied to redesign the system and/or increase resources (for example, network capacities) or change the RT constraint values to more realistic levels. Thus, a *Stress-Test based Performance Engineering (STPE)* approach can be devised to assist testers and system analysts in fixing the distribution-related faults. Following STPE, the designer uses stress test results to evaluate the performance throughout of a SUT, analyze missed Real-Time constraints, and provide guidelines to enhance performance and robustness of the system in terms of Real-Time constraints.

Recall from Section 8.4.1 that we assumed the network paths' dispatching policy does not change during the transmission of a message, i.e., the transmission shares of each of the involved networks stay the same during the entire transmission. To relax such a limitation, a future work can be to get more information from the routing and network protocols involved in a SUT so that we can calculate the transmission shares of each network/path in each time instant. This will help to derive more precise stress test requirements.

We recognized that the results of our methodology are likely to be sensitive to uncertainty in timing information: if the variance (as a measure of uncertainty) in timing information in a SUT increases, then the preciseness of the output stress test requirements generated by our methodology will decrease, i.e., generated stress test cases will not necessarily maximize traffic on a given network (or a node). Future work must investigate the sensitivity of our stress test methodology to uncertainty in timing information.

References

[1] J. J. P. Tsai, Y. Bi, S. J. H. Yang, and R. A. W. Smith, *Distributed Real-Time Systems: Monitoring, Visualization, Debugging, and Analysis*: John Wiley & Sons, 1996.

[2] E. Weyuker and F. I. Vokolos, "Experience with Performance Testing of Software Systems: Issues, an Approach and Case Study," *IEEE Transactions on Software Engineering*, vol. 26, pp. 1147-1156, 2000.

[3] R. Kuhn, "Sources of Failure in the Public Switched Telephone Network," *IEEE Computer*, vol. 30, pp. 31-36, 1997.

[4] S. Mackay, E. Wright, and J. Park, *Practical Data Communications for Instrumentation and Control*: Newnes, 2003.

[5] S. C. Bhatia, "Industrial SCADA: SCADA Control Systems In Integrated Steel Plants," in *Proceedings of Power Quality Conference*, pp. 225-233, 1998.

[6] M. Ivey, A. Akhil, D. Robinson, K. Stamber, and J. Stamp, "Accommodating Uncertainty in Planning and Operations," Technical Report, Consortium for Electric Reliability Technology Solutions 1999.

[7] F. J. Molina, J. Barbancho, and J. Luque, "Automated Meter Reading and SCADA Application for Wireless Sensor Network," *Lecture Notes in Computer Science*, vol. 2865, pp. 223-234, 2003.

[8] ABB Co., "ABB Group Annual Report," http://www.abb.com/Global/Clabb/CLABB155.NSF/viewunid/2DDB4104B522E26A04256C3000527EEB/$file/ABB_TECH_E-Annual2000.pdf 2000.

[9] K. P. Birman, J. Chen, K. M. Hopkinson, R. J. Thomas, J. S. Thorp, R. v. Renesse, and W. Vogels, "Overcoming Communications Challenges in Software for Monitoring and Controlling Power Systems," *Proceedings of the IEEE*, vol. 9, 2005.

[10] Object Management Group (OMG), "UML 2.0 Superstructure Specification," 2005.

[11] T. Pender, *UML Bible*: Wiley, 2003.

[12] Object Management Group (OMG), "UML Profile for Schedulability, Performance, and Time (v1.0)," 2003.

[13] C. S. D. Yang, "Identifying Potentially Load Sensitive Code Regions for Stress Testing," in *Proceedings of MASPLA'96 (The Mid-Atlantic Student Workshop on Programming Languages and Systems)*, State University of New York at New Paltz, NY, USA, April 1996.

[14] J. Zhang and S. C. Cheung, "Automated Test Case Generation for the Stress Testing of Multimedia Systems," *Journal on Software Practice and Experience*, vol. 32, pp. 1411-1435, 2002.

[15] A. Avritzer and E. J. Weyuker, "The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software," *IEEE Transactions on Software Engineering*, vol. 21, pp. 705-716, 1995.

[16] L. C. Briand, Y. Labiche, and M. Shousha, "Automating Stress Testing for Real-Time Systems Using Genetic Algorithms," in *Proceeding of Genetic and Evolutionary Computation Conference*, pp. 1021-1028, 2005.

[17] A. Avritzer and B. Larson, "Load Testing Software Using Deterministic State Testing," in *Proceedings of International Symposium on Software Testing and Analysis*, pp. 82-88, 1993.

[18] W. Brauer, W. Reisig, and G. R. (eds.), "Petri-nets, Central Models and their Properties," in *Advances in Petri-Nets, Part I. Proceedings of an Advanced Course, Bad Honnef (ed), Lecture Notes in Computer Science*, vol. 254: Springer, 1987.

[19] J. F. Allen, "Maintaining Knowledge about Temporal Intervals," *Communications of the ACM*, vol. 26, pp. 832-843, 1983.

[20] D. Hovemeyer and W. Pugh, "Finding Concurrency Bugs in Java," in *Workshop on Concurrency and Synchronization in Java Programs, International Symposium on Principles of Distributed Computing*, 2004.

[21] Y. Ben-Asher, Y. Eytani, and E. Farchi, "Heuristics for Finding Concurrent Bugs," in *Workshop on Parallel and Distributed Systems: Testing and Debugging, International Parallel and Distributed Processing Symposium*, 2003.

[22] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded Java Program Test Generation," *IBM Systems Journal*, vol. 41, pp. 111-125, 2002.

[23] S. D. Stoller, "Testing Concurrent Java Programs using Randomized Scheduling," *Electronic Notes in Theoretical Computer Science*, vol. 70, pp. 1-16, 2002.

[24] R. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*: Addison-Wesley, 1999.

[25] I. P. Paltor and J. Lilius, "Digital Sound Recorder: a Case Study on Designing Embedded Systems Using the UML Notation," TUCS Technical Report No. 234, Turku Centre for Computer Science, Finland 1999.

[26] B. Douglass, *Doing Hard Time, Developing Real-Time Systems with UML Objects, Frameworks, and Patterns*: Addison Wesley, 1999.

[27] D. Herzberg, "UML-RT as a Candidate for Modeling Embedded Real-Time Systems in the Telecommunication Domain," in *Proceedings of International Conference on the Unified Modeling Language*, pp. 331-338, 1999.

[28] L. Kabous and W. Neber, "Modeling Hard Real Time Systems with UML: The OOHARTS Approach," in *Proceedings of International Conference on the Unified Modeling Language*, pp. 339-355, 1999.

[29] A. Lanusse, S. Gerard, and F. Terrier, "Real-Time Modeling with UML: The ACCORD Approach," in *Proceedings of International Conference on the Unified Modeling Language*, Mulhouse, France, pp. 319-335, 1998.

[30] J. Hakansson, L. Mokrushin, P. Pettersson, and W. Yi, "An Analysis Tool for UML Models with SPT Annotations," in *International Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems*, 2004.

[31] S. Bernardi, S. Donatelli, and J. Merseguer, "From UML Sequence Diagrams and Statecharts to Analysable Petri-net Models," in *Proceedings of International Workshop on Software and Performance*, pp. 35-45, 2002.

[32] C. M. Woodside and D. C. Petriu, "Capabilities of the UML Profile for Schedulability Performance and Time (SPT)," in *Workshop SIVOES-SPT on the usage of the SPT Profile, held in conjunction with the 10th IEEE Real-Time and Embedded Technology and Applications Symposium RTAS'2004*, Toronto, Canada, May 2004.

[33] D. C. Petriu, "Performance Analysis Based on the UML SPT Profile," in *tutorial given at QEST'2004*, Enschede, The Netherlands, September 2004.

[34] D. C. Petriu and C. M. Woodside, "Extending the UML Profile for Schedulability Performance and Time (SPT) for component-based systems," in *Workshop SIVOES-SPT on the usage of the SPT Profile, held in conjunction with the 10th IEEE Real-Time and Embedded Technology and Applications Symposium RTAS'2004*, Toronto, Canada, May 2004.

[35] B. P. Douglass, "Rhapsody 5.0: Breakthroughs in Software and Systems Engineering," I-Logix Corp. whitepaper 2003.

[36] Object Management Group (OMG), "Unified Modeling Language Specification (v1.3)," 1999.

[37] Object Management Group (OMG), "Unified Modeling Language Specification (v1.5)," 2003.

[38] A. Avizienis, J.-C. Laprie, and B. Randell, "Fundamental Concepts of Dependability," Technical Report 01145, LAAS (Laboratory for Analysis and Architecture of Systems), Toulouse, France 01-145, 2001.

[39] Y. Huang, P. Jalote, and C. Kintala, "Two Techniques for Transient Software Error Recovery," *Lecture Notes in Computer Science*, vol. 774, pp. 159-170, 1994.

[40] J. Gray, "A Census of Tandem System Availability Between 1985 and 1990," *IEEE Transactions on Reliability*, vol. 39, pp. 409-418, 1990.

[41] J. Gray, "Why do Computers Stop and What Can be Done About it?," in *Proceedings of International Symposium on Reliability in Distributed Software and Database Systems*, pp. 3-12, 1986.

[42] M. Sullivan and R. Chillarege, "Software Defects and Their Impact on System Availability - A Study of Field Failures in Operating Systems," in *Proceedings of International Symposium on Fault-Tolerant Computing*, pp. 2-9, 1991.

[43] R. Chillarege, S. Biyani, and J. Rosenthal, "Measurement of Failure Rate in Widely Distributed Software," in *Proc. of 25th IEEE Intl. Symposium on Fault Tolerant Computing*, Pasadena, CA, USA, pp. 424-433, July 1995.

[44] I. Lee and R. K. Iyer, "Software Dependability in the Tandem GUARDIAN System," *IEEE Transactions on Software Engineering*, vol. 21, pp. 455-467, 1995.

[45] A. S. Tanenbaum, *Computer Networks*, Fourth ed: Prentice Hall, 2003.

[46] A. Ganesh, N. O'Connell, and D. Wischik, *Big Queues*: Springer Publication, 2004.

[47] B. P. Douglass, *Real Time UML: Advances in the UML for Real-Time Systems*, 3rd ed: Addison-Wesley Professional, 2004.

[48] J. M. Bacon, *Concurrent Systems: Operating systems, Database and Distributed Systems, an Integrated Approach*, Second ed: Addison Wesley, 1997.

[49] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*: Addison-Wesley, 2000.

[50] V. Garousi, L. Briand, and Y. Labiche, "Control Flow Analysis of UML 2.0 Sequence Diagrams," in *Proceedings of European Conference on Model Driven Architecture-Foundations and Applications, LNCS 3748*, pp. 160-174, 2005.

[51] J. Rumbaugh, I. Jacobson, and G. Booch, *UML Reference Manual*: Addison-Wesley, 1999.

[52] C. Larman, *Applying UML and Patterns*, 2nd edition ed: Prentice Hall, 2002.

[53] F. Fraikin and T. Leonhardt, "SeDiTeC-Testing based on Sequence Diagrams," in *Proceedings of International Conference on Automated Software Engineering*, pp. 261-266, 2002.

[54] Y. Wu, M.-H. Chen, and J. Offutt, "UML-based Integration Testing for Component-Based Software," in *Proceedings of International Conference on COTS (Commercial-Off-The-Shelf)-based Software Systems*, pp. 251-260, 2003.

[55] P. P. Puschner and R. Nossal, "Testing the Results of Static Worst-Case Execution-Time Analysis," in *Proc. of IEEE Real-Time Systems Symp.*, pp. 134–143, 1998.

[56] H. Thane, "Monitoring, Testing and Debugging of Distributed Real-Time Systems," in *Department of Machine Design*. Stockholm, Sweden: Royal Institute of Technology, 2000, pp. 128.

[57] J. Axelsson, "A Method for Evaluating Uncertainties in the Early Development Phases of Embedded Real-Time Systems," in *International Conference on Embedded and Real-Time Computing Systems and Applications*, 2005.

[58] N. Nissanke, L. David, and F. Cottet, "Probabilistic Uni-processor Schedulability Analysis," in *International Workshop on Probabilistic Analysis Techniques for Real-time and Embedded Systems*, 2004.

[59] V. Garousi, L. Briand, and Y. Labiche, "Control Flow Analysis of UML 2.0 Sequence Diagrams," Technical Report SCE-05-09, Carleton University, http://www.sce.carleton.ca/squall/pubs/tech_report/TR_SCE-05-09.pdf, 2005.

[60] L. Briand and Y. Labiche, "A UML-based Approach to System Testing," *Journal of Software and Systems Modeling*, vol. 1, pp. 10-42, 2002.

[61] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. GilChrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development - The Fusion Method*: Prentice Hall, 1994.

[62] R. J. A. Buhr, "Use Case Maps as Architectural Entities for Complex Systems," *IEEE Transactions on Software Engineering*, vol. 24, 1998.

[63] C. Nebut, F. Fleurey, Y. L. Traon, and J.-M. Jézéquel, "Requirements by Contracts allow Automated System Testing," in *Proceedings of International Symposium on Software Reliability Engineering*, pp. 85-96, 2003.

[64] S. Muchnick, *Advanced Compiler Design and Implementation*, First ed: Morgan Kaufmann, 1997.

[65] OMG, "UML 2.0 Superstructure Final Adopted specification," 2003.

[66] H. R. Nielson and F. Nielson, "Infinitary Control Flow Analysis: a Collecting Semantics for Closure Analysis," in *Symp. on Principles of Programming Languages*, pp. 332-345, 1997.

[67] J. Bauer, "A control-flow-analysis for multi-threaded java with security applications," Master's thesis, Universitat des Saarlandes, 2001, pp. 97.

[68] P. D. Blasio, K. Fisher, and C. Talcott, "A Control-Flow Analysis for a Calculus of Concurrent Objects," *IEEE Trans. on Soft. Eng.*, vol. 26, 2000.

[69] D. L. Long and L. A. Clarke, "Task interaction graphs for concurrency analysis," in *Proc. Int. Conf. on Soft. Eng.*, pp. 44-52, 1989.

[70] A. T. Chamillard and L. A. Clarke, "Improving the accuracy of Petri net-based analysis of concurrent programs," in *Proc. Int. Symp. on Soft. testing and analysis*, pp. 24-38, 1996.

[71] A. Rountev, S. Kagan, and J. Sawin, "Coverage Criteria for Testing of Object Interactions in Sequence Diagrams," in *Proc. Conf. Fundamental Approaches to Soft. Eng.*, pp. 289-304, 2005.

[72] M. Okazaki, T. Aoki, and T. Katayama, "Formalizing Sequence Diagrams and State Machines using Concurrent Regular Expression," in *Proceedings of International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, pp. 74-79, 2003.

[73] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*: Addison Wesley, 2003.

[74] R. E. Tarjan, "Depth-First Search and Linear Graph Algorithms," *Society for Industrial and Applied Mathematics' Journal on Computing*, vol. 1, 1972.

[75] Sun Microsystems, "Java Remote Method Invocation (RMI) Specification (version 1.4.2)," 2003.

[76] Wikipedia, "Definition of Binary Large OBject (BLOB)," in *http://en.wikipedia.org/wiki/Binary_large_object*, Last accessed: Feb. 2006.

[77] M. S. Gittens, "The Extended Operational Profile Model for Usage-Based Software Testing," Doctoral Thesis, University of Western Ontario, 2004.

[78] W. D. Shepherd, *Network and Operating System Support for Digital Audio and Video*: Springer, 1994.

[79] F. L. Presti, N. G. Duffield, J. Horowitz, and D. Towsley, "Multicast-based inference of network-internal delay distributions," *IEEE/ACM Transactions on Networking*, 2002.

[80] A. Adas, "Traffic Models in Broadband Networks," *IEEE Communications Magazine*, vol. 35, pp. 82-89, 1997.

[81] S. Sen and J. Wang, "Analyzing Peer-to-Peer Traffic across Large Networks," *IEEE/ACM Transactions on Networking*, vol. 12, pp. 219-232, 2004.

[82] V. Garousi, L. Briand, and Y. Labiche, "Traffic-aware Stress Testing of Distributed Systems based on UML Models," Technical Report SCE-05-13, Carleton University, http://www.sce.carleton.ca/squall/pubs/tech_report/TR_SCE-05-13.pdf, 2005.

[83] V. Garousi, L. Briand, and Y. Labiche, "Traffic-aware Stress Testing of Distributed Systems based on UML Models," in *Proceedings of International Conference on Software Engineering*, pp. 391-400, 2006.

[84] S. I. Gass, *Linear Programming : Methods and Applications*, Fifth ed: Dover Publications, 2003.

[85] M. J. Atallah, *Handbook of Algorithms and Theory of Computation*: CRC (Chemical Rubber Company) Press, 1999.

[86] J. W. Chinneck, "Practical Optimization: A Gentle Introduction," Systems and Computer Engineering, Carleton University. Available at: http://www.sce.carleton.ca/faculty/chinneck/po.html.

[87] R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms*: Wiley-Interscience, 1998.

[88] J. Lahtinen, P. M. Silander, and H. Tirri, "Empirical Comparison of Stochastic Algorithms," in *Proceedings of Nordic Workshop on Genetic Algorithms and their Applications*, pp. 45-60, 1996.

[89] P. Chardaire, A. Kapsalis, J. W. Mann, V. J. Rayward-Smith, and G. D. Smith, "Applications of Genetic Algorithms in Telecommunications," in *Proceedings of Applications of Neural Networks to Telecommunications*, pp. 290-299, 1995.

[90] S. W. Mahfoud and D. E. Goldberg, "Parallel Recombinative Simulated Annealing: A Genetic Algorithm," *Journal on Parallel Computing*, vol. 21, pp. 1-28, 1995.

[91] S. Y. Mahfouz, "Design Optimization of Structural Steel Work," Ph.D. Thesis, Department of Civil and Environmental Engineering, University of Bradford, 1999.

[92] K. De Jong, "Learning with Genetic Algorithms: An Overview," *Machine Learning*, vol. 3, pp. 121-138, 1988.

[93] J. J. Grefenstette and H. G. Cobb, "Genetic Algorithms for Tracking Changing Environments," in *Proceedings of International Conference on Genetic Algorithms*, pp. 523-530, 1993.

[94] J. D. Schaffer, R. A. Caruana, L. J. Eshelman, and R. Das, "A Study of Control Parameters Affecting Online Performance of Genetic Algorithms for Function Optimization," in *Proceedings of International Conference on Genetic algorithms*, pp. 51-60, 1989.

[95] M. A. Pawlowsky, "Crossover Operators," in *Practical Handbook of Genetic Algorithms Applications, L. Chambers Ed.*, pp. 101-114, 1995.

[96] T. Back, "Towards a Practice of Autonomous Systems," in *Proceedings of European Conference on Artificial Life*, pp. 263-271, 1992.

[97] H. Mühlenbein, "Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization," in *Proceedings of International Conference on Genetic algorithms*, pp. 416-421, 1989.

[98] J. E. Smith and T. C. Fogarty, "Adaptively Parameterized Evolutionary Systems: Self Adaptive Recombination and Mutation in a Genetic Algorithm," *Proceedings of International Conference on Parallel Problem Solving From Nature*, pp. 441-450, 1996.

[99] M. Wall, "GAlib: A C++ Library of Genetic Algorithm Components," Documentation version 2.4, Massachusetts Institute of Technology 1996.

[100] V. Garousi, "GARUS (Genetic Algorithm-based test Requirement tool for real-time distribUted Systems)," in *http://squall.sce.carleton.ca/tools/GARUS*, 2006.

[101] S. J. Louis and G. J. E. Rawlins, "Predicting Convergence Time for Genetic Algorithms," Technical Report 370, Computer Science Department, Indiana University 1993.

[102] A. Daneels and W. Salter, "What is SCADA?," in *Proceedings of International Conference on Accelerator and Large Experimental Physics Control Systems*, pp. 39-343, 1999.

[103] J. Brunton, G. Digby, and A. Doherty, "Design and Operational Philosophy for a Metro Power Network SCADA System," in *of International Conference on Power System Control and Management*, pp. 176-180, 1996.

[104] Y. Ebata, H. Hayashi, Y. Hasegawa, S. Komatsu, and K. Suzuki, "Development of the Intranet-based SCADA for Power System," in *Proceedings of IEEE Power Engineering Society Winter Meeting*, pp. 1656-1661, 2000.

[105] T. Seki, T. Tsuchiya, T. Tanaka, H. Watanabe, and T. Seki, "Network Integrated Supervisory Control for Power Systems based on Distributed Objects," in *Proceedings of International Symposium on Applied Computing*, pp. 620-626, 2000.

[106] M. Mavrin, V. Koroman, and B. Borovic, "SCADA in Hydropower Plants," in *Proceedings of International Symposium on Computer Aided Control System Design*, pp. 624-629, 1999.

[107] E.-K. Chan and H. Ebenhoh, "The Implementation and Evolution of a SCADA System for a Large Distribution Network," *IEEE Transactions on Power Systems*, vol. 7, pp. 320-326, 1992.

[108] D. Trung, "Modern SCADA Systems for Oil Pipelines," in *Proceedings of International Petroleum and Chemical Industry Conference*, pp. 299-305, 1995.

[109] A. J. N. Batista, A. Combo, J. Sousa, and C. A. F. Varandas, "A Low Cost, Fully Integrated, Event-driven, Real-time Control and Data Acquisition System for Fusion Experiments," *Review of Scientific Instruments*, vol. 74, pp. 1803-1806, 2003.

[110] J. A. How, J. W. Farthing, and V. Schmidt, "Trends in Computing Systems for Large Fusion Experiments," *Fusion Engineering and Design*, vol. 70, pp. 115-122, February 2004.

[111] Z. Constantinescu, P. Petrovic, A. Pedersen, D. Federici, and J. Campos, "QADPZ (Quite Advanced Distributed Parallel Zystem)," in *http://qadpz.sourceforge.net*, 2003.

[112] A. Sauvé, C. Matthews-Dickson, and O. Peterson, "Real-Time Distributed Factory Automation System," Fourth-year Engineering Project Report, Department of Systems and Computer Engineering, Carleton University 2003.

[113] European Information Society Technologies (IST), "COACH (Component Based Open Source Architecture for Distributed Telecom Applications," in *http://coach.objectweb.org*, 2003.

[114] US military, "The Joint Interoperability Test Command," in *http://jitc.fhu.disa.mil/*, 2005.

[115] "CitectSCADA," in *http://www.citect.com/products/citectscada*, 2005.

[116] BWI Co., "ElipseSCADA," in *http://www.bwi.com/proot/2775*, 2004.

[117] N. Toshida, M. Uesugi, Y. Nakata, M. Nomoto, and T. Uchida, "Open Distributed EMS/SCADA System," *Hitachi Review*, vol. 47, pp. 208-213, 1998.

[118] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns, and Java*, 2nd Edition ed: Prentice Hall, 2003.

[119] H. S. Kim, J. M. Lee, T. Park, J. Y. Lee, and W. H. Kwon, "Design of Networks for Distributed Digital Control Systems in Nuclear Power Plants," in *Proceedings of International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies*, pp. 629-633, 2000.

[120] B. Stojkovic and I. Vujosevic, "A Compact SCADA System for a Smaller Size Electric Power System Control-a Fast, Object-Oriented and Cost-Effective Approach," in *Proceedings of IEEE Power Engineering Society Winter Meeting*, pp. 695-700, 2002.

[121] Wikipedia, "Definition of Controllability," in *http://en.wikipedia.org/wiki/Controllability*, 2005.

[122] Wikipedia, "Definition of Observability," in *http://en.wikipedia.org/wiki/Observability*, Last accessed: Feb. 2006.

[123] A. Makinen, M. Parkki, P. Jarventausta, M. Kortesluoma, P. Verho, S. Vehvilainen, R. Seesvuori, and A. Rinta-Opas, "Power Quality Monitoring as Integrated with Distribution Automation," in *Proceedings of International Conference and Exhibition on Electricity Distribution*, pp. 172-172, 2001.

[124] A. Avritzer, J. P. Ros, and E. J. Weyuker, "Estimating the CPU Utilization of a Rule-based System," *ACM SIGSOFT Software Engineering Notes*, vol. 29, pp. 1-12, 2004.

[125] Y. F. Wang, M. H. Hsu, and Y. L. Chuang, "Predicting CPU Utilization by Fuzzy Stochastic Prediction," *Computing and Informatics*, vol. 20, pp. 67-76, 2001.

[126] A. Ligeza, *Logical Foundations for Rule-Based Systems*: Springer, 2006.

[127]

[128] L. Briand, Y. Labiche, and Y. Miao, "Towards the Reverse Engineering of UML Sequence Diagrams," in *Proceedings of International Working Conference on Reverse Engineering*, pp. 57-66, 2003.

[129] Object Management Group (OMG), "UML 2.0 Testing Profile Specification," 2003.

[130] V. Garousi, L. Briand, and Y. Labiche, "A Unified Approach for Predictability Analysis of Real-Time Systems using UML-based Control Flow Information," in *International Workshop on Modeling and Analysis of Real-Time and Embedded Systems (MARTES), in conjunction with International Conference on Model Driven Engineering Languages and Systems*, 2005.

[131] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*: Addison-Wesley, 1989.

[132] T.-P. Hong, H.-S. Wang, and W.-C. Chen, "Simultaneously Applying Multiple Mutation Operators in Genetic Algorithms," *Journal on Heuristics*, vol. 6, pp. 439-455, 2000.

[133] E. S. H. Hou, N. Ansari, and H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling," *IEEE Transactions on Parallel and Distributed Systtems*, vol. 5, pp. 113-120, 1994.

[134] Wikipedia, "Definition of NP-hard," in *http://en.wikipedia.org/wiki/NP_hard*, Last accessed: Feb. 2006.

Appendix A- Genetic Algorithms Overview

In [130-133], the authors describe GAs as a means of solving complex optimization problems that are often NP-hard[19] [134] in limited amounts of time. Optimization problems are those that try to reach the best solution given the measurement of the goodness of solutions. GAs are based on concepts adopted from genetic and evolutionary theories. GAs are comprised of several components: a representation of the solutions, referred to as the *chromosomes*, fitness of each chromosome, referred to as the *objective (fitness) function*, the genetic operations of *crossover* and *mutation* which generate new offspring, and selection operations which choose offspring fit for survival.

A chromosome models the problem solutions. Each element within a chromosome is known as a *gene*. The collection of chromosomes used by the GA is called a *population*.

---

[19] In computational complexity theory, NP-hard (Non-deterministic Polynomial-time hard) refers to the class of decision problems that contains all problems H such that for every decision problem L in NP there exists a polynomial-time many-to-one reduction to H, written $L \leq H$. Informally this class can be described as containing the decision problems that are at least as hard as any problem in NP. This intuition is supported by the fact that if we can find an algorithm A that solves one of these problems H in polynomial time then we can construct a polynomial time algorithm for any problem L in NP by first performing the reduction from L to H and then running the algorithm A [99] M. Wall, "GAlib: A C++ Library of Genetic Algorithm Components," Documentation version 2.4, Massachusetts Institute of Technology 1996..

Figure 150 illustrates these concepts in terms of representation of the Red/Green/Blue (RGB) makeup of a population of three pixels on a screen. The chromosome in the figure is composed of three genes. Each gene represents the red, green or blue components of a pixel on a screen. Hence, the chromosome depicts one pixel's RGB makeup. The population portrays the makeup of three pixels on the screen.

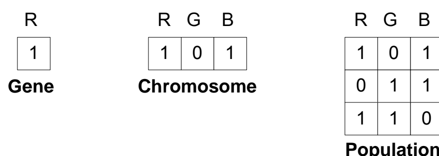


Figure 150-GA chromosome terminology.

The quality of a chromosome is its *fitness*. Fitness defines which chromosomes are closer to the optimal solution. If the optimal solution for the population of Figure 150 is a pixel with only a red component (i.e. a chromosome with RGB values 100), the first and the last chromosomes of the population would be deemed fitter than the second one.

Both crossover and mutation operators are needed to explore the problem search space. Crossover operators generate offspring from two parents based on the merits of each parent, as demonstrated in Figure 151 through *single point* crossover[20].

---

[20] Single-point crossover is on type of crossover operators. There are other types such as multi-point crossover.

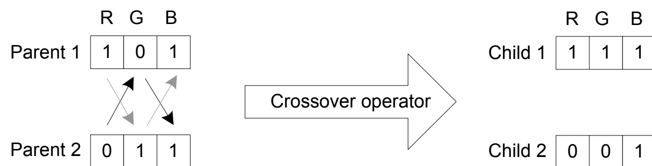


Figure 151-Illustration of crossover operator (*single point* crossover).

Taking the *G* gene of a chromosome as a division point common to both parents, the parents alternate genes with respect to the division point in creating the children. Parent 1 contributes the *RB* components of Child 1, allowing Parent 2 to contribute the *G* component. Similarly, Parent 2 contributes the RB components of Child 2, while Parent 1 contributes its *G* component. Hence, GAs use the notion of survival of the fittest by passing superior traits from one generation to the next.

Mutation operators *mutate*, or alter, a single chromosome. Mutation aids the GA in avoiding local minima. In the example in Figure 152, the red gene is mutated, resulting in a chromosome with RGB values 010.

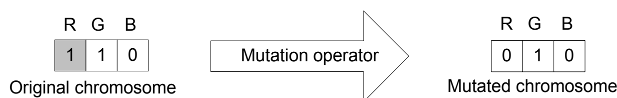


Figure 152-Illustration of mutation operator.

The process of selecting determines which individuals among the original populations, mutated and child chromosomes will survive, hence retaining a constant population size.

An initial population of individuals (usually random) is first given to a GA. Working with the population, the GA then selects and performs various crossover and mutation operations, creating new chromosomes. The fitness of the new chromosomes (using the objective function) is compared to others in the population. Fitter individuals are retained

while less fit ones are removed. The process of crossover, mutation, fitness comparison and replacement continues until a termination criterion is reached. In most cases, the termination criterion is a particular number of runs or generations of the algorithm [133]. By adopting the GA process concept from [99], we can draw an activity diagram for the process as shown in Figure 153.

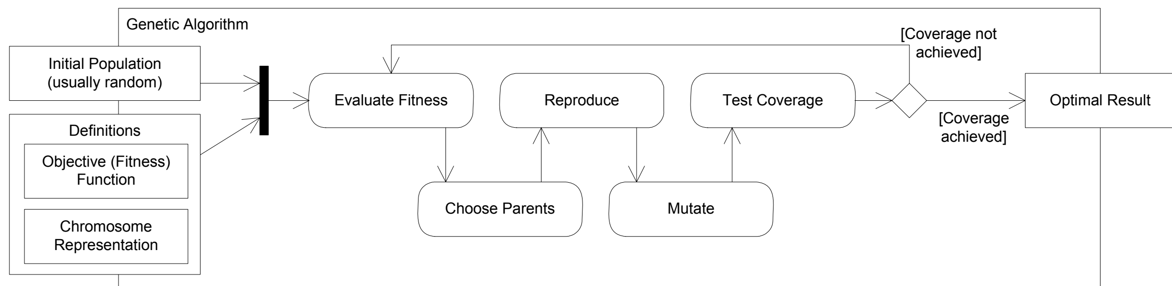


Figure 153-Activity diagram of the most general form of genetic algorithms (concept from [87]).

A variety of replacement methodologies are defined for GAs, such as simple, steady state and incremental. Each replacement methodology specifies how much of the population should be replaced with each run or generation of the algorithm. The simple GA creates an entirely new population of chromosomes with each generation of the algorithm. The steady state algorithm, on the other hand, uses overlapping populations, leaving it up to the user to determine the number of chromosomes to replace in each generation. Each generation, the steady state GA produces, are stored in a temporary location. These are then added to the population and the worst individuals are removed such that the population such that the population size remains constant. In incremental genetic algorithms, only one or two offspring chromosomes are generated. These are integrated

into the population in one of the following ways: replacing the parent, replacing a random individual in the population, or replacing an individual that is similar to the offspring.

Appendix B- Proof of the Formula to Calculate the Unbounded Range Starting Point (URSP) of a Bounded Arrival Pattern

The following is the proof of formula (Equation 9) to calculate the Unbounded Range Starting Point (URSP) of a bounded arrival pattern (AP), given the minimum and maximum inter-arrival times (*minIAT* and *maxIAT*) of the AP The formula is used in Section 10.7.4 for determining a suitable maximum search time for our GA.

Recall from Section 10.2 the concept of Accepted Time Intervals (ATI) for a bounded arrival pattern. For example, the gray eclipses in the timing diagram in Figure 59 depict the ATIs of the arrival pattern (*'bounded', (4, ms), (5, ms)),* i.e. m*inIAT=4ms, maxIAT=5ms*. To devise a formula to find the calculate the URSP of a bounded AP, we formalize bounded APs time properties as demonstrated in Figure 154.

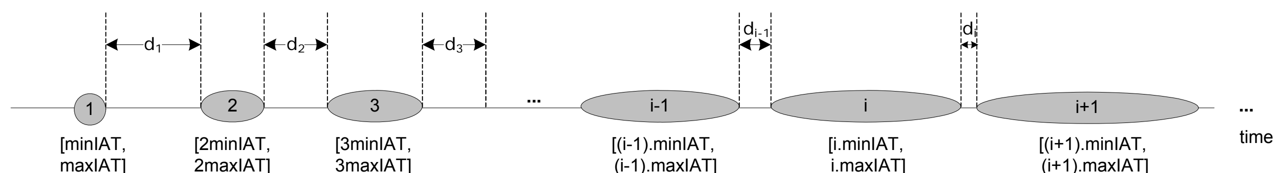


Figure 154-Formalization of bounded APs time properties.

ATIs are indexed and are referred as ATI_i. Recall from Section 10.2 (analysis of arrival patterns) that each ATI's start and end times are multiples of the AP's m*inIAT* and *maxIAT*, respectively. For example, the consecutive ATIs of the arrival pattern (*'bounded', (4, ms), (5, ms))* are: *[4 ms, 5 ms], [8 ms, 10 ms], [12 ms, 15 ms], [16 ms, 20 ms]*, and so on. In the parametric form, consecutive ATIs are shown in Figure 154 as: *[*m*inIAT, maxIAT]*, *[2*m*inIAT, 2maxIAT]*, *[3*m*inIAT, 3maxIAT]*, *[k.*m*inIAT, k.maxIAT]*, and so on. d_i denotes the closest distance between two neighboring ATIs ATI_i and ATI_{i+1}. It is obvious that:

$$d_i = (i+1).minIAT - i.maxIAT$$

The URSP of a bounded AP appears when two consecutive ATIs overlap[21], i.e., the start time of the next ATI is smaller than or equal to the end time of the current ATI. This, in turn, entails than $d_k \leq 0$ in such a case. k here denotes the index of the ATI whose start time is the URSP of the bounded AP. Such a situation is visualized in Figure 155.

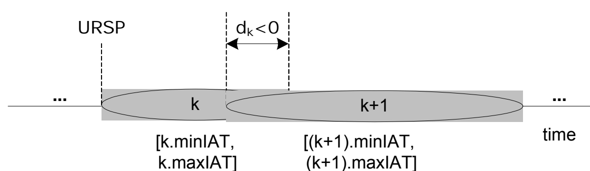


Figure 155-Two overlapping ATIs.

Therefore, in order to find the value of URSP, we should find the *k*-th ATI's start time such that $d_k \leq 0$. To prove that for every bounded AP, there exists a URSP, we should prove that there exists a $d_k \leq 0$. This can be proved if we show that the d_i values of a bounded AP are *descending,* i.e., $d_i > d_{i+1}$. Since supposing that d_i will start from a positive value and it is descending, it will at some point be equal to zero or less than zero.

Theorem. The d_i values of a bounded AP are *descending,* i.e., $d_i > d_{i+1}$.

Proof. We follow a proof-by-contradiction approach. Assume to the contrary that $d_i \leq d_{i+1}$. Then:

---

[21] Two ATIs are said to be *overlapped* if they have at least one common ATP.

$$\begin{gathered} d_i \leq d_{i+1} \\ (i+1)minIAT - i.maxIAT \leq (i+1+1)minIAT - (i+1).maxIAT \\ maxIAT \leq minIAT \end{gathered}$$

We can see that, after taking the proof-by-contradiction approach, we have got a conclusion which contradicts the assumption of m*inIAT*<*maxIAT* for bounded APs (Section 10.1). This means the proof of the main theorem, i.e., the d_i values of a bounded AP are *descending,* i.e., $d_i > d_{i+1}$.

Therefore, in order to find the value of URSP, we should find *k*-th ATI's start time such that $d_k \leq 0$ (the smallest *k*). If we find the index, *k*, of the ATI, URSP can be found easily by *URSP=k.minIAT*. The value of *k* can be found as the following:

$$\begin{aligned} & d_k \leq 0 \\ & \Rightarrow (k+1)minIAT - k.maxIAT \leq 0 \\ & \Rightarrow k(maxIAT - minIAT) \geq minIAT \\ & \Rightarrow k \geq \frac{minIAT}{maxIAT - minIAT} \\ & \Rightarrow k = \left\lceil \frac{minIAT}{maxIAT - minIAT} \right\rceil \quad \text{(since } k \text{ is an integer.)} \end{aligned}$$

Therefore, the URSP of a bounded AP can be calculated by:

$$URSP = \left\lceil \frac{minIAT}{maxIAT - minIAT} \right\rceil .minIAT$$

For example, the URSP of the bounded AP (*'bounded', (4, ms), (5, ms))* is 16 ms which can be verified visually in Figure 59.

$$URSP = \left\lceil \frac{4}{5-4} \right\rceil .4 = 16ms$$

Appendix C- Sample Test Models used in Validation of Test Requirements Generated by GARUS

The following is the input test file corresponding to test model #20 (Section 11.3.5.5). The test model does not correspond to a real system. It is among a set of experimental test models generated by a random test model generator to validate GARUS test requirements. Refer to Section 11.3.5.5 for details.

```
--ISDSs
10
ISDS0 6 SD14 SD3 SD11 SD9 SD19 SD17
ISDS1 6 SD7 SD17 SD10 SD18 SD13 SD9
ISDS2 6 SD8 SD3 SD17 SD9 SD6 SD5
ISDS3 5 SD3 SD15 SD8 SD1 SD11
ISDS4 9 SD0 SD1 SD13 SD7 SD6 SD11 SD17 SD12 SD5
ISDS5 5 SD11 SD1 SD18 SD16 SD6
ISDS6 8 SD6 SD13 SD2 SD5 SD11 SD12 SD7 SD15
ISDS7 9 SD3 SD14 SD6 SD2 SD13 SD12 SD4 SD5 SD10
ISDS8 6 SD2 SD9 SD17 SD19 SD15 SD4
ISDS9 7 SD11 SD12 SD14 SD5 SD15 SD2 SD17
--SDs
20
SD0 3 4 p0.0 p0.1 p0.2 p0.3
SD1 9 4 p1.0 p1.1 p1.2 p1.3
SD2 2 4 p2.0 p2.1 p2.2 p2.3
SD3 6 3 p3.0 p3.1 p3.2
SD4 2 4 p4.0 p4.1 p4.2 p4.3
SD5 4 2 p5.0 p5.1
SD6 2 4 p6.0 p6.1 p6.2 p6.3
SD7 9 3 p7.0 p7.1 p7.2
SD8 2 4 p8.0 p8.1 p8.2 p8.3
SD9 4 3 p9.0 p9.1 p9.2
SD10 8 2 p10.0 p10.1
SD11 9 3 p11.0 p11.1 p11.2
SD12 6 4 p12.0 p12.1 p12.2 p12.3
SD13 8 4 p13.0 p13.1 p13.2 p13.3
SD14 2 3 p14.0 p14.1 p14.2
SD15 8 2 p15.0 p15.1
SD16 5 3 p16.0 p16.1 p16.2
SD17 7 3 p17.0 p17.1 p17.2
SD18 3 4 p18.0 p18.1 p18.2 p18.3
SD19 1 4 p19.0 p19.1 p19.2 p19.3
--SD_Arrival_Patterns
SD0 no_arrival_pattern
SD1 bounded 3 5
SD2 periodic 9 1
SD3 irregular 9 14 20 6 5 27 21 7 19 25
SD4 bounded 2 5
SD5 irregular 6 3 20 22 9 22 24
SD6 no_arrival_pattern
SD7 periodic 9 1
SD8 irregular 11 5 24 1 25 16 11 12 26 17 27 26
SD9 bounded 2 6
SD10 bounded 2 5
SD11 bounded 3 6
SD12 irregular 5 10 21 11 9 21
SD13 no_arrival_pattern
SD14 irregular 11 25 29 20 2 3 1 9 1 24 29 17
SD15 periodic 6 0
```

```
SD16 bounded 2 5
SD17 no_arrival_pattern
SD18 no_arrival_pattern
SD19 no_arrival_pattern
--DCCFPs
p0.0 2 ( 42 13 ) ( 76 6 )
p0.1 7 ( 29 8 ) ( 50 14 ) ( 93 17 ) ( 113 15 ) ( 148 13 ) ( 185 15 ) ( 234 6 )
p0.2 6 ( 0 18 ) ( 28 6 ) ( 50 17 ) ( 82 17 ) ( 84 5 ) ( 130 18 )
p0.3 6 ( 2 16 ) ( 3 17 ) ( 38 17 ) ( 46 12 ) ( 48 11 ) ( 89 15 )
p1.0 3 ( 13 13 ) ( 51 15 ) ( 88 5 )
p1.1 2 ( 26 17 ) ( 33 9 )
p1.2 8 ( 31 16 ) ( 57 15 ) ( 85 8 ) ( 118 11 ) ( 163 6 ) ( 212 18 ) ( 261 8 ) ( 263 17 )
p1.3 3 ( 40 9 ) ( 49 12 ) ( 52 15 )
p2.0 5 ( 33 8 ) ( 69 11 ) ( 79 5 ) ( 115 8 ) ( 130 12 )
p2.1 7 ( 1 9 ) ( 38 7 ) ( 48 11 ) ( 96 16 ) ( 132 9 ) ( 180 15 ) ( 218 17 )
p2.2 9 ( 33 18 ) ( 77 8 ) ( 117 14 ) ( 143 12 ) ( 170 16 ) ( 189 15 ) ( 191 5 ) ( 204 15
) ( 239 8 )
p2.3 9 ( 3 12 ) ( 37 7 ) ( 77 8 ) ( 96 13 ) ( 107 12 ) ( 110 18 ) ( 111 18 ) ( 136 16 ) (
182 17 )
p3.0 2 ( 18 17 ) ( 30 6 )
p3.1 5 ( 3 9 ) ( 42 19 ) ( 46 17 ) ( 61 12 ) ( 90 11 )
p3.2 1 ( 42 19 )
p4.0 2 ( 26 14 ) ( 61 7 )
p4.1 6 ( 1 6 ) ( 2 10 ) ( 3 7 ) ( 43 9 ) ( 70 11 ) ( 102 15 )
p4.2 8 ( 11 16 ) ( 58 14 ) ( 61 5 ) ( 76 7 ) ( 83 14 ) ( 103 10 ) ( 105 18 ) ( 110 14 )
p4.3 2 ( 45 14 ) ( 74 17 )
p5.0 9 ( 23 18 ) ( 36 19 ) ( 51 5 ) ( 84 17 ) ( 118 9 ) ( 142 11 ) ( 156 14 ) ( 164 15 )
( 194 9 )
p5.1 7 ( 20 5 ) ( 28 16 ) ( 38 10 ) ( 60 11 ) ( 61 16 ) ( 79 7 ) ( 123 8 )
p6.0 9 ( 7 11 ) ( 44 5 ) ( 79 7 ) ( 112 17 ) ( 112 11 ) ( 127 12 ) ( 134 13 ) ( 157 18 )
( 179 10 )
p6.1 5 ( 23 15 ) ( 57 18 ) ( 75 19 ) ( 104 5 ) ( 136 13 )
p6.2 4 ( 45 17 ) ( 78 6 ) ( 125 6 ) ( 164 17 )
p6.3 8 ( 6 10 ) ( 42 5 ) ( 91 16 ) ( 98 9 ) ( 118 15 ) ( 125 10 ) ( 169 6 ) ( 182 16 )
p7.0 2 ( 1 8 ) ( 41 7 )
p7.1 1 ( 30 17 )
p7.2 4 ( 6 6 ) ( 42 5 ) ( 85 16 ) ( 86 19 )
p8.0 1 ( 45 19 )
p8.1 9 ( 12 16 ) ( 27 10 ) ( 59 12 ) ( 105 12 ) ( 134 13 ) ( 170 13 ) ( 179 19 ) ( 182 6
) ( 195 6 )
p8.2 1 ( 8 14 )
p8.3 8 ( 43 9 ) ( 79 9 ) ( 83 11 ) ( 112 14 ) ( 137 8 ) ( 159 5 ) ( 190 9 ) ( 229 17 )
p9.0 8 ( 26 15 ) ( 62 16 ) ( 102 7 ) ( 126 14 ) ( 143 14 ) ( 177 9 ) ( 226 19 ) ( 252 17
)
p9.1 9 ( 8 9 ) ( 32 7 ) ( 46 8 ) ( 89 5 ) ( 127 6 ) ( 166 17 ) ( 185 6 ) ( 186 15 ) ( 204
13 )
p9.2 5 ( 10 18 ) ( 20 17 ) ( 51 10 ) ( 86 5 ) ( 108 6 )
p10.0 2 ( 28 6 ) ( 29 7 )
p10.1 2 ( 34 5 ) ( 48 7 )
p11.0 9 ( 26 15 ) ( 36 13 ) ( 72 9 ) ( 113 11 ) ( 126 9 ) ( 163 6 ) ( 170 9 ) ( 212 8 ) (
238 12 )
p11.1 3 ( 45 8 ) ( 58 12 ) ( 105 19 )
p11.2 1 ( 26 8 )
p12.0 3 ( 48 13 ) ( 93 8 ) ( 121 19 )
p12.1 8 ( 36 16 ) ( 54 12 ) ( 82 18 ) ( 93 11 ) ( 132 15 ) ( 144 12 ) ( 192 11 ) ( 226 8
)
p12.2 9 ( 14 8 ) ( 56 11 ) ( 79 5 ) ( 114 5 ) ( 134 18 ) ( 160 7 ) ( 186 9 ) ( 228 15 ) (
262 6 )
p12.3 4 ( 37 15 ) ( 62 8 ) ( 101 6 ) ( 143 7 )
p13.0 2 ( 15 18 ) ( 45 19 )
p13.1 2 ( 33 8 ) ( 51 5 )
p13.2 7 ( 7 7 ) ( 46 15 ) ( 55 15 ) ( 75 17 ) ( 99 14 ) ( 103 18 ) ( 116 10 )
p13.3 9 ( 7 11 ) ( 33 13 ) ( 77 9 ) ( 78 10 ) ( 122 5 ) ( 127 6 ) ( 160 18 ) ( 200 13 ) (
246 17 )
p14.0 1 ( 3 19 )
p14.1 3 ( 8 15 ) ( 40 18 ) ( 44 10 )
p14.2 1 ( 11 11 )
p15.0 5 ( 47 8 ) ( 50 10 ) ( 76 14 ) ( 103 17 ) ( 117 6 )
p15.1 1 ( 3 18 )
p16.0 4 ( 28 15 ) ( 37 7 ) ( 44 13 ) ( 90 17 )
```

```
p16.1 9 ( 45 6 ) ( 81 9 ) ( 122 11 ) ( 128 17 ) ( 147 15 ) ( 195 17 ) ( 218 12 ) ( 223
7 ) ( 266 6 )
p16.2 9 ( 41 17 ) ( 85 10 ) ( 114 9 ) ( 125 7 ) ( 173 9 ) ( 210 12 ) ( 258 9 ) ( 259 13 )
( 284 18 )
p17.0 6 ( 10 6 ) ( 36 13 ) ( 75 16 ) ( 86 18 ) ( 103 5 ) ( 116 9 )
p17.1 2 ( 32 6 ) ( 56 10 )
p17.2 9 ( 10 8 ) ( 53 11 ) ( 78 15 ) ( 84 5 ) ( 103 9 ) ( 103 10 ) ( 120 17 ) ( 121 19 )
( 160 15 )
p18.0 7 ( 5 7 ) ( 20 6 ) ( 34 12 ) ( 80 6 ) ( 102 9 ) ( 112 10 ) ( 124 8 )
p18.1 8 ( 47 9 ) ( 53 14 ) ( 59 10 ) ( 96 12 ) ( 96 14 ) ( 139 17 ) ( 144 13 ) ( 162 17 )
p18.2 2 ( 36 13 ) ( 74 10 )
p18.3 4 ( 33 7 ) ( 80 16 ) ( 105 14 ) ( 109 12 )
p19.0 2 ( 19 14 ) ( 38 8 )
p19.1 8 ( 13 5 ) ( 41 6 ) ( 67 7 ) ( 100 12 ) ( 133 16 ) ( 162 8 ) ( 205 12 ) ( 213 10 )
p19.2 3 ( 9 17 ) ( 31 8 ) ( 58 16 )
p19.3 2 ( 0 10 ) ( 21 15 )
--GATimeSearchRange

150
```