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

Vahid Garousi, Lionel Briand and Yvan Labiche

Software Quality Engineering Laboratory (SQUALL)
www.sce.carleton.ca/squall
Department of Systems and Computer Engineering, Carleton University
1125 Colonel By Drive, Ottawa, ON K1S5B6, Canada
{vahid | briand | labiche}@sce.carleton.ca

Abstract

A stress test methodology aimed at increasing chances of discovering faults related to network 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 network 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.

Keywords:

Stress testing, performance testing, model-based testing, distributed systems, real-time systems, UML, network traffic, genetic algorithms

Table of Contents

List of Figures

List of Acronyms

AD	Activity Diagram
AOCS	Attitude and Orbit Control System
AP	Arrival Pattern
APC	Arrival-Pattern Constraint
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
NIT	Network Interconnectivity Tree
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
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

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 plan systems [3]. 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 [4], 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 [5], 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, in terms of outage numbers, although only 6% of the outages were overloads, but 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.

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 stressed 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. But 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

Assuming that the UML design model of a DRTS, sequence diagrams annotated with timing information are provided, we propose a technique to derive test requirements to stress the robustness of a system to network traffic problems in a cost-effective manner. 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 on a subset of all nodes.

The Unified Modeling Language (UML) [6-8] 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 [9]. The new version of UML, version 2.0 [8], 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, extensibility, support for component-based development, modeling of relationships and model management [9]. 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 an 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. 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:

- A 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, 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.
- More specifically, the work includes a specific technique based on Genetic Algorithms aimed at dealing with internal and external system events exhibiting complex arrival patterns. This is of the utmost importance for the testing of real-time systems (Chapter 10)

1.4 Structure

The remainder of this article 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 paper. Chapter 3 presents a 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, a necessary first step. 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-express 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. Finally, Chapter 13 concludes this article 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 article (Section 2.3), and a brief introduction to the UML profile for Schedulability, Performance, and Time (UML-SPT) [10] (Section 2.4).

2.1 Related Works

There has not been a great deal of work published on systematic generation of stress and load test suites for software systems. The works in [11-15] 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 stressed conditions. For example, Kuhn [5] 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, in terms of outage numbers, 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 [13] 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 the 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 the number of calls 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.

Yang proposed a technique [11] 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. [12] 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 [17] coupled with temporal constraints. Allen's interval temporal logic [18] was used by the authors to model temporal relationships. For example, given two media objects, *VideoA* and *VideoB*, the representation: $a_{VideoB} = b_{VideoA} + 4$ (where a_{VideoB} and b_{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 networking 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 [17] and Allen's interval temporal logic [18]. 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: towards a nod`e vs. from a node. This will be discussed in detail in our system model and methodology sections.

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 paper.

> 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 from 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 [19]. 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 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 [19].

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 system. A network may be a subnet of another network or the supernet of several other networks. A more comprehensive definition of a network is given in Section 4.2.1.

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 [20-24]. 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) [10] (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 [25-29] and the industry [30]. 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 [10].

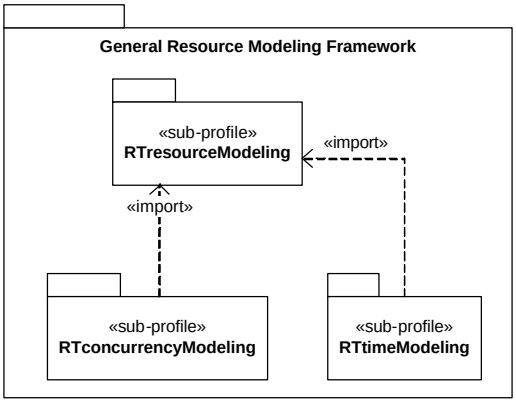


Figure 1-The structure of the GRM Framework of the UML-SPT profile [10].

According to the UML-SPT profile's specification [10], 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. The time domain model is divided into 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 the time constraints in a distributed system (Section 3.1), 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 [10].

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 to send the data to be updated in the control server (n_{cs}).

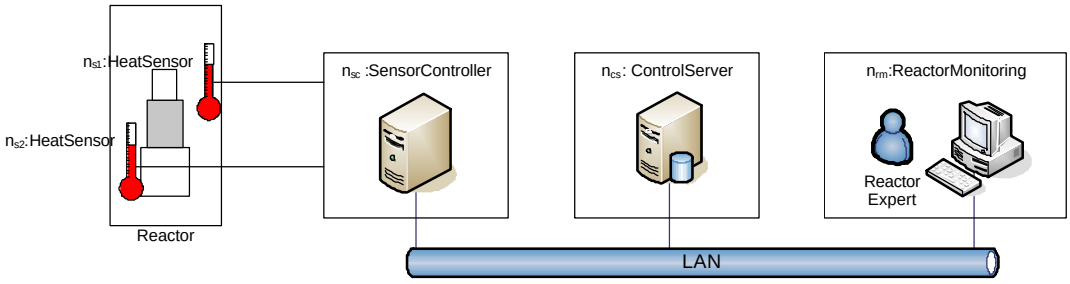


Figure 2-Part of the deployment architecture of a chemical reactor system.

The sequence diagram (SD) in Figure 3 shows the realization of the update process. The SD is using time modeling constructs in the *TimeModel* package of the UML-SPT profile and the UML 2.0 [8] notations. As a reminder, the graphical notations of UML 2.0 for synchronous, asynchronous, and reply messages are also indicated in Figure 3.

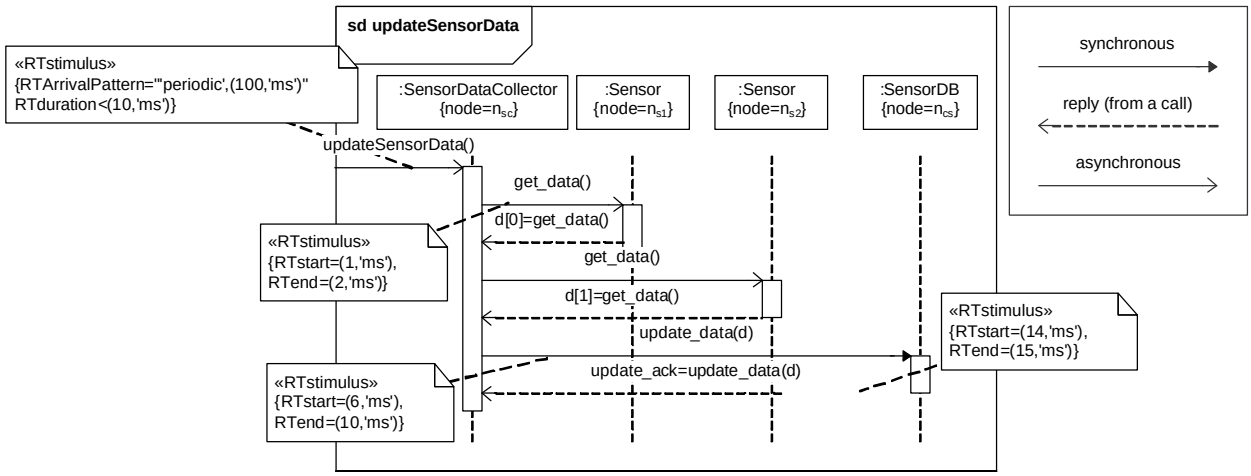


Figure 3-Example of time modeling using UML-SPT profile.

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.

Chapter 3

A FAULT TAXONOMY FOR DISTRIBUTED 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 [31]. Three fundamental categories of threats exist in the dependability theory: *failures*, *errors*, and *faults*. A failure occurs when an error reaches the service interface and alters the service. 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. [31], as depicted in Figure 4.

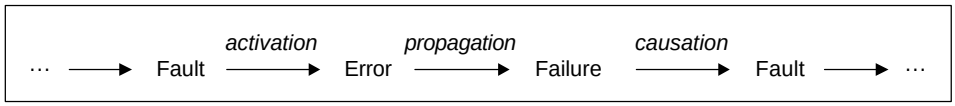


Figure 4-The fundamental chain of dependability threats.

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, where 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*".

Adopting the concept of dependability to our context, i.e., distributed systems, it would make sense to count for specific faults which occur specially in distributed systems as a different category of faults. The elementary fault classes were proposed by Avizienis et al. [31], which included classes like "domain" and "system boundary" of a fault. The larger fault classes are further categorized into subclasses. For example, the fault class of "domain" contains two subclasses of "hardware" and "software" fault subclasses. We generalize the fault category, given in [31], to incorporate the distributed faults as well. Our proposed additions are given in Figure 5, where the faults classes on the gray background are our proposed additions, while those with dotted border were given by Avizienis et al. in [31]. We have added two top-most categories: 'nature` and 'location of creation or occurrence`.

As given in Figure 5, we consider four different categories for the nature of a fault: *local*, *distributed*, *real-time* and *concurrency*. Local faults are those which occur on a node in the system and basically do not have any

thing to do with the distribution of the system. Functional faults in a single process are examples of local faults. Distributed faults are those which occur due to the distribution nature of a system. We define two types of distributed faults: unavailability and traffic, which will be described in detail in Section 3.2. As their name indicate themselves, real-time and concurrency faults relate to the real-time constraints and concurrent character of a system.

The "Location of creation or occurrence" fault class indicates the location where a fault has occurred. For the case of faults in distributed system, we assume two cases for this class: *network* and *node*. In other words, we assume that a fault (in a distributed system) may happen either in a network or in a node. This will be described in detail in Section 3.5.

In this work, the system under test is a distributed system composed of several nodes and several concurrent processes running in each node. There can also be real-time constraints in the system. The idea for classification of faults by their "nature" in such a system is that, aside from the *local* system classifications (shown as boxes with dotted border in Figure 5), a fault may have a pure distributed, real-time or concurrency cause.

In the following sections, we first revisit the persistency of faults. Then, each of our proposed fault classes will be further discussed. We also give examples for each category of faults. This will clarify the stress testing methodology in Section 8.5 and will highlight the types of faults we want to tackle in this work.

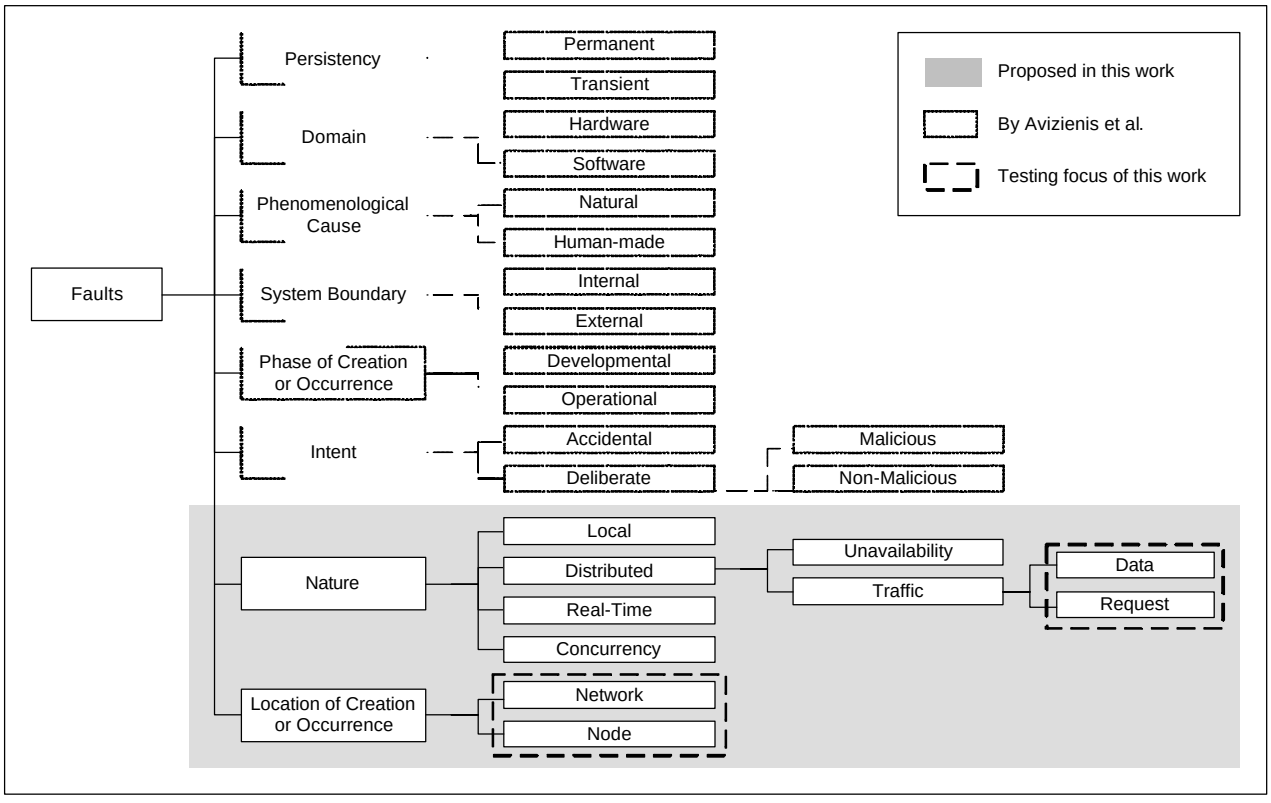


Figure 5-Tree of Fault Classes generalized for Distributed Systems.

3.1 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 [32]. Some other studies classify software faults as both permanent and transient. Gray [33] classifies software faults into *Bohrbugs* and *Heisenbugs*. Bohrbugs are essentially permanent design faults and hence almost deterministic in nature, and they correspond to permanent faults as classified by Avizienis et al. in [31], shown in Figure

5. Bohrbugs can be identified easily and can be removed during the testing and debugging phase (or early deployment phase) of the software life cycle. Heisenbugs, on the other hand, belong to the class of temporary internal faults and are intermittent. Heisenbugs correspond to transient faults as classified by Avizienis et al. in [31], shown in Figure 5. Heisenbugs are essentially permanent 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 Heisenbugs might surface are high usage loads, improper or insufficient exception handling and interdependent timing of various events. It is for this reason that Heisenbugs are difficult to identify through testing. 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 Heisenbugs than due to Bohrbugs.

Some studies on failure data have reported that a large proportion of software failures are transient in nature [33, 34], caused by phenomena such as overloads or timing and exception errors [35, 36]. 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 [37, 38]. In another recent paper by Kuhn [5], sources of failures in the United States' Public Switched Telephone Network (PSTN), as a very big distributed system, were investigated. It was reported that in the time period of 1992-1994, although only 6% of the system outages were overloads, but 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 overloads happen not frequently, but the failure costs due to them can be expensive.

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

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

The proposed technique in this work aims to cause Heisenbugs (transient) faults with network traffic overload type.

3.2 Distributed Faults

Since nodes are geographically distributed in a distributed system, there should be a communication medium connecting them. We identify faults pertaining to communication among nodes under the class of "distributed" faults, which it itself is under the class of "nature of 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 [39], 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 categorize faults with distributed nature in two groups:

- Distributed unavailability faults
- Distributed traffic faults

The reason why we do not call the above faults as "Network …" instead of "Distributed …" is that we would like to distinguish between the faults, for example, happening in nodes and network links. We discuss each of the above fault categories in the following sections.

3.2.1 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.

Basically a distributed unavailability fault 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 might have been 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 5.

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. The same thing might also be the case for a node availability fault, i.e., a particular node might fail to reply to the incoming requests while other nodes are functioning properly. Therefore, all different types and combinations 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 network link, the source or the destination nodes exhibits unavailability faults. A schematic notation of possible distributed unavailability faults in a simple distributed message scenario is shown in Figure 6.

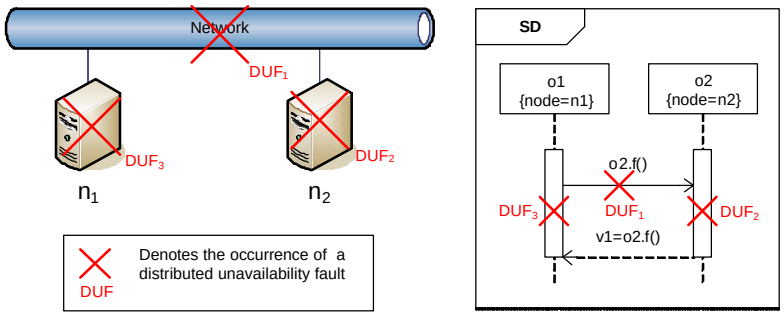


Figure 6-Occurrences of Distributed Unavailability Faults (DUF).

In the simple distributed message scenario of Figure 6, 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 6. 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.

3.2.2 Distributed Traffic Faults

A distributed traffic fault occurs when a system failure is because at least one of system components (either a network or a node) does not function correctly under heavy network traffic. The root cause for distributed traffic faults might be due to many network-related issues in the system, such as network congestion, buffer overflows, and processing delay in a software module. 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 [40]. Discussion on the basic reasons which cause distributed traffic faults is outside the scope of this paper. Among the main causes, we only consider two cases: either when large amounts of data (network packets) or high number of requests (messages) are sent over in a communication scenario between two nodes. For the location of a fault, just like the case of distributed unavailability fault, one can consider two possibilities where a distributed traffic fault might happen: a network or a node. Because of this distinction, the heuristic for a stress test strategy will be to enforce simultaneous traffic (either data or number of messages) to go through a network or towards/from a node. More details on this will be given in Section 8.5, where the stress testing strategy is presented.

As an example of a scenario when a distributed traffic fault might happen, consider the network schematic shown in Figure 7. 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" from our stress testing standpoint. 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 4. 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.

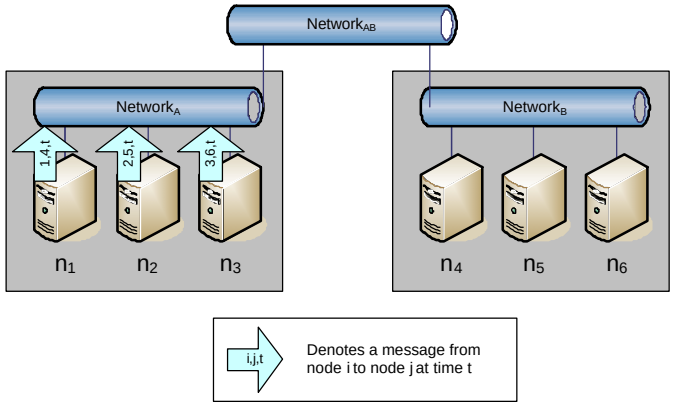


Figure 7-An example scenario showing how a distributed traffic fault might happen.

As discussed, in addition to amount of data transmitted over a network or from/to a node, we further assumed that high number of simultaneous messages might also be a potential cause of traffic faults. Considering the example scenario in Figure 7, 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 in a single time instant. Since there can be large number of concurrent processes on each node, so 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. Therefore, a different stress

test strategy will be to select a set of sequence diagrams and schedule them such that maximum numbers of messages go along a network, or from / to a node, on a single time instant.

3.3 Real-Time Faults

A real-time fault is said to have occurred if the root cause of a system failure is missing a real-time deadline. As discussed, 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 [41]. A missed hard deadline results in a system failure. A system with hard deadlines is called a hard real-time system. In hard real-time systems, *late* data is *bad* data. 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.

Several techniques have been proposed to maximize the chances of real-time faults. Briand, Labiche and Shousha's work in [16] proposes a methodology for the derivation of test cases that aims at maximizing the chances of critical (hard) deadline misses within a system. A deadline missing can be interpreted as an occurrence of a real-time fault. Zhang and Cheung [12] describe a procedure for automating stress test case generation for multimedia systems. The authors considered a multimedia system consisting of a group of servers and clients connected through a network as a SUT. The goal of their stress test case methodology was to schedule the multimedia objects such that the CPU usage of a node is maximized in a single time instant. This high load of CPU usage might lead to potential violations of timing constraints in a multimedia system. Therefore, the technique in [12] can also be considered as a way to maximize the chances of real-time faults.

3.4 Concurrency Faults

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.

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 [42]. 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 [42]. 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.

Several techniques have been proposed to find concurrent faults, such as [43-46] which aim at finding data-race related faults. For example, Ben-Asher et al. [44] 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 number of times they are accessed by different processes. This ordering is done using what the authors call *cross-run monitoring*. 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 [44] 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 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.

3.5 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.

3.6 Chain of Distribution Faults

As shown in the fundamental chain of dependability threats in Figure 4, 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 will not be 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. By this criterion, the data traffic fault is the first fault in the chain and the real-time one is the second in the above example.

3.7 Class of Faults Considered in this Work

As shown by dashed boxes in Figure 5, the classes of faults targeted by the stress testing technique of this work are data and request traffic faults with distributed nature, and the location of faults can be either networks or nodes.

Different variations of the proposed stress testing strategy will be given to accommodate different fault types. The system model is given in Chapter 5 and the stress testing methodology is proposed in Section 8.5.

Chapter 4
OVERVIEW OF THE STRESS TEST METHODOLOGY

Section 4.1 presents the overview of our model-based stress test process. The overview of metamodels used in the stress test methodology is discussed in Section 4.2.

4.1 Stress Test Process

The overview of our model-based stress test methodology is shown using an activity diagram in Figure 8. Note that only the steps in gray background are addressed by the current paper. 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).

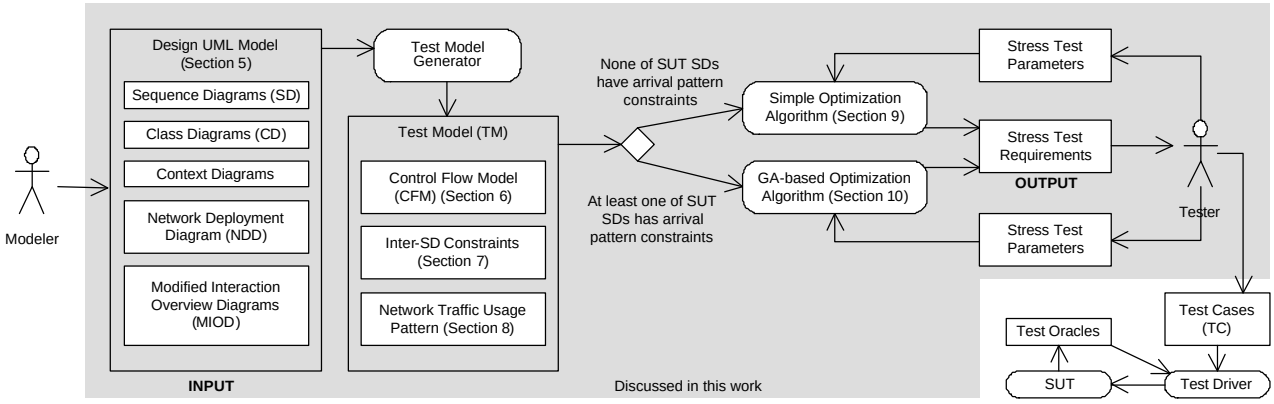


Figure 8- Overview of our model-based stress test methodology.

As we will discuss in Section 5.3, triggering SDs may not be allowed in any time instance. 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 Metamodels in the Stress Test Methodology

The overview of metamodels used in the stress test methodology is shown in Figure 9. The metamodels are grouped into three packages: network topology, input system UML model, and test model. Network

topology is a metamodel for network topology (distributed architecture) of a system under test (SUT). The metamodels packages are described next.

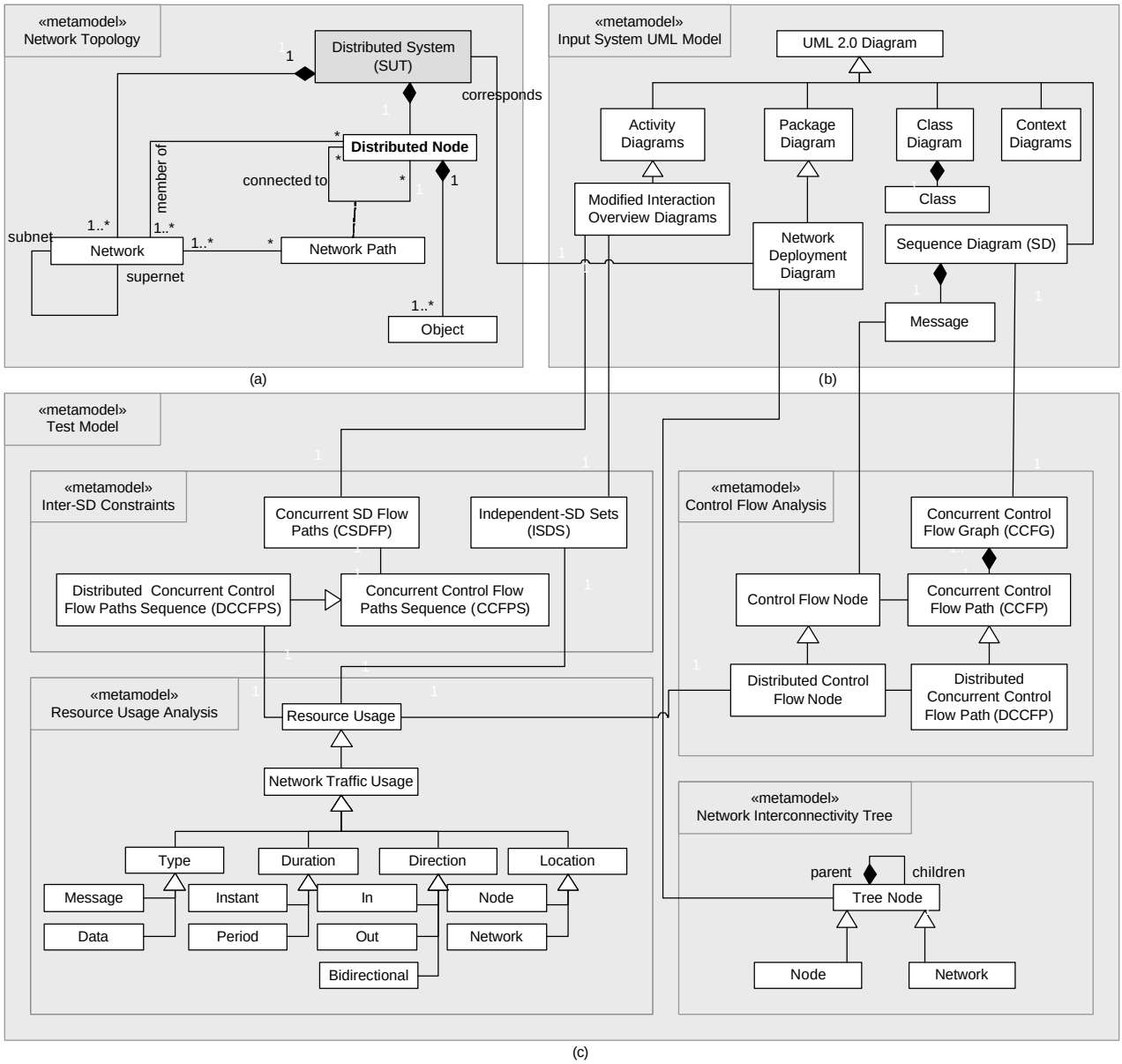


Figure 9- Metamodels in the Stress Test Methodology.

4.2.1 Network Topology Metamodel

The structure of the distributed architecture of a SUT as we need it to be described is shown in Figure 9-(a) 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 10.

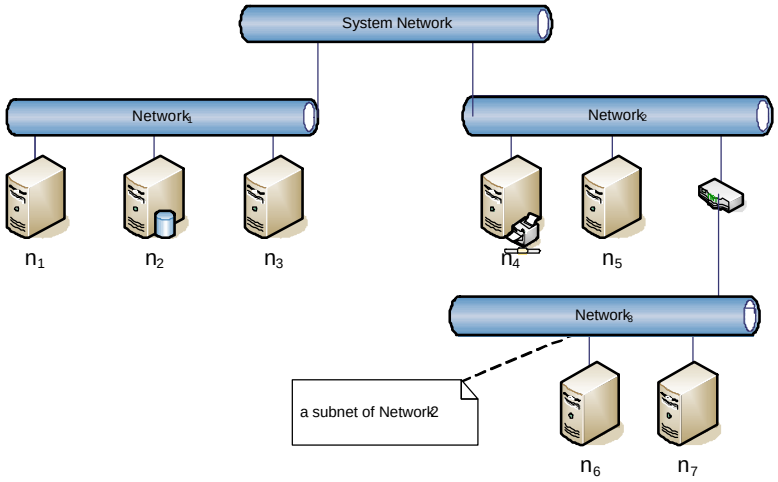


Figure 10-A network topology diagram.

In the example of Figure 10, 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 is a n*etwork path* defined 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 10 is <*Network*$_1$, *SystemNetwork*, *Network*$_2$, *Network*$_3$>. A function to derive the network path between two nodes will be described in Section 8.2. A UML package-based notation, referred to as Network Deployment Diagram (NDD), will be used to model a network topology.

4.2.2 Input System Metamodel

Sequence diagrams model the behavior of a SUT. Class diagrams will be used to estimate the data size of message in SDs. A Network Deployment Diagram (NDD) will model the network topology of a SUT. A context diagram [47] will be used to provide the number of multiple invocations of a SD. A Modified Interaction Overview Diagram (MIOD) will model the constraints between SDs. More details will be discussed in Chapter 5.

4.2.3 Test Metamodel

The Test Metamodel (TM) is shown in is shown in Figure 9-(b). It consists of four sub-models: control flow analysis model, network traffic usage model, network interconnectivity tree and inter-SD constraints, which are described in the next subsections.

4.2.3.1 Control Flow Analysis

In UML 2.0 [48], 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 [49], asynchronous messages and parallel combined fragments entail concurrency inside SDs.

In a SD of a DS, some messages are *local* (sent form 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 form 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.

4.2.3.2 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.

4.2.3.3 Network Interconnectivity Tree

A Network Interconnectivity Tree (NIT) is an equivalent data structure to the package-based representation of a network topology. A NIT is built from a NDD using the technique presented in Section 5.5.2.

4.2.3.4 Inter-SD Constraints

These constraints are derived from a given Modified Interaction Overview Diagram (MIOD), which models constraints among SDs of a SUT. The constraints are used as part of the test model to represent the sequential and conditional constraints among SDs. We propose four elements to analyze such constraints in our methodology, which will be described in detail in Chapter 7.

- Independent-SD Sets (ISDS)
- Concurrent SD Flow Paths (CSDFP)
- Concurrent Control Flow Paths Sequence (CCFPS)
- Distributed Concurrent Control Flow Paths Sequence (DCCFPS)

Chapter 5

INPUT SYSTEM MODEL

In this work, stress test input data is assumed to be UML 2.0 [8] 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 [9]. The new version, UML 2.0 [8], proposed by 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 rationales 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 5.2)
- A modified UML 2.0 diagram: modified interaction overview diagram (Section 5.3)
- A context diagram [47] (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. As UML 2.0 sequence diagrams are used as the main behavior model, an overview on SDs is presented in Section 5.7.

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 [8], seven UML diagrams can be used to specify the behavior of a system. As shown in Appendix A of [8], they are Activity, Sequence, Collaboration (or called Communication in Section 14 of [8]), 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 [50, 51]. The diagrams are particularly well-suited for object-oriented software, where they represent the flow of control during object interactions [52]. A SD shows a set of interacting objects and the sequence of messages exchanged among them. The diagram may also contain additional information about the flow of control during the interaction, such as if-then conditions ("*if c send message m*") and iteration ("*send message m multiple times*") or state-dependent behavior [50]. SDs have been the basis for several approaches for testing of object-oriented software [8, 19, 51, 53-56]. Some of existing approaches test the interactions among collaborating objects using SDs. SDs are used to determine the interactions that must be exercised. For

example, it may be required to cover all relationships of the form *"object X sends message m to object Y"*. Sequences of messages for example, all possible beginning-to-end message sequences in the diagram may also be considered for coverage. We choose SDs as the source of information for dynamic behavior of a SUT.

According to the new features of SDs in UML 2.0, Section 14 of [8], SDs can call each other through a mechanism which is called *InteractionOccurrence* in the specification. Due to the conditional constructs in SDs, there can be multiple flows of control in a SD. Therefore, network-traffic stress conditions might happen in only subsets of the possible control flows of a SD. Thus, to derive network-aware stress test requirements, we will need to analyze control flow in SDs. We have presented a control flow analysis technique based on SDs in [2], 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. An example is shown in Figure 11.

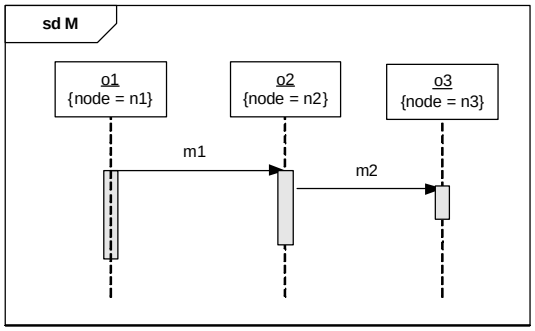


Figure 11-Modeling the deployment node of an object using *node* tagged value.

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 in runtime or real-time faults will occur. It was also discussed in Chapter 3 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, we assume that the timing information of all messages in SDs is given. By timing information of a message, we basically mean the start and end times of a message. As discussed in Section 3.3, out of all messages in a typical DRTS, some might have hard and some have soft real-time constraints. There might be also messages that do not possess any real-time constraints. However, in order to give a scheduled stress test requirement that will cause stress on network traffic on a predicted time instant (or period), we require that all messages have precise or statistical timing information.

The start and end times of messages with hard deadlines can be modeled in the UML model of a SUT using the UML-SPT profile notations [10]. As discussed in Section 3.3, messages with soft deadlines can be stochastically characterized. Another common definition for such messages is that they are constrained only by average time constraints. For the case of messages with no time constraints, runtime monitoring techniques (such as [57]) can be utilized to get a statistical view of the time length of such messages in 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 the expected values of start and end times can 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. In this work, we do not go into details on the issue of timing information. We assume that a time measurement technique has been utilized for the messages in the SUT and such information is already available.

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 [9].

The stress testing technique in this paper 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 [2], 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.

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 and also certain conditions may have to be satisfied before a particular SD can be executed.

Different types of such SD constraints might exist in different systems. We identify three of those types of constraints.

- *Sequential* constraints [54]: Constraints which 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: 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: These constraints relate to timing of SDs. The time instant when a SD can start running might be constrained in a system. Considering each SD alone, it might only be allowed to be executed in some particular time instants. For example in a replicated distributed database server system, where the data on the main server should be 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.

Our approach in considering the above set of constraints when generating stress test requirements is as the following. We propose a test requirement generation technique, as an optimization problem, in Chapter 9 which takes into account the first two types of constraints (sequential and conditional) between SDs. We refer to the technique as *Time-Shifting Stress Test Technique.* A more complex optimization algorithm, based on Genetic Algorithms, will be presented in Chapter 10 which will consider *all* three types of constraints (sequential and conditional and arrival-pattern), and will be referred to as *Genetic Algorithm-based Stress Test Technique*.

The reasons why we intend to propose two optimization algorithms are:

- In systems where only sequential and conditional constraints are to be considered, the technique of Chapter 9 can be used, which is expected to generate more accurate results, i.e.,

test requirements, than the technique of Chapter 10. This is because the later technique is based on genetic algorithms, which do not always find the most optimum results.

– As we will discuss, the technique of Chapter 9 is less complex, easier to comprehend, and in fact a simpler type of the one in Chapter 10.

The approach in which the different SD constraint types are considered by the two optimization algorithms in this work is visually depicted in Figure 12.

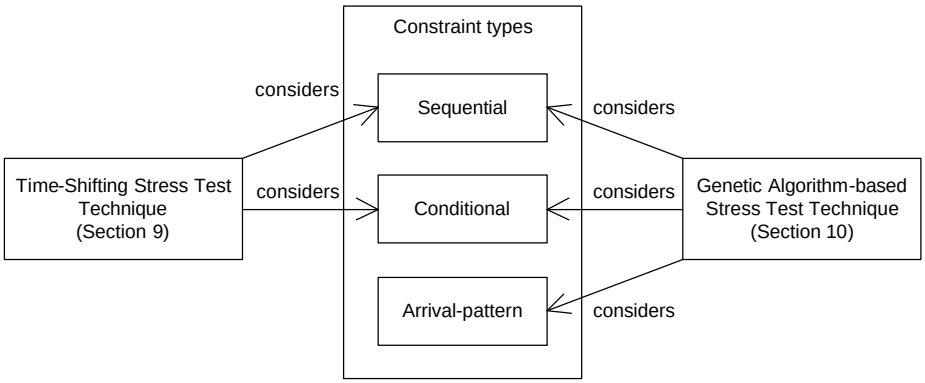


Figure 12-The approach in which the different SD constraint types are considered by the two optimization algorithms in this work.

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, thus allowing any test generation technique to use them to derive test cases that comply with such constraints. The arrival-pattern constraints apply to each SD and they can be modeled using UML-SPT profile, as explained in Section 2.4.

Sequential and conditional constraints are between SDs. Therefore, we refer to them as *inter-SD* constraints. In the following, we first discuss the existing techniques and representations to model and formalize the inter-SD constraints and we will then choose the one which suits best our context. We also propose a method to derive all possible (allowed) SD sequences in Chapter 7.

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. Refer to the SD in Figure 3 for an example.

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 [58] proposed the notion of *life-cycle model* which bears some similarity in concepts to what we call SD sequential constraints now. Even recently, there have been works [59, 60] on Fusion, where Fusion calculus together with a notation called *Synchronized Hyperedge Replacement (SHR)* are compared and challenges of applying Fusion calculus to distributed systems are discussed.

Use-Case Maps (UCMs) [61] are also one of the notation which started to evolve before UML, although it bears some similarities to UML activity diagrams, since its later design phases coincided with the UML's. 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 [61].

Allen's interval temporal logic [62] 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 [12] to model the temporal constraints among objects in multimedia presentations. Having modeled these temporal constraints, they presented a technique to stress test the CPU load of a multimedia system using linear programming optimization technique.

Petri-nets [17] can also be used to model sequential constraints among SDs. For example, Zhang and Cheung [12] 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.

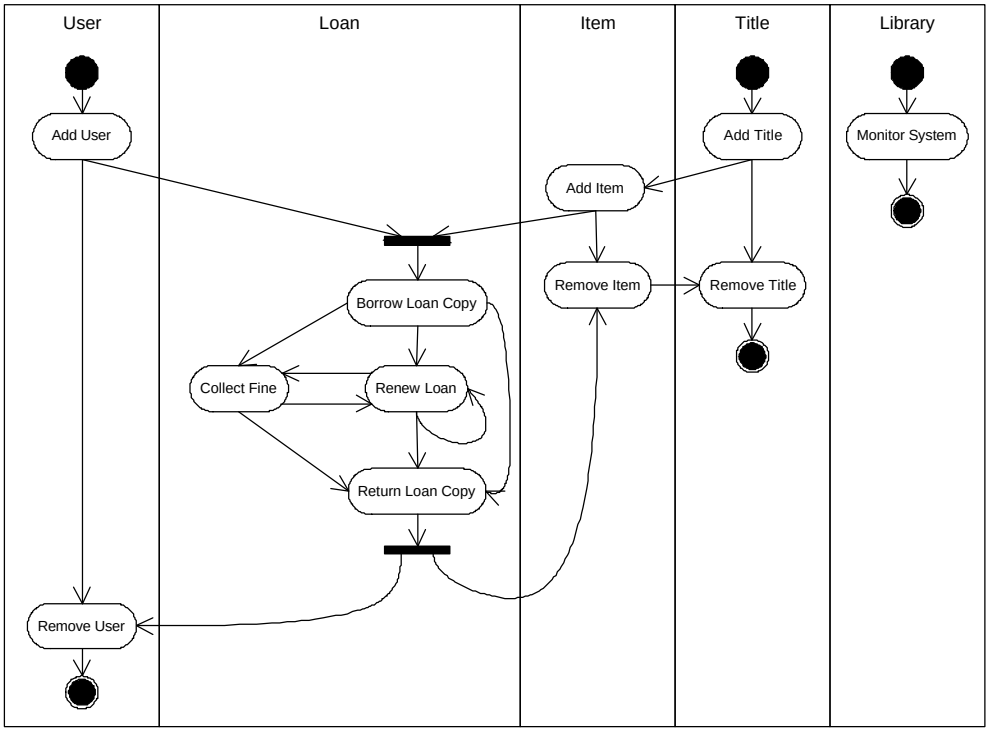


Figure 13- Use Case Sequential Constraints for the Librarian actor (adopted from [54]).

In the context of UML and SDs, there have also been techniques and representations to model and formalize constraints among SD, [54] 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 [54] report that when planning test cases for UCs, all possible execution *sequences* for UCs have to identified. The authors present principles underlying the representation and generation of possible UC test sequences. In order to do that, they use a model to represent the sequential dependencies of UCs. Such sequential dependencies are represented by the means of an activity diagram, in which the vertices are UCs and the 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. As an example, the authors evaluated the technique on a library system. Based on [54], the UC sequential constraints for the Librarian actor (in a library system) is shown in Figure 13 (formal parameters of the UCs are not shown for clarity). The authors also discuss the dependencies in terms of actual parameter values between the use cases in a path. For instance, in a path like *AddTitle.AddItem.RemoveItem.RemoveTitle* in Figure 13, parameter *isbn* for UC *AddItem* must be identical to parameter *isbn* in *AddTitle*. The authors of [54] also propose an algorithm to derive all possible sequences of UCs to test.

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 are attached as UML notes to each use case, and are expressed with logical expressions. The sequential constraints among SDs can then be deduced from the set of contracts.

OMG introduces a new UML diagram in the new 2.0 version: *Interaction Overview Diagram* (IOD), Section 14.4 of [8]. IODs "define interactions through a variant of activity diagrams in a way that promotes overview of the control flow" [8]. IODs are specializations of Activity Diagrams (AD) that represent interactions. IODs focus on the overview of the flow of control where the nodes are Interactions or InteractionOccurrences (refer to Interactions as defined by UML 2.0 [8]). The lifelines and the messages (of each interaction diagram) do not usually appear at this overview level.

5.3.2 Our Choice: IODs

We surveyed some of the existing techniques and representations to model constraints among SD in the previous section. As mentioned in Section 2.1, one fundamental constraint is that the entire system modeling should be performed using UML. Therefore, we have to find ways to derive the valid orders of SDs' execution in a system using its design UML model. IODs are the most suitable means in UML 2.0 to model the sequential and conditional constraints among SDs.

In this section, we present a brief overview on IODs from the UML 2.0 specification [8]. The metamodel of IODs is not directly given in the UML 2.0 specification. However it is mentioned that IODs are specialization of activity diagrams that represent interactions [8]. IODs differ from ADs in some respects.

1. In place of AD object nodes, IODs can only have either (inline) Interactions or Interaction Occurrences.
2. Alternative CombinedFragments are represented by a decision node and a corresponding merge node.
3. Parallel CombinedFragments are represented by a fork node and a corresponding join node.
4. Loop Combined Fragments are represented by simple cycles.
5. Branching and joining of branches must in IODs be properly nested. This is more restrictive than in ADs.
6. IODs are framed by the same kind of frame that encloses other forms of interaction diagrams.

All of the above constraints are adequate in our context[1]. It should be mentioned that we will require having only one IOD for a system, since we only need to know if any two SDs in a system have a dependency relationship or not. IODs are similar to activity diagrams, proposed by Briand and Labiche [54] for use case sequential constraints. However, the important additions in IODs are AD conditionals (Section 12.3.11 of [8]) and loops (Section 12.3.28 of [8]).

In the following sections, we rephrase the definition of dependent/independent SDs and an ISDS in the context of a MIOD and then discuss a method for the derivation of ISDSs from a MIOD.

As an example, the IOD of an ATM system is depicted in Figure 14. The IOD is composed of interaction occurrences which refer to the corresponding 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.

[1] As a reminder, what we mean by our context is the SD constraints to be modeled.

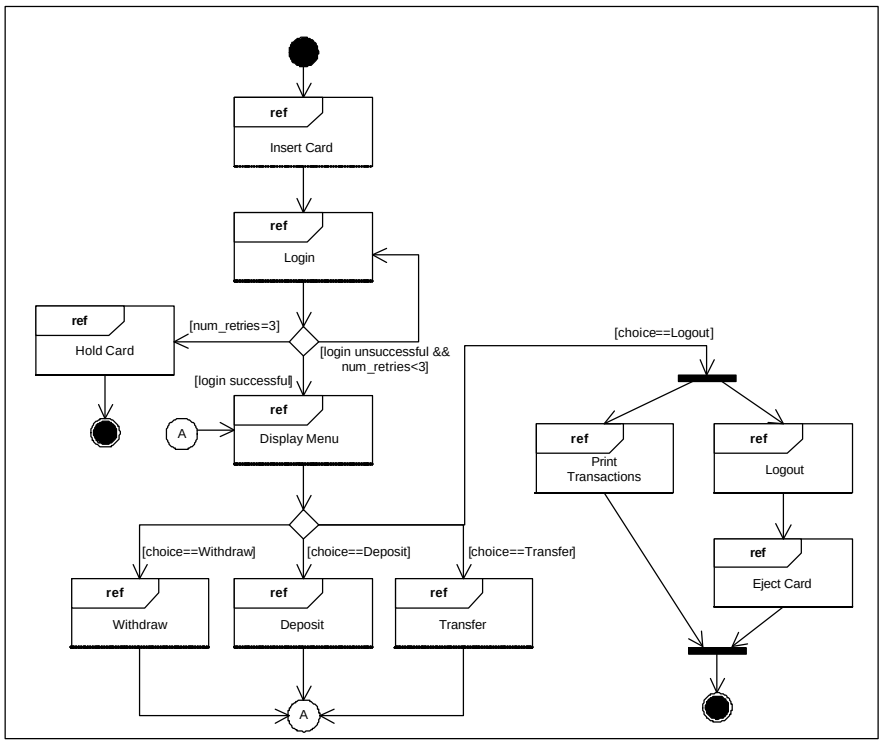


Figure 14-Interaction Overview Diagram (IOD) of a simplified ATM system.

5.3.3 Modified Interaction Overview Diagrams

To model which actor or sub-system invokes a particular SD, we slightly modify IODs to include activity partitions and refer to the modified IOD metamodel as *Modified Interaction Overview Diagrams (MIOD)*. Activity partitions are modeling features which include AD swimlanes. In a MIOD, SDs are grouped into swimlanes, according to actors triggering each SD. For example, the MIOD of the IOD in Figure 14 is shown in Figure 15.

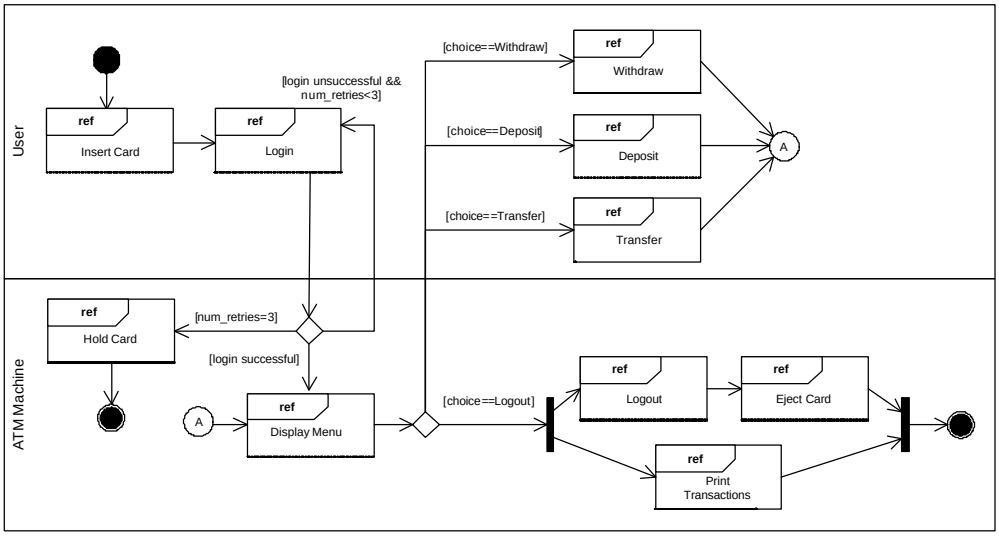


Figure 15- Modified Interaction Overview Diagram (MIOD) of a simplified ATM system.

The differences of our MIOD modeling notation with the use-case sequential-constraints modeling done in [54] are: (1) the MIOD is a notation for system-wide sequential constraint modeling for SDs, while the

notation in [54] was per actor. (2) the MIOD takes into account the conditional constraints (defined in Section 5.3) among SDs, while the work in [54] 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. We will then propose a way to derive the set of *independent SDs* in a SUT which will be used by our stress test methodology.

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 test technique.

To model concurrent invocations of SDs, we using the information provided in a Context Diagram [47]. The concept of context diagrams was proposed in the COMET (Concurrent Object Modeling and Architectural Design Method) framework [47]. For example, a context diagram is shown in Figure 16-(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* in this system, Figure 16-(b). Therefore, at one time instance, up to three concurrent instances of the SD can be executed.

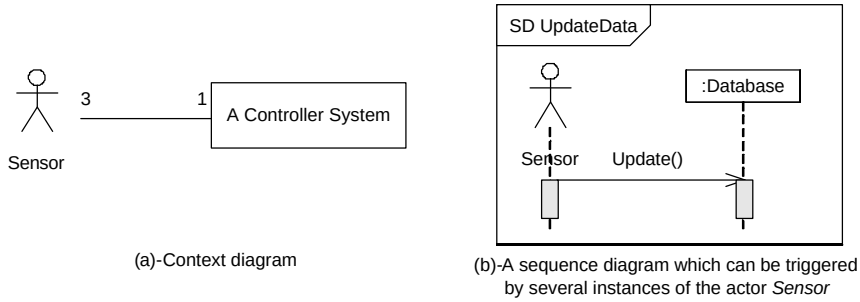


Figure 16-A controller system made of several sensors.

Alternatively, the number of concurrent instances of a SD may be modeled inside 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 17-(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.

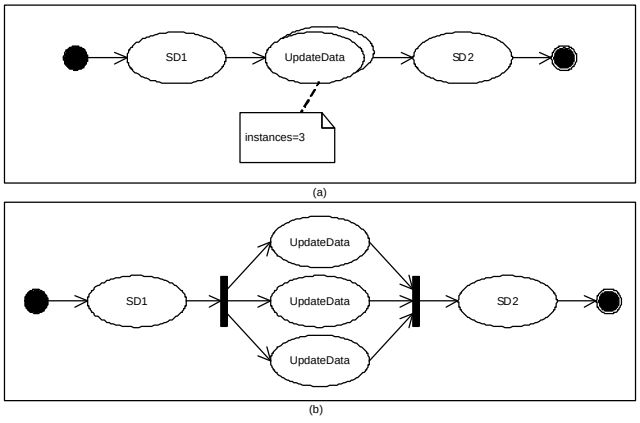


Figure 17-(a): Modeling concurrent instances of SDs inside MIOD. (b): Equivalent in meaning to (a).

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 CCD is used to model such information, our technique needs to look and match the SDs actors with the actors in the CCD of a system to extract the information.

It is good to note that the MIOD in Figure 17-(a) is equivalent in meaning to Figure 17-(b). In other words, 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*.

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 and the system topology.

In UML 2.0 [8], there has been a significant change in support for modeling application architecture, nodes and communication paths, compared to UML 1.x [9]. Modelers can model complicated deployment scenarios such as nested and generalized nodes. Network topology modeling has also enhanced. *CommunicationPath*, Section 10.3.2 of [8], generalized from standard UML's "Association" is a new concept for modeling the communication path between distributed nodes of a system. As defined by the specifications [8]: "A communication path is an association between two nodes, through which nodes are able to exchange signals and messages." For example, Figure 18 represents a simple network deployment of an online shopping system where client workstations, servers and printers are collaborating together.

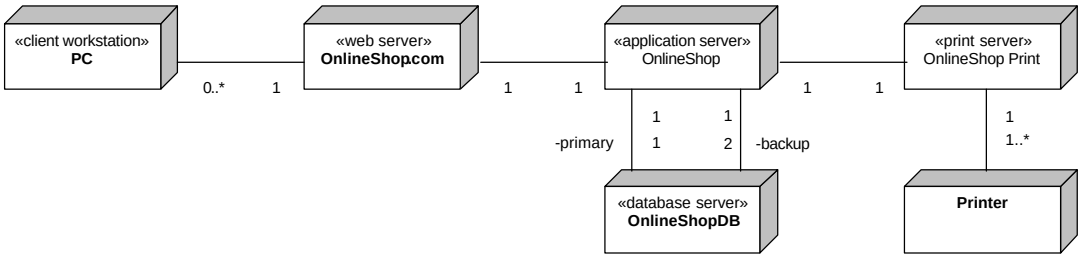


Figure 18-A simple network deployment for an online shopping service.

However, to the knowledge of the authors, modeling a hierarchical set of networks and their inter-connectivity is not directly stated in the UML 2.0 specification [8]. Suppose we want to model a system, composed of several networks with the interconnectivity scheme as shown in Figure 10.

5.5.1 Using the Notation of Package Diagrams

Interestingly, we can think of the system network as a package structure where the whole system network is the root (high level) package and other networks and nodes are the sub-packages in a hierarchical manner. Having made this assumption, we can use the notation of packages and sub-packages (or called *nested packages* in the specification) of the UML 2.0, Section 7.13 of [8], to model network interconnectivity. Our suggested modeling approach can be proposed by modeling the example topology of Figure 10 in Figure 19 using the notation of packages. In Figure 19, packages represent networks of the system and the solid lines between nodes and packages mean the connection of a node to a network. Nested relationships among the packages symbolize nested networks. For example, as $Network_3$ is a subnet of $Network_2$ in Figure 10, therefore the package representation of $Network_3$ is inside $Network_2$ in Figure 19.

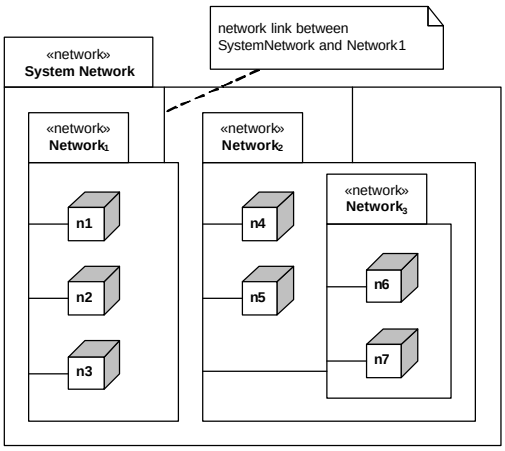


Figure 19-Using UML packages to model network interconnectivity of Figure 10.

In order to model and quantify bandwidth (capacity) values of each network, we can define a *bandwidth* tagged value for the «network» stereotype in the above package 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 and also 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 as well. Let us also make the assumption 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 20, which depicts the network interconnectivity of a nodes and networks in a distributed system running in a typical university network. The system is deployed in three buildings (*Building*$_i$), where each building may have its own subnets in different floors. Each floor also has its own network and consists of one or more nodes. Each node (workstation) is represented as *w(building_number). (floor_number). (node_number)*, such as *w3.1.2*.

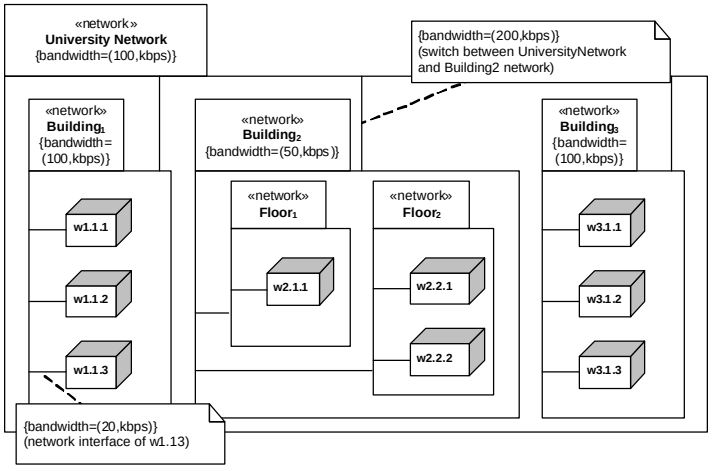


Figure 20-Modeling network interconnectivity of a University Network.

Therefore, we assume that the network interconnectivity model of the SUT is done using the above notation. As a more efficient representation which will be used by our testing technique, we propose a tree data structure for representing the interconnectivity, which will be an internal notation for our technique, i.e., the modelers and testers do not need to use this in their models. We refer to the new notation as *Network Interconnectivity Tree (NIT)*, which is described next.

5.5.2 Network Interconnectivity Tree

A Network Interconnectivity Tree (NIT) is an equivalent data structure to the package-based representation of a network interconnectivity mention above. The root of the tree is always the whole system network while system networks and nodes are its children. In a NIT, networks and nodes are shown as rectangles and circles, respectively. For example, the NIT of the network interconnectivity model of the Figure 10 (or equivalently Figure 19) is shown in Figure 21. The rationale of having NIT 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 path 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 21, the test strategy will only look for nodes under $Network_2$ in the NIT tree and will generate the test cases by considering only those nodes.

Generating the NIT from a network topology diagram 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 NIT data structure's elements (rectangles for networks and circles for nodes) and edges (representing switch / router between networks).

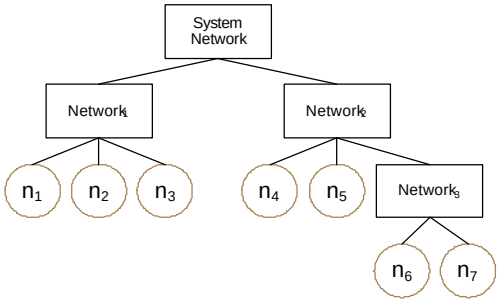


Figure 21-Network Interconnectivity Tree (NIT) of the topology in Figure 10.

5.6 Modeling Real-Time Constraints

As discussed in Section 3.3, 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) [10] proposes comprehensive modeling constructs to model timing information. Although UML-SPT briefly mentions hard RT constraints (Section 2.2.3 of [10]), it doesn't propose any stereotype or tagged value to distinguish between hard and soft RT constraints in UML models.

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 in a 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. On a similar note, we consider a tagged value referred to as *criticality* for HRT constraints. Criticality is defined as the degree to which the consequences of missing a hard deadline are unacceptable. As we define, the closer to one the criticality of a HRT constraint, the more severe will be the consequences of missing it. For example, if missing a HRT constraint

may cause life-threatening situations, it would be better to set criticality=1. Conversely, if for example the cost of missing 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. Note that, with the above definitions, there is similarity in the concepts of HRT constraints with low criticality and SRT constraints. *HRTaction* and *SRTaction* stereotypes are presented in Table 1 and Table 2, which are similar to the representation used in the UML-SPT [10].

Stereotype	Base Class	Tags
SRTaction	Message	RTduration
	MessageSequence	RTmissProb
	Action	
	ActionSequence	

Table 1-A stereotype to model SRT constraints.

Stereotype	Base Class	Tags
HRTaction	Message	RTduration
	MessageSequence	RTcriticality
	Action	
	ActionSequence	

Table 2-A stereotype to model HRT constraints.

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 [10]. The SRT» and «HRT» stereotypes have two associated tagged values each, which are defined in Table 3.

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. *RTduration* tagged values is an instance of the *RTtimeValue* data type (Section 4.2.2.4 of [10]). *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.

Furthermore, we divide the RT constraints into two levels: *SD-level* and *MIOD-level*. SD-level constraints are applied to *Message* and *MessageSequence*, while MIOD-level constraints are applied to *Action*, and *ActionSequence* (since MIOD is a subtype of activity diagrams). As this idea has been used in [10], though unnamed, these two levels provide enough flexibility in modeling RT constraints by annotations on either messages (in SD level) or on SDs (in MIOD level). Examples of a SD-level SRT constraint and a MIOD-level HRT constraint is shown in Figure 22.

The tagged-values of SRT and HRT constraints can help our stress testing technique to order the constraints in terms of importance and test order. Such a technique will be proposed in Section 9.14.

5.7 An Overview on UML 2.0 Sequence Diagrams

The UML 2.0 [8] syntax of SDs is used in this work. Comparing to UML 1.x [6, 7], UML 2.0 have proposed a set of new features to SDs. In the following we provide a brief definition of the new features and then we state the features that are supported in this work.

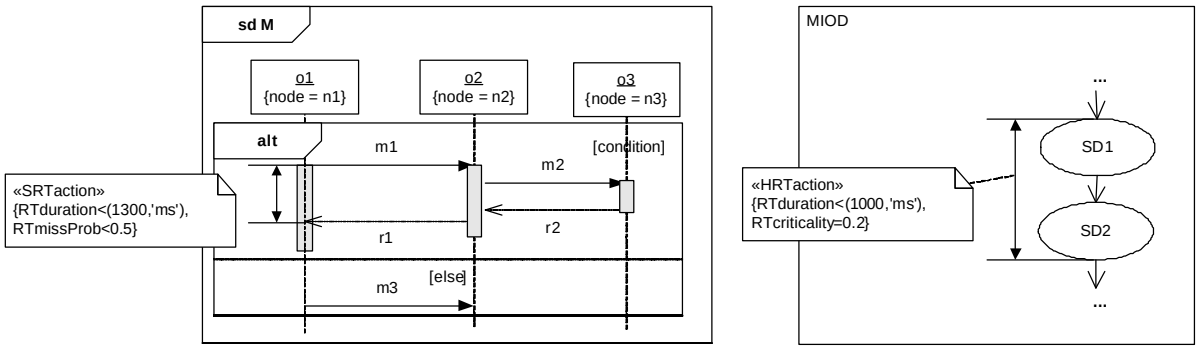


Figure 22-Examples of SD- and MIOD-level SRT and HRT constraints.

A glimpse of SD new features is shown in Table 4. Some of the new features are illustrated with an example in Figure 24. The SD metamodel showing the class diagram of SD features is shown in Figure 23.

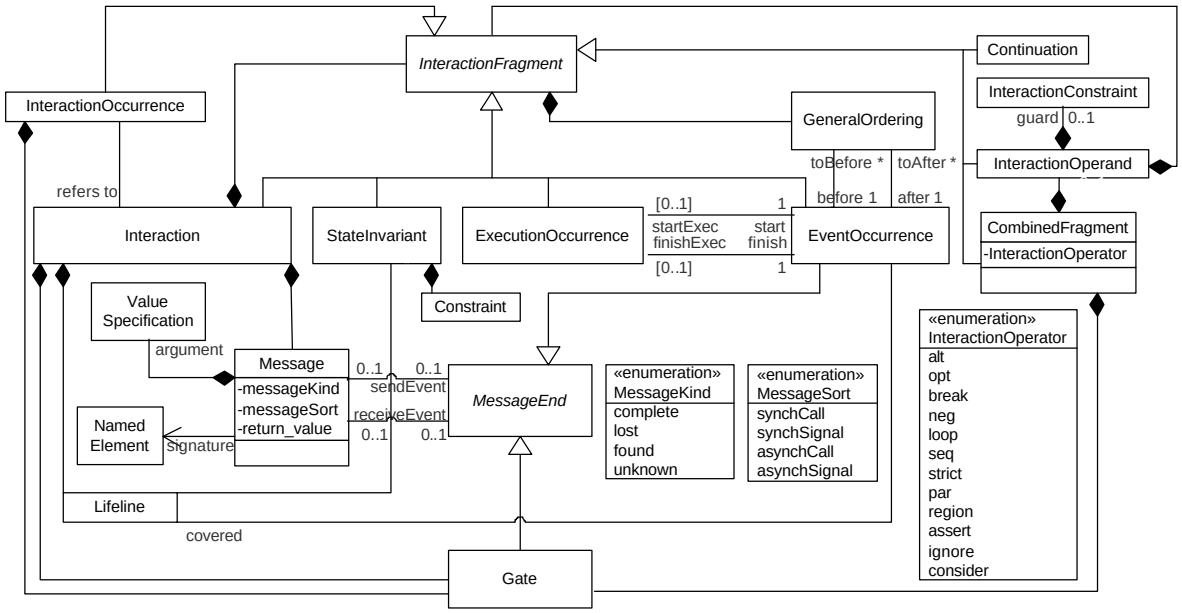


Figure 23-UML 2.0 Sequence Diagram Metamodel.

- *Interaction*: An interaction is a sequence of messages passed between objects to accomplish a particular task. The rational behind defining interactions is to reuse them in other contexts as *InteractionOccurrences.* For example, SD *seqname1* in Figure 24 is an interaction.
- *InteractionOccurrence*: An InteractionOccurrence is a symbol that refers to an interaction that is used within another interaction or context. *seqname1* and *seqname2* are two interaction occurrences which refer to SDs *seqname1* and *seqname2*.
- *EventOccurrence*: EventOccurrences represents moments in time to which actions are associated. An EventOccurrence is the basic semantic unit of Interactions. The sequences of EventOccurrences are the meanings of interactions. EventOccurrences are ordered along a Lifeline. A message has two types of EventOccurrences: *sendEvent* and *receiveEvent*. The SendEvent is at the base of the message arrow where the message departs from the lifeline of the sending object, while ReceiveEvent is at the point of the message arrow where the arrow hits the lifeline of the receiving object. The ReceiveEvent of message *m3* is pointed in Figure 24.

Concept	New/Existing
Object Lifeline	Existing
Stimulus (Message)	Existing (but just called Message in the new version)
Time observation and constraint	Existing
Activation	Existing
Interaction	New
InteractionOccurrence	New
EventOccurrence	New
CombinedFragment	New
InteractionOperator	New
InteractionOperand	New
Duration observation and constraint	New

Table 4-New and existing features of the UML 2.0 SDs, compared to UML 1.x (taken from [64]).

- *CombinedFragment*: A CombinedFragment consists of one or more InteractionOperands. A CombinedFragment has an InteractionOperator that defines the number of allowed InteractionOperands and also how the messages in the CombinedFragment will be treated. As an example, a CombinedFragment with *alt* InteractionOperator is shown in Figure 24.
- *InteractionOperand*: An InteractionOperand describes a grouping mechanism inside combined fragments. Interaction operands are features similar to Interactions, except the fact that they are part of a CombinedFragment. As example, the *alt* CombinedFragment in Figure 24 has two interaction operands.
- *InteractionOperator*: An InteractionOperator defines how to use the interaction operands within the context of the combined fragment. The following interaction operators are defined: alternatives (*alt*), option (*opt*), break (*break*), parallel (*par*), weak sequence (*seq*), strict sequence (*strict*), negative (*neg*), critical region (*region*), ignore / consider (*ignore/consider*), assertion (*assert*), and loop (*loop*). Description of each of the interaction operators can be found in [65]. We briefly describe next the ones we intend to use in our MBCFA technique:
    - Alternatives (*alt*): Provides alternatives, only one of which will be taken. The InteractionOperands are evaluated on the basis of guards. An *else* guard is provided that evaluates to TRUE if and only if all guards of the other InteractionOperands evaluate to FALSE.
    - Option (*opt*): Defines an optional interactions segment. The model for an *opt* combined fragment looks like an *alt* that offers only one interaction.
    - Break (*break*): Is a shorthand for an Alternative operator where one operand is given and the other assumed to be the rest of the enclosing InteractionFragment. In the course of executing an interaction, if the guard of the break is satisfied, then the containing interaction abandons its normal execution and instead performs the clause specified by the *break* fragment.
    - Parallel (*par*): Supports parallel execution of a set of InteractionOperands.
    - Loop (*loop*): Indicates that the interaction operand will be executed repeatedly and also includes a mechanism to stop the iteration.
- *Duration observation and constraint*: UML 2.0 provides two types of constraints on the performance characteristics of interactions: duration and time. Furthermore, these features are enhanced by the UML-SPT profile [66].

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.

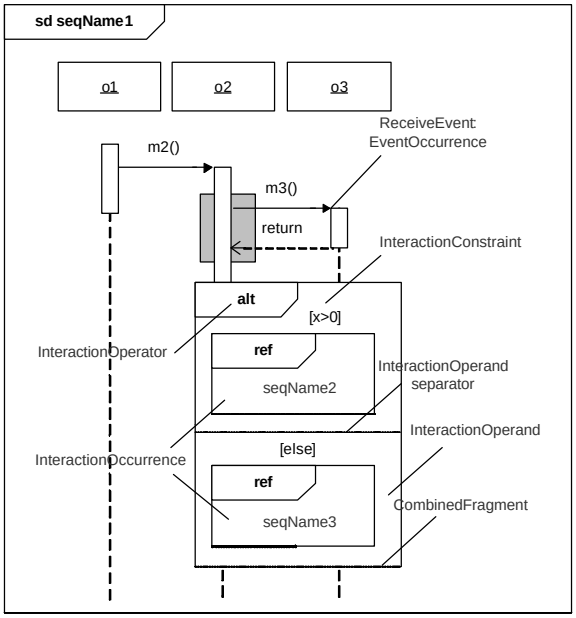


Figure 24-An example illustrating the new features of the UML 2.0 SDs.

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 [65]) as:

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

The notational representations of reply and asynchronous messages in UML 2.0 have changed compared to UML 1.x, as shown in Figure 25.

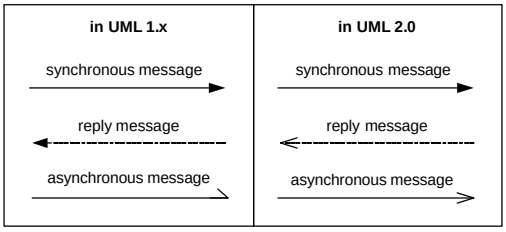


Figure 25-Notations for synchronous/asynchronous messages and replies in UML 1.x and 2.0.

As another property for messages is the so-called message *kind*. UML 2.0 defines the following message kinds:

- *complete*: sendEvent and receiveEvent are present
- *lost*: sendEvent present and receiveEvent absent
- *found*: sendEvent absent and receiveEvent present
- *unknown*: sendEvent and receiveEvent absent (should not appear)

The difference between message sort and kind properties is that message sort specifies the synchrony of a message, while message kind categorizes a message by on its message ends.

The SD metamodel is not shown in one place in UML 2.0 specification [65], rather divided in several small metamodel diagrams, since it is composed of many elements. By omitting unnecessary details, we show the complete metamodel generated from the specification in Figure 23. For space limitations, only some of the role names and multiplicities are shown in this figure.

Chapter 6

CONTROL FLOW ANALYSIS OF SEQUENCE DIAGRAMS

We presented a Control Flow Analysis (CFA) technique in [2] 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 23), 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 [2].

We review here some of the discussions in [2] which are used by the current work. More details on our control flow analysis technique can be found in [2].

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

We proposed CCFGs to analyze the concurrent control flow of SDs. A CCFG will be generated for each SD. In cases where a SD calls (refers to) another SD, there will be control flow edges connecting their corresponding CCFGs to form an *Inter-SD* CCFG. Inter-SD CCFG here is similar to the concept of inter-procedural CFG [67].

As discussed in Section 4.3.2 of [2], we extended CCFG from the UML *IntermediateActivities* activity package. As an example, considering the SD in Figure 26, the corresponding CCFG is shown in Figure 27.The procedure to map the SD to the CCFG is discussed in detail in [2].

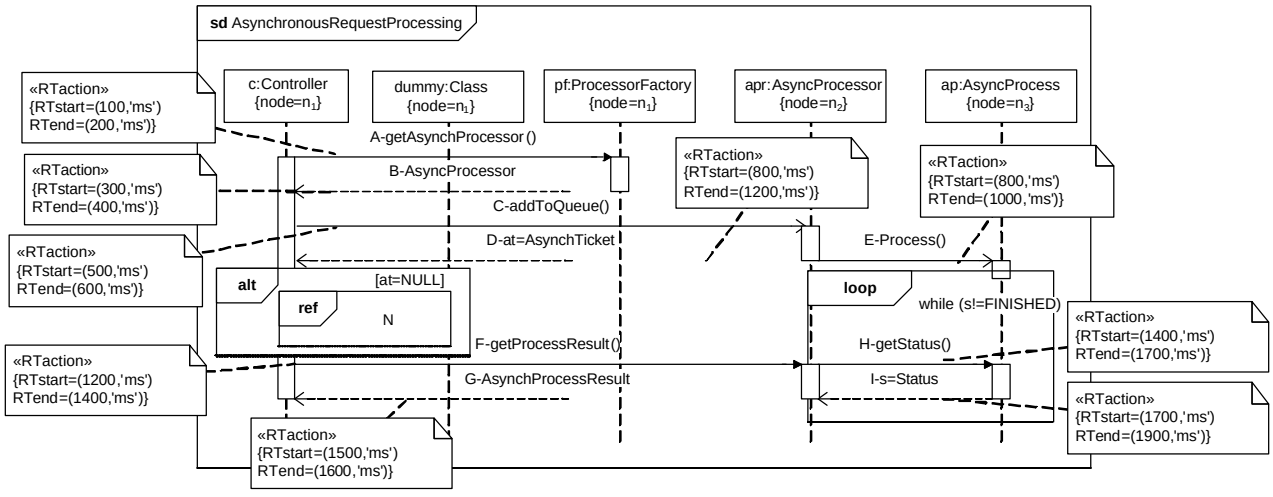


Figure 26-A SD with asynchronous messages.

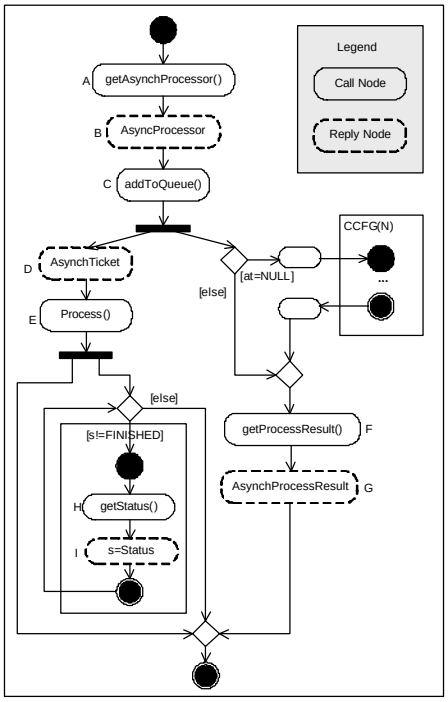


Figure 27-CCFG of the SD in Figure 26.

6.2 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 [2]. We presented a grammar in [2] to derive all different CCFPs of a CCFG.

For example, by using such grammar, some of the CCFPs of the CCFG in Figure 27 can be derived as shown in Figure 28. The symbol r will be used in the rest of this article to refer to CCFPs.

$$r_1 = ABC\left(DE\left(\right)FGHI\right) \qquad r_2 = ABC\left(DE\left(JK\right)FGHI\right)$$

$$r_3 = ABC\left(DE\left((JK)^2\right)FGHI\right) \qquad r_4 = ABC\left(DE\left((JK)^3\right)FGHI\right)$$

Figure 28-CCFPs of the CCFG in Figure 27.

Four CCFPs for the CCFG in Figure 27 are due to the decision node (corresponding to a loop) in the CCFG. According to the grammar of CCFPs (Equation 1 of [2]), a loop can either be bypassed (e) – if possible, taken only once, a representative or average number, and a maximum number of times. These possibilities have derived the four CCFPs: r_1, r_2, r_3 and r_4. The loop is bypassed in r_1, taken once in r_2, repeated twice in r_3, and a maximum number (m) of times in r_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.

6.3 Incorporating Distribution and Timing Information in CCFPs

The discussions in [2] 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 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 [10].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.

6.4 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 [48], each message annotated with timing information (using the UML-SPT profile [48]) 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[1] of each message in the UML metamodel can be used to distinguish signal and call messages, the metamodel does not provide a 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 [49].

[1] The *messageSort* attribute of a message specifies the type of communication reflected by the message [48], and can be any of these values: *synchCall* (synchronous call), *synchSignal* (synchronous signal), *asynchCall*, or *asynchSignal*

6.5 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 6.3 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 *r* by removing all local messages and keeping the distributed ones. As an example, let us assume the CCFPs given in Figure 28. In order to derive their DCCFPs, we should first judge each messages as local or distributed. According to the corresponding SD (Figure 26), all the messages except the messages *A* and *B* are distributed. Therefore, in the CCFG of Figure 27, only control nodes *A* and *B* are local, and the rest are distributed. Hence, the DCCFPs corresponding to the CCFPs given in Figure 28 are shown in Figure 29.

$$DCCFP(r_1) = C\left(DE\left(\begin{array}{c}\\ FGHI\end{array}\right)\right) \qquad DCCFP(r_2) = C\left(DE\left(\begin{array}{c}JK\\ FGHI\end{array}\right)\right)$$

$$DCCFP(r_3) = C\left(DE\left(\begin{array}{c}(JK)^2\\ FGHI\end{array}\right)\right) \qquad DCCFP(r_4) = C\left(DE\left(\begin{array}{c}(JK)^3\\ FGHI\end{array}\right)\right)$$

Figure 29- DCCFPs of the CCFPs in Figure 28.

6.6 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 [8]). 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 26 and $DCCFP(r_2)$ in Figure 29. Timed inter-node representation of $DCCFP(r_2)$ in shown in Figure 30, 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.

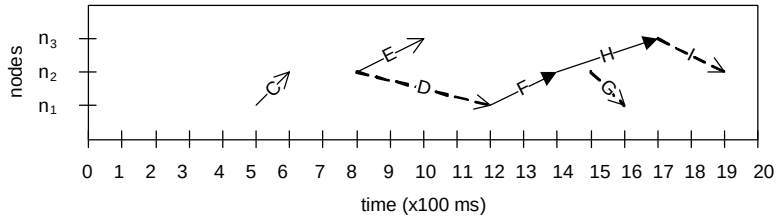


Figure 30-Timed inter-node representation of $DCCFP(r_2)$ in Figure 29.

Suppose the NIT of this system is as the one shown in Figure 31. The inter-network representation of the *DCCFP*(r_2) can be derived using the node information in SD, the inter-node representation (Figure 30) and the system NIT.

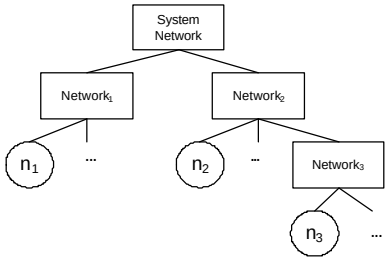


Figure 31-A simple system NIT.

The inter-network representation of the *DCCFP*(r_2) is drawn in Figure 32. 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 NIT, but are in the network path 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 NIT.

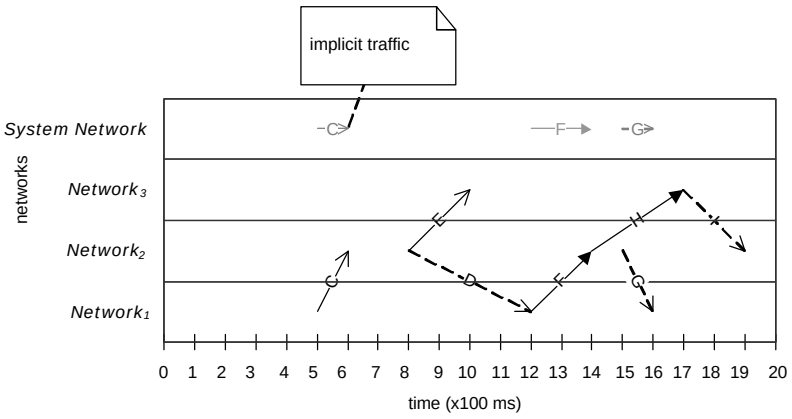


Figure 32-Timed inter-network representation of a DCCFP.

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 instance.

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 33. This MIOD is the completed version[1] of the activity diagram shown in [54]. 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 [54] and some typical business logic assumptions of the library system.

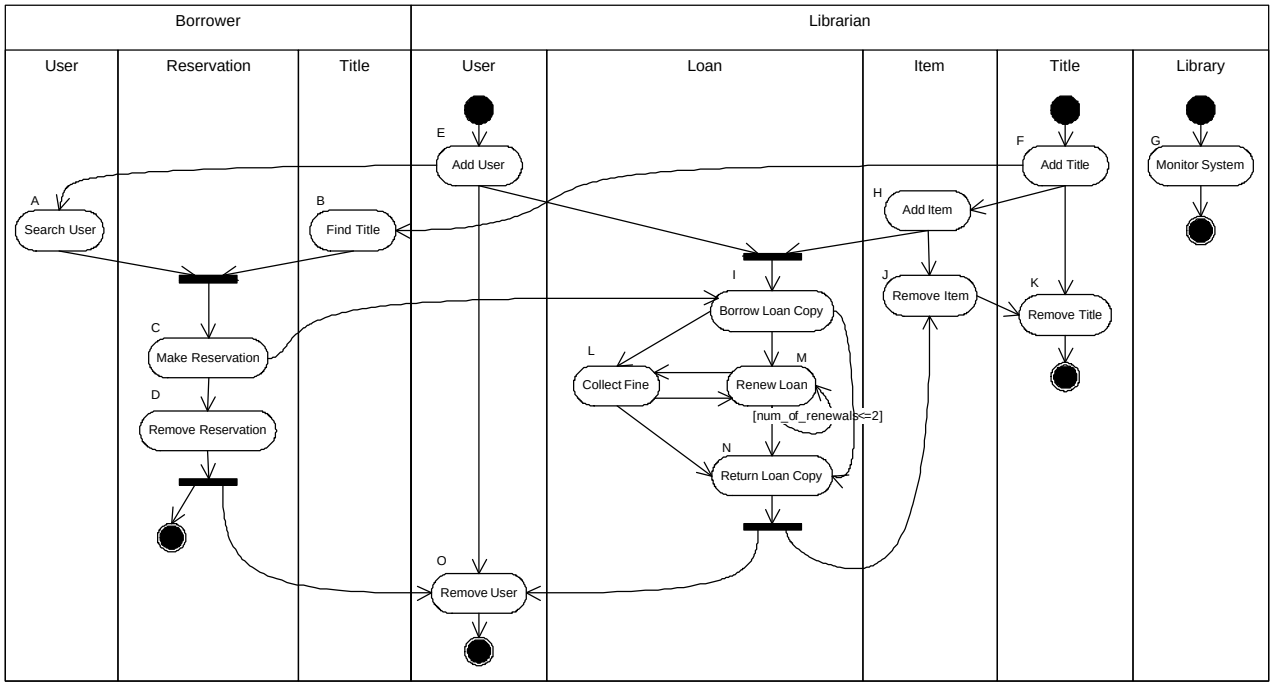


Figure 33- The MIOD of a library system.

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 33. 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 33. 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 current set. For example, the set of SDs {*AddUser*, *AddTitle*} is a set of

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

independent SDs in Figure 33, however it is not maximal according to our definition, since a SD like *MonitorSystem* can still be added to this set while the independence relationship still holds among all the SDs in the set. An ISDS, in this case, can be {*AddUser, AddTitle, MonitorSystem*}. In the following section, we discuss a method to systematically derive all the ISDSs of a MIOD. Note that there can be several ISDSs in a MIOD.

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. We propose a graph-based algorithm here to derive ISDSs of a MIOD.

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 33 is shown in Figure 34.

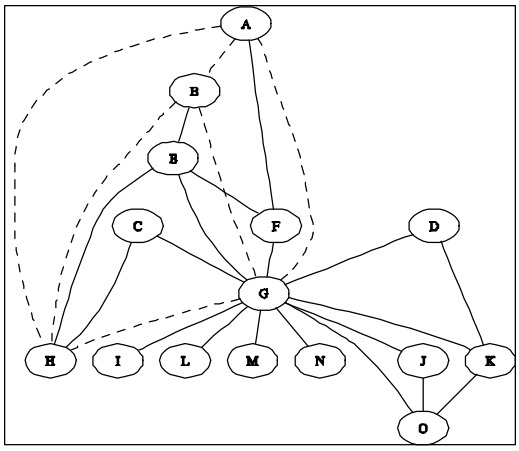


Figure 34-The Independent SD Graph (ISDG) corresponding to the MIOD in Figure 33. The ISDS=*{A,B,G,H}* is shown with dashed edges.

Every *strongly connected component* of an ISDG is an ISDS. A strongly connected component of a graph is a maximal sub-graph of a graph such that for every pair of vertices *u*, *v* in the sub-graph, there is an edge between *u* and *v* [68]. For example, the strongly connected component *{A,B,G,H}* is shown with bold edges in the ISDG of Figure 34, which corresponds to a ISDS.

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 [2] 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 35. In this MIOD, SD *A* is a multi-SD where three concurrent instances of it can be executed.

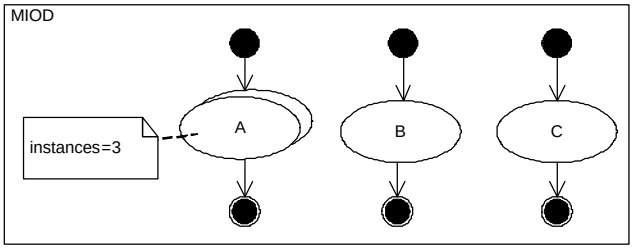


Figure 35-A MIOD with a multi-SD construct.

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 $O(n^2)$, where n is the number of SDs.

Tarjan [69] has devised an $O(n)$ algorithm for determining strongly connected components of a graph. Therefore consider the complexity to build an ISDG, $O(n^2)$, and the complexity to derive its strongly connected components, 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 36, where a MIOD (a) and the CCFG (b) of one of the SDs in the MIOD are shown.

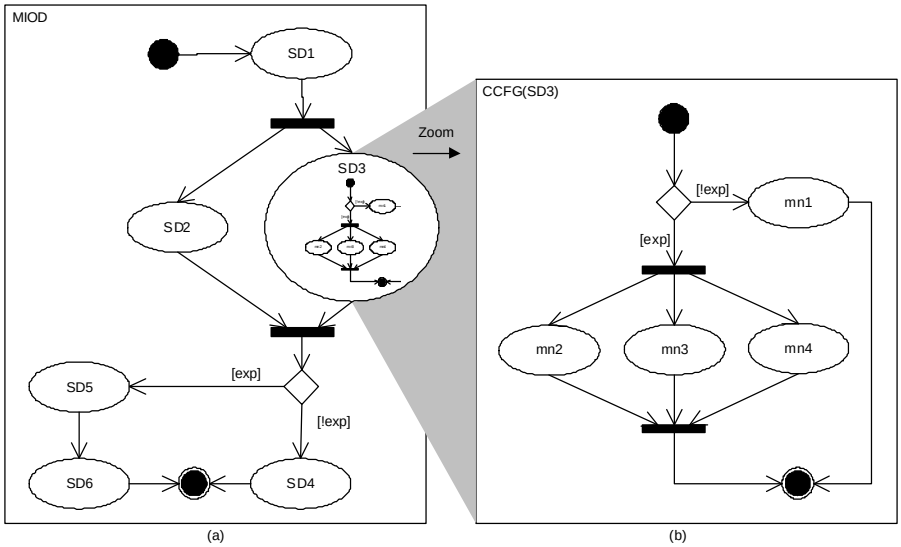


Figure 36-An example MIOD and the CCFG of one of its SDs.

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 36 to better represent the hierarchical relationship.

In order to find CSDFPs of a MIOD, we use the same technique as we used in [2] to derive CCFPs of a SD. This is doable since both MIOD and CCFG are extensions of ADs. For example, the MIOD in Figure 36 has the following two CSDFPs:

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

$$CSDFP_2 = SD_1\begin{pmatrix} SD_2 \\ SD_3 \end{pmatrix}SD_5SD_6$$

As another example, we list here some of the CSDFPs (out a total of 62) which can be derived from the MIOD in Figure 33:

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

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

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 represent 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 36. SD_3 has two CCFPs as:

$$CCFP_{3,1} = mm_1$$

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

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 show the corresponding DCCFPs by $DCCFP_{i,j}$.

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

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

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

$$CSDFP_2 = SD_1\begin{pmatrix} SD_2 \\ SD_3 \end{pmatrix}SD_5SD_6$$

By substituting each SD of *CSDFP1* 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 *CSDFP2*:

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

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

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 impossible CCFPSs.

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_{∀m∈ccfps} (m.endTime)
4.      else if ccfps is the serial concatenation of several CCFPSs (i.e., ccfps = ccfps_1 ··· ccfps_n)
5.          return Duration(ccfps_1)+…+ Duration(ccfps_n)
6.      else if ccfps is the concurrent combination of several CCFPSs (i.e., ccfps = (ccfps_1 ··· ccfps_n))
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 26 as an example). Therefore, the duration of a CCFP is equal to the end time of its latest message. 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 *CCFPS1* 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 *CCFPS1* are given as:$CCFP_{1,3}$ (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 37. 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 ($\begin{pmatrix} r_{2,2} \\ r_{3,1} \end{pmatrix}$), 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.

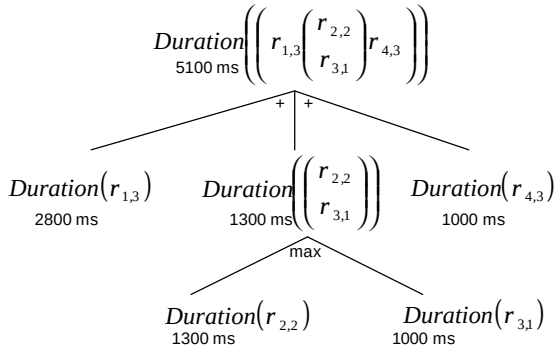


Figure 37-The call tree of the recursive algorithm *Duration* applied to $CCFPS_1$.

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 6.5, a DCCFP is just a filtered CCFP where only distributed messages are selected.

Chapter 8

NETWORK TRAFFIC USAGE ANALYSIS

As we saw in the system model of this work (Figure 9), 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 synchronous message (usually with parameters); the call request is handled (executed) by the receiver object, and 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 network traffic on the connecting networks. We assume two network 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 network traffic usage in the current context and to devise network-aware stress test requirement in a SUT, this section aims to formalize the network 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 network 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 instance or during a period of time.

We will then discuss the effect of concurrent processes on network traffic in Section 8.4. Finally, a class of traffic functions for DCCFPs will be given in Section 8.5. The resource usage analysis technique presented in this section will be used in Chapter 9 and Chapter 10. Furthermore, the definitions and discussions, given in this section, might be useful in other works aiming at studying network traffic of a distributed system by CFA of SDs.

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 6.3:

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

By looking at the above two forms, 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*ameterList=*$<(p_1,C_1,[in\,|\,out]),...,(p_n,C_n,[in\,|\,out])>$ 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. 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 Network Traffic Usage (NTU) functions for different types of messages are presented in Equation 1.

$$NTU : Message \rightarrow Real$$

$$\forall msg \in Message : NTU(msg) = \begin{cases} SignalDT(msg) & ;\text{if } msg.msgType = 'Signal' \\ CallDT(msg) & ;\text{if } msg.msgType = 'Call' \\ ReplyDT(msg) & ;\text{if } msg.msgType = 'Reply' \end{cases}$$

$$SignalDT(msg) = dataSize(msg.methodOrSignalName)$$

$$CallDT(msg) = \sum_{C_i \mid (-, C_i) \in msg.parameterList} dataSize(C_i)$$

$$ReplyDT(msg) = \sum_{C_i \mid (-, C_i) \in msg.returnList} dataSize(C_i)$$

$$\forall C \in classDiagram : dataSize(C) = \sum_{a_i \in C.attributes} dataSize(a_i)$$

Equation 1- Network Traffic Usage (NTU) 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 [70]). 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.

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 [70]).

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 38. Using the class specifications of *A* and *B*, we can estimate the size of the message msg_1 as:

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

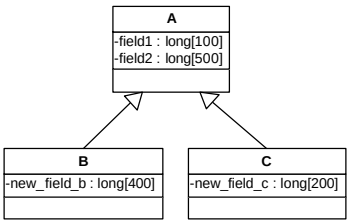


Figure 38-A class diagram showing three classes with data fields.

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 particular 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 38. 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) in its own local database. Apparently, similar to the case of *String* class 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 [71]. Runtime monitoring techniques (such as [57]) can be utilized to monitor and derive such distributions. Then after, the expected value of the distribution can be used as the estimated data size of a message.

8.2 Formal Node and Network Relationships

To facilitate our discussions in the next sections, two formal node and network relationships are defined in this section. We saw earlier in Section 5.5 that a tree structure named NIT (Network Interconnectivity Tree) can be generated from UML network deployment diagrams, to represent the interconnection of the nodes and networks in a system. The NIT of a system is shown in Figure 39, where there are seven nodes ($n_1,\ldots,n_7$) and four networks (including system network).

We define two formal relationships here which are described next.

- Node-network and network-network membership
- Network path function

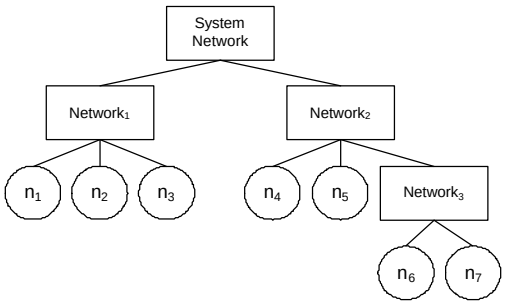


Figure 39-A Network Interconnectivity Tree (NIT).

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:

$$isMemberOf(nod, net) = \begin{cases} true; \text{if network } net \text{ is a precesessor of node } nod \text{ in NIT} \\ false; \text{otherwise} \end{cases}$$

Equation 2-Node-network membership function.

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

$$isMemberOf(net_{sub}, net_{super}) = \begin{cases} true; \text{if network } net_{super} \text{ is a precesessor of network } net_{sub} \text{ in NIT} \\ false; \text{otherwise} \end{cases}$$

Equation 3-Network-network membership function.

For example, the following relations hold in the NIT shown in Figure 39:

- *isMemberOf(n_2, $Network_1$)= true*
- *isMemberOf(n_3, $Network_3$)= false*
- *" n_i: isMemberOf(n_i, SystemNetwork)= true*
- *isMemberOf(n_7, $Network_2$)= true*
- *isMemberOf($Network_2$, SystemNetwork)= true*

8.2.2 Network Path 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. Given a sender (n_s) and a receiver node (n_r), the network path function is an ordered set of networks, which a message sent from n_s will go through in order until it reaches n_r. NIT can be used to derive network paths. For example assuming the NIT of Figure 39, the network path between n_4 (as the sender node) and n_6 (as the receiver node) will be:

$$getNetworkPath(n_4, n_6) = <Network_2, Network_3>$$

Derivation of this network path is shown schematically in Figure 40.

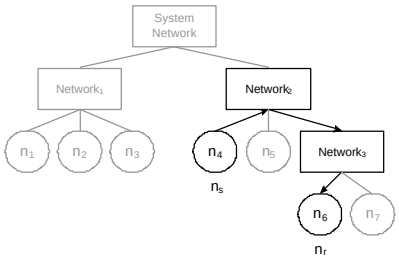


Figure 40-Derivation of network path between two nodes from NIT (*getNetworkPath(n_s, n_r)* function).

8.3 Network Traffic Usage Attributes

In the current resource usage analysis, we consider four attributes for network 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, network 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. It then 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. Traffic location 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 41. Topological and NIT 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*.

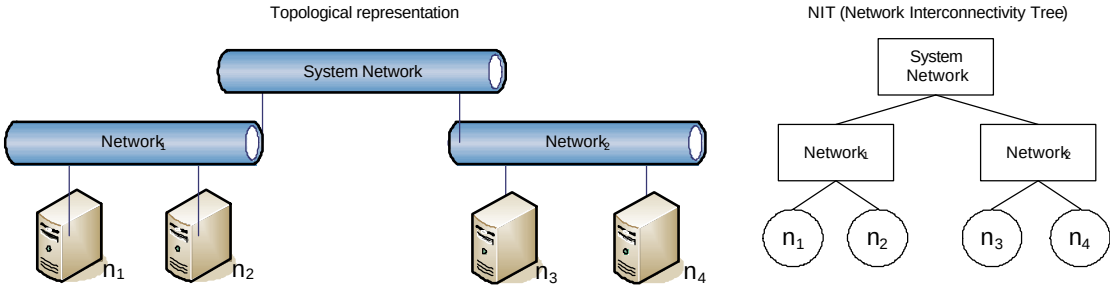


Figure 41-A system made up of four nodes and three networks.

Considering the system topology shown in Figure 41, 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 6.6) of the three mentioned DCCFPs are shown in Figure 42-(a) and (b), respectively.

As it can be seen in Figure 42-(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 NIT in Figure 41). Therefore, traffic entailed by *DCCFP*$_1$ only goes through *Network*$_1$, as shown in timed inter-network representation of *DCCFP*$_1$ in Figure 42-(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 42-(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*.

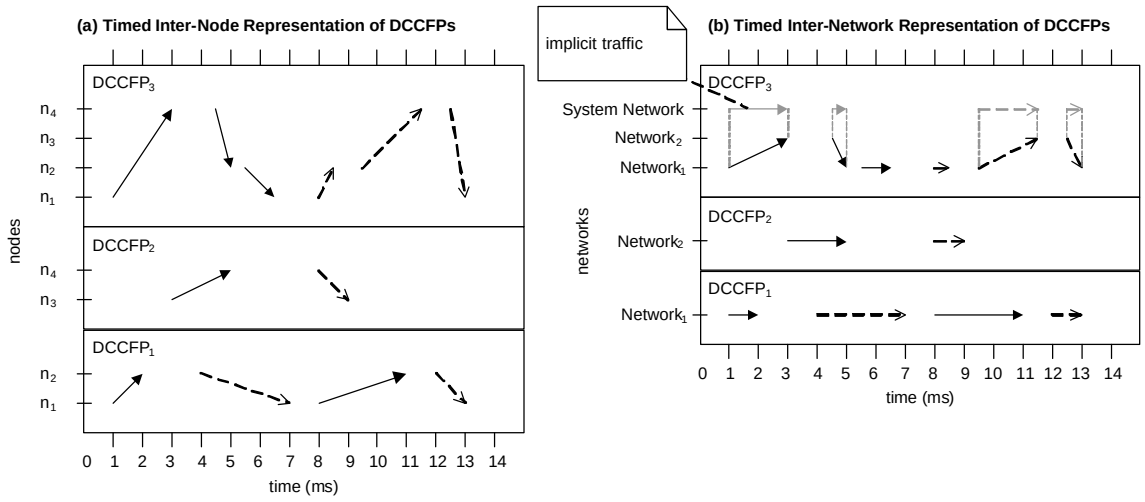


Figure 42-Timed inter-node and inter-network representations of three DCCFPs.

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 42-(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, network 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 network 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 network 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 43.

To compare data versus message traffic, let us look at the control flow path $CCFP_2$ in *CCFG(M)* shown in Figure 43. 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 43 and not those belonging to *CCFG(P)* and *CCFG(N)*. If we consider data traffic as the network 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 43), $DCCFP_2$ 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 network traffic diagram showing data traffic for $DCCFP_2$ as shown in Figure 44. The x-axis is time in milliseconds and only the time interval shown in the SD *M* is considered.

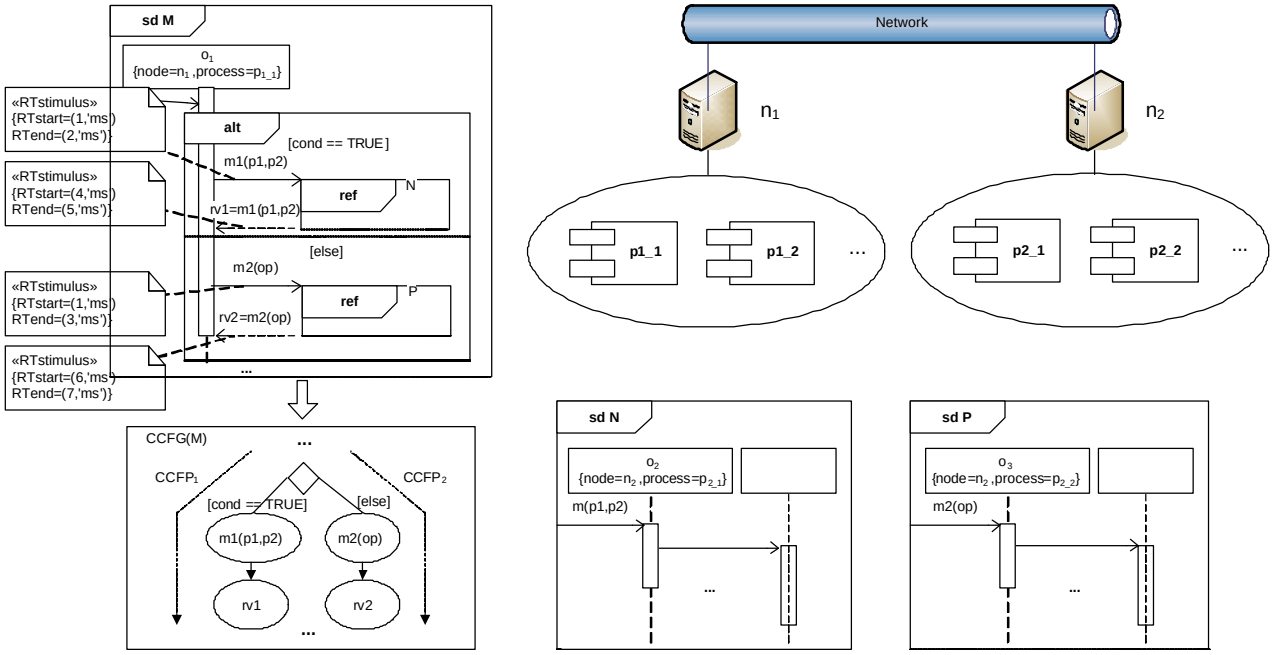


Figure 43-A typical system composed of two nodes and four processes.

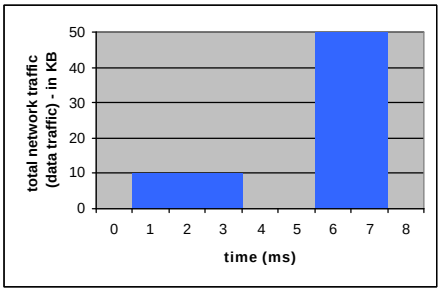


Figure 44-Network traffic diagram (data traffic) of $DCFP_2$ in Figure 43.

On the other hand, if we consider number of distributed messages as the network traffic, the network traffic diagram of $DCCFP_2$ will be as Figure 45 shows. Each call or reply message counts for one unit of distributed message in this analysis.

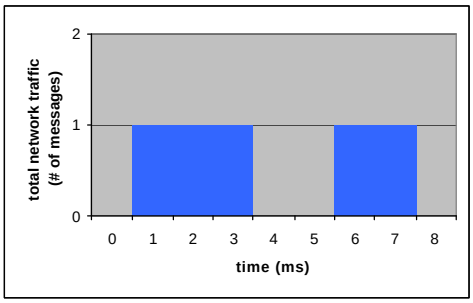


Figure 45-Network traffic diagram (number of distributed messages) of $DCFP_2$ in Figure 43.

Each of the above two network 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 9. 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 network 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 network 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 network 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 44 and Figure 45, 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 44, 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 46, which is actually derived by adding all message-level traffic values for all the messages sent to *n*.

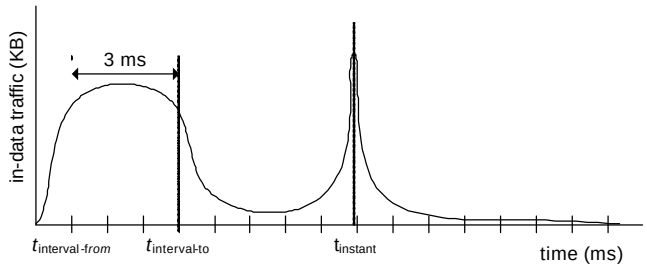


Figure 46-"In-data" traffic diagram of a node, highlighting difference between *instant* and *interval* (3ms) traffic.

According to Figure 46, 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 Effect of Concurrent Processes

According to the SUT model in Figure 9, several processes can run concurrently on a single node. Each of the processes might be in the process of running a method. Therefore, the network 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 47. 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.

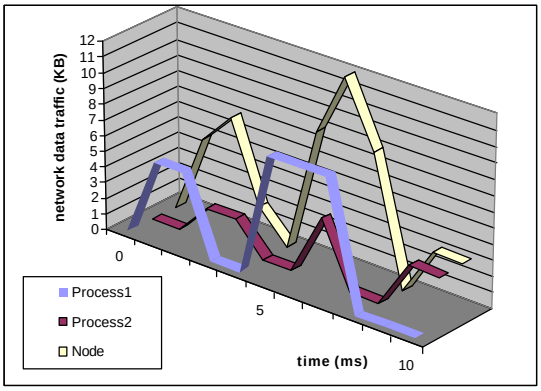


Figure 47-The data traffic diagram of a node with two processes.

8.5 A Class of Traffic Functions for Distributed Concurrent Control Flow Paths

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 network 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 network 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 is shown in Figure 48. The root node of the tree has a null label. A function name is formed by traversing from the root to a leaf node and concatenating all the node labels in order.

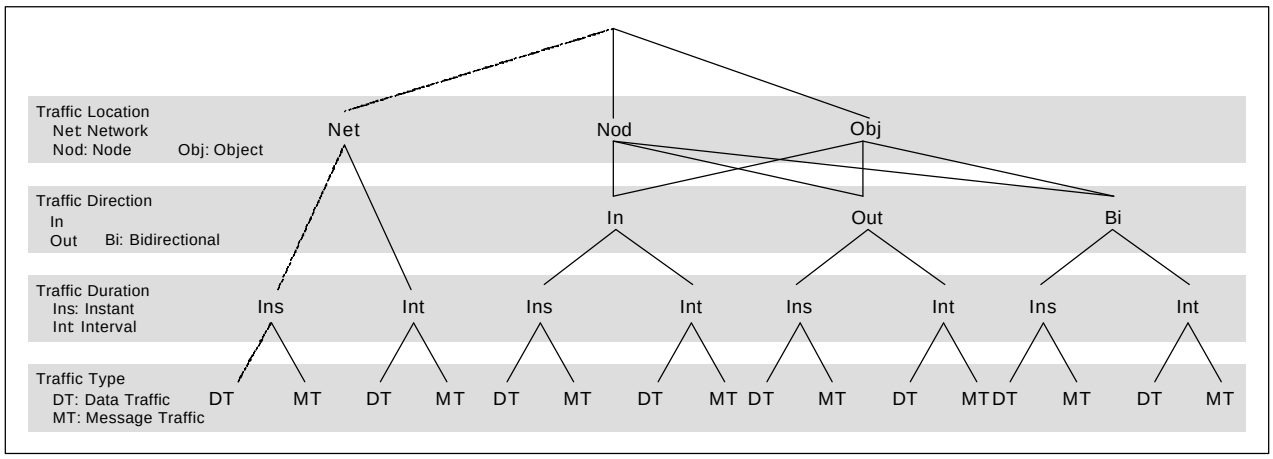


Figure 48-Naming convention for traffic usage functions.

Four layers are shaded in the tree. They correspond to four traffic attributes discussed in Section 8.3. One addition made to the traffic location layer is that objects (*obj*) are also considered to be parts of the traffic location attribute. In this context, objects are processes on nodes of a distributed system. This generalization, by no means, violates our previous categorization of traffic location, given in Section 8.3.1. We believe that, in addition to network- and node-level traffic, it might be useful to analyze traffic at the object level as well. This adds more location granularity in traffic analysis and we believe that a distributed object or a node might behave differently (and exhibits faults) if we direct the traffic towards them in

different scenarios. The other three layers (direction, duration and type) fully conform to the discussions 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 we will define 28 different traffic functions.

In addition to the tree notation used above, the general form of a function can also be specified using the Backus-Nauer Form (BNF) [67] as shown in Equation 4.

$$
\begin{aligned}
&functionName ::= LocationDirectionDurationType \\
&Location ::= Net \mid Nod \mid Obj \\
&Direction ::= \begin{cases} In \mid Out \mid Bi & ;\text{if } Location \in \{Nod, Obj\} \\ \text{null} & ;\text{else} \end{cases} \\
&Duration ::= Ins \mid Int \\
&Type ::= DT \mid MT
\end{aligned}
$$

Equation 4-BNF to generate traffic usage function names.

Equation 5 gives a BNF for the input parameters of a traffic function based on its name.

$$
\begin{aligned}
inputParameters(functionName) ::= &\begin{cases} (r, net, t) & ;\text{if } functionName == NetInsT \\ (r, net, interval) & ;\text{if } functionName == NetIntT \\ (r, nod, t) & ;\text{if } functionName == NodDInsT \\ (r, nod, interval) & ;\text{if } functionName == NodDIntT \\ (r, obj, t) & ;\text{if } functionName == ObjDInsT \\ (r, obj, interval) & ;\text{if } functionName == ObjDIntT \end{cases} \\
&D ::= * \ ;\text{Direction} \\
&T ::= * \ ;\text{Type} \\
&interval ::= (start, end)
\end{aligned}
$$

Equation 5-BNF to generate the input parameters of traffic usage functions.

For example, consider the path specified by the dashed line in Figure 48. This path represents function *NetInsDT*. According to the syntax form for the input parameters of traffic functions (given in Equation 5), the input parameters of this function would be (*r*, *network*, *t*). The explanation of this function will be that it returns the instant (Ins) data traffic (DT) value of a given DCCFP (*r*) for a given network (*network*) at a given time (*t*). *per::=(start,end)* in Equation 5 means that, for functions with interval duration, the start and end times of an interval should be given as input. More detailed descriptions of the functions will be given and the ways to calculate their values will be discussed next.

8.5.2 Functions

In this section, traffic functions are listed using the naming convention given in Figure 48. The functions are grouped according to the top layer (traffic location) of the tree in Figure 48. Mathematical formulas to calculate some traffic functions will be given. The rest can be derived in similar fashion. In the following mathematical formulas, for brevity, *msg.start* and *msg.end* have been used instead of *msg.startTime* and *msg.endTime*. *msg.s.n* and *msg.r.n* have been used 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 instance *t* to *t+1*.

$$NetInsDT(r,net,t)=\begin{cases}\sum_{msg_i} size(msg_i)/dur(msg_i) & ;\exists msg_i \mid msg_i \in r \wedge \\ & msg_i.start \le t \le msg_i.end \wedge \\ & net \in getNetworkPath(msg_i.s.n, msg_i.r.n) \\ 0 & ;\text{otherwise}\end{cases}$$

where *s*, *r* and *n* are shorthand notations for sender, receiver and node fields of a message. *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.

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

$$NetInsMT(r,net,t)=\left|\begin{array}{l}\forall msg_i \mid msg_i \in r \wedge \\ msg_i.start \le t \le msg_i.end \wedge \\ net \in getNetworkPath(msg_i.s.n, msg_i.r.n)\end{array}\right|$$

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*.

$$NetIntDT(r,net,int)=\sum_{t=int.start}^{t=int.end} NetInsDT(r,net,t)$$

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

$$NetIntMT(r,net,int)=\sum_{t=int.start}^{t=int.end} NetInsMT(r,net,t)$$

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.

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

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

$$NodInInsMT(r,nod,t)=\left|\begin{array}{l}\forall msg_i \mid msg_i \in r \wedge \\ msg_i.start \le t \le msg_i.end \wedge \\ msg_i.r.n = nod\end{array}\right|$$

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

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

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

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

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.

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

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

$$NodOutInsMT(r, nod, t) = \begin{vmatrix} \forall msg_i \mid msg_i \in r \wedge \\ msg_i.start \le t \le msg_i.end \wedge \\ msg_i.s.n = nod \end{vmatrix}$$

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

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

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

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

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.

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

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.

$$NodBiInsMT(r, nod, t) = \begin{vmatrix} \forall msg_i \mid msg_i \in r \wedge \\ msg_i.start \le t \le msg_i.end \wedge \\ (msg_i.s.n = nod \vee msg_i.r.n = nod) \end{vmatrix}$$

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.

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

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.

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

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.

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

   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.

$$ObjInInsMT(r,obj,t)=\left|\begin{array}{l}\forall msg_i \mid msg_i \in r \wedge \\ msg_i.start \le t \le msg_i.end \wedge \\ msg_i.r.o = obj\end{array}\right|$$

An Example

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

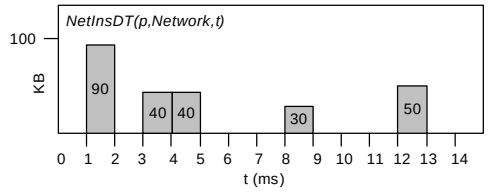
$CM_1=((o_1,O_1,n_1),(o_2,O_2,n_2),t,<(p1:-),(p2:-)>,1,2)$

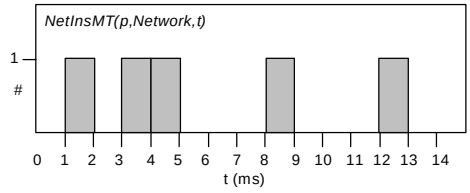
$CM_2=((o_2,O_2,n_2),(o_3,O_3,n_3),u,<(p3:-),(p4:-)>,3,5)$

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

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

Let us suppose a SUT's NIT to be the one shown in Figure 31. Also suppose that the sizes of the four messages of DTCCFP *r* 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.





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

Chapter 9

TIME-SHIFTING STRESS TEST TECHNIQUE

This section describes the first (and the simpler) stress test technique to stress test network traffic. The technique is an optimization technique which is based on shifting DCCFPs along time axis to find the time instance when maximum possible stress can occur.

The chapter is structured as the following. The problem statement is revisited in Section 9.1, where we express the problem using the formalism given in Chapter 5-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-Chapter 8. Test objectives are discussed in Section 9.2. Section 9.3 presents the heuristic of our stress test strategy. An example is presented in Section 9.4 to visualize the heuristic. Then after, different strategies of the proposed stress testing approach are discussed in Section 9.5, which closely match the different network traffic attributes discussed in Section 8.3. We discuss in Section 9.6 how we take into account the inter-SD constraints in the generation of stress test requirements. Section 9.7 formulates the stress test generation problem as an optimization problem. Section 9.8 presents the high-level stress testing algorithm. Input and pre-processing steps of the high-level algorithm are discussed in Section 9.9. The general form of a stress test requirement (the output of the technique) is given in Section 9.10. Finally techniques to derive test requirements depending on different stress strategies are proposed in Section 9.11. The algorithm complexities are discussed in Section 9.12. A variation of the technique, referred to as *Real-Time Constraint-Oriented Stress Test (RTCOST)* is presented in Section 9.13, which aims at generating test requirements for a given Real-Time constraint. Section 9.14 discusses how the derivation process of test elements can be automated.

9.1 Problem Statement: Revisited

Having formalized the input and test modeling needed for our stress test technique in Chapter 5-Chapter 8, we state the problem statement in a more precise manner in this section. The rephrased problem statement is as the following:

> 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 (to be used for polymorphic CFA of SDs, Section 5 of [2], and data size estimation, Section 8.1), and the network deployment diagram(s) showing interconnectivity and the network hierarchy of the system (as discussed in Section 5.5). We also require the inter-SD constraints are given using a MIOD (Section 5.3). Suppose the CFA of system's SDs is done using the techniques in Chapter 6. Inter-SD constraints are analyzed according to the techniques in Chapter 7 and SUT's Independent-SD Sets (ISDSs), Section 7.1, and Set of SD Sequences (SSDS), Section 7.2, are derived. The network traffic of the system is formalized as stated in Chapter 8. The problem is to find a schedule to run 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 network traffic faults (defined in Section 3.2.2).

9.2 Test Objectives

The fundamental test objective is to perform a network-aware stress testing on a distributed system. The overall goal in our ongoing research is to propose stress test techniques for different aspects of a distributed system. In this work, we only consider the network traffic of such systems, and in particular we propose a stress test technique to maximize the chance of detecting distributed traffic faults (as described in Section 3.2.2). Our objective would be to propose a systematic testing technique to automatically generate test requirements to stress test the network traffic of a system, based on a UML 2.0 design model of the system. The test requirements will essentially be a set of selected DCCFPs along with a schedule to execute them. The general form of stress test requirements is precisely specified in Section 9.10.

9.3 Stress Test Heuristic

As discussed in Chapter 5, we assumed that a system may have several nodes, where each node is running several concurrent processes. We also assumed that there can be several SDs running concurrently on the system nodes. In Chapter 6, each SD was assigned a corresponding CCFG, where each CCFG could have one or more CCFPs. A Distributed CCFP (DCCFP) was defined as a CCFP where only distributed messages are considered. Depending on the UML 2.0 interaction constraints [8], executing a SD might follow different CCFPs in its CCFG. These different CCFPs will yield different DCCFPs. Different DCCFPs of a SD will cause different traffic on different networks and nodes of the system.

Given a specific network or node to stress test, our heuristic is to choose a message (or a set of messages) in a particular DCCFP of a SD which imposes maximum traffic (either in terms of data or number of messages) on the given network or node. Let us refer to such messages as *maximum stress messages*. Intuitively, if none of the DCCFPs of a SD has any message going through a given network or to/from a given node, it means that this particular SD does not have any network traffic on the network or node and hence it will not be included in the output stress test schedule. Afterwards, using the start times of the maximum stress messages selected in each DCCFP, the selected set of DCCFPs can be scheduled in a way that the maximum stressing messages all can run concurrently. We believe that this concurrent schedule of DCCFPs will cause a maximum possible traffic on a particular node or network, which in turn will increase the probability of exhibiting distributed traffic faults in the node or network under stress test.

The heuristic can be visualized by an example. The example below is just to illustrate the heuristic and does not include a formal description of the test requirement derivation process. It will be given in Section 9.11. It should also be mentioned that the stress test has different strategies as discussed in Section 9.5.

9.4 An Example to Visualize the Heuristic

Suppose a typical SUT whose NIT is shown in Figure 49. According to the NIT, there are three nodes n_1, n_2 and n_3 and a network (*SystemNetwork*) in the system, where all nodes are connected to.

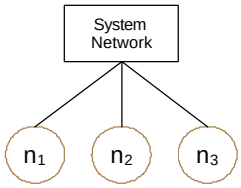


Figure 49-A simple system NIT.

For simplicity, let us assume the system has several SDs which CFA has yielded four DCCFPs (*DCCFP*$_1$,...,*DCCFP*$_4$). The timed inter-node representations (described in Section 6.6) of DCCFPs are shown in Figure 50-(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.

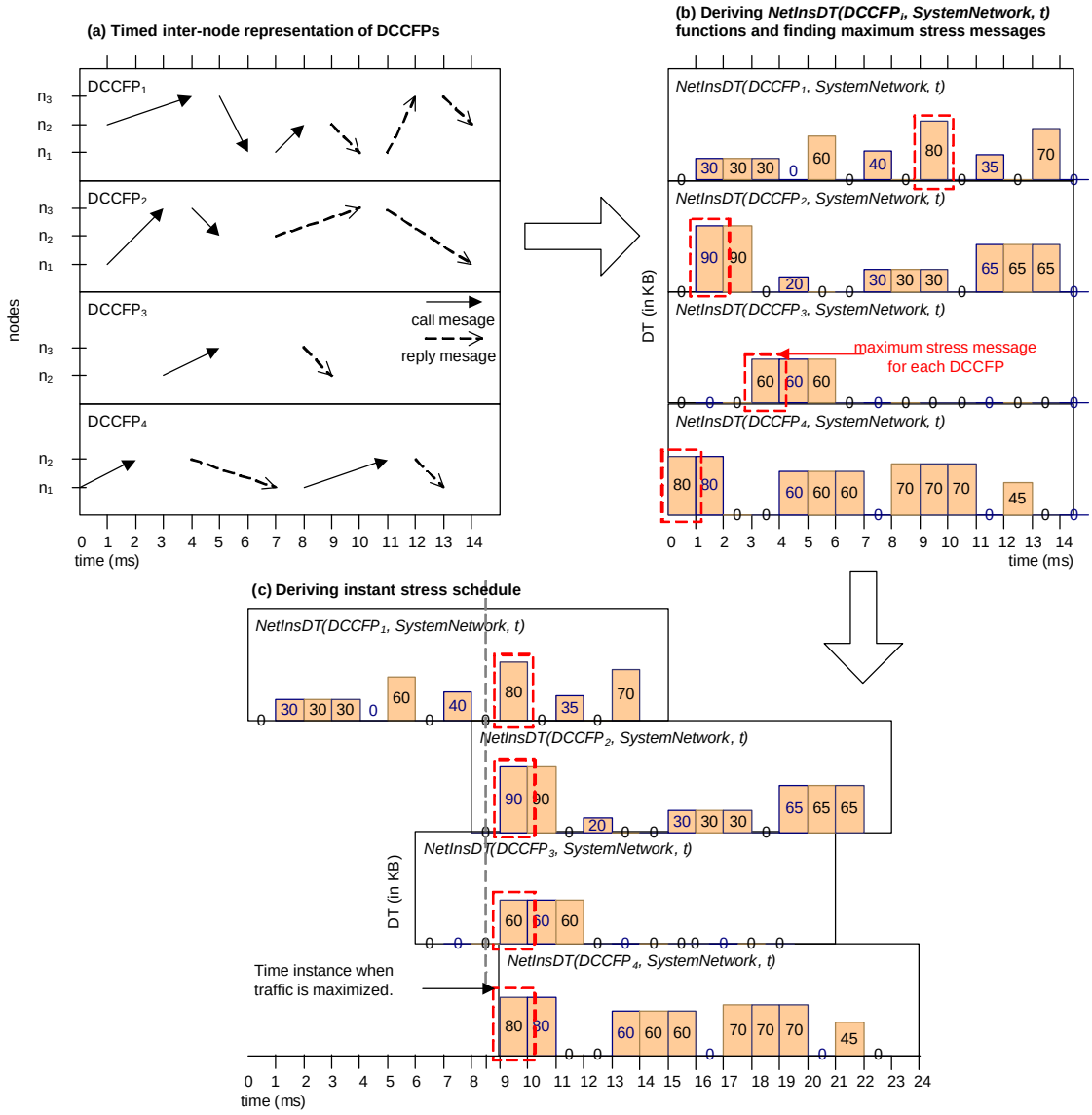


Figure 50-Heuristic to stress test instant data traffic on a network.

In this example, let us suppose that we want to derive test requirements for what we call *network instant data traffic (NetInstDT)* stress for network *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 50-(b) for each DCCFP. Again for simplicity, the calculation steps of those functions are not shown.

Next step of the heuristic is to find the *maximum stress messages* of each DCCFP. This is shown graphically in Figure 50-(b) by dashed lines around such messages. Recalling that the criterion to find these messages is that the given network is 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 selected in each DCCFP to schedule the selected set of DCCFPs such that these maximum stress messages all run concurrently. This is illustrated in Figure 50-(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 ms (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.

9.5 Different Stress Testing Strategies

As discussed in the fault taxonomy, Chapter 3, nodes or networks of a system may not be robust to faults with distributed nature. As we categorized in Section 3.2, one type of such faults was distributed traffic faults, which occur when a system failure is due to the fact that a network or a node does not function correctly under heavy traffic.

As mentioned in Section 9.1, our stress test heuristic is to enforce heavy network workloads on system components to increase the probability of exhibiting a distributed traffic fault. We investigate different strategies for heavy network workloads: stress location (networks or nodes), traffic direction (in and out – only applies when testing nodes), stress type (data traffic or number of requests), and stress duration (instant versus interval stress) strategies. The reason why traffic direction (in and out) do not apply in our stress testing approach when testing networks is that networks, unlike nodes, are not the end points of traffic, i.e. any traffic entering a network goes out of it, therefore distinguishing in and out traffic in case of a network is unnecessary. Different stress testing strategies are discussed in following sections.

9.5.1 Location: Nodes vs. Networks

As discussed in faults taxonomy, Section 3.5, a distributed fault might happen in one of the following locations in a distributed messaging scenario between two nodes:

- A network in the network path between two nodes
- Sender or receiver node

As discussed in Section 5.5, a network path between two nodes is the unique path extracted from the Network Interconnectivity Tree (NIT). As discussed in Section 8.3.1, network traffic was also analyzed in two strategies (nodes and networks) in terms of its location.

The rational behind this classification of stress testing strategy is that either a network (in a network path) or one of the two end nodes becomes the cause of a failure when excessive traffic goes through it.

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

As discussed above, the stress test target can be either a network, sender or receiver node of a distributed message. In case of a node, we can think of two scenarios for stress testing in terms of traffic direction. Traffic can be maximized either *towards* a node or *from* a node. In other words, in one scenario, we might schedule all DCCFPs such that all distributed messages *towards* (arriving at) a node be sent all at once. While in the second scenario, all distributed messages *from* a node will be scheduled to be sent concurrently.

The rationale for this distinction in stress test strategy is that a traffic fault might be revealed in the network component of a node if a large amount of traffic arrives from other nodes to the node all at once. Conversely, a network traffic fault might occur if large instantaneous traffic is originated from the node to other nodes in the system. The low level faults causing this malfunction might include: insufficient network buffer / frame sizes, high CPU load of the node and failing to process traffic on time.

9.5.3 Type: Amount of Data vs. Number of Messages

Dividing stress type in two categories: amount of data and number of messages corresponds to the distinction made for traffic types in Section 8.3.3. Networks or nodes of a system might exhibit faults when they are faced with big amounts of data or large number of message going through them (networks), from or towards them (nodes).

9.5.4 Duration: Instant vs. Interval

In term of duration of the stress, we consider two approaches: *instant* and *interval*. These two approaches correspond to the distinction made for traffic duration in Section 8.3.4. We define an *instant* network stress test requirement to be a schedule of DCCFPs of a system's SDs that imposes a traffic stress in a given unit of time. The used unit of time is assumed to be the smallest unit of time defined in a system, such as millisecond. This is mentioned in tagged values of messages in SDs. For example, an example of modeling time using the UML-SPT profile was shown in the SD in Figure 3, where the smallest unit of time among all tagged-values is ms (millisecond). An *interval* stress test requirement, on the other hand, is a set of SDs and their execution schedule that, when applied to the SUT, causes a stress in a network or a node during an interval of time.

9.6 Taking into Account the Inter-SD Constraints

As we discussed in Chapter 7, executing any arbitrary sequence of SDs in a SUT might not be always valid or allowed. 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. A Modified Interaction Overview Diagram (MIOD) was proposed in Section 5.3 to model sequential and conditional inter-SD constraints.

The duration strategy of stress testing in our system model affects the way we should take into account the inter-SD constraints, because:

- In case of instant stress testing, a set independent SDs, which entails the maximum stress should be triggered concurrently, and,
- In case of interval stress testing, the objective is to trigger a sequence of SDs with maximum stress.

The first type of stress test from duration point of view is *instant* stress (Section 9.5.4). We discussed in Section 9.5.4 that instant network stress is a schedule of DCCFPs that entail a maximum traffic stress in an instant time. One important consideration in generating such schedule is that the SDs that are to be executed altogether should not be dependent (Section 7.1). We proposed a method in Section 7.1 to derive so called *Independent-SD Sets* (*ISDSs*) in a SUT. As defined, an ISDS is a set of SDs that can be run concurrently, i.e., there are no inter-SD constraints between any two of the SDs in the set.

The other type of stress test from duration point of view is *interval* stress. As defined, a interval stress test is a set of SDs and their execution schedule that, when applied to a SUT, causes maximum stress in a network or a node during an interval of time. One important consideration in this case is to choose and schedule those sequences of SDs that are allowed in the SUT. A method was proposed in Section 7.2 to derive *Concurrent SD Flow Paths* (*CSDFP*). As defined, a CSDFP is a sequence of SDs that are allowed to be executed in a SUT (according to the constraints modeled in the MIOD). According to this definition, any sequence of SD in a SUT which is not a CSDFP is not allowed to be executed and hence can not be used in a stress test scenario.

In our stress test mythology, we assume that the inter-SD constraints are given in a form of a MIOD. MIOD is then used to generate the SUT's Independent-SD Sets (ISDS) and Concurrent SD Flow Paths (CSDFP). ISDSs and CSDFPs will be used in Section 9.11 by instant and interval stress test techniques to generate valid test requirements, respectively.

If we consider the stress test generation problem as an optimization problem, ISDSs and CSDFPs will be the *constraints* of the optimization problem. We discuss in next section how the stress test generation problem can be formulated as an optimization problem.

9.7 Formulating the Stress Test Generation Problem as an Optimization Problem

The stress test requirement generation problem can be formulated an optimization problem. The general formulated optimization problem, without taking into account the different testing strategies (Section 9.5), is presented in Figure 51.

Objective: Maximize the traffic on a specified network or node (at a time instant or a period of time)
Variables:
- A subset of DCCFPs (one DCCFP from each SD) with maximum traffic on a specified network or node
- Schedule to run the selected DCCFPs

Constraints:
- Inter-SD sequential and conditional constraints

Figure 51-Formulating the Stress Test Generation Problem as an Optimization Problem.

9.8 High-level Algorithm

The high-level algorithm for the derivation of stress test requirements is given in Algorithm 2. Steps 1 and 2 in Algorithm 2 are briefly described next. After that, the general form of a stress test requirement (as the output of the technique) is defined in Section 9.10. The last step of the high-level algorithm (Step 3) is described in Section 9.11.

Step 1. Input (Chapter 5):

- Use the UML design model of the SUT as the input model. The UML design model should include:
  - Network Deployment Diagram (NDD): for deriving NIT (Section 5.5)
  - SDs: SD_1,…, SD_n (system has *n* SDs.). SDs should have timing information. RT constraints should be modeled as stated in Section 5.6.
  - Class diagram(s) (Section 5.2): to be used for polymorphism-dependent CFA (Section 5 of [2]) and also calculation of message sizes (Section 8.1)
  - Modified Interaction Overview Diagrams (MIOD): to model the inter-SD constraints (Section 5.3)
- A list of test objectives where each objectives is a tuple of four fields:
  - A stress location: either a network or a node name
  - A stress direction (only for nodes): in, out or bidirectional
  - A stress type: data or number of messages:
  - A stress duration: instant or period

Step 2. Building the Test Model-(Chapter 5-Chapter 7):

- Build the system's Network Interconnectivity Tree (NIT) based on the Network Deployment Diagram in the UML model (Section 5.5).
- Control Flow Analysis of SDs:
  - Transform each of the system's SDs into its corresponding CCFG (Chapter 6).
  - Derive CCFPs and then DCCFPs of each CCFG (Chapter 6).
- Taking into consideration the inter-SD constraints:
  - Derive Independent-SD Sets (ISDSs) (Section 7.1).
  - Derive Concurrent SD Flow Paths (CSDFP) (Section 7.2).

Step 3. Derivation of Test Requirements (Section 9.11):

- For each entry in the test objectives list, depending on the stress location and test direction, derive the test requirements using the following algorithms:
  - If the stress location is a network, use the algorithm in Section 9.11.2
  - If the stress location is a node:
    - If test direction is "in", use the algorithm in Section 9.11.3.1.
    - If test direction is "out", use the algorithm in Section 9.11.3.2.
    - If test direction is "bidirectional", use the algorithm in Section9.11.3.3.

Algorithm 2-High level algorithm for derivation of stress test requirements

9.9 Input and Building the Test Model

We comprehensively described the system model of a SUT and the UML syntax to be used in Chapter 5. We also discussed how to build the test model from the given UML design model in Chapter 5-Chapter 7.

The CFA procedures to convert the SUT's UML model into CCFGs and DCCFPs were presented in Chapter 6. Those methods should be used to convert a given system model into a NIT and a set of DCCFPs, which will be used in Step 3 of Algorithm 2 to derive test requirements.

We also discussed in Chapter 7 how to derive the Independent-SD Sets (ISDSs) and Concurrent SD Flow Paths (CSDFP) of a SUT. As discussed in Section 9.6, these two will be taken into account as inter-SD constraints in the generation of stress test requirements.

9.10 Output Stress Test Requirements

For each test element in the test objective list of Algorithm 2, a stress test requirement set will be generated by our technique. Assuming that a 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:

$$<(r_{1max}, ar_{1max}), \ldots, (r_{nmax}, ar_{nmax})>$$

where *i*-th entry of the set is a tuple of r_{imax} and ar_{imax}. r_{imax} is a DCCFP in the DCCFP set of SD_i, $DCCFP(SD_i)$, that entails a stress traffic over the given system component (network or a node) with a given stress flavor (direction, type and duration). ar_{imax} is the start time of DCCFP r_{imax}, i.e., the time to trigger r_{imax}, that together with all DCCFPs in the test requirement set, will maximize traffic over a given network or node. A stress test requirement set is the output of our methodology. Algorithms to derive test requirements for different stress test strategies will be given in the next section. Intuitively, if none of the DCCFPs of SD_i has any message going through the given network or to/from the given node, it means that that SD_i does not have any network traffic on the given network or node and hence it will not be included in the output stress test set. In such a case, the *i*-th entry of the test requirement set (corresponding to SD_i) will be null.

9.11 Derivation of Stress Test Requirements

Various algorithms (corresponding to different stress test strategies) to derive stress test requirements are given in this section. We use a set of mathematical functions ,in our algorithms. First the naming convention of such functions will be given in Section 9.11.1. Algorithms will be then presented in Sections 9.11.2 and 9.11.3.

As an example of how stress test requirements can be generated using algorithms in this section (such as Algorithm 3 and Algorithm 4), refer to Section 12.4 where test requirements derivation process of our case study has been explained in detail. We consider different test elements with different stress test strategies in our case study.

9.11.1 Naming Convention

Naming convention of functions used in different stress test algorithms can be described using the tree in Figure 52. Note that these functions return maximum message(s) and DCCFP(s) of a given DCCFP and SD, respectively. In such a sense, these functions are different than the functions presented in Figure 48, which were intended to measure traffic entailed by a DCCFP.

The root node of the tree in Figure 52 is *StressTest*, indicating that all functions return a value with a particular maximization criterion, which depends on the function name. A function name is made by traversing from the root to a leaf node and concatenating all the node titles in order. Five layers are shaded in the tree. Four top layers specify the four different stress test strategies mentioned in Section 9.5. The lowest layer (function's output type) indicates the return type of the function whose name is generated by traversing from the root to a leaf node and concatenating all the node labels in order. As mentioned in Section 9.5.1, since we do not consider stress direction ("in", "out" and "bidirectional") for networks, the network sub-tree (nodes under *Net*) is not divided into two branches in the second layer. Since there are 32 leaf nodes in Figure 52, there will be 32 different functions.

For example, consider the path specified by a dashed line in Figure 52. This path represents function *MaxNetInsDTMsg*. This function finds the message (Msg) in a given DCCFP that yields the maximum (Max) instant (Ins) data traffic (DT) on a given network (Net).

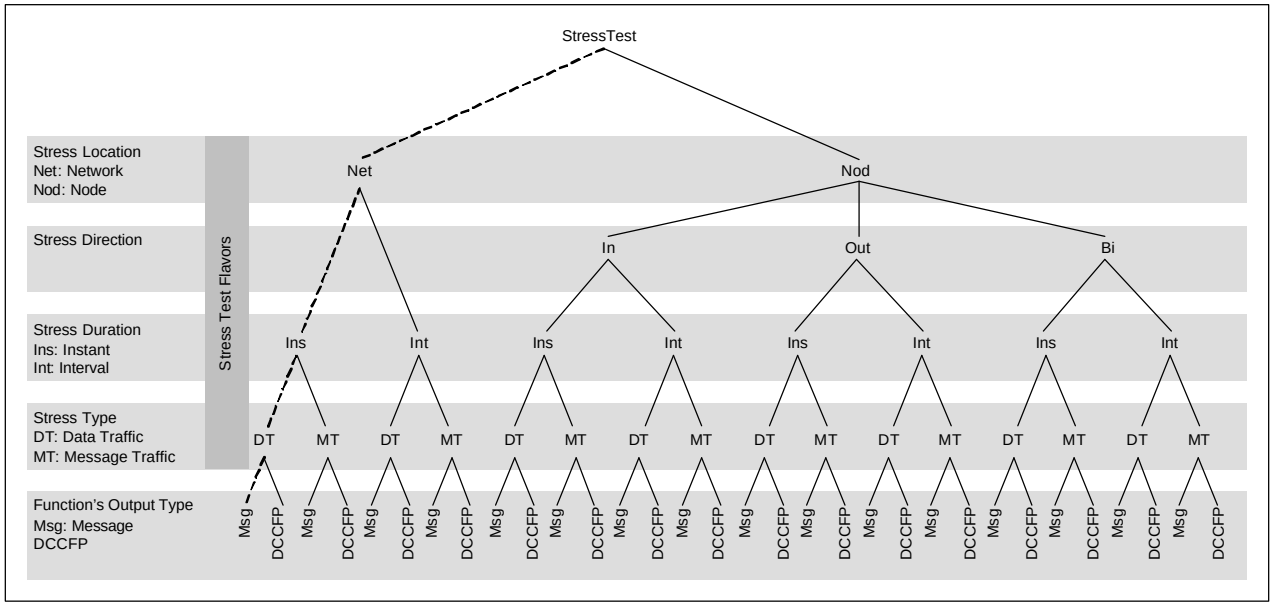


Figure 52-Naming conventions of functions used in various stress test algorithms.

The general form of a function can also be given using the BNF in Equation 6.

$$
\begin{aligned}
&functionName ::= StressTestLocationDirectionDurationTypeOutput\\
&Location ::= Net \mid Nod\\
&Direction ::= \begin{cases} In \mid Out \mid Bi & ;\text{if } Location == Nod\\ \text{null} & ;\text{else} \end{cases}\\
&Duration ::= Ins \mid Per\\
&Type ::= DT \mid MT\\
&Output ::= Msg \mid DCFG
\end{aligned}
$$

Equation 6-BNF to generate stress test function names.

The BNF in Equation 7 can be used to determine the input parameters of a function based on its name.

$$
inputParameters(\ functionName) ::= \begin{cases} (r_{ij}, network) & ;\text{if } functionName == MaxNet * Msg\\ (SD_i, network) & ;\text{if } functionName == MaxNet * DCFP\\ (r_{ij}, node) & ;\text{if } functionName == MaxNod * Msg\\ (SD_i, node) & ;\text{if } functionName == MaxNod * DCFP \end{cases}
$$

Equation 7-BNF for the list of input parameters of a stress test function.

where input parameter r_{ij} is a DCCFP in the *CCFG* of SD_i. Input parameters *network* and *node* are given to be stress tested and *interval* is the time interval for which the interval stress test should be derived for. The functions and input parameters usage will be clear in the next sections, where algorithms for the derivation of test requirements complying with different test strategies will be given.

There are four layers in the stress test strategies block in Figure 52. These four layers indicate the variations of stress test strategy we can apply on a system of nodes and networks. Counting the different paths from the root of the tree to the nodes of the lowest of these four layers (stress type), 16 different strategies can be considered. Similar to the naming convention of functions explained above, the BNF in Equation 8 specifies various stress test strategies.

$$stressTestStrategyName ::= StressLocationDirectionDurationType$$
$$Location ::= Net \mid Nod$$
$$Direction ::= \begin{cases} In \mid Out \mid Bi & ;\text{if } Location == Nod \\ \text{null} & ;\text{else} \end{cases}$$
$$Duration ::= Ins \mid Per$$
$$Type ::= DT \mid MT$$

Equation 8-BNF to generate various stress test strategies.

Using the above naming convention for stress test strategies, example strategies are: *StressNetInsDT*, *StressNetInsMT*, *StressNetPetDT*, *StressNetPetMT*, and *StressNodInInsDT*.

9.11.2 Test Requirements for a Network

According to the naming tree in Figure 52, we consider four stress test strategies for a network:

1. *StressNetInsDT(net)*
2. *StressNetInsMT(net)*
3. *StressNetIntDT(net)*
4. *StressIntMT(net)*

In the following, we discuss each of the above and give algorithms to derive stress test requirements for each stress test strategy.

1. *StressNetInsDT(net)*

To better understand this stress test strategy, a top-level UML activity diagram is shown in Figure 53, which depicts the flow of activities to generate stress test requirements of the stress test strategy *StressNetInsDT(net)*.

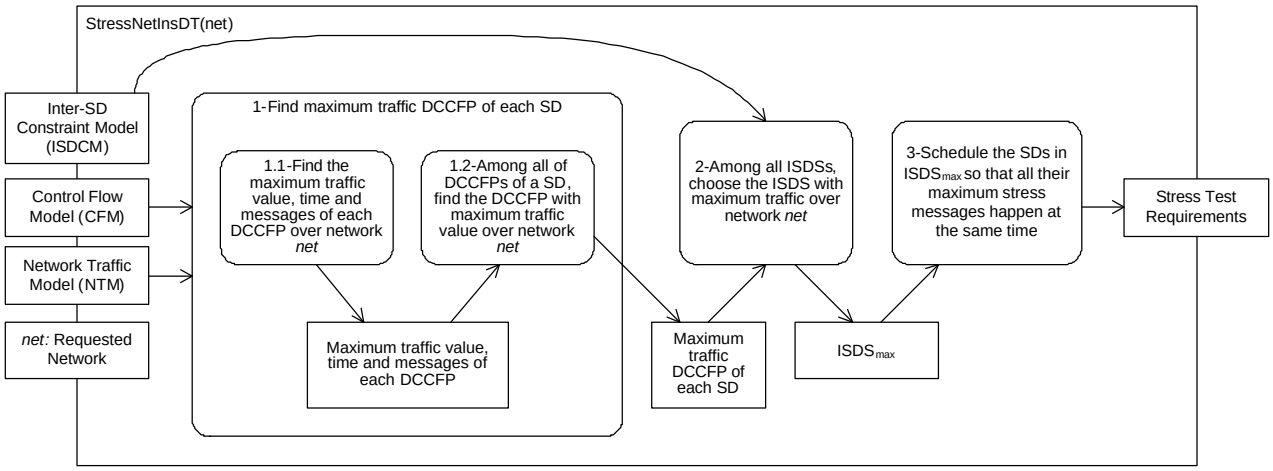


Figure 53-Activity diagram of stress test strategy *StressNetInsDT(net)*.

To better clarify the idea, let us consider the hierarchical relationships between messages, DCCFPs, SDs, ISDSs and a MIOD; and see how the algorithm works in extracting hierarchical maximums. Such hierarchical relationships are illustrated using the concept of composition association in Figure 54–(a). 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.

The dashed arrow beside the hierarchical relationships represents the direction of hierarchical maximum selection by the algorithm. To generate stress test requirements, the algorithm (in activity 1.1 of Figure 53) first finds the maximum traffic messages of each DCCFP. Using the maximum traffic message, the maximum traffic DCCFPs of each SD are then chosen (in activity 1.2). Then after, among all ISDSs of a MIOD, the ISDS with maximum traffic is chosen. If all the entitles in the hierarchical relationship are

consider as sets, the hierarchical maximum selection process can be shown using a *Venn diagram*[1] in Figure 54–(b). All entities are sets of entities inside them, except messages (shown as *msg*) which are not sets. The hierarchical maximum selection process starts from messages and continues on to ISDSs in MIOD level. The detailed pseudo-code of the algorithm is then given in Algorithm 3.

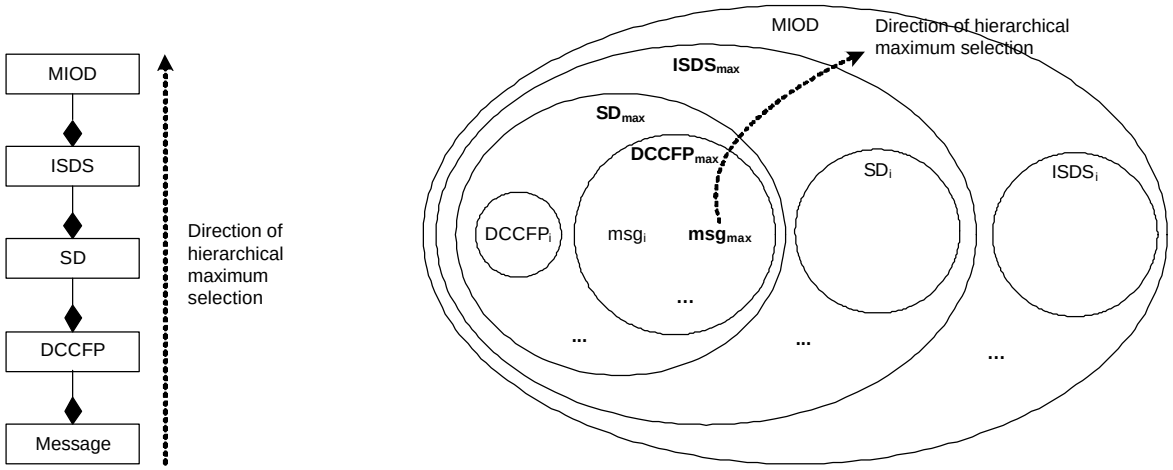


Figure 54-(a): Hierarchical relationships between messages, DCCFPs, SDs, ISDSs and a MIOD. (b): Using a Venn diagram to represent how the hierarchical maximum selection process works.

Here we describe the rationale behind Algorithm 3 and each of its steps. Our goal in this case is to derive stress test requirements with network stress location, data traffic type and instant stress duration. Step 1 finds the DCCFP, for each SD, which entails the maximum stress on the given network. Step 1.1.1 finds the maximum stress message of each DCCFP first. Step 1.1.2 then finds the DCCFP with maximum message size (maximum of maximums) among all of DCCFPs of a SD.

In order to consider the inter-SD constraints in finding maximum instant stress, Step 2 chooses an ISDS (Independent-SD Set) with maximum stress. In order to do so, Step 2.1 first calculates each ISDS's *MaxNetInsDT* using the values calculated in Step 1. Using these values, Step 2.2 then finds the ISDS with maximum stress and labels this ISDS as $ISDS_{max}$. $ISDS_{max}$ will be used is Step 3 when scheduling all DCCFPs to force the maximum stress.

Step 3 is the final step of the algorithm which schedules the selected DCCFPs of SDs in $ISDS_{max}$ to force the maximum stress on the given network. In order to schedule the selected SDs, Step 3.1 first calculates the latest start time among the selected DCCFPs of all SDs in $ISDS_{max}$. The latest start time is saved in *DCCFPsLateStartTime*. Step 3.2 generates the actual schedule by using the *DCCFPsLateStartTime* variable and shifting the start times of the selected DCCFPs (and their corresponding SDs) to enforce concurrent execution of the maximum stressing messages of such DCCFPs. This was shown earlier graphically in Figure 50-(c). The maximum stress will occur in *DCCFPsLateStartTime* time instance after starting the test.

[1] A graphical representation in which sets are represented by closed areas. The closed regions may bear all kinds of relations to one another, such as be partially overlapped, be completely separated from one another, or be contained totally one within another. All members of a set are considered to lie within or be contained within the closed region representing the set. The diagram is used to facilitate the determination of whether several sets include or exclude the same members.

1. Finding maximum stress DCCFP of each SD
   1.1. For each SD_i
      1.1.1. For each DCCFP r_{ij} of SD_i / / Finding the maximum DT value, stress time and stress messages of each DCCFP

Find the maximum DT value and time in r_{ij}, using:

$$MaxNetInsDTValue(r_{ij}, net) = \max_{t}\left(NetInsDT(r_{ij}, net, t)\right)$$

$$MaxNetInsDTTime(r_{ij}, net) = t_{\max} \mid \forall t : MaxNetInsDTValue(r_{ij}, net, t_{\max}) \geq MaxNetInsDTValue(r_{ij}, net, t)$$

Find the set of messages in r_{ij}, which put maximum traffic on network *net*, using:

$$MaxNetInsDTMsgs(r_{ij}, net) = (msg_{\max 1}, \ldots, msg_{\max n}) \left| \begin{array}{l} msg_{\max i} \in r_{ij} \wedge \\ msg_{\max i}.start \leq MaxNetInsDTTime(r_{ij}, net) \leq msg_{\max i}.end \wedge \\ net \in getNetworkPath(msg_{\max i}.s.n, msg_{\max i}.r.n) \end{array} \right.$$

where first condition ensures that the message putting maximum traffic is returned. *dur(msg)* is the duration of a message and can be calculated as *dur(msg)=msg.end-msg.start*. The last two conditions make sure that only messages going through the given network *net* are compared to yield the maximum. *s*, *r* and *n* are shorthand notations for sender, receiver and sender.node fields of a message.

If no message in r_{ij} satisfies the *network_path* condition, the function returns null.

      1.1.2. Among all of SD_i's DCCFPs r_{ij}, find the DCCFP with maximum stress value:

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

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

2. Choosing the ISDS (Independent-SD Set) with maximum stress: / / Considering inter-SD constraints
   2.1. For each $ISDS_i$ / / Calculate each ISDS's MaxNetInsDT

$$MaxNetInsDTValue(ISDS_i) = \sum_{\forall SD \in ISDS_i} MaxNetInsDTValue\left(MaxNetInsDTDCCFP(SD, net), net\right)$$

   2.2. Among all ISDSs, find the ISDS with maximum $MaxNetInsDTValue(ISDS_i)$ and refer to it as $ISDS_{max}$

3. Derivation of stress test requirements (scheduling SDs in the ISDS with maximum stress – found in Step 2):
   3.1. Calculate the latest start time among the selected DCCFPs r_{imax} of all SDs in $ISDS_{max}$:

$$DCCFPsLatestStartTime = \max_{\forall SD_i \in ISDS_{\max}} \left(\min_{\forall message \in MaxNetInsDTMsgs\left(MaxNetInsDTDCCFP(SD_i, net), net\right)} \left(message.start\right) \right)$$

   3.2. For each SD in $ISDS_{max}$
      3.2.1. If $r_{i\max} = MaxNetInsDTDCFP(SD_i, net)$ is not null

$StressTestSchedule_i = (r_{i\max}, ar_{i\max})$ where $ar_{i\max}$ is $r_{i\max}$'s start time and is equal to:

$$ar_{i\max} = DCFPsLatestStartTime - \min\left(MaxNetInsDTMsgs(r_{i\max}, net).start\right)$$

      3.2.2. Else

$StressTestSchedule_i = null$

Algorithm 3-Derivation of instant stress test requirements for data traffic on a given network (*StressNetInsDT*). Stress traffic will occur in time=*DCCFPsLatestStartTime*.

We gave the details of the algorithm for *StressNetInsDT(net)* strategy, where *net* is the input parameter to the algorithm. Algorithm 3 can be modified to give algorithms for the other instant network stress test strategy (*StressNetInsMT*) as described below. The algorithms for two other network stress test strategies (*StressNetIntDT* and *StressNetIntMT*) will be slightly different since they have to generate *interval* stress tests.

2. *StressNetInsMT(net)*

The following changes should be made in Algorithm 3:

- Formulas in Step 1.1.1:

$$MaxNetInsMTValue(r_{ij}, net) = \max_{t}\left(NetInsMT(r_{ij}, net, t)\right)$$

$$MaxNetInsMTTime(r_{ij}, net) = t_{\max} \mid \forall t : MaxNetInsMTValue(r_{ij}, net, t_{\max}) \geq MaxNetInsMTValue(r_{ij}, net, t)$$

$$MaxNetInsMTMsgs(r_{ij}, net) = (msg_{\max 1}, \ldots, msg_{\max n}) \left| \begin{array}{l} msg_{\max i} \in r_{ij} \wedge \\ msg_{\max i}.start \leq MaxNetInsMTTime(r_{ij}, net) \leq msg_{\max i}.end \wedge \\ net \in getNetwork Path(msg_{\max i}.s.n, msg_{\max i}.r.n) \end{array} \right.$$

- Formula in Step 1.1.2:

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

- The statements in Steps 2 and 3 stay the same. Only the variable and function names with *DT* pattern should be changed to *MT* instead.

3. *StressNetIntDT(net)*

The algorithms for the two network stress test strategies *StressNetIntDT* and *StressNetIntMT* will be slightly different than the two mentioned before (*StressNetInsDT* and *StressNetInsMT*), since these two have to generate *interval* stress tests. The UML activity diagram of stress test strategy *StressNetIntDT(net)* is shown in Figure 55. The Pseudo-code of the algorithm is then given in Algorithm 4.

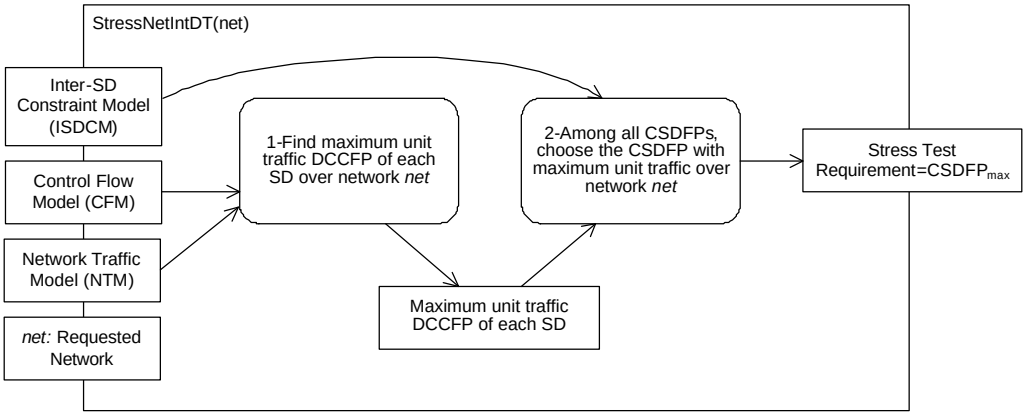


Figure 55- Activity diagram of stress test strategy *StressNetIntDT(net)*.

Here we describe the rationale of the Algorithm 4. Our goal in this case is to derive stress test requirements with network stress location, data traffic type and instant stress duration. Step 1 of the Algorithm 3 finds the DCCFP, for each SD, which entails the maximum stress on the given network. Step 1.1.1 finds the maximum stress message of each DCCFP first. Step 1.1.2 then finds the DCCFP with maximum message size (maximum of maximums) among all of DCCFPs of a SD.

1. Finding the DCCFP of each SD with maximum unit data traffic
   1.1. For each SD_i
      1.1.1. For each DCCFP r_{ij} of SD_i / / Finding maximum stress message of each DCCFP
      Calculate Unit Data Traffic (UDT) of r_{ij}, using:

$$NetUDT(r_{ij}, net) = \frac{\sum_{t} (NetInsDT(r_{ij}, net, t))}{Duration(r_{ij})}$$

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

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

      where $CCFP(r_{ij})$ is the CCFP corresponding to DCCFP r_{ij}.

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

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

      If no DCCFP in SD_i is found with the above criteria, the function returns null.
2. Choosing a CSDFP (Concurrent SD Flow Path) with maximum stress: / / Considering inter-SD constraints
   2.1. For each $CSDFP_i$ / / Calculate each CSDFP's Unit Data Traffic (UDT)

$$NetUDT(CSDFP_i, net) = \frac{\sum_{\forall SD \in CSDFP_i} \sum_{\forall t} NetInsDT(MaxNetPerDTDCCFP(SD, net), net, t)}{Duration(BuildDCCFPS(CSDFP_i, MaxNetPerDT, net))}$$

      where *Duration* (presented in Section 7.2.3) is a function that calculates the time length of a DCCFPS (DCCFP Sequence). *BuildDCCFPS* is function that builds a DCCFPS from the given $CSDFP_i$ using the given criteria: $\forall SD \in CSDFP : MaxNetPerDTDCCFP(SD_i, net)$.
   2.2. Among all CSDFPs, find the sequences with maximum $NetUDT(CSDFP_i, net)$ and return it as output ($CSDFP_{max}$)

Algorithm 4-Derivation of interval stress test requirements for data traffic on a given network (*StressNetIntDT*).

4. *StressNetIntMT(net)*

In Algorithm 4, the name of the functions with pattern **DT** should be replaced with **MT**.

9.11.3 Test Requirements for a Node

Algorithm 3 and Algorithm 4 can be modified to provide algorithms for stress test strategies of nodes. We group node stress test strategies into three groups: *in*, *out* and *bidirectional.*

9.11.3.1 Stress Direction: In

Node stress test strategies with "in" stress direction can be extracted from the naming tree in Figure 52:

1. *StressNodInInsDT (nod)*
2. *StressNodInInsMT(nod)*
3. *StressNodInIntDT(nod)*
4. *StressNodInIntMT(nod)*

Modifications to the Algorithm 3 and Algorithm 4 to devise algorithms for node test strategies with "in" stress direction are described in this section.

1. *StressNodInInsDT(nod)* strategy

In Algorithm 3, the name of the functions with pattern **Net** should be replaced with **NodIn**.

2. *StressNodInInsMT(nod)* strategy

In Algorithm 3, the name of the functions with pattern **Net*DT* should be replaced with **NodIn*MT*.

3. *StressNodInIntDT(nod)* strategy

In Algorithm 4, the name of the functions with pattern **Net** should be replaced with **NodIn**.

4. *StressNodInIntMT(nod)* strategy

In Algorithm 4, the name of the functions with pattern **Net*DT* should be replaced with **NodIn*MT*.

9.11.3.2 Stress Direction: Out

Node stress test strategies with "in" stress direction can be extracted from the naming tree in Figure 52:

1. *StressNodOutInsDT (nod)*
2. *StressNodOutInsMT(nod)*
3. *StressNodOutIntDT(nod)*
4. *StressNodOutIntMT(nod)*

In the following, we discuss each of the above and give the details of the algorithms to derive stress test requirements.

1. *StressNodOutInsDT(nod)* strategy

In Algorithm 3, the name of the functions with pattern **Net** should be replaced with **NodOut**.

2. *StressNodOutInsMT(nod)* strategy

In Algorithm 3, the name of the functions with pattern **Net*DT* should be replaced with **NodOut*MT*.

3. *StressNodOutIntDT(nod)* strategy

In Algorithm 4, the name of the functions with pattern **Net** should be replaced with **NodOut**.

4. *StressNodOutIntMT(nod)* strategy

In Algorithm 4, the name of the functions with pattern **Net*DT* should be replaced with **NodOut*MT*.

9.11.3.3 Stress Direction: Bidirectional

Node stress test strategies with "bidirectional" stress direction can be extracted from the naming tree in Figure 52:

1. *StressNodBiInsDT (nod)*
2. *StressNodBiInsMT(nod)*
3. *StressNodBiIntDT(nod)*
4. *StressNodBiIntMT(nod)*

In the following, we discuss each of the above and give the details of the algorithms to derive stress test requirements.

1. *StressNodBiInsDT(nod)* strategy

In Algorithm 3, the name of the functions with pattern **Net** should be replaced with **NodBi**.

2. *StressNodBiInsMT(nod)* strategy

In Algorithm 3, the name of the functions with pattern **Net*DT* should be replaced with **NodBi*MT*.

3. *StressNodBiIntDT(nod)* strategy

In Algorithm 4, the name of the functions with pattern **Net** should be replaced with **NodBi**.

4. *StressNodBiIntMT(nod)* strategy

In Algorithm 4, the name of the functions with pattern **Net*DT* should be replaced with **NodBi*MT*.

9.12 Algorithms Complexity

The steps to calculate the complexities of Algorithm 3 and Algorithm 4 are shown in Figure 56 and Figure 57 respectively, where the calculations are performed in a bottom-up manner (from sub-steps, to steps, and then to the whole algorithm). The variables are defined in Table 6.

Variable	Description
s	Number of SDs
m	Average number of messages per each DCCFP
p	Average number of DCCFPs per each SD
i	Number of ISDSs
y	Average number of SDs per each ISDS
t	Average time duration of each DCCFP
c	Number of CSDFPs

Table 6-Description of the variables used in calculating algorithms complexity.

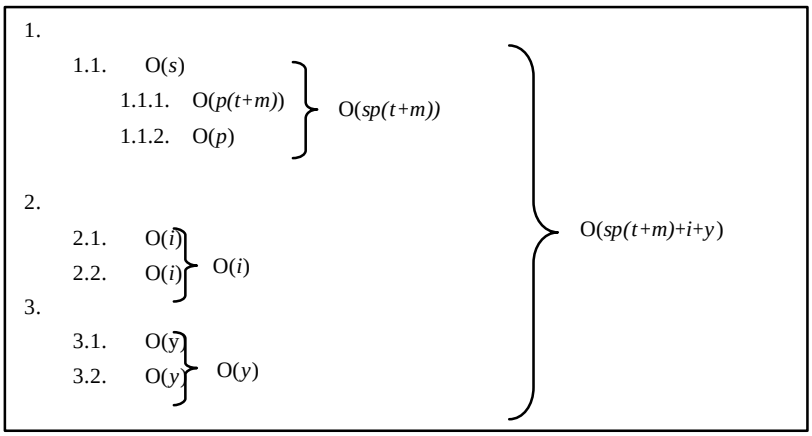


Figure 56-Calculating complexity of Algorithm 3.

For example, as it has been shown in Figure 56, Step 1.1.1 of Algorithm 3 has the complexity of O(*p*(*t*+*m*). This is because there is a loop on all DCCFPs of a SD (*p*), and there are two loops (maximum functions) on all time instances (*t*) and messages (*m*) of each DCCFPs in Step 1.1.1.

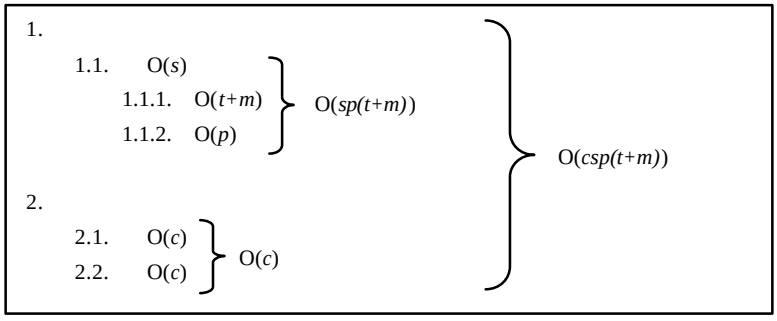


Figure 57-Calculating complexity of Algorithm 4.

9.13 Real-Time Constraint-Oriented Stress Test

The stress test methodology, discussed in the previous sections, finds the maximum possible traffic portion of every SD/DCCFP and schedules SDs in a way that the maximum possible stress happens in a time instances or during a time interval. Let us refer to such methodology as g*lobal* stress test. Such stress test targets to find any network traffic fault in a SUT. In this way, a network traffic fault might be revealed in SUT's network component, e.g., buffers, queues or in its business logic. For example, by applying a specific

stress test flavor, we might find out that the length of a network buffer of a server in a safety-critical system is not enough in stress conditions. Furthermore, different strategies of global stress test can usually enforce violations only in a subset of the RT constraints. This is because the test requirements generated by the global stress test methodology, i.e., Independent-SD Sets (ISDSs) or Concurrent SD Flow Paths (CSDFP), usually cover subsets of all RT constraints in a SUT. Therefore, some RT constraints might never be exercised by a global stress test.

As an example, consider the MIOD in Figure 58, which has two MIOD-level HRT constraints. One of the constraints is on the duration of *SD10* and the other one is bound to the length of *SD5* and *SD6* together (from the beginning of *SD5* to the end of *SD6*). Let us refer to the former as *HRTC1* and to the later as *HRTC2*. Suppose that the data and message traffic of messages in different SDs are such that, no matter which stress test flavor is applied to this SUT, *SD6* (i.e., any of its DCCFPs) never gets chosen as part of the test requirement. This is possible since the data and message traffic values of messages in *SD6* might be less than all messages in all other SDs. In such a situation, *HRTC1* will never get a chance to be exercised (tested), since *SD6* never gets executed by any of the stress test strategies. However, as it can be seen in the MIOD, *HRTC1* has more criticality value than *HRTC2*, which means the former has more critical consequences if it happens to be missed in the field.

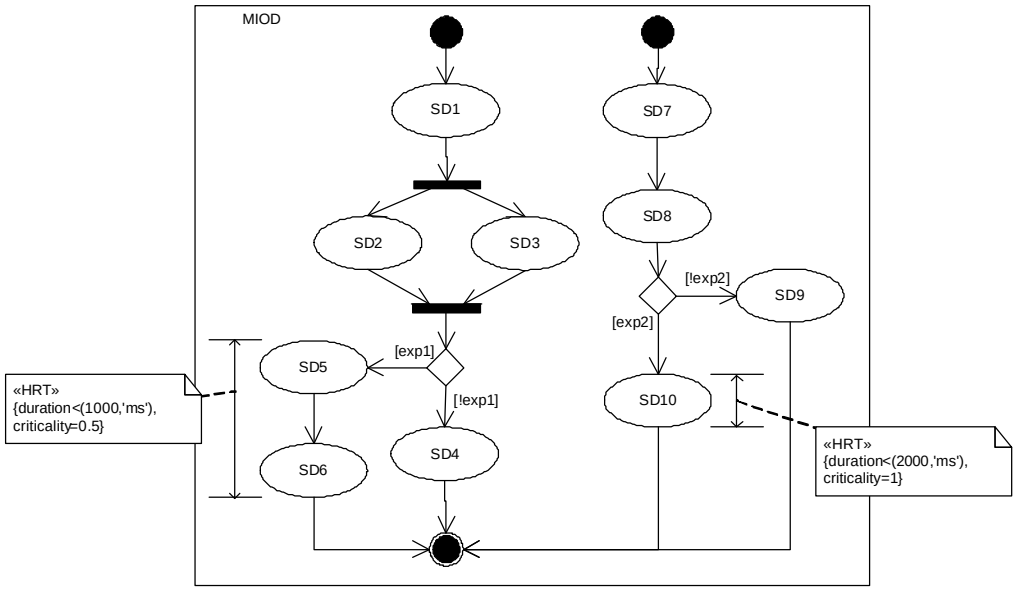


Figure 58-An example MIOD with two MIOD-level HRT constraints.

To address the above issue, we propose a modified stress test technique, referred to as *Real-Time Constraint-Oriented Stress Test (RTCOST)*. Given a RT constraint, RTCOST derives stress test requirements which target the given constraint in particular, and maximize the chances of violating it. By using RTCOST, all RT constraints can be checked one by one to make sure they hold in most stressed conditions of a system. The concept of global stress test and RTCOST are briefly compared in Figure 59.

Luckily, the RTCOST technique can be devised by minor modifications to the global stress test requirement generation algorithms, presented in Section 9.11. As we will discuss, the modifications are slightly different for SD-level and MIOD-level constraints.

We present next algorithms for some of the RTCOST variations (strategies), grouped by the level of the given constraint (SD-level or MIOD-level). The rest of the RT constraint-oriented stress test strategies can be derived in a similar fashion. The strategies are prefixed by "*SDRT*" (for SD-level constraints) and "*MIODRT*" (for MIOD-level constraints); and are derived from their global-stress-test counterparts.

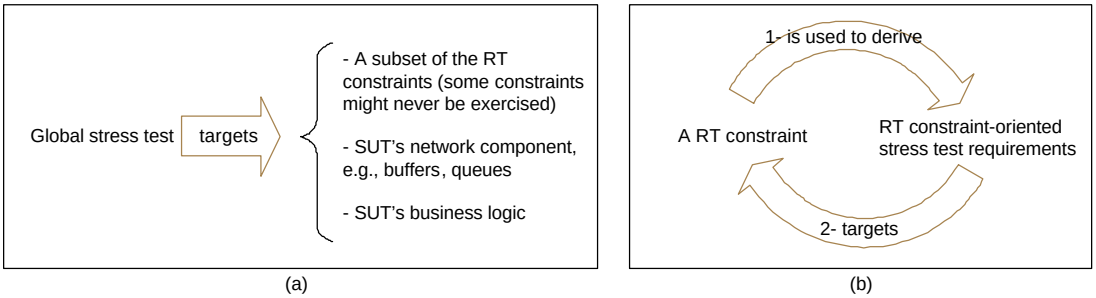


Figure 59-(a): Global stress test versus (b): RT constraint-oriented stress test.

9.13.1 SD-Level RTCOST

We present here the algorithm for *SDRTStressNetInsDT(RTC)* test flavor, where *RTC* is a given SD-level RT constraint. The algorithm can be derived from *StressNetInsDT* global stress test flavor (Section 9.11.2) and is shown in Algorithm 5. Algorithm for the other stress test strategies can be derived in a similar fashion, as were presented for the global stress test algorithms in Section 9.11.

In Algorithm 5, *Msgs(RTC)*, *SD(RTC)* and *Nets(RTC)* are utility functions and are defined as follows. *Msgs(RTC)* and *SD(RTC)* return the messages the *RTC* is connected to and the SD enclosing the *RTC*. These functions are easy to implement since *RTC* is a SD-level constraint. For example, considering the SD-level constraint *RTC* in Figure 60-(a), *Msgs(RTC)* and *SD(RTC)* will return the values: *Msgs(RTC)={m1,m2,r2,r1}* and *SD(RTC)=M*.

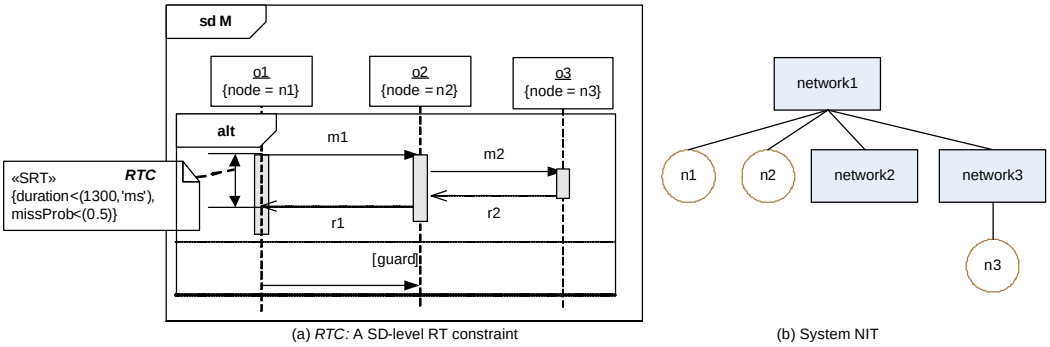


Figure 60-An example of a SD-level RT constraint.

Nets(RTC) returns the set of networks which messages of set *Msgs(RTC)* go through. For example considering the RT constraint in Figure 60-(a) and the system NIT in Figure 60-(b), *Nets(RTC)* will return *{network1, network3}*. The pseudo code of function *Nets(RTC)* is given in Algorithm 6. The output of this function is the union of all network paths of all messages bound by the given RT constraint.

We now briefly discuss the rational of Algorithm 5. Only a RT constraint is given as the input to the algorithm and the algorithm is supposed to find, among networks in *Nets(RTC)*, the network which receives the maximum instant data traffic. The reason why we limit the search domain to networks in *Nets(RTC)* is that the stress test strategy is RT-constraint oriented and we need to find a network stress situation targeting the given constraint. In Step 1.1.1.1 of the algorithm, we do a slightly modified search among DCCFPs of *SD(RTC)*. Since for *SD(RTC)*, we want the stress test to happen in the portion where *RTC* is located, we therefore assign the stress values only for DCCFPs covering messages in *Msgs(RTC)*.

Algorithm 5 can be also modified to account for network capacities while searching for net_{max} (network with maximum stress on *RTC*). The modifications should be made in Step 2 of the algorithm, so that the network, where the ratio of instant data traffic to its capacity is the highest, is chosen as net_{max}. Alternatively, all the networks in *Nets(RTC)* can be stress tested, if resources permit.

1. For each network *net* in *Nets(RTC)*
   1.1. Find maximum stress DCCFP of each SD for *net*
      1.1.1. For each SD_i where SD_i
         1.1.1.1. For each DCCFP r_{ij} of SD_i

If $SD_i <> SD(RTC)$ then

$$MaxNetInsDTValue(r_{ij}, net) = \max_t\left(NetInsDT(r_{ij}, net, t)\right)$$

$$MaxNetInsDTTime(r_{ij}, net) = t_{\max} \mid \forall t: MaxNetInsDTValue(r_{ij}, net, t_{\max}) \geq MaxNetInsDTValue(r_{ij}, net, t)$$

$$MaxNetInsDTMsgs(r_{ij}, net) = \left(msg_{\max 1}, \ldots, msg_{\max n}\right) \left| \begin{array}{l} msg_{\max i} \in r_{ij} \wedge \\ msg_{\max i}.start \leq MaxNetInsDTTime(r_{ij}, net) \leq msg_{\max i}.end \wedge \\ net \in getNetworkPath(msg_{\max i}.s.n, msg_{\max i}.r.n) \end{array} \right.$$

Else If $SD_i = SD(RTC$

If $Msgs(RTC) \in r_{ij}$ then

$$MaxNetInsDTValue(r_{ij}, net) = \max_t\left(NetInsDT(r_{ij}, net, t) \mid \min_{m \in Msgs(RTC)}(m.start) \leq t \leq \max_{m \in Msgs(RTC)}(m.end)\right)$$

$$MaxNetInsDTTime(r_{ij}, net) = t_{max} \left| \begin{array}{l} \forall t: MaxNetInsDTValue(r_{ij}, net, t_{max}) \geq MaxNetInsDTValue(r_{ij}, net, t) \wedge \\ \min_{m \in Msgs(RTC)}(m.start) \leq t \leq \max_{m \in Msgs(RTC)}(m.end) \end{array} \right.$$

$$MaxNetInsDTMsgs(r_{ij}, net) = \left(msg_{max1}, \ldots, msg_{maxn}\right) \left| \begin{array}{l} msg_{maxi} \in Msgs(RTC) \wedge \\ msg_{maxi}.start \leq MaxNetInsDTTime(r_{ij}, net) \leq msg_{maxi}.end \wedge \\ net \in getNetworkPath(msg_{maxi}.s.n, msg_{maxi}.r.n) \end{array} \right.$$

Else

$$MaxNetInsDTValue(r_{ij}, net) = 0$$

$$MaxNetInsDTTime(r_{ij}, net) = null$$

$$MaxNetInsDTMsgs(r_{ij}, net) = null$$

1.1.1.2. Among all DCCFPs, of SD_i's, find the DCCFP with maximum stress value:

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

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

2. Choose the ISDS (Independent-SD Set) and network with maximum stress on *RTC*
   2.1. For each network *net* in *Nets(RTC)*
      2.1.1. For each $ISDS_i$ such that $SD(RTC) \in ISDS_i$

$$MaxNetInsDTValue(ISDS_i, net) = \sum_{\forall SD \in ISDS_i} MaxNetInsDTValue\left(MaxNetInsDTDCCFP(SD, net), net\right)$$

   2.2. For each network *net* in *Nets(RTC)*
      2.2.1. For each $ISDS_i$ such that $SD(RTC) \in ISDS_i$

Find the maximum $MaxNetInsDTValue(ISDS_i, net)$ and refer to it the selected ISDS and network as $ISDS_{max}$ and net_{max}.

3. Schedule SDs in the ISDS with maximum stress ($ISDS_{max}$) in the same way as Step 3 of .
4. Return all $StressTest\ Schedule_i = \left(r_{i\max}, ar_{i\max}\right)$ and net_{max} .as outputs.

Algorithm 5-Derivation of RT constraint-oriented stress test requirements with network instant data traffic flavor targeted to a SD-level RT constraint *RTC*.

Function *Nets(rtc: a RT constraint):Set of networks*

1. Output=Empty set
2. For each message *m* in *Msgs(RTC)*
   2.1. Find maximum stress DCCFP of each SD for *net*
   2.2. Output= Output ∪ getNetworkPath(m.sender.node, m.receiver.node)
3. Return Output

Algorithm 6-Pseudo code of function *Nets(RTC).*

9.13.2 MIOD-Level RTCOST

MIOD-level RT constraint-oriented stress test algorithms can be devised similarly as the SD-level RTCOST algorithms, since MIOD-level RT constraints are similar to SD-level constraints and the former ones are only one level higher than the later ones (Section 5.6).

We present the algorithm for *MIODRTStressNetInsDT(RTC)* test flavor, where *RTC* is a given MIOD-level RT constraint. The algorithm can be derived from *StressNetInsDT* global stress test flavor (Section 9.11.2) and is shown in Algorithm 7. Algorithm for the other stress test strategies can be derived in similar ways, as were presented for the global stress test algorithms in Section 9.11.

5. For each network *net* in *Nets(RTC)*
   5.1. Find maximum stress DCCFP of each SD for *net*
      5.1.1. For each SD_i where SD_i
         5.1.1.1. For each DCCFP r_{ij} of SD_i

$$MaxNetInsDTValue(r_{ij}, net) = \max_t \left(NetInsDT(r_{ij}, net, t) \right)$$

$$MaxNetInsDTTime(r_{ij}, net) = t_{\max} \mid \forall t : MaxNetInsDTValue(r_{ij}, net, t_{\max}) \geq MaxNetInsDTValue(r_{ij}, net, t)$$

$$MaxNetInsDTMsgs(r_{ij}, net) = \left(msg_{\max 1}, \ldots, msg_{\max n} \right) \left| \begin{array}{l} msg_{\max i} \in r_{ij} \wedge \\ msg_{\max i}.start \leq MaxNetInsDTTime(r_{ij}, net) \leq msg_{\max i}.end \wedge \\ net \in getNetworkPath(msg_{\max i}.s.n, msg_{\max i}.r.n) \end{array} \right.$$

         5.1.1.2. Among all DCCFPs, of SD_i's, find the DCCFP with maximum stress value:

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

         If no DCCFP in SD_i is found with the above criteria, the function returns null.
6. Choose the ISDS (Independent-SD Set) and network with maximum stress on *RTC*
   6.1. For each network *net* in *Nets(RTC)*
      6.1.1. For each $ISDS_i$ such that $SD_{RTC} \in ISDS_i$ and $SD_{RTC} \in SDs(RTC)$

$$MaxNetInsDTValue(ISDS_i, net) = \sum_{\forall SD \in ISDS_i} MaxNetInsDTValue\left(MaxNetInsDTDCCFP(SD, net), net \right)$$

   6.2. For each network *net* in *Nets(RTC)*
      6.2.1. For each $ISDS_i$ such that $SD_{RTC} \in ISDS_i$ and $SD_{RTC} \in SDs(RTC)$

      Find the maximum $MaxNetInsDTValue(ISDS_i, net)$ and refer to it the selected ISDS and network as $ISDS_{max}$ and net_{max}.
7. Schedule SDs in the ISDS with maximum stress ($ISDS_{max}$) in the same way as Step 3 of .
8. Return all $StressTest\ Schedule_i = \left(r_{i\max}, ar_{i\max} \right)$ and net_{max}.as outputs.

Algorithm 7-Derivation of RT constraint-oriented stress test requirements with network instant data traffic flavor targeted to a MIOD-level RT constraint *RTC*.

Msgs(RTC) and *Nets(RTC)* are utility functions as described in Section 9.13.1. *SDs(RTC)* is similar to *SD(RTC)*, described above, however it returns the set of SDs a RT constraint is bound to. For example, Considering the part of a MIOD in Figure 61, *SDs(RTC)* will return {*SD1*, *SD2*}.

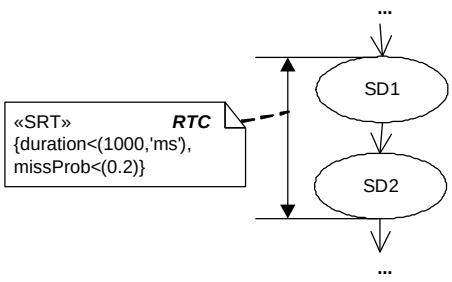


Figure 61-An example of a MIOD-level RT constraint (only part of the MIOD is shown).

Since a MIOD-level RT constraint does not apply to individual messages in a SD, therefore we do not need a modified search among DCCFPs of *SD(RTC)*, as done in Step 1.1.1.1 of Algorithm 5 (*SDRTStressNetInsDT*). Two modifications made while deriving Algorithm 7 from *StressNetInsDT* (Section 9.11.2) are:

1. The loop in Step 1 to go over networks in *Nets(RTC)*.
2. Limiting ISDSs to only those which include at least one of the SDs in *SDs(RTC)* (Step 2.1.1 of Algorithm 7).

9.13.3 The Feasibility of Full Automation

We investigated the feasibility of a method to determine the order of different stress test strategies in terms of importance, given a RT constraint. However, finding such order of stress test strategies (such as *NetInsDT*, *NetInsMT*, *NodInInstDT*, or *NodBiInstMT*) is not possible, since different network/nodes might exhibit network traffic failures in different data/message traffic thresholds compared to others. The best practice is to apply all possible stress test strategies (they are only 16 according to Section 9.5) for a given RT constraint.

9.14 Automating the Derivation Process of Test Elements

We saw in Algorithm 2 that test elements are parts of the input to our stress testing technique, where the rest of the steps are done automatically. We discuss here if the test elements 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. Note that this automated process can be applied to both global stress test and the RT constraint-oriented stress test techniques.

We discussed in Section 5.6 that RT constraints are modeled by two stereotypes: *SRT* (soft) and *HRT* (hard). SRT constraints have an upper bound probability (*missProb*) up to which they can be missed in a series of executions. Besides, we assumed that a criticality value is assigned to each HRT constraint. Criticality was defined as the degree to which the consequences of missing a hard deadline are unacceptable. The closer the criticality of a HRT constraint to one, the more severe will be the consequences of missing it. For example, if missing a HRT constraint may cause life-threatening situations, it would be better to assign criticality=1 for it. Conversely, if the cost of missing 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 should have a lesser value of criticality. Note that, with the above definitions, there is similarity in the concepts of HRT constraints with low criticality and SRT constraints.

As an overview, we present the heuristics of our automated derivation process of test elements. Further details are given next.

1. HRT constraints should be verified before SRT constraints. In other words, system components (networks or nodes) associated with the HRT constraints should be stress tested before those components related with the SRT constraints. The association relationship will be described below.
2. Among HRT constraints, the constraints with higher criticality should be verified first.
3. Among SRT constraints, the constraints with lower missing probability (*missProb*) should be verified first.
4. For each HRT or SRT constraint, the total instant/interval data/message traffic for each of its associated networks and nodes can be calculated and sorted in the descending order. The suggested order to generate test elements is in the order of sorted components.

A RT constraint (hard or soft) is said to be associated with a system component (networks or nodes), if the network behavior of the component affects the duration of the constraint. We derive such associations of a constraint from MIOD or SD if the constraint is MIOD-level or SD-level, respectively. For example, consider the MIOD-level constraint *SRTC1* and SD-level constraints *HRTC1* and *SRTC2*, shown in Figure 62.

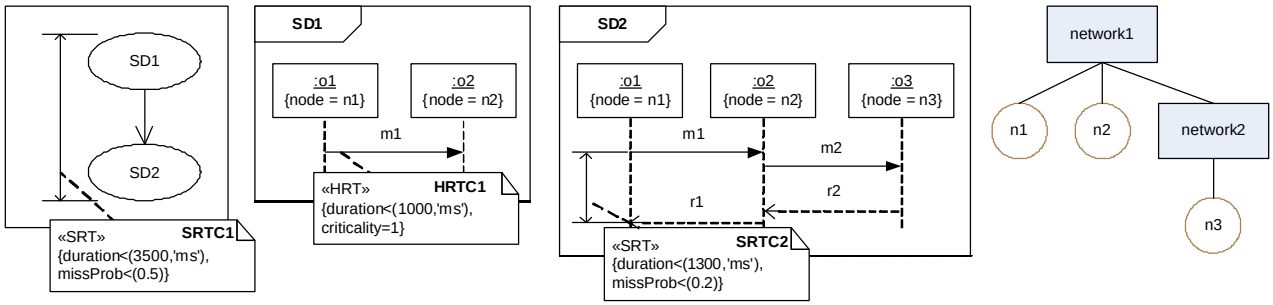


Figure 62-Association of RT constraints in SD and MIOD levels.

According to our definition of "association" between RT constraint and network/nodes, the set of associated network/nodes for constraints *SRTC1*, *SRTC2* and *HRTC1* is shown in Table 7. To derive the associated networks of a constraint, we use the *networkPath* function to find the list of networks connecting two nodes in a NIT. For example the associated networks of constraint *SRTC2* are: *network1* and *network2*, since the path from node *n2* to *n3* goes through these networks in the NIT of Figure 62-(d).

RT constraint	Associated nodes and networks
SRTC1	*n1, n2, n3, network1, network2*
SRTC2	*n1, n2, n3, network1, network2*
HRTC1	*n1, n2, network1*

Table 7-Associated nodes and networks of the RT constraints in Figure 62.

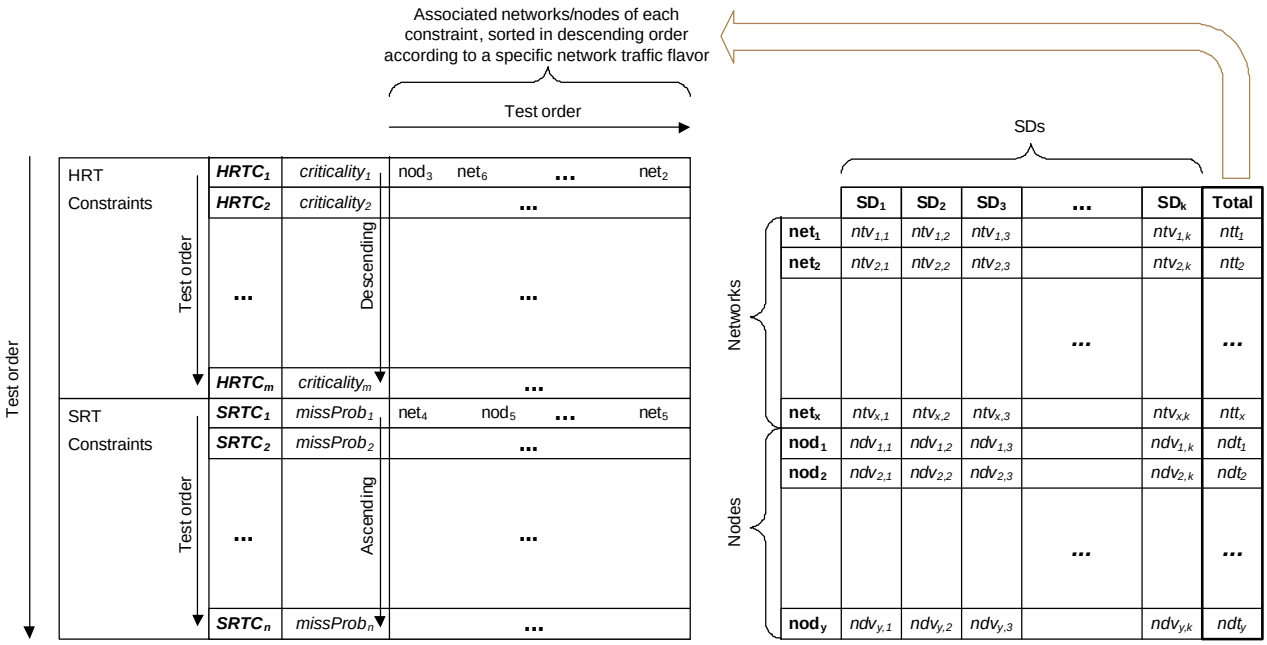
Now, let us return back to the heuristic to automate the derivation process of test elements. To better illustrate the idea, the list of heuristic can be depicted graphically in Figure 63.

Figure 63 shows the general procedure to derive the order of test elements. Suppose this SUT has *m* HRT constraints ($HRTC_1$, …, $HRTC_m$), *n* HRT constraints ($HRTC_1$, …, $HRTC_m$), *x* networks, *y* nodes and *k* SDs. The table shown in Figure 63-(a) suggests an order of constraints to test. For each constraint in this order, the order of elements to test

the associated set of networks/nodes can be derived as discussed above. To realize the last heuristics, we use a matrix, referred to as *SD-Network Usage Matrix (SDNUM)*, an example of which is shown in Figure 63-(b). In a SDNUM, each row corresponds to a network or a node in the SUT. SDNUM rows are divided in to groups: networks and nodes. Columns correspond to SDs of the SUT, where the last column is the summation of all the values in a row, i.e.:

$$ntt_i = \sum_{j=1}^{k} ntv_{i,j}$$

$$ndt_i = \sum_{j=1}^{k} ndv_{i,j}$$



(a) Order of test elements (b) SD Network Usage Matrix

Figure 63-Heuristics to Automate the Process of Test Elements Derivation.

Each element of a SDNUM, corresponding to a SD *M* and a component c (network or node), is equal to the value of a network traffic function which is entailed by executing SD *M* on the component c. Based on the discussions in Section 9.5, all stress functions can used. This idea can be formalized with the BNF shown in Figure 64.

$ntv_{i,j} ::= NetInsFunction(SD_j, net_i) \mid NetPerFunction(SD_j, net_i)$

$ndv_{i,j} ::= NodInsFunction(SD_j, nod_i) \mid NodPerFunction(SD_j, nod_i)$

$NetInsFunction(SD_j, net_i) ::= MaxNetInsDTValue(MaxNetInsDTDCCFP(SD_j, net_i)) \mid MaxNetInsMTValue(MaxNetInsMTDCCFP(SD_j, net_i))$

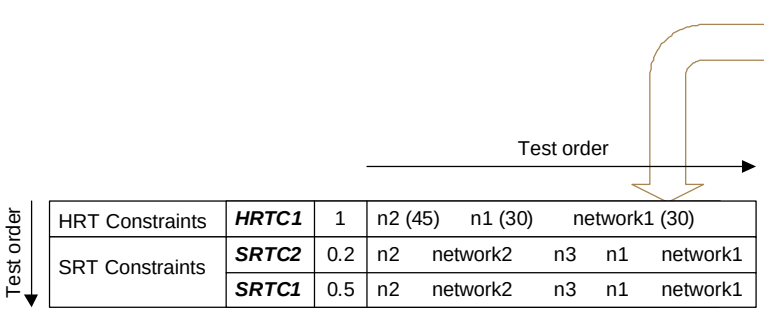
$NodInsFunction(SD_j, nod_i) ::= MaxNodBiInsDTValue(MaxNodBiInsDTDCCFP(SD_j, nod_i)) \mid MaxNodBiInsMTValue(MaxNetInsMTDCCFP(SD_j, nod_i))$

$NetPerFunction(SD_j, net_i) ::= NetUDT(MaxNetPerDTDCCFP(SD_j, net_i)) \mid NetUMT(MaxNetPerMTDCCFP(SD_j, net_i))$

$NodPerFunction(SD_j, nod_i) ::= NodBiUDT(MaxNodBiPerDTDCCFP(SD_j, nod_i)) \mid NodBiUMT(MaxNodBiPerMTDCCFP(SD_j, nod_i))$

Figure 64-BNF for the elements of SD-Network Usage Matrix (SDNUM).

For example, we show here how the test order of test elements corresponding to Figure 62 can be derived. We derived the set of associated network / nodes for constraints *SRTC1*, *SRTC2* and *HRTC1* of the example of Figure 62 in Table 7. Let us assume functions *MaxNetInsDTValue* and *MaxNodBiInsDTValue* are chosen for the test elements derivation process. Let us further assume that the values of these functions for each pair of SDs and components (networks or nodes) are calculated as shown in Figure 65-(b). Using these values and the automatic test elements derivation process strategy, stated above, the order of test elements is shown in Figure 65-(a). The only HRT constraint *HRTC1* is ordered before the SRT constraints *SRTC1*, *SRTC2*. SRT constraints are ordered in ascending order of their missing probabilities. For each constraint, the set of its associated networks / nodes are ordered in the descending order of the corresponding total function value, extracted from the SDNUM in Figure 65-(b).



(a) Order of test elements

	SD1	SD2	Total
network1	10	20	**30**
network2	null	35	**35**
n1	10	20	**30**
n2	10	35	**45**
n3	null	35	**35**

(b) SDNUM with functions MaxNetInsDTValue and MaxNodBiInsDTValue

Figure 65-An example showing how the automated test element derivation heuristics works.

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.
- 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 the following. 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 section which will consider *all* three types of constraints (sequential and conditional and arrival-pattern). The ideas of the optimization algorithm in this section are based 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 section. 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 provides a general overview of the formulated optimization problem, which basically adds the arrival-pattern group of constraints to the optimization problem, presented in Section 9.7. Section 10.5 describes the impacts of arrival patterns on various stress test strategies (Section 9.5).

Based on such impacts, we separate instant and interval stress test strategies with arrival patterns, and address them 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 section, 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 below, genetic algorithms will be chosen as the optimization technique type and the optimization problem will be formulated to be solvable by a genetic algorithm in Section 10.7. Section 10.8 presents a variation of the TSSTT to derive interval stress test requirements.

10.1 Types of Arrival Patterns

Arrival-Pattern constraints (APC) relate to 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 [10], *RTarrivalPattern* tagged-values can be used to model the pattern in which a SD is triggered. Five arrival patterns are defined in [10] using the following BNF (Backus-Nauer Form) forms:

- `<bounded> ::= 'bounded', <time-value>, <time-value>`

  Describes a bounded inter-arrival pattern, where the left time value is the minimal interval between successive arrivals and the one on the right is the maximum; both values are expressed using the *RTtimeValue* type. *RTtimeValue* type is another tagged-value which is a general format in the UML-SPT profile [10] for expressing time value expressions, e.g. *(20, ms)*.

  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, 15, 16>, where all times values are in ms, satisfies the arrival pattern.

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

  The BNF describes a bursty inter-arrival pattern, where the time value is the burst interval expressed using the *RTtimeValue* type and the integer identifies 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, 6, 7, 12, 14>, where all times values are in ms, satisfies the arrival pattern.

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

  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 instances.

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

  Describes periodic inter-arrival patterns, where the left time value defines the period and the optional second time value represents the maximal deviation (from the period value). Both values are expressed using the *RTtimeValue* type.

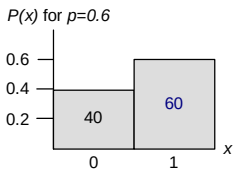
  For example, (*'periodic', (6, ms), (1, ms))* specifies a periodic inter-arrival pattern, where the period and the deviation values are 6 and 1 ms.

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

  Describes a pattern specified by a *Probability Distribution Function (PDF)* defined in *RTtimeValue* in Section 4.2.2 of [10]. The types of PDFs supported are: Bernoulli, binomial, exponential, gamma, geometric, histogram, normal (Gaussian), Poisson, and uniform. Different PDF types are explained below with the corresponding modeling BNFs, the mathematical PDF formulas and an example graph of the PDF.

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

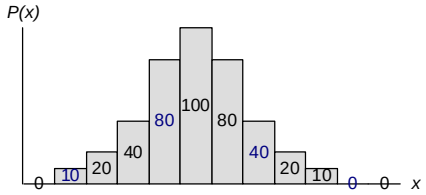
    `<bernoulliPDF> ::= 'bernoulli', <Real>`

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

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

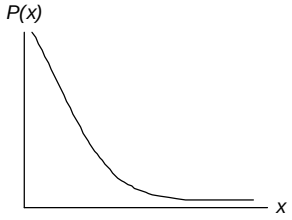
<binomialPDF> ::= 'binomial', <Real>, <Integer>[1]



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

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

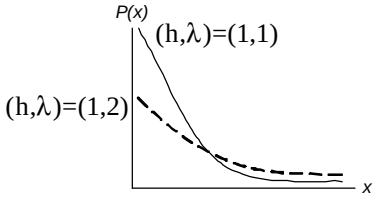
<exponentialPDF> ::= 'exponential', <Real>



$$P(x) = l\, e^{-lx}$$

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

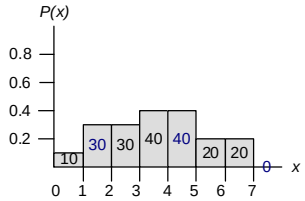
<gammaPDF> ::= 'gamma', <Integer>, <Real>



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

- 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>



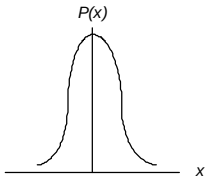
An example:

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

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

[1] This is written in the UML-SPT as <binomialPDF> ::= " 'binomial' ," <Integer>[1], in page 4-33 of [10]. We have altered the BNF to conform to the PDF's mathematical definition.

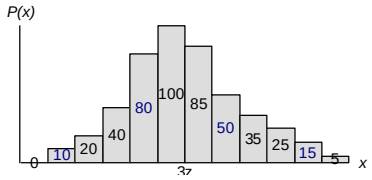
<normalPDF> ::= 'normal', <Real>, <Real>



$$P(x) = \frac{1}{s\sqrt{2p}} e^{\frac{-(x-m)^2}{2s^2}}$$

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

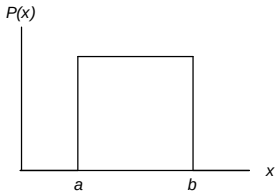
<poissonPDF> ::= 'poisson', <Real>



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

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

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



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

10.2 Analysis of Arrival Patterns

In order to study the arrival patterns and their impact on our test techniques, their timing characteristics 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 possible given the AP.

The pseudo-code, shown in Figure 66, determines if a DCCFP arrival time satisfies an AP. The AP can be any of these {'*bounded*', '*bursty*', '*irregular*', '*periodic*', '*unbounded*'}. The pseudo-code is described in detail next.

```
Function IsAPCSatisfied(arrivalTime, AP)
AP∈{'bounded', 'bursty', 'irregular', 'periodic', 'unbounded'}
1   If (AP='bounded' )
2       If arrivalTime is in one of the intervals of the bounded pattern: Return True
3       Else: Return False
4   If (AP='bursty'): Return Ture
5   If (AP='irregular')
6       If arrivalTime is one of the time values in the AP list: Return True
7       Else: Return False
8   If (AP='periodic')
9       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: Return True
10      Else: Return False
11  If (AP='unbounded'), i.e., AP has a Probability Distribution Function (PDF): Return True
```

Figure 66- Pseudo-code to check if the arrival pattern *AP* is satisfied by an arrival time.

If *AP* is *bounded*, the function 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. This is because if the two values are equal, the arrival pattern will be 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 67 depict the Accepted Time Intervals (ATI) of the arrival pattern. ATI here means the time intervals where an arrival pattern is satisfied.

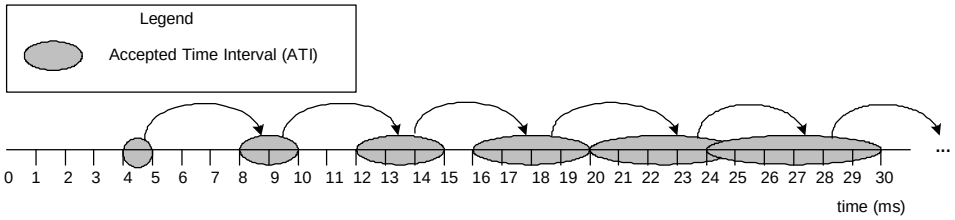


Figure 67-Accepted Time Intervals (ATI) of a *bounded* arrival pattern (*'bounded', (4, ms), (5, ms))*, i.e. *MinIAT=4ms, MaxIAT=5ms*.

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 67 denote how a 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 next arrival time to be in the range of [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 66 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 68 depict the Accepted Time Intervals (ATI) of this arrival pattern.

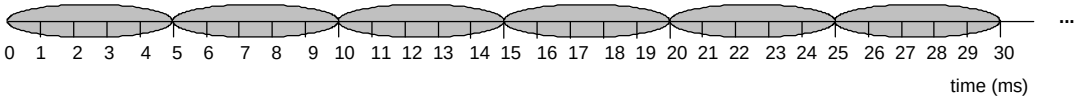


Figure 68-Accepted Time Intervals (ATI) of the bursty arrival pattern (*'bursty', (5, ms), 2)*.

As it can be seen, given a bursty pattern, a single arrival can happen in any time instance, 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 instance (to generate a stress test requirement), we can choose any time instance.

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 time instances specified. 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, and rather time instants, we refer to them as *Accepted Time Points (ATP)*. An example is shown in Figure 69.

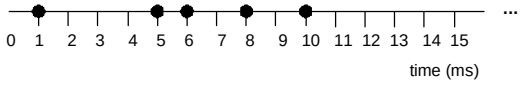


Figure 69-Accepted Time Point (ATP) of the irregular inter-arrival pattern (*'irregular', (1, ms), (5, ms), (6, ms), (8, ms), (10, ms))*.

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 70. Only arrival times in any of the ATIs are accepted.

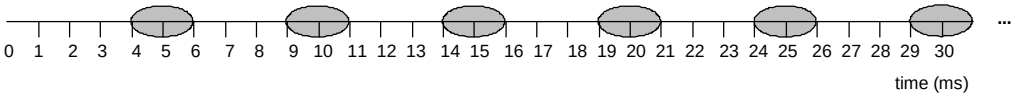


Figure 70-Accepted Time Intervals (ATI) of the periodic interarrival pattern (*'periodic', (5, ms), (1, ms))*.

If the arrival pattern is *unbounded*, the function *IsAPCSatisfied* in Figure 66 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 instance. For example, the PDF of (*'poisson', (5, ms))* arrival pattern is shown in Figure 71.

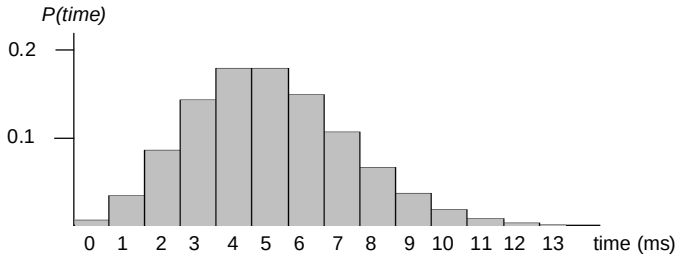
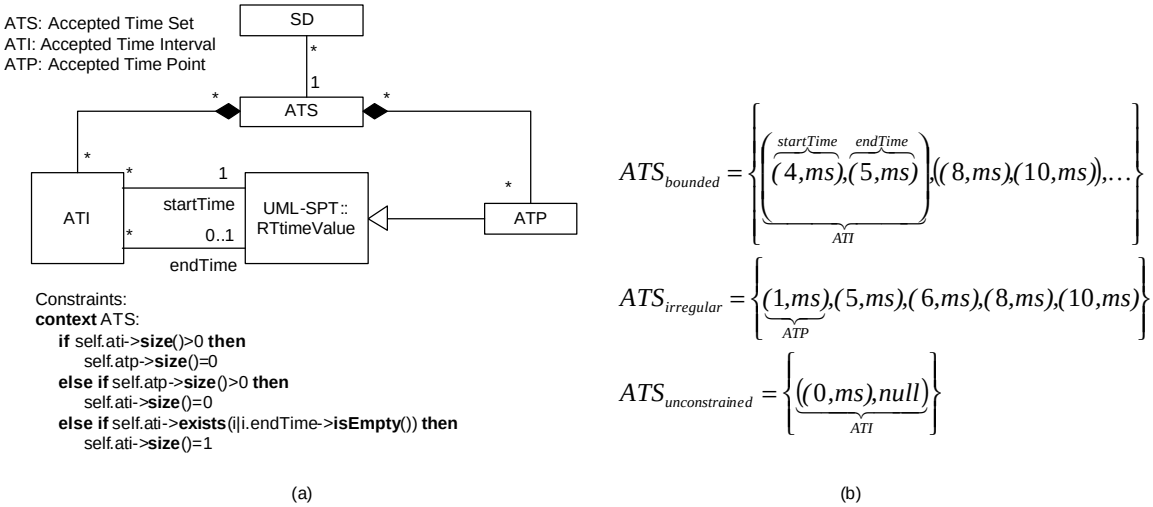


Figure 71-Probability Distribution Function (PDF) of (*'poisson', (5, ms))* arrival pattern.

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 instances 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 each SD. The ATS metamodel in Figure 72-(a) defines the fundamental concepts.



$$ATS_{bounded} = \left\{ \underbrace{(\overbrace{(4,ms)}^{startTime},\overbrace{(5,ms)}^{endTime})}_{ATI}, ((8,ms),(10,ms)), \ldots \right\}$$

$$ATS_{irregular} = \left\{ \underbrace{(1,ms)}_{ATP}, (5,ms), (6,ms), (8,ms), (10,ms) \right\}$$

$$ATS_{unconstrained} = \left\{ \underbrace{((0,ms),null)}_{ATI} \right\}$$

Figure 72-(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 instances 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 the OCL constraints (the first two *if* conditions) in Figure 72-(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. End time of an ATI can be null, which denotes an ATI which has no upper bound (described below in more detail).

Three instances of the metamodel are shown in Figure 72-(b). $ATS_{bounded}$ is the ATS corresponding to the arrival pattern whose timing diagram was shown in Figure 67. $ATS_{irregular}$ corresponds to the arrival pattern in Figure 69. $ATS_{unconstrained}$ 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. Alternatively, we can use ∞ as the end time of the interval. However, as we use the *RTtimeValue* type (from the UML-SPT's for time, we choose the first option (leaving the end time of an interval as null) to represent an unconstrained ATS, as ∞ is not supported in the UML-SPT. Such an ATS has only one ATI which starts from time 0 until ∞ (never ends). This constraints has been formalized by the last (third) *if* condition in the OCL expression in Figure 72-(a). Note that there can be what we refer to as *partly-constrained* ATSs such as:

$$ATS_{partly\text{-}constrained} = \{((0,ms),(3,ms)),((5,ms),)\}$$

where the corresponding SD can be triggered in all times, except interval [3…5ms]. In such an ATS, there are more than one ATI where each ATI's 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, therefore we assume that there will not be any SD with a partly-constrained ATS.

Our GA-based algorithm in Section 10.7 will require intersection of two ATSs. Therefore, in order to find the intersection of two ATSs, we define an intersection operator (∩) for any pair of ATSs. As discussed above, ATSs are sets of time intervals / values. The formula to calculate the intersection of two ATSs is given in Equation 9. 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 9-Intersection of two ATSs.

The membership operators (∈) 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 72-(a), $(1,ms) \hat{I} ATS_{irregular}$.

The output of the formula is the union of three sets: common ATPs, common ATPs in ATIs, and common ATIs of the two ATSs. Common ATPs set is self-explanatory. Common ATPs in ATIs are the set of ATPs in one ATS for which there exists an ATI in the other ATS, such that the point is inside the interval. The formula uses a newly-defined in-range (∠) function between a point and a time interval as:

$$\forall atp \in ATP, ati \in ATI : ati.startTime \le atp \le ati.endTime \Leftrightarrow atp \angle ati$$

Common ATIs of two ATSs are the overlapping intervals in both ATSs. The rationale for finding overlapping (common) intervals of two ATSs is shown in Figure 73.

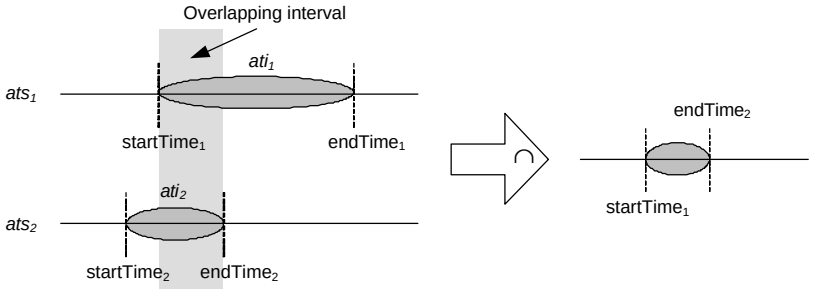


Figure 73- Rationale for finding overlapping (common) intervals of two ATSs.

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 72-(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 ATPs can be the operands of an union operators, yielding an ATS. Two examples, showing how intersections of two ATSs can be calculated using Equation 9, are illustrated in Figure 74.

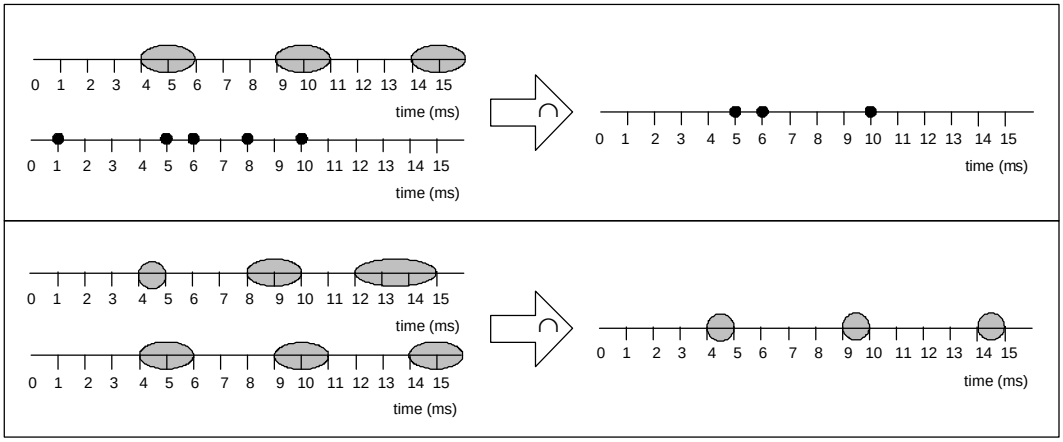


Figure 74-Two examples showing how intersections of two ATSs can be calculated.

Based on the 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 as an Optimization Problem

The problem of generating stress test requirements can be formulated as an optimization problem. The general formulated optimization problem is presented in Figure 75. This formulation has the same objective and variables as the formulated optimization problem of Chapter 9 (Figure 51), however the one here has one more constraint: SD arrival patterns.

Objective: Maximize the traffic on a specified network or node (at a time instant or a period of time)
Variables:
- A subset of DCCFPs (one DCCFP from each SD) with maximum traffic on a specified network or node
- Schedule to run the selected DCCFPs

Constraints:
- Inter-SD sequential and conditional constraints
- **SD arrival patterns**

Figure 75-Formulating the problem of generating stress test requirements as an optimization problem.

10.5 Impact of Arrival Patterns on Stress Test Strategies

We discussed 32 stress test strategies such as: instant stress test towards a node with maximum data (*StressNodInInsDT*) in Section 9.5. 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 impact on TSSTT test strategies, which assume non-constrained start times for DCCFPs. Being more specific, since TSSTT test strategies were grouped into two categories in term of time (duration): instant and interval test strategies, we expect that arrival patterns will impact differently the two groups of strategies. By using the illustrations in Figure 76, we discuss below the impacts of arrival patterns on the two groups of strategies in terms of duration.

10.5.1 Impact on Instant Stress Test Strategies

As we discussed in Section 9.5, 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 76-(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 have arrival pattern constraints. Conversely, consider Figure 76-(b) with the same SDs, but this time, the SDs have arrival pattern constraints, as shown by the ATIs. Due to time constraints from ATIs, SDs can not be scheduled freely in any arbitrary time instants. The heuristics to find maximum possible stress while respecting arrival patterns, in this case, will be to search among the ATI 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 76-(b).

We now discuss the extent to which the impacts of arrival patterns complicate the optimization technique of the instant stress test strategies. As discussed above, deriving instant stress test requirements while considering arrival patterns need 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 in the time axis.

10.5.2 Impact on Interval Stress Test Strategies

We now discuss the impact of arrival patterns on interval stress test strategies. Interval stress test strategies (Section 9.5) aim at increasing the chances of traffic faults by invoking a sequence of SDs, referred to as Concurrent SD Flow Paths (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, we assume that as soon as all the previous SDs were executed (thus satisfying the sequential constraints of a SD), the SD can start its first execution according to its arrival pattern. For example, consider Figure 76-(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:

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

As the CSDFP indicates, SD_5 can start as soon as SD_4 is finished. This is shown in Figure 76-(c), where no SD has arrival patterns and SD_5 can start immediately as soon as SD_4 is finished. However, in the case when SD_5 has an arrival pattern, it cannot start until the first time instant in its ATS to start. Considering the fact that the goal of the interval stress test strategies is to maximize interval stress (maximize possible stress in the shortest possible time of a CSDFP), the impact of arrival patterns on interval stress test strategies will be that the optimization technique can only schedule each SD in its earliest ATS. SDs with arrival patterns can no longer start immediately after all their previous SDs (in the MIOD) have been completed.

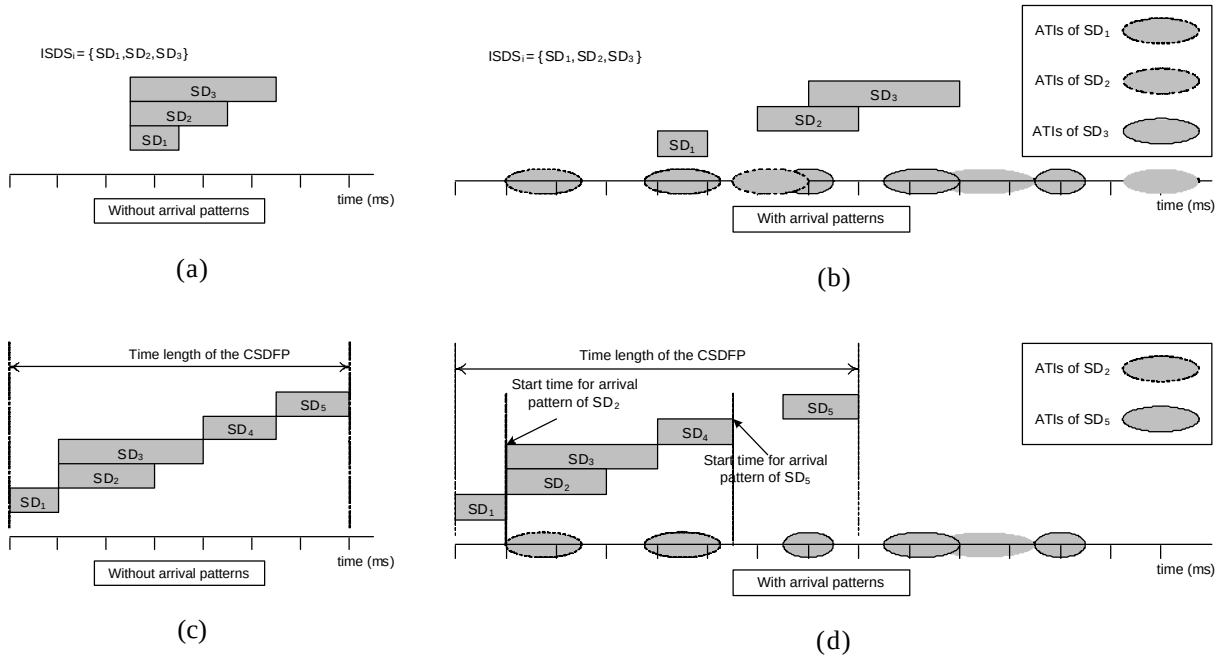


Figure 76-Impact of arrival patterns on instant (a)-(b) and interval (c)-(d) stress test strategies.

We now discuss the extent to which the impacts of arrival patterns complicate the optimization technique of the interval stress test strategies. Considering arrival patterns, 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). Therefore, no complicated global search is required in this case. The time length of CSDFPs will increase in such a case, compared to the case when none of the SD of a CSDFP has an arrival pattern (refer to Figure 76-(c) and Figure 76-(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 77-(a) depicts two (partial) SDs, each having an arrival pattern constraint. We described in Section 10.1 the types of arrival patterns as presented by the UML-SPT profile and we consider in this section. The arrival pattern of SD_1 in Figure 77-(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 77-(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 77-(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 in 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 9.10, <(SD_1, 10 ms), (SD_2, 14ms)>[1] will be a VSDS. On the other hand, if the start time of any SD in a SD schedule does not satisfy its arrival pattern, the schedule is referred to as an *Invalid SD Schedule (IVSDS)*. For example, <(SD_1, 0 ms), (SD_2, 0ms)> will be an IVSDS, considering the arrival patterns in Figure 77. These two schedules are visualized in Figure 77-(c).

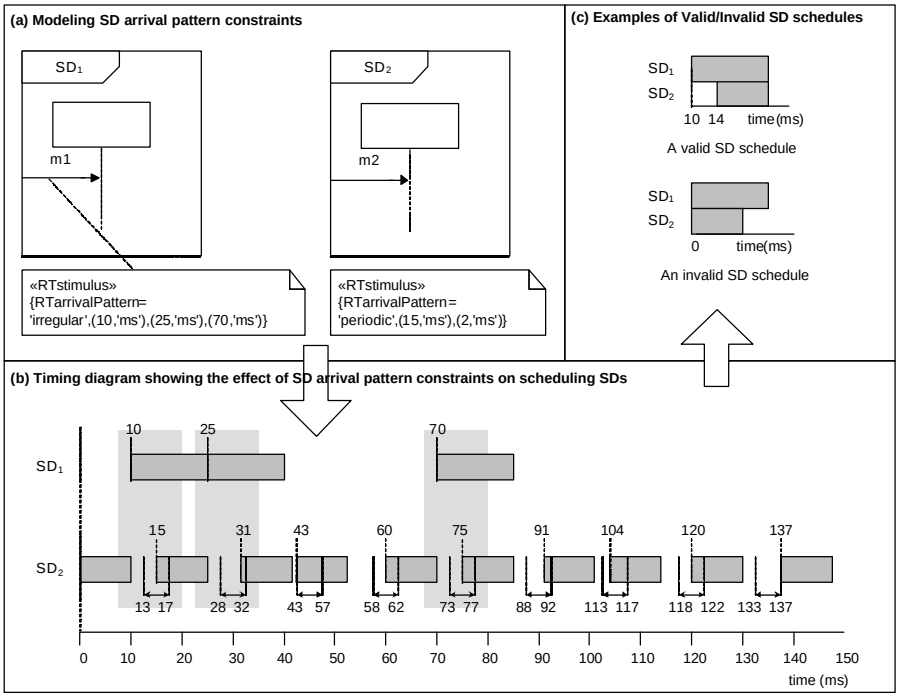


Figure 77-SD arrival pattern constraints.

Now let us discuss why the outputs of the technique in Chapter 9 might not comply with arrival pattern constraints. Suppose that the control flow analysis of SD_1 and SD_2 have yielded two DCCFPs for each: $DCCFP_{1,1}$ and $DCCFP_{1,2}$ for SD_1, $DCCFP_{2,1}$ and $DCCFP_{2,2}$ for SD_1. Using the network traffic formalism presented in Chapter 8, assume the instant network data traffic (*NetInsDT*) function values in Figure 78-(a) for these four DCCFPs, which are entailed on a *SystemNetwork*. By applying the TSSTT technique in to this example to derive stress test requirements, we will get the test requirements in Figure 78-(b). Recalling that the technique first finds the maximum stress messages of each DCCFP, it then finds the DCCFP of each SD with highest maximum stress value. The last step (Step 3 in Algorithm 3) is to schedule DCCFPs such that the maximum stress messages happen at the same time. Note that scheduling DCCFPs is actually scheduling the SDs corresponding to DCCFPs, and having control flow of each SD to follow an specific DCCFP.

By applying the scheduling step of the technique in Chapter 9 to the example in Figure 78, we will get <($DCCFP_{1,2}$, 0 ms), ($DCCFP_{2,2}$, 0ms)> as the stress test schedule, which is equivalent to <(SD_1, 0 ms), (SD_2, 0ms)>. However, as discussed above, such a schedule is an Invalid SD Schedule (IVSDS). This means that

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

triggering SDs with such a schedule violates the SD arrival patterns and may not be allowed in a SUT. Therefore, we see that the outputs of the technique in Chapter 9 might not comply with 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 case of interval stress test strategies, while considering arrival patterns in instant stress test strategies needs global search for an optimum result all across the 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. Section 10.8 presents a variation of the technique in Chapter 9 to derive interval stress test requirements while preserving arrival patterns.

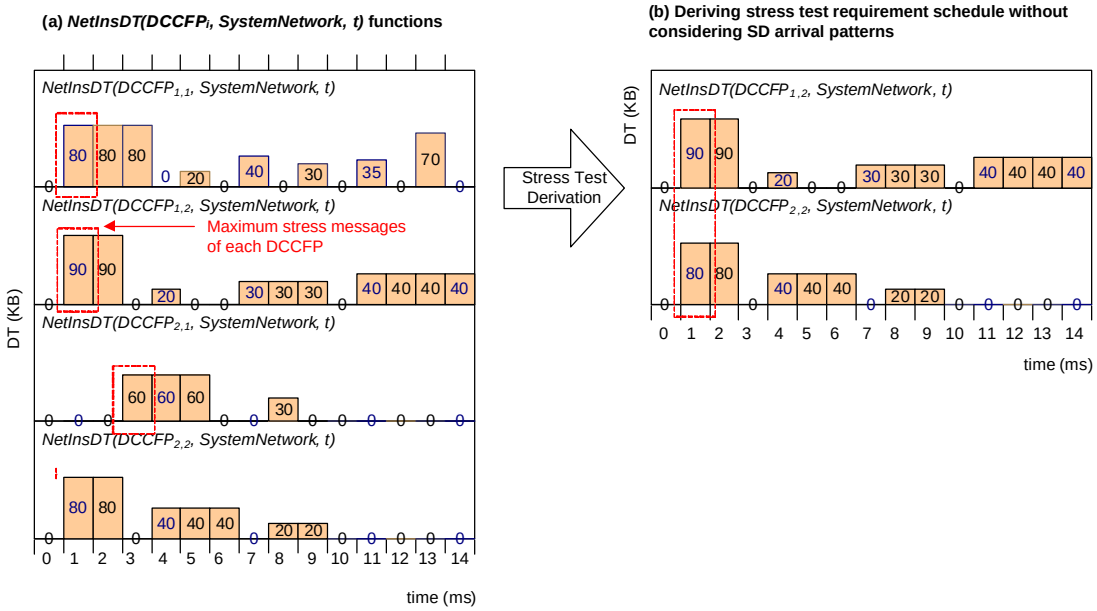


Figure 78-An example stress test requirement which is an invalid schedule, considering SD arrival patterns.

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 flat regions, namely at the intersections of the constraints. Global optimization solutions, also known as meta-heuristic solutions, continually search for better solutions by altering a set of current solutions [72]. The solutions lie on an uneven solution space, characterized by multiple peaks and valleys. These peaks and valleys can result in locally optimum solutions; one where no other solution in the vicinity have better solutions. Global optimization solutions aim at avoiding local optima solutions, reaching global ones instead. Stimulated annealing, tabu search 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 are not fixed. As the number of SDs and DCCFPs increases and their arrival patterns change, the different combinations representing solutions can grow exponentially. As a result, linear

programming cannot be used, as they would lead to combinatorial explosion problem [73]. Furthermore, for the scheduling problem at hand, any 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. A global optimization technique is thus needed.

Genetic Algorithms (GA) are based on concepts adopted from evolutionary theory [74]. GAs involve a search from a population of solutions rather than a single solution like SA. 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 Stimulated 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 the 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 [72].

According to the global optimization literature, GAs and SA are very similar. Some studies, such as [75] indicate that SA outperforms GAs, while others, such as Chardaire et al. [76] claim that GAs produce solutions equivalent to 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 [77, 78]. 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 if they are fitter than previous solutions. Furthermore, GAs lend themselves to parallelism. Because they manipulate whole populations with both mutation and crossover operators, they can readily be implemented on multiple processors. SA, on the other hand, cannot easily run on multiple processors because only one solution is constantly manipulated [77]. Hence, we adopt GA as our optimization technique methodology. An overview on Genetic Algorithms is provided in Appendix A.

10.7 Components of the 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 the problem. We define chromosomes representation in Section 10.7.1. Constraints (in the context of a chromosome) are formulated in Section 10.7.2. Derivation of the initial GA population is discussed in Section 10.7.3. The objective (fitness) function is described in Section 10.7.4. GA operators (crossover and mutation) are finally presented in Section 10.7.5.

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 [74]. We discuss the chromosomes representation of our application in Section 10.7.1.1. The chromosomes' length is described in Section 10.7.1.2.

10.7.1.1 Representation

In our application, the values to be optimized, or the genes of a chromosome, are the selected DCCFPs of SDs and 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 ($r_{i,selected}$, $ar_{i,selected}$), where $r_{i,selected}$ is a selected DCCFP of SD_i, and $ar_{i,selected}$ is the start time of the DCCFP. 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 representation is same as the general form of a stress test requirement (the output of the technique in Chapter 9).

The formal metamodel of chromosomes and genes in our GA algorithm is shown in Figure 79-(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 10.7.5.

Each *Gene* has an association to zero or one DCCFP, and has two attributes *startTime* and *numOfMutipleSDInstances*. *dccfp* is a selected DCCFP of a SD and *startTime* is the time value to trigger *dccfp*, and is of type *RTtimeValue* (defined in the UML-SPT). *numOfMutipleSDInstances* is the number of multiple instances of the SD corresponding to the gene which are allowed to be triggered concurrently ((Section 5.4). Each DCCFP belongs to a SD, whereas each SD can have several DCCFPs. Each SD can be a member of several ISDS. Each ISDS can have one or more SDs.

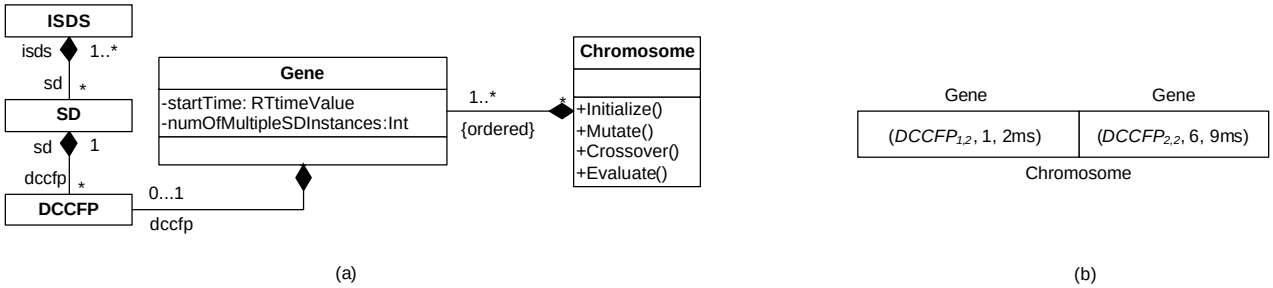


Figure 79-(a): Metamodel of chromosomes and genes in our GA algorithm. (b): Part of an instance of the metamodel.

Part of an instance (considering chromosome and gene only) of the metamodel is depicted in Figure 79-(b). The instance relates to the example in Figure 78. The chromosome is composed of two genes, since there are two SDs in the SUT in Figure 78. $DCCFP_{1,2}$ 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 in our application 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 fixed number of SDs. Note that the numbers of multiple instances of SDs are kept in each gene. 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 corresponding gene is presented as null. Therefore, the chromosome length will not be affected in such cases.

10.7.2 Constraints

Inter-SD and arrival pattern constraints should be satisfied when generating new chromosomes from parents, otherwise, GA *backtracking* procedures [74] should be used to backtrack from a newly generated chromosomes which violates the constraints. Backtracking, however, has its drawbacks: it is deemed expensive as well as 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 rephrase inter-SD and arrival pattern constraints in the context of our GA in this section.

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 be executed 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 the same ISDS. In other words, each chromosome corresponds to one ISDS. We can formulate the above constraint as an OCL expression as presented in Figure 80, which relates to the metamodel in Figure 79-(a).

```
1 Chromosome.allInstances->forAll(c|
2    ISDS.allInstances->exits(isds|
3        c.gene->forAll(g|
4            if (g.dccfp.size()<>0)
5                isds.sd->includes(g.dccfp)
6        )
7    )
8 )
```

Figure 80- 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 81 can be used to determine if the chromosome (the scheduling of its genes) satisfies the Arrival Pattern Constraints (APC) of SDs. The function *IsAPCSatisfied(c :Chromosome)* returns true if all genes of the chromosome satisfy the APCs. The OCL function makes use of the function *IsAPCSatisfied(startTime, AP)*, defined in Section 10.2. Suppose *AP(g.dccfp.sd)* returns the arrival pattern information of the SD associated with the gene *g*.

```
1   IsAPCSatisfied(c:Chromosome)
2       post:
3           if c.gene->exits(g|not IsAPCSatisfied(g.startTime,AP(g.dccfp.sd))
4               result=false
5           else
6               result=true
```

Figure 81-Constraint #2 of the GA (an OCL function).

10.7.3 Initial Population

We discuss in this section the initial population size of our GA and how it is generated. Determining the population size of the GA is challenging [72]. A small population size will cause the GA to quickly converge on a local minimum because it insufficiently samples the parameter space. A large population, on the other hand, causes the GA to run longer in search for an optimal solution. Haupt and Haupt in [74] list a variety of works that suggests an adequate population size. The authors in [74] reveal that the work of De Jong [79] suggests a population size ranging from 50 to 100 chromosomes. Grefenstette et al. [80] recommend a range between 30 and 80, while Schaffer and his colleagues [81] suggest a lower population size: between 20 and 30.

However as discussed in Section 10.7.2.1, each chromosome in our GA corresponds to an ISDS. If the number of chromosome in the initial population is less than number of ISDSs in a system, as we will discuss in Section 10.7.5.1, our crossover operator can not guarantee that all ISDSs are searched. Therefore, the population size we apply is *max(2.numOfISDS,80)*. We choose 80 as it is consistent with most of these findings, and twice the number of ISDSs is because we initialize two chromosomes from each ISDS. In case

one of them disappears due to the crossover operator, the other has a chance to stay and play the role of a parent.

The GA initial population generation process should ensure that both two constraints of Section 10.7.2 are met. The pseudo-code to generate the initial set of chromosomes is presented in Figure 82. As discussed in Section 10.7.2, each chromosome corresponds to an ISDS. Furthermore, our intention is to include all ISDSs in the initial population. Therefore, assuming that ISDSs of a SUT are indexed, line 1 of the pseudo-code chooses the next ISDS in the sequence and the initialization algorithm continues with the selected ISDS to create an initial chromosome.

For each SD in the selected ISDS, lines 2-3 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 4-5). An initial scheduling is done on genes in lines 6-13. 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 6), 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 instance), a random time instance is selected from the intersection set (lines 7-8). All DCCFPs of the genes are then scheduled to this time instance (line 10). For each such gene, line 11 sets the value for the number of multiple instances of the corresponding SD.

If the intersection set is null, it means that the ATSs do not have any overlapping time instance. In such a case, the DCCFP of every gene is scheduled differently, by scheduling it to a random time instance in the ATS corresponding to its SD (lines 15-16). Following the algorithm in Figure 82, we ensure the initial population of chromosomes comply with both constraints of Section 10.7.2.

```
Function CreateAChromosome: Chromosome
c: Chromosome
1   ISDS=next ISDS in the sequence of ISDSs
    // selecting genes (DCCFPs)
2   For all SDi∈ISDS
3       c. genei.dccfp= a random DCCFP from SDi
4   For all SDi∉ISDS
5       c. genei=null
    // initial scheduling of genes (DCCFPs)
6   Intersection=ATS(SD1) ∩ ATS(SD2) ∩… ∩ATS(SDi), where SDi∈ISDS
7   If Intersection≠ϕ
8       Choose a random time instance t_schedule in Intersection
        // schedule all genes' start time to t_schedule
9       For all c. genei≠null {
10          c. genei.startTime= t_schedule
11          c. genei. numOfMutipleSDInstances=SDi. numOfMutipleSDInstances
12      }
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. genei≠null {
15          c. genei.startTime= A random time instance ti in ATS(SDi)
16          c. genei. numOfMutipleSDInstances=SDi. numOfMutipleSDInstances
17      }
18  Return c
```

Figure 82-Pseudo-code to generate chromosomes of the GA's initial population.

In the case when the intersection of SD ATSs is null, one might wonder whether there care still any possibilities to run SDs concurrently to have a 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 83, 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 for example) have non-null ATS intersections, which allow them to be triggered concurrently. This situation can be made possible in a chromosome by our mutation operator (Section 10.7.5.2), since as we will discuss, our mutation operator will shift each of the SD in its ATSs and the GA will then assess the new resulted offspring.

- Another situation when the data-centric messages of a set of SDs with null 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 83 has been triggered in one of its allowed times and it has continued until the ATS intersection of SD_1 and SD_2. In such a case, messages from all three SDs overlap (in time domain) and thus triggering high stress scenarios is possible. Similar to the previous item, this situation can also be made possible in a chromosome by our mutation operator.

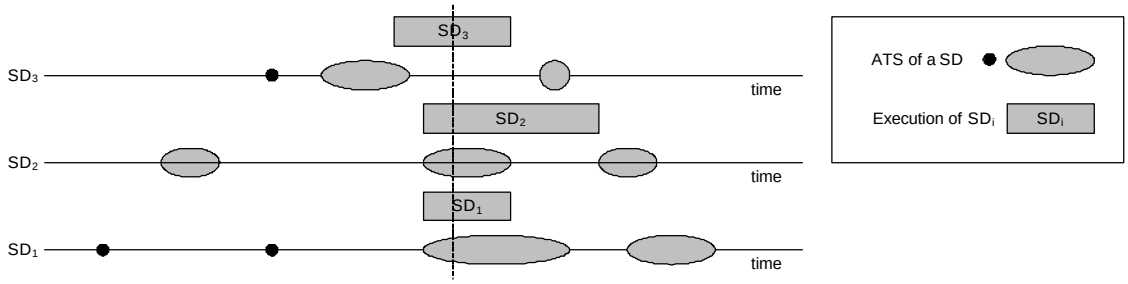


Figure 83-An example where the ATS intersection of all SDs is null, but they can overlap.

10.7.4 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 [72]. 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: *What selection and what schedule of DCCFPs maximize the traffic on a specified network or node (at a 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 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*. Depending on other values for those parameters of a test objective, ISTOF should be measured differently by simply using other network traffic usage functions from the set of functions defined in Section 8.5.

$$ISTOF : Chromosome \rightarrow Real$$

$$\forall c \in Chromosome : ISTOF(c) = \max_{t \in SearchRange} \sum_{g \in Genes(c)} NetInstDT(g.dccfp, net, t).g.numOfMultipleSDInstances$$

$$SearchRange = [\min_{g \in Genes(c)} (g.startTime) \ldots \max_{g \in Genes(c)} (g.startTime + Length(g.dccfp))]$$

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

where the first line indicates that the input and output domains of ISTOF are chromosomes and real numbers. *Genes(c)* is the set of genes in chromosome *c*. *Length(dccfp)* is a function to calculate the time length of a DCCFP. Such calculation was presented in Chapter 9 (Algorithm 4.). *net* is the given network to stress test. *NetInsDT* is the network traffic usage function to measure the instant data traffic (Section 8.5.2.1). The value of the *NetInsDT* function is multiplied by the gene's *numOfMultipleSDInstances* value. This is so because, when multiple instances of a DCCFP are triggered at the same time, the entailed traffic by the all instances of the same DCCFPs at each time instant will be multiplied by the number of them..

The heuristic of the above ISTOF formula is that it tries to find the maximum instant data traffic considering all genes of 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 the ISTOF, let us discuss how the ISTOF of the chromosome in Figure 79-(b) is calculated. The calculation process is shown in Figure 84. The chromosome is given as the input on the left side, where the timed-traffic representations of the genes have also been depicted. The search range is [2ms…20ms]. The ISTOF sums the *NetInsDT* values in this range and finds the maximum value. The output value of the ISTOF is 110 KB.

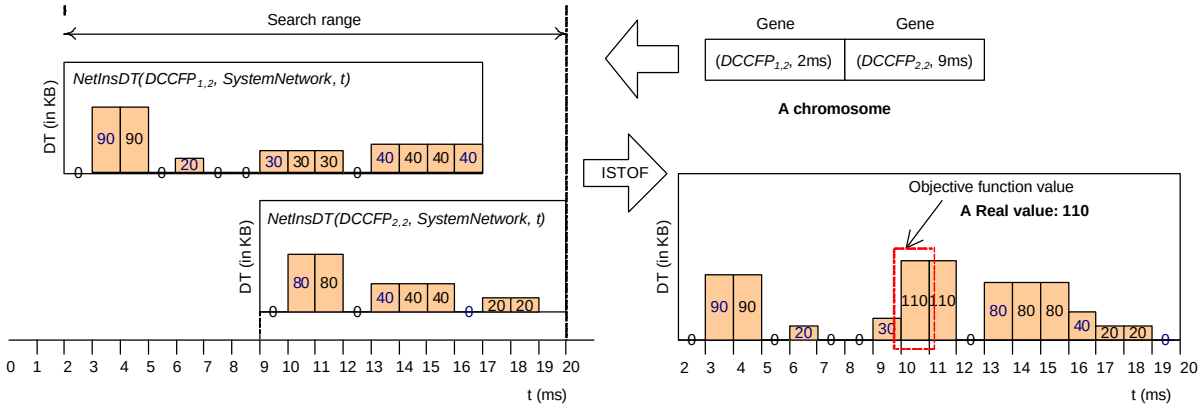


Figure 84-Computing the Instant Stress Test Objective Function (ISTOF) value of a chromosome.

10.7.5 Operators

Operators are the ways GAs explore a solution space [74]. Hence, they must be formulated in such a way that they efficiently and exhaustively explore the solution space. 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.

Formulating operators is rather a difficult task, as genetic operators must maintain *allowability*. In other words, genetic operators must be designed in such a way that if a constraint is not violated by the parents, it will not be violated by the children resulting from the operators [82]. Furthermore, operators should be formulated such that they explore the whole solution space. We define the crossover and mutation operators next.

10.7.5.1 Crossover Operator

Crossover operators aim at passing on desirable traits or genes from generation to generation [74]. Varieties of crossover operators exist, such as sexual, asexual and multi-parent [1]. The former uses two parents to pass traits to the two resulting children. Asexual crossover involves only one parent and produces one child that is a replica of the parent. Multi-parent crossover, as the name implies, combines the genetic makeup of three or more parents when producing offspring. 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 [74]. Pawlowsky dubs this n-point crossover. 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 [83].

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 [84]) is (*n-1*)-point, and is defined *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 from parents with a probability of 50%. This can be visualized as a coin flip. When alternating the genes of each parent, a coin is flipped. Every time the coin lands on heads, the gene is inherited from one parent by the child. Otherwise, the gene is inherited from the other parent.

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 a different chromosome (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 will for sure 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 85. 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).

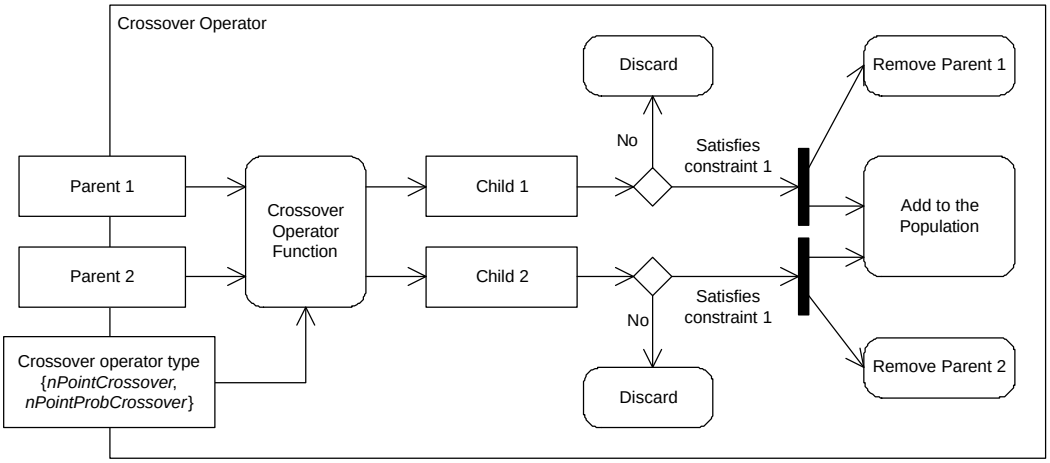


Figure 85-Activity diagram of the crossover operators.

Let us consider the example in Figure 86 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 SDs 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 86.

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 86. The coin flips are assumed to land on heads, tails, tails, tails, and then heads for the five successive fragments for both children. All four generated children conform to constraint 1 (i.e., the SD corresponding to their genes belong to one ISDS ($ISDS_1$)).

The advantages of *nPointProbCrossover* are twofold. It introduces further randomness to the crossover operation. By doing so, it allows further exploration of the solution space. Furthermore, *nPointProbCrossover* is a generalized version of *nPointCrossover*; if the coin flip for each fragment alternates between tails and heads (in that order), we obtain *nPointCrossover*. However, *nPointProbCrossover* has its disadvantages. If the result of all coin flips in a given operation is always tails or always heads, the resulting children are replicas of the parents, with no alteration occurring. This is never the case with *nPointCrossover*; resulting children are always genetically distinct from their parents.

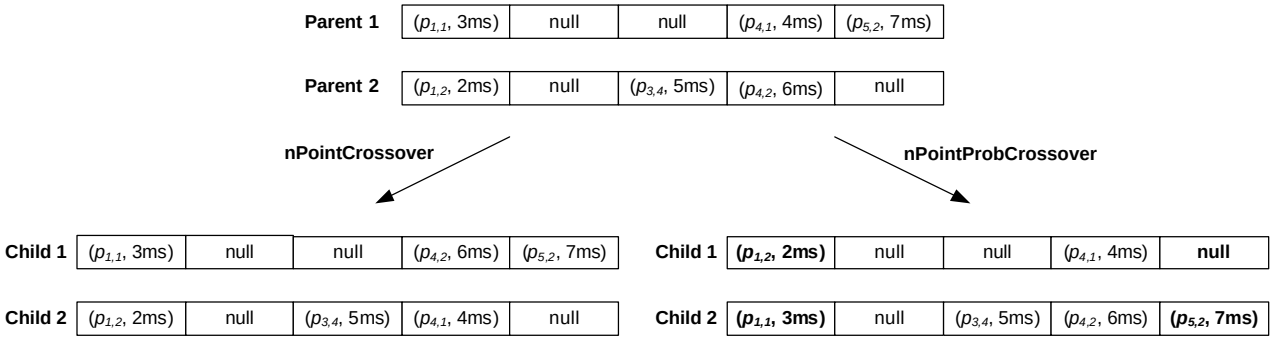


Figure 86-Two example uses of the crossover operators.

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 [74]. De Jong [79] concluded that a desirable crossover rate should be about 60%. Grefenstette et al. [80] 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%.

10.7.5.2 Mutation Operator

Mutation aims at altering the population to ensure that the genetic algorithm 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 two mutation operators that mutate a non-null gene (a gene with an already assigned DCCFP) in a chromosome by altering either: (1) its DCCFP or, (2) its start time. The mutation operators are referred to as *DCCFPMutation* and *startTimeMutation*, 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 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 on the chromosomes are met (Section 10.7.2).

Since the mutation operators alter non-null genes only, the set of SDs corresponding to a chromosome will not be altered by them. Therefore, there is no way for the altered chromosome to violate constraint 1. However, start times are changed by the mutation operator *startTimeMutation*. Hence it should be made

sure that the arrival pattern constraint (constraint 2) is upheld. 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 two mutation operators can be illustrated as two activity diagrams in Figure 87 and Figure 88.

Mutation rates are critical. Mutation rate is the percentage of chromosomes in a population being selected for a mutation operation. Throughout the GA literature, various mutation rates have been used to transform chromosomes. If the rates are too high, too many good genes of a chromosome are mutated and the GA will stall in converging [74]. Back [85] enumerates some of the more common mutation rates used. The author states that De Jong [79] suggests a mutation rate of 0.001, Grefenstette [80] suggests a rate of 0.01, while Schaffer et al. [81] formulated the expression $1.75/l\sqrt{length}$ (where *?* denotes the population size and *length* is the length of chromosomes) for the mutation rate. Mühlenbein [86] suggests a mutation rate defined by $1/length$. Smith and Fogarty [87] 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.: $1.75/l\sqrt{length}$.

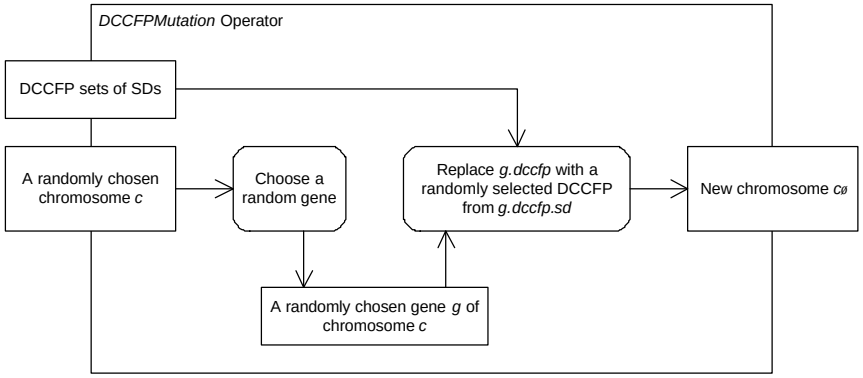


Figure 87-Activity diagram of the *DCCFPMutation* operator.

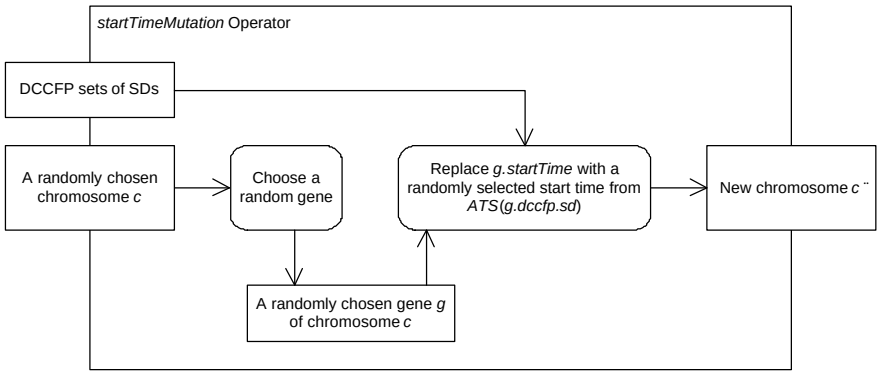


Figure 88-Activity diagram of the *startTimeMutation* operator.

10.8 Interval Stress Test Strategies considering 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). 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 none of the SD of a CSDFP has an arrival pattern (refer to Figure 76-(c) and Figure 76-(d) as an example). We present a modified version of Algorithm 4 in Sections 9.11.2 in Algorithm 8, referred to as *APStressNetIntDT*, which takes into account the arrival patterns.

Where *minAPDuration(aDCCFPS)*, in Step 2.1, 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: $\forall SD \in CSDFP : MaxNetPerDTDCCFP(SD_i, net)$. The pseudo-code of *minAPDuration()* is shown in Algorithm 9 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 89 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.

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

3.1. For each SD_i

3.1.1. For each DCCFP r_{ij} of SD_i / / Finding maximum stress message of each DCCFP

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

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

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

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

where $CCFP(r_{ij})$ is the CCFP corresponding to DCCFP r_{ij}.

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

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

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

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

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

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

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: $\forall SD \in CSDFP : MaxNetPerDTDCCFP(SD_i, net)$.

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

Algorithm 8-Derivation of period stress test requirements for data traffic on a given network, considering arrival patterns (*APStressNetIntDT*).

The idea is formulated in the calculation of the function *earliestAT* (arrival time) in Figure 90 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.

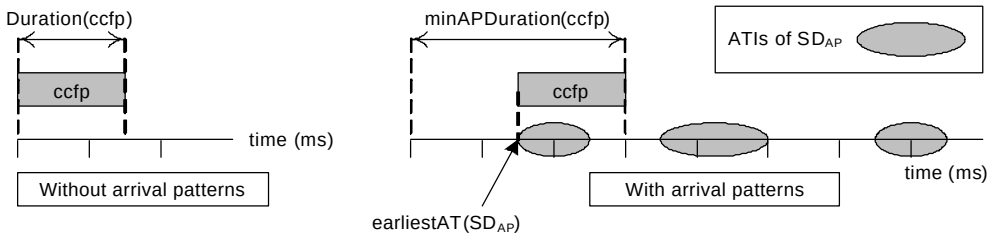


Figure 89-An illustration to show the impact of arrival patterns in the actual duration of a CCFP.

9. Function minAPDuration(*ccfps*: CCFPS): integer
10. if *ccfps* is atomic (only made of one CCFP)
11. return $\max_{\forall m \in ccfps}(m.endTime) + earliestAT(SD_{ccfps}) \mid ccfp = \text{the only CCFP of } ccfps$
12. else if *ccfps* is the serial concatenation of several CCFPSs (i.e., $ccfps = ccfps_1 \cdots ccfps_n$)
13. return minAPDuration ($ccfps_1$)+…+ minAPDuration ($ccfps_n$)
14. else if *ccfps* is the concurrent combination of several CCFPSs (i.e., $ccfps = \begin{pmatrix} ccfps_1 \\ \cdots \\ ccfps_n \end{pmatrix}$)
15. return max(minAPDuration ($ccfps_1$),…, minAPDuration ($ccfps_n$))
16. End Function

Algorithm 9-Calculating the minimum duration of a Concurrent Control Flow Path Sequence (CCFPS), considering arrival patterns.

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

Figure 90- 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:

$$CCFPS = r_1 \begin{pmatrix} r_2 \\ r_3 \end{pmatrix} r_4$$

where each r_i is a CCFP of SD_i. Assume the duration of each of the individual CCFPs is given as in Table 8. Also, the arrival patterns of SD_i are also given in Table 8. The given arrival patterns can be analyzed using the discussions in Section 10.2 and 10.3 to get 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 (i.e. $\max_{\forall m \in CCFPS}(m.endTime)$)
$CCFP_1$	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 8-(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 91. 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 ($\left(\begin{matrix} r_2 \\ r_3 \end{matrix}\right)$), 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. For example $minAPDuration(r_3)$=1000+400=1400ms.

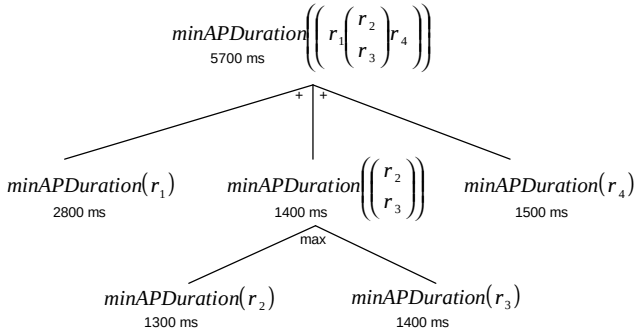


Figure 91-Call tree of the recursive algorithm *minAPDuration* applied to a CCFPS.

Chapter 11

TOOL SUPPORT

To improve automation for the two stress test techniques (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 TSTT 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 [1], an open source C++ library for GAs, in implementing GARUS. An overview on GAlib is presented in Section 11.1. Section 11.2 describes our tool. Section 11.3 reports how we validated test requirements generated by GARUS for a case study.

11.1 GAlib

The library used to implement our GA-based tool was GAlib [1]. 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 92 illustrates the basic GAlib class hierarchy. Only the major classes of the library are shown. For complete class listing, the reader is referred to [1].

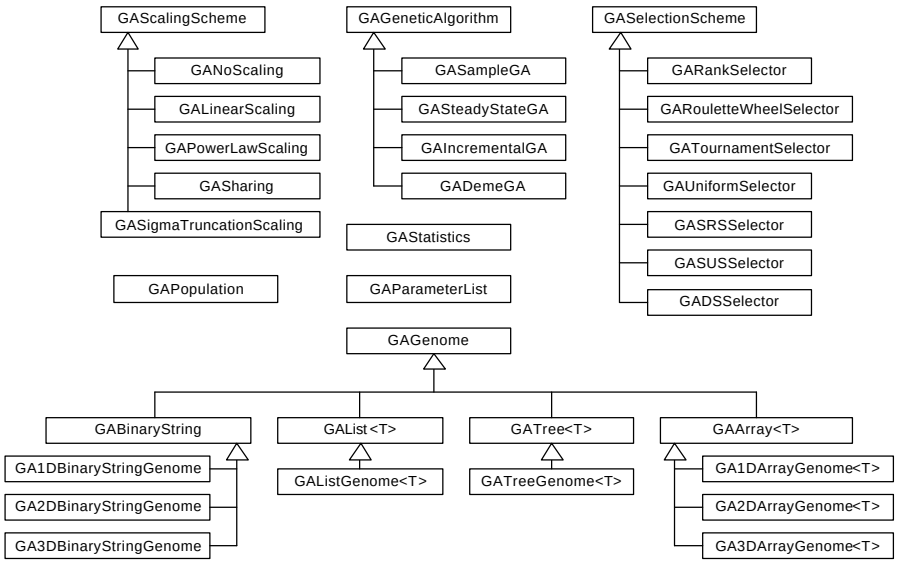


Figure 92-Basic GAlib class hierarchy (adopted from [1]).

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 [1]. 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

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 93. The classes in the class diagram are grouped in two packages: *TestModel* and *GA*. 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 in runtime for a SUT. The connection between the two packages (*TestModel* and *GA*) is via class DCCFP (in the *TestModel* package) and class *GARUSGene* (in the *GA* package).

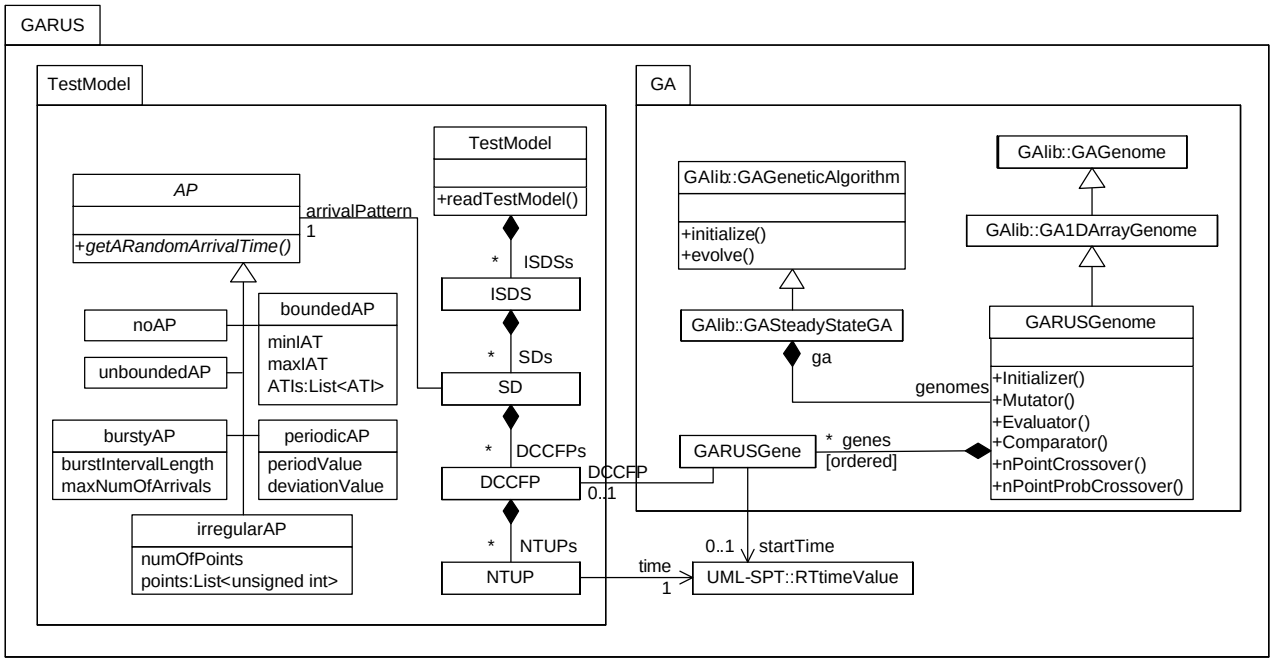


Figure 93-Simplified class diagram of GARUS.

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 94. 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 a ordered set of genes of type *GARUSGene* (refer to the class diagram in Figure 93). 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 10.7.5). When the evolution of *ga* finishes, the tool's task is done and the best individual of *ga* (accessible by `ga.statistics().bestIndividual()`) is saved in the output file, with a format explained in Section 11.2.4.

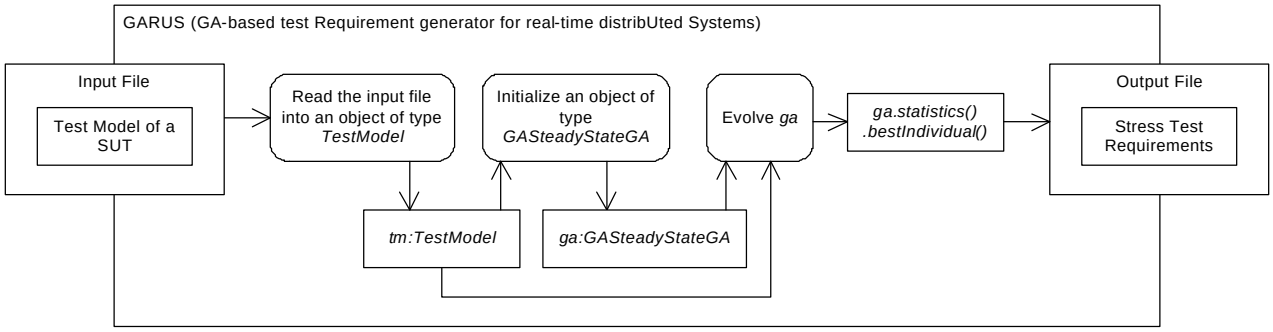


Figure 94-Overview activity diagram of GARUS.

11.2.3 Input File Format

Input file provided to GARUS contains the test model (TM) of a SUT. As it was shown in Figure 8 (overview of our model-based stress test methodology), a TM consists a CFM (including DCCFPs), inter-SD constraint (ISDSs) and network traffic usage patterns.

Referring to Figure 8, stress test parameters are also considered be part of the input to our methodology. 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 network traffic usage analysis (e.g. location, direction, and period).

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 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 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 is in a format to accommodate such filtered TM. 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 95.

The input file format can be best described using an example. An example input file is shown in Figure 96. Different blocks are separated with a gray highlight. The TM starts with a block of two ISDSs *ISDS0* and *ISDS1* (*ISDSsBlock* in Figure 95). For example, *ISDS0* consists of three SDs: *SD0*, *SD1*, and *SD2*.

The second block of the input file is SDs (*SDsBlock* in Figure 95). 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*.

$$inputFileFormat ::= ISDSsBlock \;\; SDsBlock \;\; SDAPsBlock \;\; DCCFPsBlock$$

$$ISDSsBlock ::= nISDSs \;\; ISDS_1 \ldots ISDS_{nISDSs}$$

$$ISDS_i ::= ISDSName_i \;\; nSDsInISDS_i \;\; SDName_1 \ldots SDName_{nSDsInISDS_i}$$

$$SDsBlock ::= nSDs \;\; SD_1 \ldots SD_{nSDs}$$

$$SD_i ::= SDName_i \; nMultipleInstances_i \;\; nDCCFPsInSD_i \;\; DCCFPName_1 \ldots DCCFPName_{nDCCFPsInSD_i}$$

$$SDAPsBlock ::= SDAP_1 \ldots SDAP_{nSDs}$$

$$SDAP_i ::= SDName_i \;\; APType_i \;\; APParameters_i$$

$$APType_i ::= no\_arrival\_pattern \mid periodic \mid bounded \mid irregular \mid bursty \mid unbounded$$

$$APParameters_i ::= \begin{cases} \in & ;\text{if } APType_i \in \{no\_arrival\_pattern, bursty, unbounded\} \\ periodValue_i \;\; deviationValue_i & ;\text{if } APType_i = periodic \\ minIAT_i \;\; maxIAT_i & ;\text{if } APType_i = bounded \\ nArrivalPointsInAP_i \;\; APoint_1 \ldots APoint_{nArrivalPointsInAP_i} & ;\text{if } APType_i = irregular \end{cases}$$

$$DCCFPsBlock ::= \underbrace{DCCFP_1 \ldots DCCFP_{nDCCFPs}}_{nDCCFPs = \sum_{\forall SD_i} nDCCFPsInSD_i}$$

$$DCCFP_i ::= DCCFPName_i \;\; nNTUPPsInDCCFP_i \;\; NTUPP_1 \ldots NTUPP_{nNTUPPsInDCCFP_i}$$

$$NTUPP_i ::= (\; time_i \;\; value_i \;)$$

Figure 95-GARUS input file format.

The third block is SD Arrival Pattern (AP) - (*SDAPsBlock* in Figure 95). 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.

```
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 )
```

Figure 96-An example input file of GARUS.

The last 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 96, 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 NTUP (Network Traffic Usage Pattern). The format for specifying NTUP of a DCCFP is described next. As discussed in Section 8.5, the NTUP 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 NTUP 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 *NTUPP (Network Traffic Usage Pattern Point)*. For example, NTUPPs of the NTUP in Figure 97 are: (1, 90), (3, 40), (4, 40), (8, 30), and (12,

50). For example, in the input file in Figure 96, p41 has two NTUPPs: (4, 20) and (7, 4). The "," symbol between time and traffic values is eliminated in the input file to ease the parsing process.

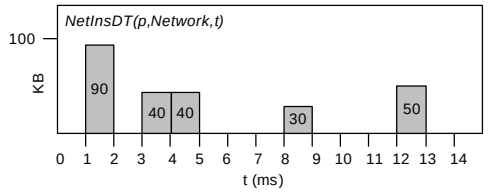


Figure 97-An example NTUP of a DCCFP.

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, standard GAlib statistics are also exported to the output file. GAlib standard statistics include number 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 98-(a). An example set of stress test requirements is presented in Figure 98-(b), which is generated by GARUS for the input file in Figure 96.

SD	DCCFP	start time
-------	-------------	------
$SDName_1$	$SelectedDCCFPName_1$	$startTime_1$
…	…	…
$SDName_{nSDs}$	$SelectedDCCFPName_{nSDs}$	$startTime_{nSDs}$

ISTOF = a float value

Max. stresstime = 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 98-(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. Refer to Section 9.10 for the formalized representation of a stress test requirement. For example, the example in Figure 98-(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 DCCFPs have been specified to be triggered from *SD3* and *SD4*. This is because a set of stress test requirements corresponds to an ISDS in a SUT, and as shown in Figure 96, the SUT has two ISDSs, where *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 to be triggered, due to the arrival pattern of its corresponding SD.

11.3 Validation of Test Requirements Generated by GARUS

GARUS outputs the maximum traffic value and time by triggering SDs according to the given stress schedule. 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 the maximum traffic time for the SUT specified by the input file in Figure 96 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 in at least four ways:

1. *Satisfaction of ATSs by start times of DCCFPs in the generated stress test requirements*: 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*: 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 the correctness of GA results. 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:* 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*: 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 [88]. 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.

Using the above four metrics, we analyze the stress test requirements generated by GARUS using an example: the input file in Figure 96. To assess the variability of the GA's outputs, it was run 1000 times. The variability in the objective function and start times as well as detailed information for the first five runs are reported in Table 9. The results from the entire 1000 runs are further discussed in Section 11.3.3, where we discuss the repeatability of our GA. As a time complexity indicator, the average execution time over all the runs, by running GARUS on an 863MHz Intel Pentium III processor with 512MB DRAM memory, was between 6 (minimum) and 10 seconds (maximum).

Run #	Generation #	Mean	Max (ISTOF)	Min	Deviation	Best individual			
1	0	36.74	55	30	7.95	SD	DCCFP	start time	
	10	44.47	58	38	8.04	----	----	----------	
	20	52.46	61	41	8.44	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	7.01	SD4	p42		2
	70	82.03	88	72	8.95				
	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	7.82	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	7.45	SD1	none		
	40	64.70	73	59	7.47	SD2	p21		8
	50	72.52	76	62	7.44	SD3	p31		5
	60	80.50	82	80	2.39	SD4	p44		8
	70	81.34	84	80	3.78				
	80	83.78	86	80	5.58	ISTOF=92			

	90	91.64	92	80	2.05	Max stress time=12			
	100	92.00	92	92	0.00				
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	8.04	SD	DCCFP	start time	
	10	45.37	58	37	8.94	----	----	----------	
	20	55.14	60	43	9.21	SD0	p05		15
	30	66.63	69	52	7.08	SD1	none		
	40	73.29	78	70	4.22	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 9-Summary of GARUS results.

11.3.1 Satisfaction of ATSs by Start Times of DCCFPs in the Generated Stress Test Requirements

Our first validation check is 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 of Figure 96. Consistent with discussions in Section 10.3, they are shown in Figure 99.

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 9 has been depicted in Figure 99. 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 instances 10, none, 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 99, 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 10.7.5.2). 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 93 for details.

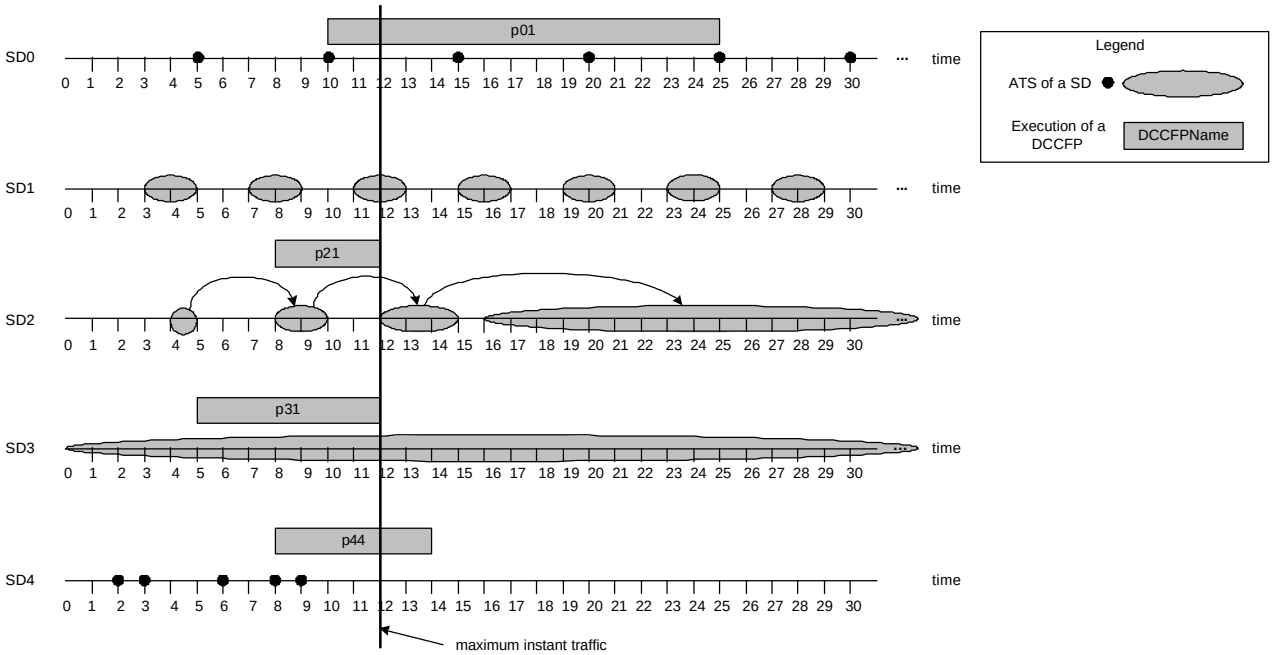


Figure 99-ATSs of the SDs in the TM in Figure 96, and a stress test schedule generated by GARUS.

11.3.2 Checking of ISTOF Values

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 NTUPs of the DCCFPs in the test model of Figure 96 such that one DCCFP of each SD gets a much higher peak value in its NTUP. The modified values are shown in bold in Figure 100.

For example, the NTUP 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 NTUP 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 NTUP 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 the following table, where *x* stands for values which changed across different runs.

```
--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 100-Modified DCCFPs of the test model in Figure 96.

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

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 NTUP point is at time=5.

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

To investigate the repeatability of GA results across multiple runs, Figure 101-(a) depicts the distributions of maximum ISTOF and stress time values for 1000 runs on the example. From the ISTOF distribution, we can see that the maximum fitness values for most of the runs are between 70 and 92 units of traffic. Descriptive statistics of the fitness values for the 1000 runs are shown in Table 10. Average and median values are very close, thus indicating that the distribution is almost symmetric.

Min	Max	Average	Median	Standard Deviation
65	112	81.66	81	7.05

Table 10-Descriptive statistics of the maximum ISTOF values over 1000 runs. Values are in units of data traffic (e.g. KB).

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 101-(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 one of the frequent maximum ISTOF values (75 units of traffic) have been highlighted in black in Figure 101-(b). As we can see, these maximum stress time values are scattered across the time scale (e.g., from 10 to over 20 units of time). This highlights that a single ISTOF value (maximum stress traffic) can happen in different time instances, thus suggesting the search landscape for the GA is rather complex for this type of problem. Thus, a strategy to further explore for comprehensive stress testing would be to try all (or a subset of) such test requirements in different time instances. Indeed, although the maximum ISTOF value in all such test requirements are the same, a SUT's reaction to different test requirements might be different, since each test requirement triggers a different DCCFP (and hence set of messages) in a different stress time instance than others. This might lead to uncovering different RT faults in the SUT.

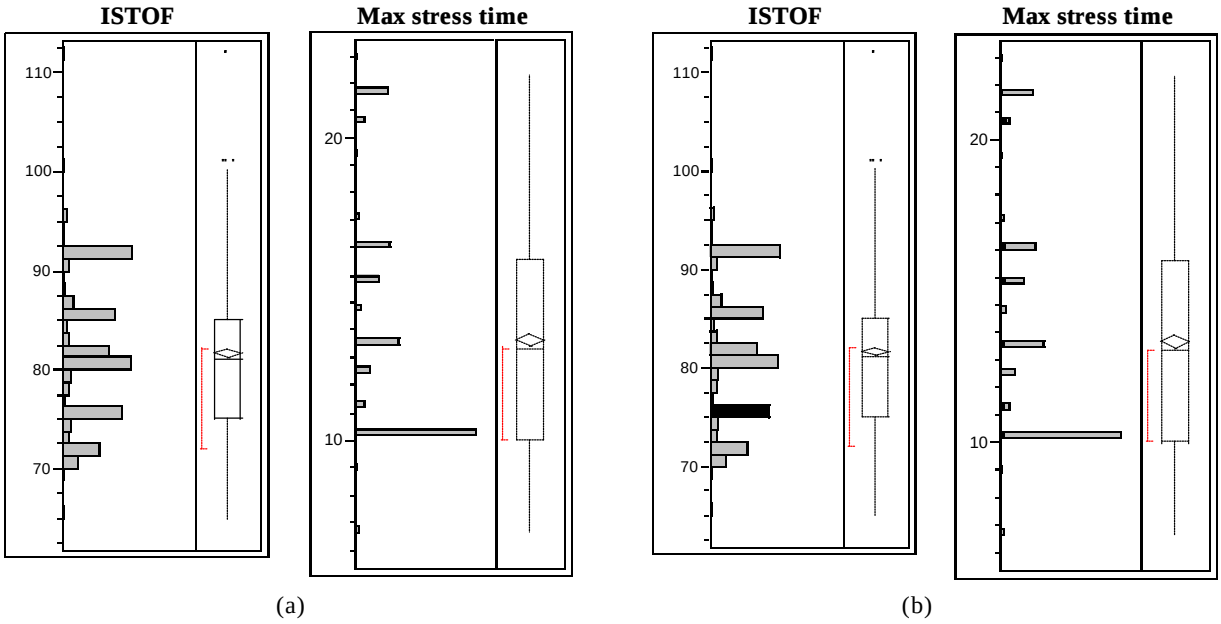


Figure 101-(a): Histogram of maximum ISTOF and stress time values for 1000 runs (b): Corresponding max stress time values for one of the frequent maximum ISTOF values.

11.3.4 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. The distribution of these generation numbers over 1000 runs is shown in **Error! Reference source not found.**, 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 [74] but is however likely to be dependent on the number and complexity of SDs as well as their ATSs.

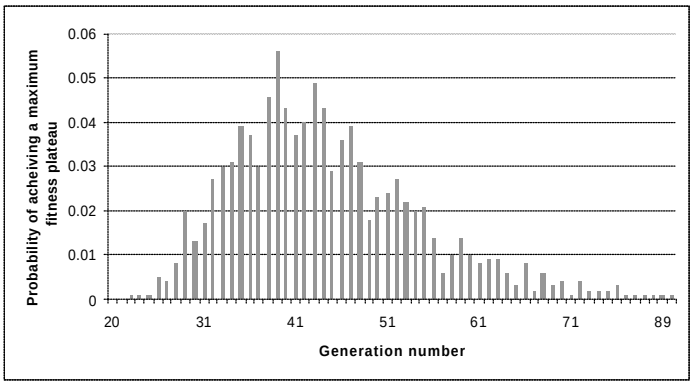


Figure 102-Histogram of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of the example by GARUS.

From the initial to the final populations, 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.

Chapter 12

CASE STUDY

A comprehensive case study is presented in this section. An overview of target systems of our stress test technique is described in Section 12.1. Section 12.2 discusses the requirements of a suitable target system as the 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. Derivation of network-aware stress test requirement and cases for SCAPS are explained in Section 12.4. Section 12.5 presents the stress test architecture used in our case study. Some descriptions of the stress test execution environment are given in Section 12.6. Test results are reported in Section 12.7, where we assess the effectiveness of our stress test technique at triggering network traffic-related failures.

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 the DCS and SCADA categories, it is more convenient to discuss them separately.

12.1.1 Distributed Control Systems

Distributed control systems (DCS) [89] are computer-based control systems where several sections of plant have their own processors, linked together to provide both information dissemination and manufacturing coordination. DCS systems are used in industrial and civil engineering applications to monitor and control distributed equipment with remote human intervention.

DCS systems are 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 (HMI) or control consoles.

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

12.1.2 Supervisory Control and Data Acquisition Systems

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 the supervisory level. As such, it is a software package that is positioned on top of hardware to which it is interfaced, in general via Programmable Logic Controllers (PLCs), or other commercial hardware modules [90]. 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 [91], power generation (conventional and nuclear) and distribution [92-96], chemistry and oil [97], but also in facilities such as nuclear fusion [98, 99]. The size of such plants ranges from a few to several thousands I/O channels. However, 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. DCS is process-oriented as it focuses at the control process (such as a chemical plant), and presents data to operators. On the other hand, SCADA 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 process control.

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.

SCADA must operate reasonably when field communications have failed. The quality of the data shown to the operator is an important facet of 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 103.

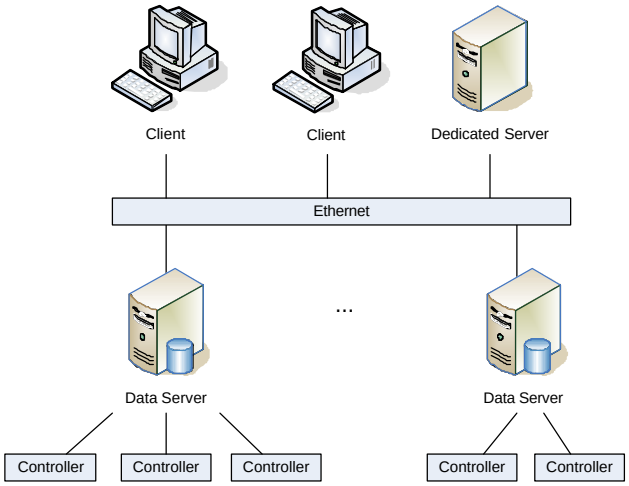


Figure 103-A typical architecture of SCADA systems.

12.1.3 Use of UML and OO Concepts in DCS and SCADA Systems

As UML and OO-driven system development are getting more popular, recent DCS and SCADA systems are no exceptions. We survey here some of the recent works on DCS and SCADA systems which use UML and OO concepts in their design.

Stojkovic and Vujosevic [100] report a prototype SCADA system for a smaller size electric power plant. They refer to their prototype as a fast, object-oriented and cost-effective approach, which has been developed with Microsoft VisualBasic, a rapid application development environment under Microsoft Windows.

To address the need for fast, reliable and RT DBMSs (DataBase Management System) in SCADA and DCS applications, Wakizono et al. [101] present and evaluate an OO DBMS for process control systems. The authors evaluate the time taken to perform a typical complex DBMS query. As their comparisons show, the query execution time in the OO DBMS is much faster in than a rational DBMS. This quick DBMS response can be useful in many RT applications.

Thramboulidis [102] presents a UML-based Engineering Support System (ESS) for Industrial Process Measurement and Control Systems (IPMCSs)[1], where an OO notation is proposed along with a network topology and an internetworking unit architecture to form the infrastructure that is necessary for the development of the new generation ESSs.

Thramboulidis [103] presents CORFU (a Common Object-oriented Real-time Framework for the Unified development of distributed IPMCS applications). As reported, this framework can assist process and system engineers in the development, configuration, and operation of distributed IPMCSs.

Brown et al. [104] present a concept for integrating the embedded programming methodology Giotto [105] and the object-oriented Attitude and Orbit Control System (AOCS) framework [106] to create an environment for the rapid development of distributed software for safety-critical embedded control systems with hard real-time requirements of the kind typically found in aerospace applications.

Brand et al. [107] present a case study on how to use the ObjectVIEW toolkit [108] within the graphical language LabVIEW [108] to execute a UML design model prior to system implementation. As an example of this approach, the application layer of the control system of the PHELIX (Petawatt High Energy Laser for heavy Ion eXperiments) [109] facility is presented.

12.1.4 Failures and Disasters due to Overload

Reports such as [110], [111], [112], [113] indicate the high risk of failures due to network overload, while [114] actually report failures and disasters which have happened due to network overload.

12.2 Choosing a Target System as Case Study

There are various distributed, real-time prototype systems in academia (e.g. [115], [116], [117]) and also real systems in industry (e.g. [118], [119], [120], [121]), which are currently in use.

12.2.1 Requirements of a Suitable System

We group the requirements of a suitable system (to be selected as our case study) into two groups: (1) system's functional features and behaviors, and (2) its model requirements.

A suitable case study should have the following functional features and behaviors:

- *Requirement 1*: It should be a distributed, hard real-time system, and preferably safety-critical, in which deadline misses can lead to catastrophic results. This reason is because 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. What we mean by a data intensive, in this context, is a distributed system in which most (or at least some) of the messages exchanged among distributed nodes usually have large data sizes. The rationale for this requirement is again due to the nature of our stress test technique, which tries to find the most data intensive distributed messages and produce schedules so that such messages run concurrently.

[1] Similar to DCS systems

- *Requirement 3*: It should be possible to run a system in the typical hardware/software platform of 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 its model:

- *Requirement 4*: 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 codes 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 can also be accepted.

12.2.2 None of the Systems in our survey Meets the Requirements

None of the existing systems we are aware of met all of the above requirements. We provide a brief, structured summary below:

- *Requirement 1*: Not all distributed systems, we surveyed, were hard RT, safety-critical such as QADPZ (Quite Advanced Distributed Parallel Zystem) [115].
- *Requirement 2*: Similarly, a good target system should be data intensive. None of the systems, under study, which met other requirements met this one, such as the RT distributed factory automation system [116] 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 institute, like ours. We are even flexible is replacing the embedded components and special hardware with test stubs or component simulators, if possible. However, doing this for a complex system is not easily possible, for example COACH (Component Based Open Source Architecture for Distributed Telecom Applications) [118].
- *Requirement 4*: The systems models/source codes are not freely available or even not available at all. The can be either due to being sensitive and classified information, such as JITC (The Joint Interoperability Test Command) [119] and [117], or systems are very expensive, such as CitectSCADA [120] and ElipseSCADA [121].
- *Requirement 5*: As a corollary of our discussion on requirement 4, no UML 2.0 model of the systems in our selection pool was accessible.

12.3 Our Prototype System: A SCADA-based Power System

Because none of the systems we surveyed meet the requirements (Section 12.2.1), we decided to analyze, design, and build a prototype system by using the ideas and concepts from existing distributed system technologies.

Section 12.3.1 presents an overview on SCADA-based power systems. We designed and developed a SCADA-based power system, which is described in Section 12.3.2. In Section 12.3.3, we discuss how and why SCAPS meets our case study requirements (described in Section 12.2.1). We present the SCAP's UML design model in Section 12.3.4. Relevant implementation issues are presented in Section 12.3.5. Section 12.3.6 provides a brief description of SCAP's hardware and configuration. SCAPE is then used as the SUT in Section 12.4 by our stress test technique.

12.3.1 SCADA-based Power Systems

SCADA for power systems was developed in the 1960's and has been improving ever since. The architecture of power SCADA systems has changed from the mainframe-dominated, centralized computing systems to network-based distributed computing in the early 1990's [117]. A new class of

SCADA systems that is called *open distributed systems* [122] has been designed based on this new architecture. Fundamental features of open distributed systems that distinguish it from the previous design are the use of industry-standard, local area network (LAN) and the distribution of functions among several computers or workstations on a LAN or WAN (Wide Area Network).

SCADA systems have been used in both nuclear and hydro power generation plants [95, 123] and distribution grids [92-94, 96, 100]. As discussed in Section 12.2.2, most of the SCADA power systems require dedicated and special-purpose hardware to run and none of the systems are made public (even those made for research purposes in articles). However, the overview descriptions of the SCADA systems are usually available. Figure 104 shows a typical SCADA model of a power distribution system [93].

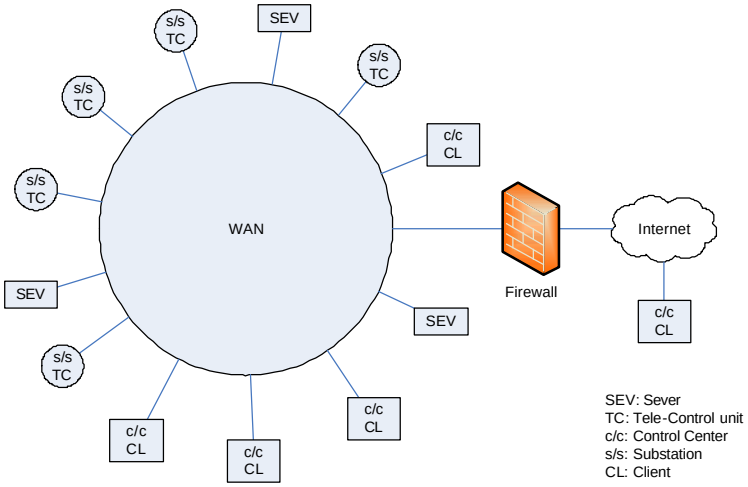


Figure 104-Power systems SCADA model [93].

The model consists of TCs (Te1e-Control units) that sends data of power system to servers. SCADA applications execute in servers. Clients (CLs) are used by operators in control centers (c/c) inside or outside the WAN. Operators monitor and control the power system through the software installed on clients. Each TC sends data related to the component of the power system to servers through WAN. Multicast communication based on IP is applied to the communication between TC and servers, and all servers can receive data from every TC. The location of servers is transparent to clients. Critical functions of SCADA can be installed in servers that can be backed up. WAN-based SCADA connects to the Internet through a firewall. Communication model between tele-control units (TCs) and servers (SEVs) in a SCADA system [93] is shown in Figure 105.

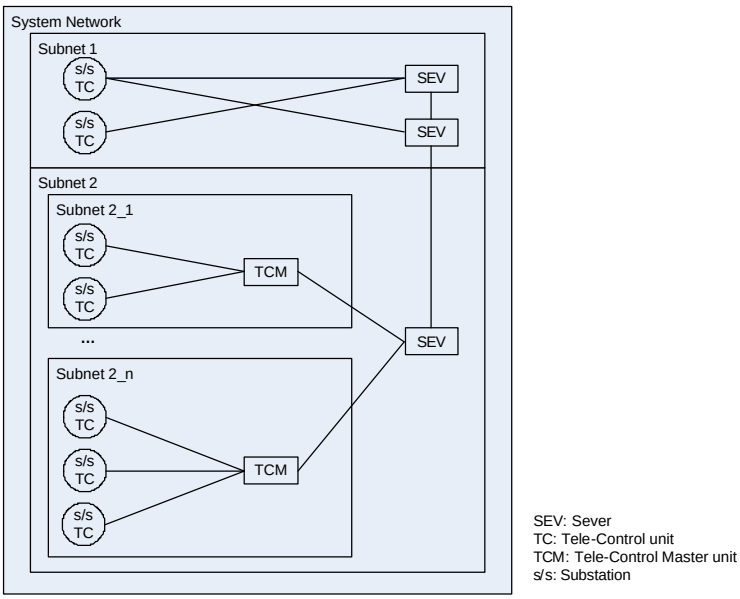


Figure 105- Communication model between tele-control units and servers in a SCADA system [93].

Power systems usually have a hierarchical operational organization [94]. A typical operational organization of power systems is shown in Figure 106. This helps to make them a good candidate for our case study, as they fit well to our discussions on Network Deployment Diagram and Network Interconnectivity Tree (NIT) in Section 5.5.

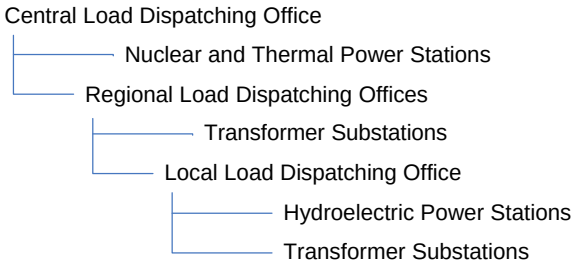


Figure 106-A typical operational organization for power systems [94].

12.3.2 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*: 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.
- *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.

- *Power restoration after network failure*: Presents 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.

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 are typical functions performed by SCADA power systems [93, 122], and will be shown in a use case diagram (Section 12.3.4), 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 [93, 122]:

- *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 to provide 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.
- *Data maintenance*: Enables operator to modify database of power apparatus and network topology by drawing single line diagrams on the control screen and defining parameters.
- *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.

12.3.3 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 a 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 (Section 12.3.1). 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.

12.3.4 Partial UML Model

Consistent with the SCAPS specification in Section 12.3.2, 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)

12.3.4.1 Use-Case Diagram

The SCAPS use-case diagram is shown in Figure 107.

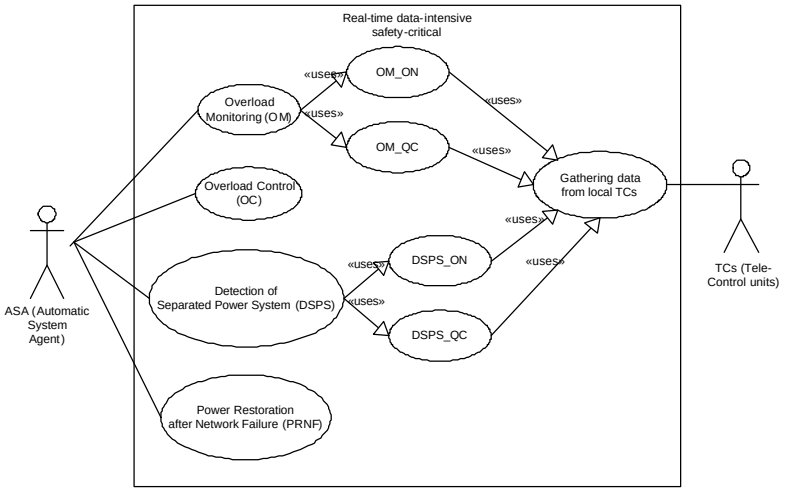


Figure 107- SCAPS use-case diagram.

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.

12.3.4.2 Network Deployment Diagram

The Network Deployment Diagram (NDD) of SCAPS is shown in Figure 108.

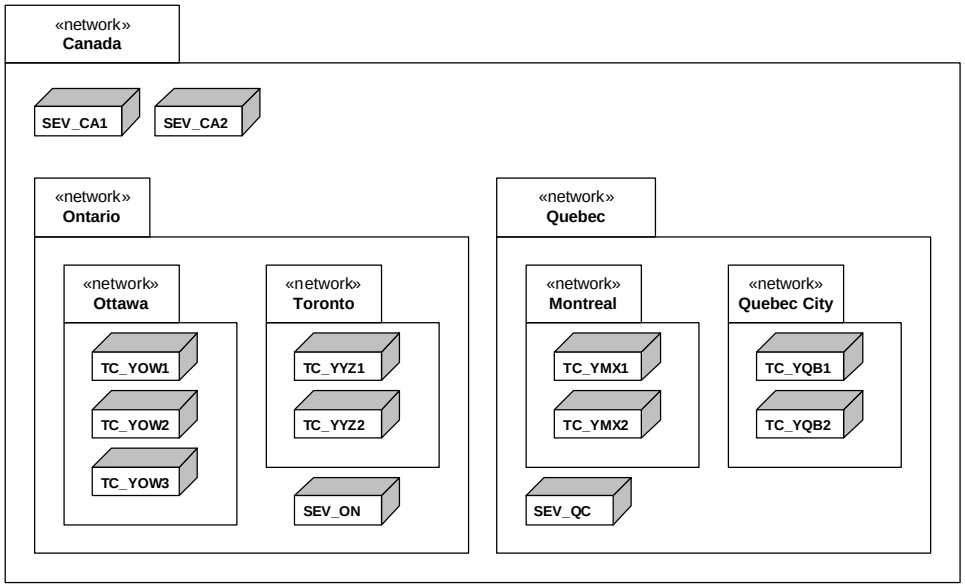


Figure 108- SCAPS network deployment diagram.

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 (*SEV_CA1* and *SEV_CA2*) 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 (*SEV_CA1*) fails.

12.3.4.3 Class Diagram

Part of the SCAPS class diagram which is required to demonstrate the case study is shown in Figure 109. The classes are grouped in two groups: entity and control classes [47, 124]. Entity classes are those which are used either as parameters (by inheriting from *SetFuncParameter*) or return values (by inheriting from *QueryFuncResult*) of the method 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 (by inheriting from *Data-Intensive*). 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 *ProvController* class. *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.

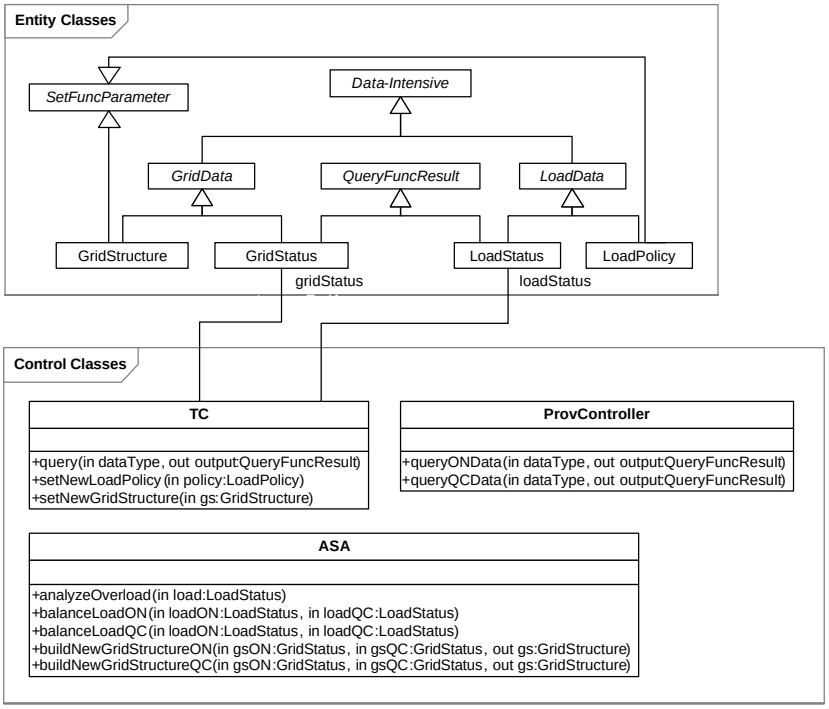


Figure 109-SCAPS partial class diagram.

12.3.4.4 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 functionality 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 [10] to extend the SDs.

Eight SDs are presented in Figure 110-Figure 115. They correspond to use-cases in the SCAPS use-case diagram (Figure 107). SDs *OM_ON* and *OM_QC* in Figure 110 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*.

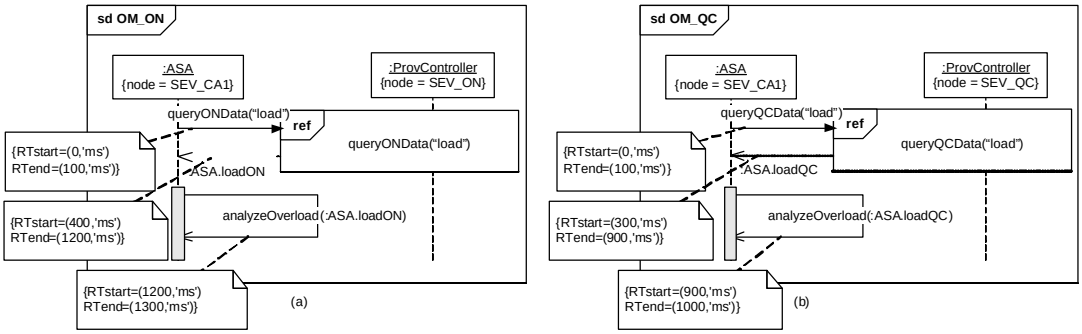


Figure 110- SDs *OM_ON* and *OM_QC* (Overload Monitoring).

The two SDs in Figure 111 (*queryONData(dataType)*) and Figure 112 (*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 108), 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 111 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.

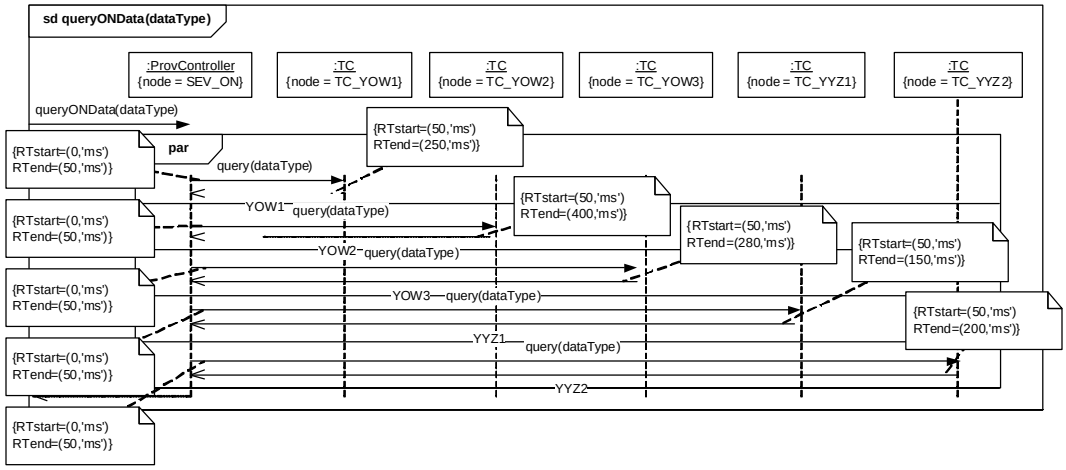


Figure 111-SD *queryONData(dataType)*.

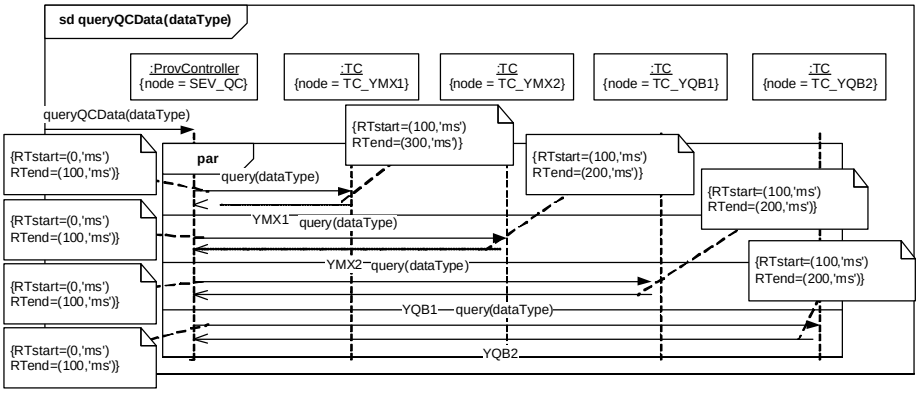


Figure 112-SD *queryQCData(dataType).*

OC (Overload Control) SD (Figure 113) checks if there is overload situation in any of the two provinces (using *overloadIn()* as a condition). If such a case has occurred 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 114) fetch grid connectivity data from the provincial controllers and check to see if there is any separated power system (using *detectSeparatedPS()*).

Similar to the *OC* SD (Figure 113), *PRNF* (Power Restoration after Network Failure) SD (Figure 115) checks if there is any separated power system in any of the two provinces (using *anySeparationIn()*as a condition). If such a case has occurred 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()*).

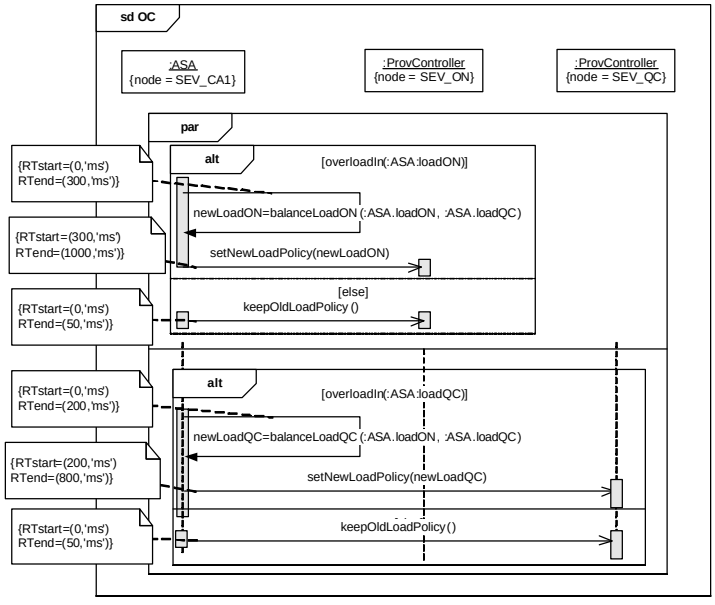


Figure 113- SD *OC* (Overload Control).

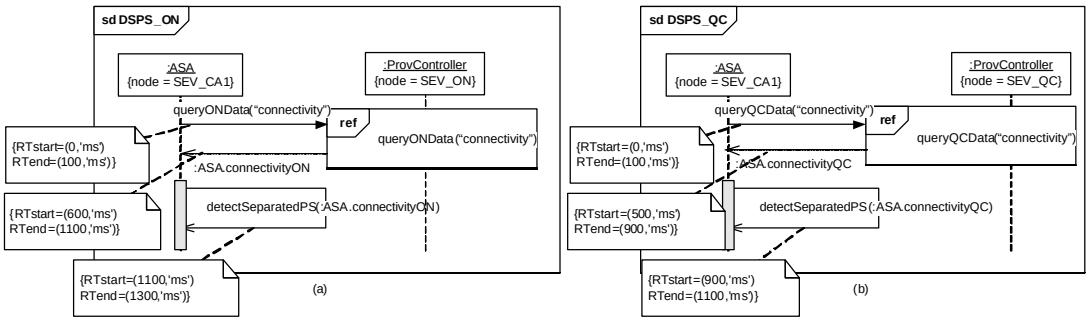


Figure 114-SD *DSPS_ON* and *DSPS_QC* (Detection of Separated Power System).

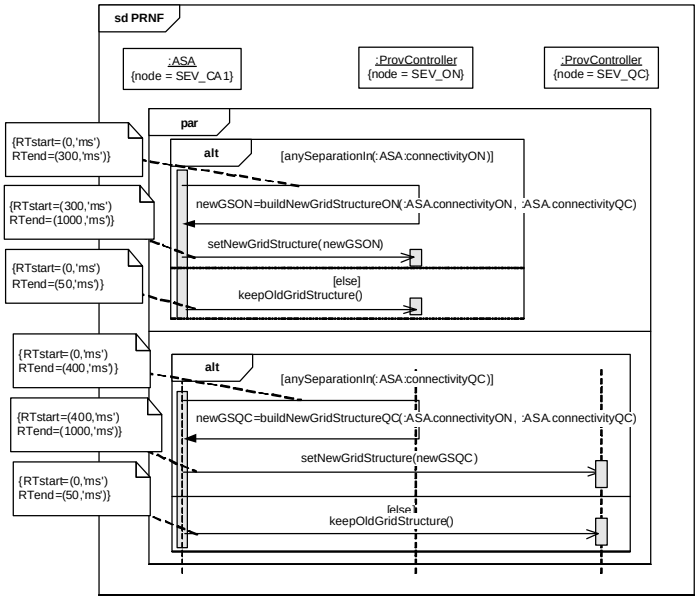


Figure 115-SD *PRNF* (Power Restoration after Network Failure).

12.3.4.5 Modified Interaction Overview Diagram

The MIOD of SCAPS is shown in Figure 116. As denoted in the SCADA-based power systems literature (e.g. [92-94, 96, 100]), 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.

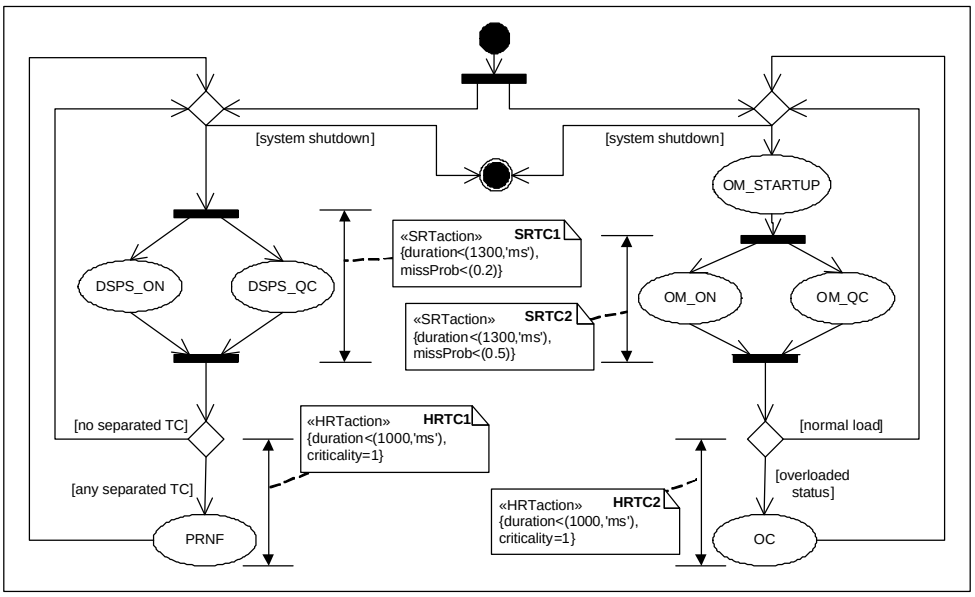


Figure 116-SCAPS Modified Interaction Overview Diagram (MIOD).

We consider four MIOD-level RT constraints for SCAPS. Figure 116 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 with numbers to make it easier to refer to them later, and are explained below.

1. SRT constraints
    a. $SRTC_1$: Overload monitoring (concurrent runs of *OM_ON* and *OM_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. $SRTC_2$: Detection of separated power systems (concurrent runs of *DSPS_ON* and *DSPS_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 an overload situation is detected, overload control policy (*OC* SD) should be executed in less than 1000 ms. We assign criticality[1]=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).
    b. $HRTC_2$: 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.

12.3.5 Implementation

SCAPS was developed using Borland Delphi[2], 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

[1] As defined by UML SPT profile [10], criticality determines the extent to which the consequences of missing a hard deadline are unacceptable.

[2] www.borland.com/delphi

enables rapid development of prototype applications without spending extensive time on programming details.

We developed only one 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 in line, the system then starts functioning. A screenshot of the main screen of SCAPS is shown in Figure 117, where the application is running as a *SEV_CA1* node and has just accepted a connection from the *TC_YOW1* node.

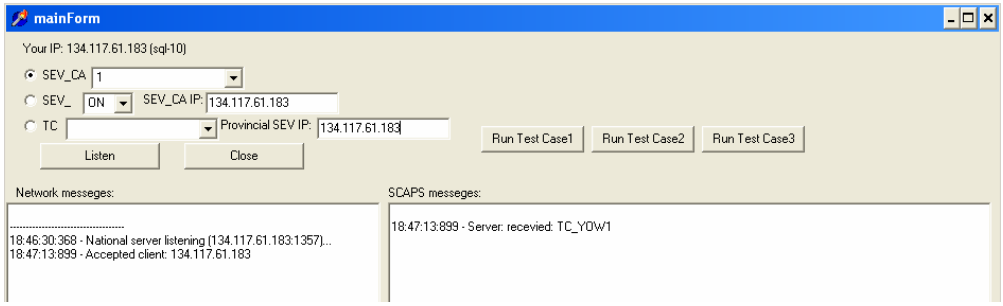


Figure 117-A screenshot of the main screen of SCAPS.

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 their 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 107), 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 109).

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 Sections 12.4 and 12.7, testing SCAPS in this work is based on triggering specific DCCFPs in specific time instances. 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 a specific edge 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 also 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 [92-94, 96, 100], e.g., we assumed that function *analyzeOverload* of class *ASA* takes 100 ms to run (refer to the SDs *OM_ON* in Figure 110). 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 Derivation of Network-aware Stress Test Cases

Using the given UML design model in Section 12.3.4, we first derive the test model required by our test technique (Sections 12.4.1-12.4.5). We then consider three stress test objectives in Section 12.4.7. Section 12.4.8 and Section 12.4.9 describe how the stress test requirements and test cases corresponding to the chosen test objectives are derived, respectively.

12.4.1 Network Interconnectivity Tree

The Network Interconnectivity Tree (NIT) of SCAPS can be derived from the Network Deployment Diagram (NDD) in Figure 108. The NIT is shown in Figure 118.

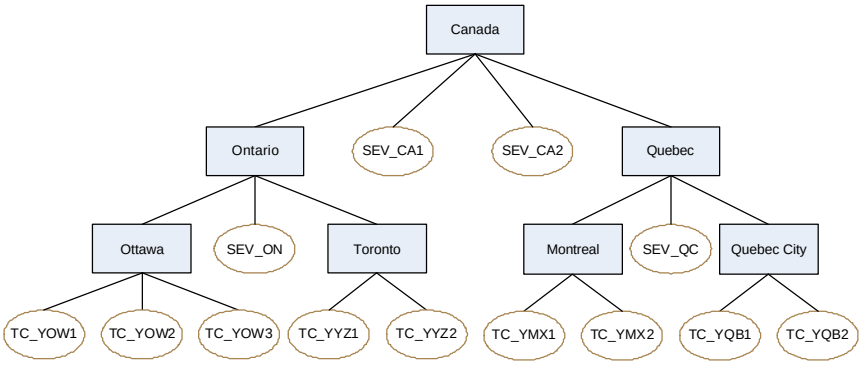


Figure 118- SCAPS Network Interconnectivity Tree (NIT).

12.4.2 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 12.3.4.4. CCFGs shown in Figure 119 to Figure 124 correspond to SDs in Figure 110 to Figure 115. 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.

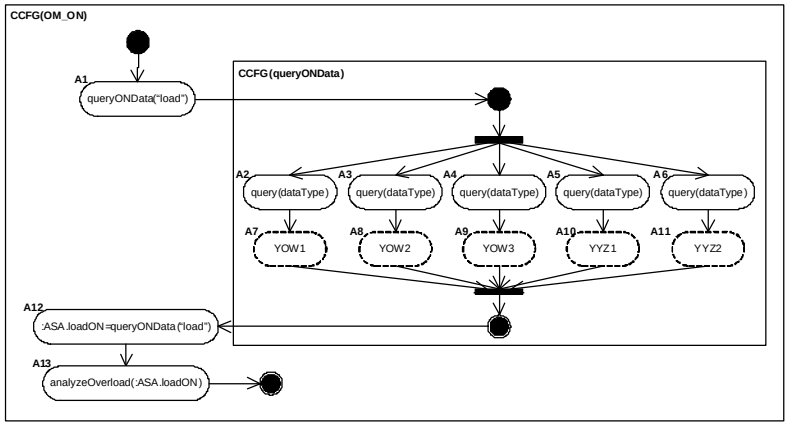


Figure 119-CCFG(OM_ON).

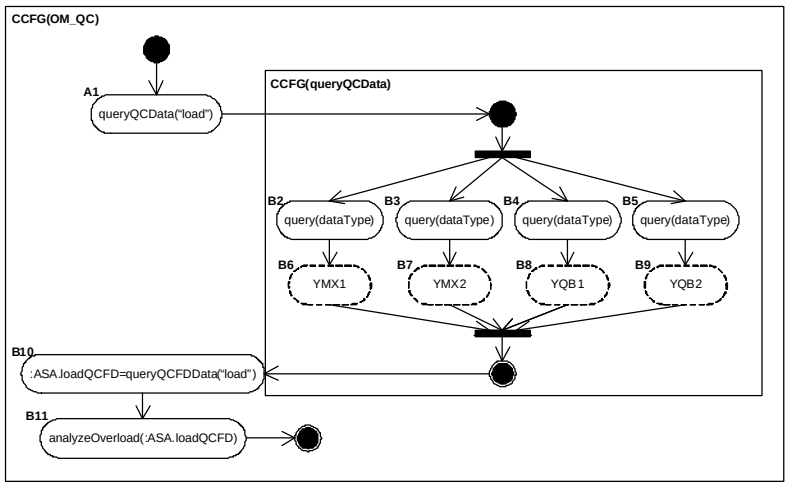


Figure 120-CCFG(OM_QC).

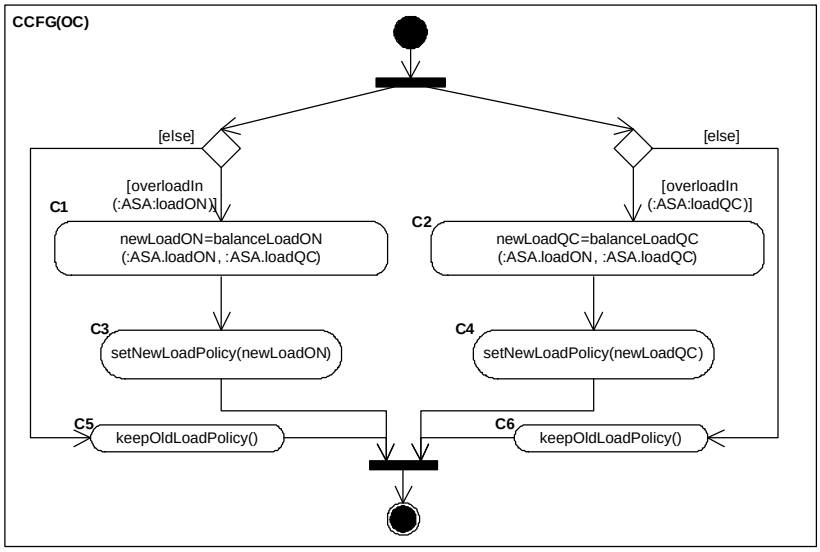


Figure 121-CCFG(OC).

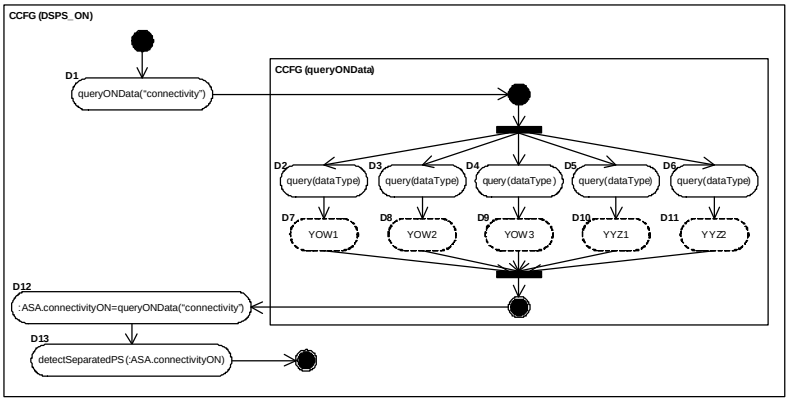


Figure 122-CCFG(DSPS_ON).

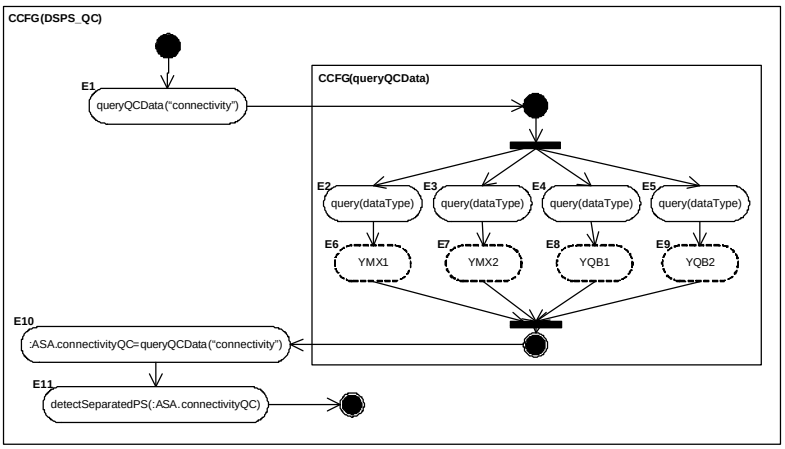


Figure 123-CCFG(DSPS_QC).

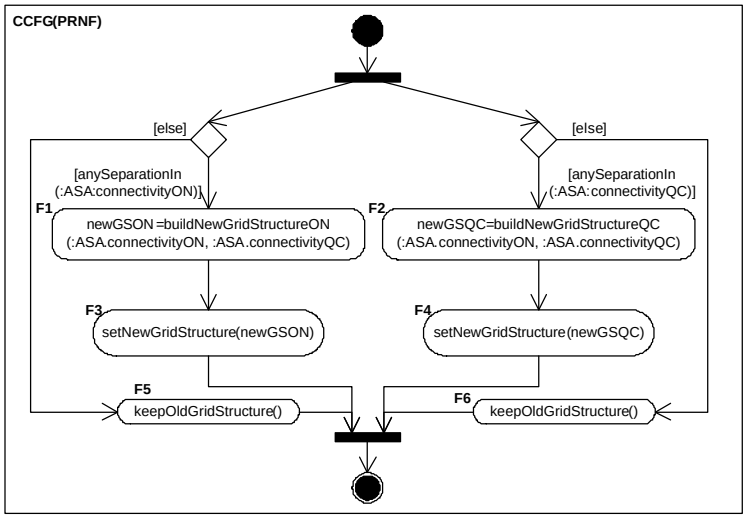


Figure 124-CCFG(PRNF).

12.4.3 Derivation of Distributed Concurrent Control Flow Paths

Using the technique presented in Chapter 6, we derive the CCFPs and DCCFPs of the CCFGs shown in Figure 119 to Figure 124. The CCFPs and DCCFPs are shown in Figure 125. To ease future references, we assign SD_i and $r_{i,j}$ indices to SDs and the DCCFPs of each SD, respectively. Let us assign $r_{0,0}$ to the only CCFP of SD *OM_STARTUP*, which does not contain any distributed message.

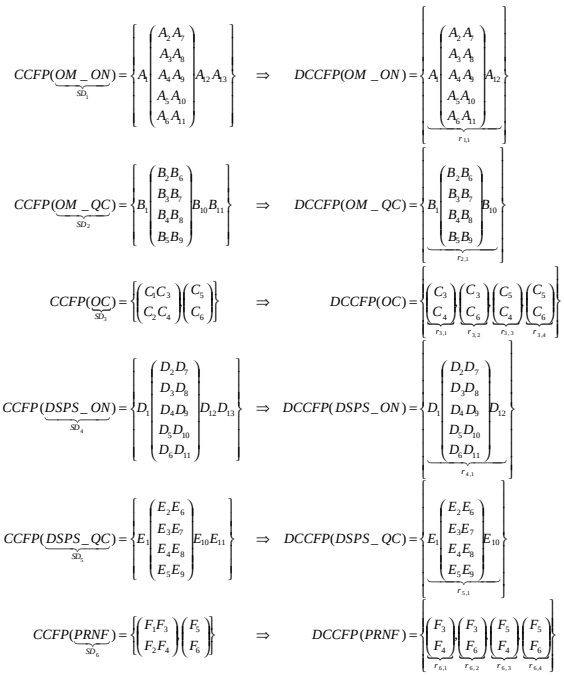


Figure 125-CCFP and DCCFP sets of SDs in SCAPS.

12.4.4 Derivation of Independent-SD Sets

Using the method in Section 7.1 and the SCAPS MIOD (Figure 116), 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 126 is derived from the MIOD of Figure 116. Note that we do not include SD *OM_STARTUP* in this ISDT, since it does not have any distributed messages.

As discussed in the algorithm presented in Section 7.1.2, every strongly connected component of an ISDG is an ISDS. By finding the strongly connected component of the ISDG in Figure 126, the Independent SD Sets of SCAPS can be derived. SCAPS has seven ISDSs:

$$ISDS_1 = \{OM\_ON, OM\_QC, DSPS\_ON\} \qquad ISDS_2 = \{OM\_ON, OM\_QC, DSPS\_QC\}$$
$$ISDS_3 = \{OM\_ON, OM\_QC, PRNF\} \qquad ISDS_4 = \{OM\_ON, DSPS\_ON, DSPS\_QC\}$$
$$ISDS_5 = \{OM\_QC, DSPS\_ON, DSPS\_QC\} \qquad ISDS_6 = \{OC, DSPS\_ON, DSPS\_QC\}$$
$$ISDS_7 = \{OC, PRNF\}$$

12.4.5 Derivation of Concurrent SD Flow Paths

Using the method in Section 7.2 and the SCAPS MIOD (Figure 116), 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.

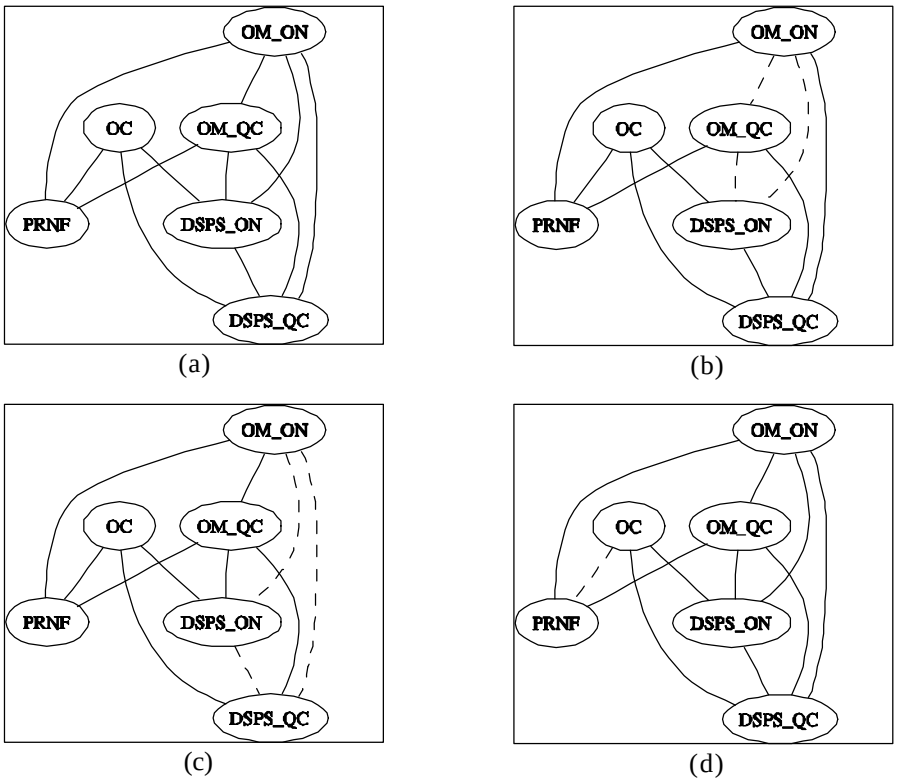


Figure 126-(a):Independent-SDs Graph (ISDG) corresponding to the MIOD of Figure 116. (b), (c) and (d): Three of the strongly connected components of the ISDG (shown with dashed edges), yielding three ISDSs.

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 116), 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 127.

$$CSDFP = \left(OM\_STARTUP \begin{pmatrix} OM\_ON \\ OM\_QC \end{pmatrix} \begin{pmatrix} DSPS\_ON \\ DSPS\_QC \end{pmatrix} \right) CSDFP \mid \left(OM\_STARTUP \begin{pmatrix} OM\_ON \\ OM\_QC \end{pmatrix} OC \begin{pmatrix} DSPS\_ON \\ DSPS\_QC \end{pmatrix} \right) CSDFP \mid$$
$$\left(OM\_STARTUP \begin{pmatrix} OM\_ON \\ OM\_QC \end{pmatrix} \begin{pmatrix} DSPS\_ON \\ DSPS\_QC \end{pmatrix} PRNF \right) CSDFP \mid \left(OM\_STARTUP \begin{pmatrix} OM\_ON \\ OM\_QC \end{pmatrix} OC \begin{pmatrix} DSPS\_ON \\ DSPS\_QC \end{pmatrix} PRNF \right) CSDFP \mid e$$

Figure 127-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 assume that the number of cycles to derive CSDFPs is given by the tester. Some of the CSDFPs which can be derived from the grammar in Figure 127 are:

$$CSDFP_1 = \begin{pmatrix} OM\_STARTUP\begin{pmatrix} OM\_ON \\ OM\_QC \end{pmatrix} \\ \begin{pmatrix} DSPS\_ON \\ DSPS\_QC \end{pmatrix} \end{pmatrix} \quad CSDFP_2 = \begin{pmatrix} OM\_STARTUP\begin{pmatrix} OM\_ON \\ OM\_QC \end{pmatrix} OC \\ \begin{pmatrix} DSPS\_ON \\ DSPS\_QC \end{pmatrix} \end{pmatrix}$$

$$CSDFP_3 = \begin{pmatrix} OM\_STARTUP\begin{pmatrix} OM\_ON \\ OM\_QC \end{pmatrix} \\ \begin{pmatrix} DSPS\_ON \\ DSPS\_QC \end{pmatrix} PRNF \end{pmatrix} \quad CSDFP_4 = \begin{pmatrix} OM\_STARTUP\begin{pmatrix} OM\_ON \\ OM\_QC \end{pmatrix} OC \\ \begin{pmatrix} DSPS\_ON \\ DSPS\_QC \end{pmatrix} PRNF \end{pmatrix}$$

$$CSDFP_5 = \begin{pmatrix} OM\_STARTUP\begin{pmatrix} OM\_ON \\ OM\_QC \end{pmatrix} OC \\ \begin{pmatrix} DSPS\_ON \\ DSPS\_QC \end{pmatrix} PRNF \end{pmatrix} \begin{pmatrix} OM\_STARTUP\begin{pmatrix} OM\_ON \\ OM\_QC \end{pmatrix} OC \\ \begin{pmatrix} DSPS\_ON \\ DSPS\_QC \end{pmatrix} \end{pmatrix}$$

$$CSDFP_6 = \begin{pmatrix} OM\_STARTUP\begin{pmatrix} OM\_ON \\ OM\_QC \end{pmatrix} OC \\ \begin{pmatrix} DSPS\_ON \\ DSPS\_QC \end{pmatrix} \end{pmatrix} \begin{pmatrix} OM\_STARTUP\begin{pmatrix} OM\_ON \\ OM\_QC \end{pmatrix} \\ \begin{pmatrix} DSPS\_ON \\ DSPS\_QC \end{pmatrix} PRNF \end{pmatrix}$$

Figure 128-Some of the CSDFPs of SCAPS derived from the grammar in Figure 127.

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.

12.4.6 Data Size of Messages

Note that, for brevity, we do not discuss the data structure of the entity data classes in SCAPS (Figure 109). But according to the literature on SCADA-based power systems [92-94, 96, 100], 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 11 as the mean data sizes of the entity data classes in Figure 109. These values are realistic size estimates of real grid and load values according to the literature on SCADA-based power systems [125]. 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 specialized literature.

Note that we assume the data sizes in Table 11 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.

Data Class	Mean Data Size
LoadStatus	4 MB
LoadPolicy	2 MB
GridStatus	3 MB
GridStructure	1 MB

Table 11-Mean data sizes of the entity data classes of SCAPS.

12.4.7 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. In our assumptions, only bidirectional stress direction is applicable to a network stress location. Since networks are not end points of communication, therefore "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, let us consider the following three examples of test objectives:

(Canada, -, data, instant)

(SEV_CA1, in, data traffic, interval)

(SEV_ON, bidirectional, message traffic, instant)

Note that the main criterion in choosing good test objectives is to look for vulnerable (to distributed faults) networks and nodes, given the hard real-time constraints in a system. As discussed in Section 12.3.4.5 and modeled in the SCPAS MIOD (Figure 116), there are two hard real-time constraints in SCAPS: (1) the power of any separated (disconnected) grid should be restored within 1000 ms, after detection, and (2) any overload case in the grid should be controlled by the central server in less than 1000 ms. Failure of SCAPS to meet any of these two requirements is unacceptable. Therefore, networks and nodes which are utilized by SDs and are in the hard real-time region of a MIOD should be stress tested first. We choose the above test objectives with this heuristic in mind.

12.4.8 Derivation of Test Requirements

We discuss here how the corresponding test requirements can be derived from the above three test objectives.

12.4.8.1 Test Objective 1: (Canada, -, data traffic, instant)

The stress test location in this element is network *Canada.* Stress type is "data traffic", and stress duration is "instant". We use the *StressNetInsDT(net)* stress test requirement generation technique, described in Section 9.11.2. We present below the steps of the algorithm to derive test requirements.

Step 1

In this step, maximum stress DCCFP of each SD is chosen. In order to do so, we need to find the maximum data traffic (DT) value of each DCCFP first, whose goes over network *Canada* in NIT of Figure 118. We consider all the DCCFPs of SCAPS (Figure 125).

For example, we show how the maximum DT value of the only DCCFP of SD *OM_ON* (SD_1), which is $r_{1,1}$, is calculated. Finding the *MaxNetInsDTValue* (Algorithm 3) requires the values of function *NetInsDT*. Using the timing information of the messages (Section 12.3.4.4) and their data sizes (Section 12.4.6), we can derive the values of $NetInsDT(r_{1,1},"Canada",t)$ for all t. To better illustrate this, the timed-DT value representation of DCCFP $r_{1,1}$ and the resulting $NetInsDT(r_{1,1},"Canada",t)$ values are shown in Figure 129.

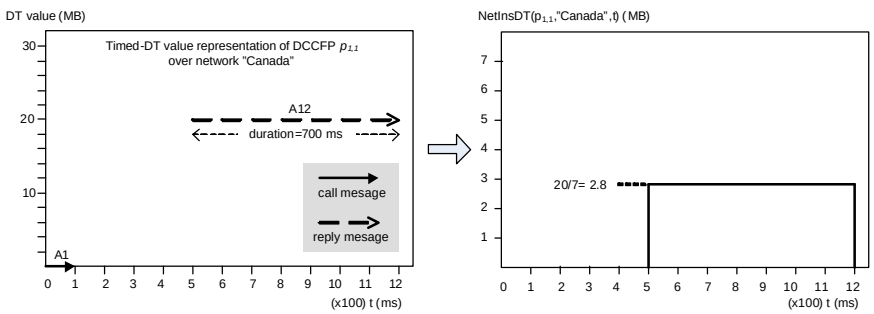


Figure 129-(a): The timed-DT value representation of DCCFP $r_{1,1}$, (b): The resulting $NetInsDT(r_{1,1},"Canada",t)$ function values.

Note that as Algorithm 3 points out, only those distributed message which go through the network *Canada* are considered in calculating *NetInsDT(r1,1,"Canada",t)*, and thus *MaxNetInsDTValue(r1,1,"Canada")*. We can now calculate the *MaxNetInsDTValue* as:

$$MaxNetInsDTValue(r_{1,1},"Canada") = \max_t\left(NetInsDT(r_{1,1},"Canada",t)\right) = 2.8MB$$

We present another example by showing how *MaxNetInsDTValue(r3,1,"Canada")* is calculated, where $r_{3,1} = \begin{pmatrix} C_3 \\ C_4 \end{pmatrix}$. The timed-DT value representation of DCCFP $r_{3,1}$ and the resulting *NetInsDT(r3,1,"Canada",t)* function values are shown in Figure 130.

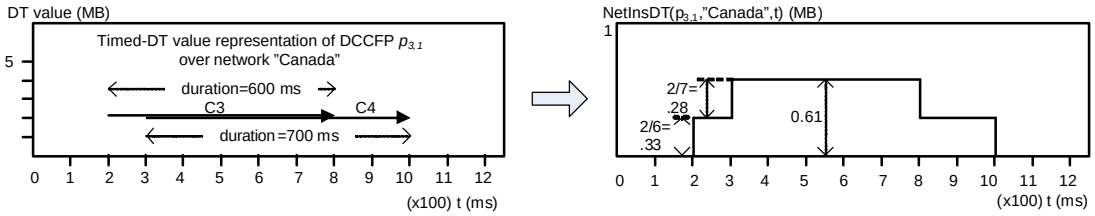


Figure 130-The timed-DT value representation of DCCFP $r_{3,1}$, (b): The resulting *NetInsDT(r3,1,"Canada",t)* function values.

As seen in *CCFG(OC)*, Figure 121, Control flow nodes *C3* and *C4* correspond to messages *setNewLoadPolicy(newLoadON)* and *setNewLoadPolicy(newLoadQC)*, in SD *OC* (Figure 113), respectively. Furthermore, as modeled in SD *OC*, the latter two messages are RT and their start and end time pairs are [300ms, 1000ms] and [200ms, 800ms], respectively. Therefore, as shown in the timed-DT value representation in Figure 130, data traffic due to *C3* and *C4* overlap each other in time interval [300ms, 800ms]. Each of *C3* and *C4* DT values are 2 MB, since they have an object of type *LoadPolicy* as the parameter, and the data size of this class is 2 MB. The total DT during the interval [300ms, 1000ms], when *C3* and *C4* overlap, is 4 MB. Considering the *NetInsDT(r3,1,"Canada",t)* values in Figure 130, it is obvious that:

$$MaxNetInsDTValue(r_{3,1},"Canada") = \max_t\left(NetInsDT(r_{3,1},"Canada",t)\right) = 0.61MB$$

Maximum DT values of other DCCFPs can be found in a similar manner. These values are shown below.

$$MaxNetInsDTValue(r_{2,1},"Canada") = 2.66MB \quad MaxNetInsDTValue(r_{3,2},"Canada") = 0.28MB$$
$$MaxNetInsDTValue(r_{3,3},"Canada") = 0.33MB \quad MaxNetInsDTValue(r_{3,4},"Canada") = 0$$
$$MaxNetInsDTValue(r_{4,1},"Canada") = 3MB \quad MaxNetInsDTValue(r_{5,1},"Canada") = 3MB$$
$$MaxNetInsDTValue(r_{6,1},"Canada") = 0.3MB \quad MaxNetInsDTValue(r_{6,2},"Canada") = 0.14MB$$
$$MaxNetInsDTValue(r_{6,3},"Canada") = 0.16MB \quad MaxNetInsDTValue(r_{6,4},"Canada") = 0MB$$

By comparing the *MaxNetInsDTValue*'s of DCCPs, we now find the maximum stress DCCFP of each SD. All SDs, except *OC* and *PRNF*, have only one DCCFP, therefore their maximum stress DCCFP will be their only one DCCFP.

$$MaxNetInsDTDCCFP(OM\_ON,"Canada") = r_{1,1} \quad MaxNetInsDTDCCFP(OM\_QC,"Canada") = r_{2,1}$$
$$MaxNetInsDTDCCFP(DSPS\_ON,"Canada") = r_{4,1} \quad MaxNetInsDTDCCFP(DSPS\_QC,"Canada") = r_{5,1}$$

The maximum stress DCCFP of SDs *OC* and *PRNF* can be calculated as:

$$MaxNetInsDTDCCFP(OC,"Canada") = r_{3,1} \quad MaxNetInsDTDCCFP(PRNF,"Canada") = r_{6,1}$$

Step 2

According to Step 2 of Algorithm 3, we should now choose an ISDS (Independent-SD Set) which entails maximum instant stress on network *Canada*. We derived the Independent-SD Sets of SCAPS in Section 12.4.4. Now, we have to find the *MaxNetInsDT* of every ISDS. For example, for $ISDS_1 = \{OM\_ON, OM\_QC, DSPS\_ON\}$, $MaxNetInsDTValue(ISDS_1)$ can be calculated as:

$$\begin{aligned} MaxNetInsDTValue(ISDS_1) &= MaxNetInsDTValue(MaxNetInsDTDCCFP(OM\_ON,"Canada"),"Canada") \\ &+ MaxNetInsDTValue(MaxNetInsDTDCCFP(OM\_QC,"Canada"),"Canada") \\ &+ MaxNetInsDTValue(MaxNetInsDTDCCFP(DSPS\_ON,"Canada"),"Canada") \\ &= MaxNetInsDTValue(r_{1,1},"Canada") + MaxNetInsDTValue(r_{2,1},"Canada") \\ &+ MaxNetInsDTValue(r_{4,1},"Canada") \\ &= 2.8 + 2.6 + 3 = 8.4MB \end{aligned}$$

Similarly, the *MaxNetInsDT* of other ISDS's can be calculated as:

$$MaxNetInsDTValue(ISDS_2) = 2.8+2.6+3 = 8.4MB \quad MaxNetInsDTValue(ISDS_3) = 2.8+2.6+0.3 = 5.7MB$$
$$MaxNetInsDTValue(ISDS_4) = 2.8+3+3 = 8.8MB \quad MaxNetInsDTValue(ISDS_5) = 2.6+3+3 = 8.6MB$$
$$MaxNetInsDTValue(ISDS_6) = 0.6+3+3 = 6.6MB \quad MaxNetInsDTValue(ISDS_7) = 0.6+0.3 = 0.9MB$$

We now need to choose the ISDS which has the maximum value of the above function. We refer to this ISDS as $ISDS_{max}$. As calculated above, it is evident that $ISDS_4$ is $ISDS_{max}$.

Step 3

Step 3 of Algorithm 3 is to schedule the SDs of $ISDS_{max}$, chosen in the Step 2, so that all maximum stress messages execute at the same time. First, the latest start time among the selected DCCFPs of all SDs in $ISDS_{max}$ should be calculated.

$$\begin{aligned} DCCFPsLatestStartTime &= \max_{\forall SD_i \in ISDS_{max}=\{OM\_QC, DSPS\_ON\}} \left(\min_{\forall m \in MaxNetInsDTMsgs(MaxNetInsDTDCCFP(SD_i,"Canada"),"Canada")} (m.start) \right) \\ &= max\left(\min_{\forall m \in MessageSet_1}(m.start), \min_{\forall m \in MessageSet_2}(m.start), \min_{\forall m \in MessageSet_3}(m.start) \right) \\ &= max(A_{12}.start, D_{12}.start, E_{10}.start) \\ &= max(500ms, 600ms, 500ms) = 600ms \end{aligned}$$

where:

- $MessageSet_1 = MaxNetInsDTMsgs(MaxNetInsDTDCCFP(OM\_ON,"Canada"),"Canada")$
- $MessageSet_2 = MaxNetInsDTMsgs(MaxNetInsDTDCCFP(DSPS\_ON,"Canada"),"Canada")$
- $MessageSet_3 = MaxNetInsDTMsgs(MaxNetInsDTDCCFP(DSPS\_QC,"Canada"),"Canada")$

Now, we use *DCCFPsLatestStartTime* to schedule those SDs of $ISDS_{max}$, which have a DCCFP going through network *Canada* (Step 3.2 of Algorithm 3). As $ISDS_{max} = \{OM\_ON, DSPS\_ON, DSPS\_QC\}$, we need to schedule SDs *OM_ON*, *DSPS_ON* and *DSPS_QC*. As presented in Step 3.2 of Algorithm 3, stress test schedule is an ordered set of tuples $STS_i = (r_{i\max}, ar_{i\max})$ where $r_{i\max}$ is the maximum DCCP of SD_i, calculated using $r_{i\max} = MaxNetInsDTDCCFP(SD_i, net)$, and $ar_{i\max}$ is $r_{i\max}$'s start time and is equal to $ar_{i\max} = DCFPsLatestStartTime - \min(MaxNetInsDTMsgs(r_{i\max}, net).start)$. If SD_i is not a member of the selected $ISDS_{max}$ or it does not have a DCCFP going through network *Canada*, its tuple in the stress test schedule will be null. Therefore, the stress test schedule for the current test objective will be:

$$StressTestSchedule_{TestElement1} = < (r_{1,1}, ar_{1,1}), null, null, (r_{4,1}, ar_{4,1}), (r_{5,1}, ar_{5,1}), , null >$$

where $ar_{1,1}$ and $ar_{4,4}$ can be calculated as:

$$\begin{aligned} ar_{1,1} &= DCFPsLatestStartTime - \min(MaxNetInsDTMsgs(r_{1,1},"Canada").start), \\ &= 600ms - 500ms = 100ms \end{aligned}$$

$$\begin{aligned} ar_{4,1} &= DCFPsLatestStartTime - \min(MaxNetInsDTMsgs(r_{4,1},"Canada").start) \\ &= 600ms - 600ms = 0ms \end{aligned}$$

and

$$\begin{aligned} ar_{5,1} &= DCFPsLatestStartTime - min(MaxNetInsDTMsgs(r_{5,1},"Canada").start) \\ &= 600ms - 500ms = 100ms \end{aligned}$$

Therefore:

$$StressTest\ Schedule_{TestElement1} = < (r_{1,1}, 100ms), null, null, (r_{4,1}, 0ms), (r_{5,1}, 100ms), null >$$

The stress test schedule indicates that in order to *instant* stress test network *Canada* in terms of *data* traffic, we have to execute DCCFPs $r_{1,1}$ of SD *OM_ON*, $r_{4,1}$ of SD *DSPS_ON* $r_{5,1}$ of SD *DSPS_QC* and in time=100ms, time=0ms and time=100ms, respectively. As discussed in Chapter 9, these are test requirements and we need to derive appropriate test cases for them.

12.4.8.2 Test Objective 2: (SEV_CA1, in, data traffic, interval)

The stress test location in this element is node *SEV_CA1*. Stress direction is "in", stress type is "data traffic", and stress duration is "interval".

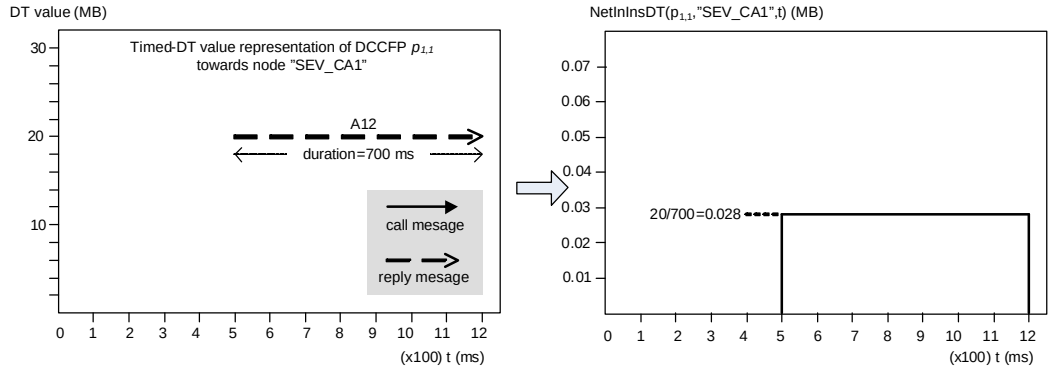
We use the *StressNodInIntDT(nod)* stress test requirement generation technique, described in Chapter 9. We present below the steps of the algorithm to derive test requirements.

Step 1

Calculate Unit Data Traffic (UDT) of each DCCFP towards node *SEV_CA1* using:

$$NodInUDT(r_{ij}, nod) = \frac{\sum_t (NodInInsDT(r_{ij}, nod, t))}{Duration(r_{ij})}$$

We will need the values of the function under sigma to calculate the UDT value of each DCCFP. As an example, we present here how to calculate the value of this function for $r_{1,1}$. The values of function $NodInInsDT(r_{1,1}, SEV\_CA1, t)$ in different time instances are derived from the Timed-DT value representation of the DCCFP towards node *SEV_CA1* and are sketched below.



$\sum_t (NodInInsDT(r_{ij}, nod, t))$ yields 20 MB. Hence:

$$NodInUDT(r_{1,1}, "SEV\_CA1") = \frac{\sum_t (NodInInsDT(r_{1,1}, SEV\_CA1, t))}{Duration(r_{1,1})} = \frac{20}{1200} \approx 0.016 MB/ms$$

Calculating the Unit Data Traffic (UDT) of other DCCFP's will yield us:

$$NodInUDT(r_{2,1},"SEV\_CA1") = \frac{16}{1000} \approx 0.016\,MB/ms \qquad NodInUDT(r_{5,1},"SEV\_CA1") = \frac{12}{1100} \approx 0.0109\,MB/ms$$
$$NodInUDT(r_{3,1},"SEV\_CA1") = \frac{0}{1000} \approx 0MB/ms \qquad NodInUDT(r_{6,1},"SEV\_CA1") = \frac{0}{1000} \approx 0MB/ms$$
$$NodInUDT(r_{3,2},"SEV\_CA1") = \frac{0}{1000} \approx 0MB/ms \qquad NodInUDT(r_{6,2},"SEV\_CA1") = \frac{0}{1000} \approx 0MB/ms$$
$$NodInUDT(r_{3,3},"SEV\_CA1") = \frac{0}{800} \approx 0MB/ms \qquad NodInUDT(r_{6,3},"SEV\_CA1") = \frac{0}{1000} \approx 0MB/ms$$
$$NodInUDT(r_{3,4},"SEV\_CA1") = \frac{0}{50} \approx 0MB/ms \qquad NodInUDT(r_{6,4},"SEV\_CA1") = \frac{0}{50} \approx 0MB/ms$$
$$NodInUDT(r_{4,1},"SEV\_CA1") = \frac{15}{1300} \approx 0.0115\,MB/ms$$

Now, using the above values, we find, among all DCCFPs of each SD, the one with maximum unit data traffic.

$$MaxNodInPerDTDCCFP(OM\_ON,"SEV\_CA1") = r_{1,1}$$
$$MaxNodInPerDTDCCFP(OM\_QC,"SEV\_CA1") = r_{2,1}$$
$$MaxNodInPerDTDCCFP(OC,"SEV\_CA1") = null$$
$$MaxNodInPerDTDCCFP(DSPS\_ON,"SEV\_CA1") = r_{4,1}$$
$$MaxNodInPerDTDCCFP(DSPS\_QC,"SEV\_CA1") = r_{5,1}$$
$$MaxNodInPerDTDCCFP(PRNF,"SEV\_CA1") = null$$

Note that we arbitrarily choose one of the DCCFPs of a SD, in case if they have equal *NodInUDT* values. Also, the output of the above function for a SD, which does not have any DT towards node *SEV_CA1*, is *null*.

Step 2

We calculate each CSDFP's UDT value towards node *SEV_CA1* using:

$$NodInUDT(CSDFP_i, nod) = \frac{\sum_{\forall SD \in CSDFP_i} \sum_{\forall t} NodInInsDT(MaxNodInPerDTDCCFP(SD, nod), nod, t)}{Duration(SelectCCFPS(CSDFP_i, MaxNodInPerDT, nod))}$$

As discussed in Section 12.4.5, SCAPS has unlimited number of CSDFPs. We presented the primitive ones in Figure 128. Here we calculate the UDT value of the primitive CSDFPs and discuss how the UDT value of other CSDFPs can be computed.

As an example, we show here how to calculate the value of this function for:

$$CSDFP_2 = \begin{pmatrix} OM\_STARTUP\begin{pmatrix} OM\_ON \\ OM\_QC \end{pmatrix} OC \\ \begin{pmatrix} DSPS\_ON \\ DSPS\_QC \end{pmatrix} \end{pmatrix}.$$

$$NodInUDT(CSDFP_2,"SEV\_CA1") = \frac{\sum_{\forall SD \in CSDFP_2} \sum_{\forall t} NodInInsDT(MaxNodInPerDTDCCFP(SD,"SEV\_CA1"),"SEV\_CA1",t)}{Duration(SelectCCFPS(CSDFP_2, MaxNodInPerDT,"SEV\_CA1"))}$$
$$= \frac{\begin{pmatrix} \sum_{\forall t} NodInInsDT(MaxNodInPerDTDCCFP(OM\_STARTUP,"SEV\_CA1"),"SEV\_CA1",t) + \\ \sum_{\forall t} NodInInsDT(MaxNodInPerDTDCCFP(OM\_ON,"SEV\_CA1"),"SEV\_CA1",t) + \\ \sum_{\forall t} NodInInsDT(MaxNodInPerDTDCCFP(OM\_QC,"SEV\_CA1"),"SEV\_CA1",t) + \\ \sum_{\forall t} NodInInsDT(MaxNodInPerDTDCCFP(DSPS\_ON,"SEV\_CA1"),"SEV\_CA1",t) + \\ \sum_{\forall t} NodInInsDT(MaxNodInPerDTDCCFP(DSPS\_QC,"SEV\_CA1"),"SEV\_CA1",t) + \\ \sum_{\forall t} NodInInsDT(MaxNodInPerDTDCCFP(OC,"SEV\_CA1"),"SEV\_CA1",t) \end{pmatrix}}{Duration(SelectCCFPS(CSDFP_2, MaxNodInPerDT,"SEV\_CA1"))}$$
$$= \frac{\begin{pmatrix} 0 + \sum_{\forall t} NodInInsDT(r_{1,1},"SEV\_CA1",t) + \\ \sum_{\forall t} NodInInsDT(r_{2,1},"SEV\_CA1",t) + \sum_{\forall t} NodInInsDT(r_{3,1},"SEV\_CA1",t) + \\ \sum_{\forall t} NodInInsDT(r_{4,1},"SEV\_CA1",t) + \sum_{\forall t} NodInInsDT(r_{5,1},"SEV\_CA1",t) \end{pmatrix}}{Duration(SelectCCFPS(CSDFP_2, MaxNodInPerDT,"SEV\_CA1"))} = \frac{0+20+16+0+15+12}{2800} \approx 0.0225 MB/ms$$

We describe here how we calculate the value of the function *Duration* in the expression above. To do so, we need to describe how a CCFPS is built from $CSDFP_2$, with the criterion *MaxNodInIntDT*. Let us refer to this CCFPS as $CCFPS_2$. In order to build a corresponding CCFPS, each of the SDs in $CSDFP_2$, are replaced with the CCFP associated with maximum stress DCCFPs which are results of the *MaxNodInIntDTDCCFP* function. Therefore:

$$SelectCCFPS(CSDFP_2, MaxNodInPerDT, "SEV\_CA1")$$
$$= \begin{pmatrix} MaxNodInPerDTDCCFP(OM\_STARTUP, "SEV\_CA1") \\ \begin{pmatrix} MaxNodInPerDTDCCFP(DSPS\_ON, "SEV\_CA1") \\ MaxNodInPerDTDCCFP(DSPS\_QC, "SEV\_CA1") \end{pmatrix} \end{pmatrix} \begin{pmatrix} MaxNodInPerDTDCCFP(OM\_ON, "SEV\_CA1") \\ MaxNodInPerDTDCCFP(OM\_QC, "SEV\_CA1") \end{pmatrix} MaxNodInPerDTDCCFP(OC, "SEV\_CA1")$$
$$= \begin{pmatrix} CCFP(r_{0,0}) \begin{pmatrix} CCFP(r_{1,1}) \\ CCFP(r_{2,1}) \end{pmatrix} CCFP(r_{3,1}) \\ \begin{pmatrix} CCFP(r_{4,1}) \\ CCFP(r_{5,1}) \end{pmatrix} \end{pmatrix}$$

where *CCFP(aDCCFP)* returns the CCFP associated with a DCCFP.

We now briefly show how the duration of the above CCFPS is calculated. Since as discussed in Section x, the *Duration* function is a recursive one, we represent the call tree of the recursive algorithm in Figure 131 to better illustrate the idea. Note we assume duration of 500 ms for the only CCFP of SD *OM_STARTUP*, $CCFP_0 = CCFP(r_{0,0})$.

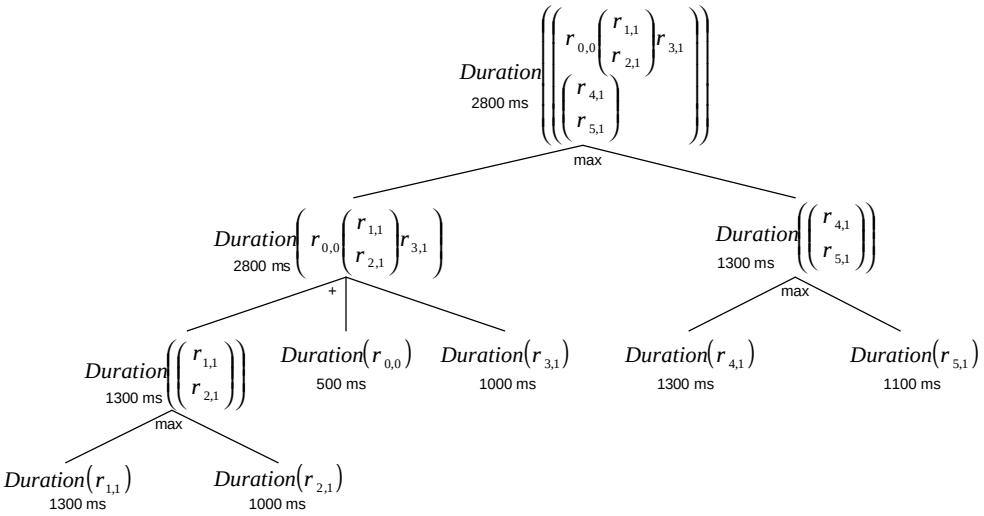


Figure 131-The call tree of the recursive algorithm *Duration* applied to $CCFPS_2$.

The UDT values of the other primitive CSDFPs towards node *SEV_CA1* can be calculated in a similar way.

$$NodInUDT(CSDFP_1, "SEV\_CA1") = \frac{20+16+15+12}{1800} \approx 0.035 MB/ms$$
$$NodInUDT(CSDFP_3, "SEV\_CA1") = \frac{20+16+15+12+0}{2800} \approx 0.0225 MB/ms$$
$$NodInUDT(CSDFP_4, "SEV\_CA1") = \frac{20+16+0+15+12+0}{2800} \approx 0.0225 MB/ms$$

The last step of the algorithm is to choose a CSDFP which has the highest UDT value among all CSDFPs. Therefore, we choose $CSDFP_1$ as $CSDFP_{max}$ for this test objective. Furthermore, the corresponding maximum stress DCCFPS is the test requirement for the current test objective. The corresponding DCCFPS is:

$$SelectCCFPS(CSDFP_2, MaxNodInPerDT, "SEV\_CA1") = \begin{pmatrix} CCFP(r_{0,0}) \begin{pmatrix} CCFP(r_{1,1}) \\ CCFP(r_{2,1}) \end{pmatrix} CCFP(r_{3,1}) \\ \begin{pmatrix} CCFP(r_{4,1}) \\ CCFP(r_{5,1}) \end{pmatrix} \end{pmatrix}$$

12.4.8.3 Test Objective 3: (SEV_ON, bidirectional, message traffic, instant)

The stress test location in this element is node *SEV_ON*. Stress direction is "bidirectional", stress type is "message traffic", and stress duration is "instant".

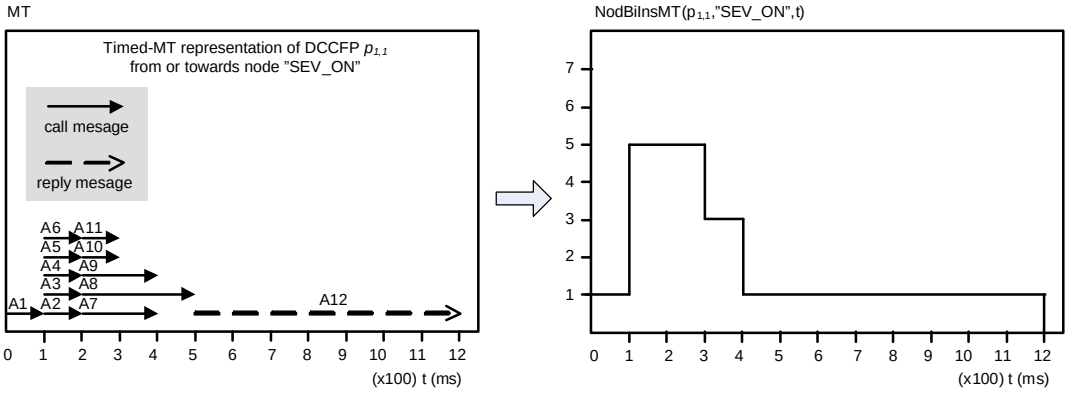
We use the *StressNodBiInsMT(nod)* stress test requirement generation technique, described in Chapter 9. We present below the steps of the algorithm to derive test requirements.

Step 1

Maximum node bidirectional instant message traffic (*MaxNodBiInsMT*) values of all DCCFPs are calculated. These values are shown below.

$$MaxNodBiInsMTValue(r_{1,1}, "SEV\_ON") = 5 \quad MaxNodBiInsMTValue(r_{4,1}, "SEV\_ON") = 5$$
$$MaxNodBiInsMTValue(r_{2,1}, "SEV\_ON") = 0 \quad MaxNodBiInsMTValue(r_{5,1}, "SEV\_ON") = 0$$
$$MaxNodBiInsMTValue(r_{3,1}, "SEV\_ON") = 1 \quad MaxNodBiInsMTValue(r_{6,1}, "SEV\_ON") = 1$$
$$MaxNodBiInsMTValue(r_{3,2}, "SEV\_ON") = 1 \quad MaxNodBiInsMTValue(r_{6,2}, "SEV\_ON") = 1$$
$$MaxNodBiInsMTValue(r_{3,3}, "SEV\_ON") = 1 \quad MaxNodBiInsMTValue(r_{6,3}, "SEV\_ON") = 1$$
$$MaxNodBiInsMTValue(r_{3,4}, "SEV\_ON") = 1 \quad MaxNodBiInsMTValue(r_{6,4}, "SEV\_ON") = 1$$

As an example on how these values are calculated, we discuss $MaxNodBiInsMTValue(r_{1,1}, "SEV\_ON")$. Timed-MT representation of DCCFP $r_{1,1}$ from or towards node *SEV_ON* and the value of function *NodBiInsMT($r_{1,1}$,"SEV_ON",t)* are shown below.



Among DCCFPs of each SD, we now find the one with maximum stress value.

$$MaxNodBiMTDCCFP(OM\_ON, "SEV\_ON") = r_{1,1} \quad MaxNodBiMTDCCFP(DSPS\_ON, "SEV\_ON") = r_{4,1}$$
$$MaxNodBiMTDCCFP(OM\_QC, "SEV\_ON") = null \quad MaxNodBiMTDCCFP(DSPS\_QC, "SEV\_ON") = null$$
$$MaxNodBiMTDCCFP(OC, "SEV\_ON") = r_{3,1} \quad MaxNodBiMTDCCFP(PRNF, "SEV\_ON") = r_{6,1}$$

Step 2

We should now choose an ISDS (Independent-SD Set) which entails maximum bidirectional instant message stress on node *SEV_ON*. We derived the Independent-SD Sets of SCAPS in Section 12.4.4. We have to find the *MaxNodBiInsMTValue* of every ISDS. For example, for $ISDS_1 = \{OM\_ON, OM\_QC, DSPS\_ON\}$, $MaxNodBiInsMTValue(ISDS_1)$ can be calculated as:

$$\begin{aligned}MaxNodBiInsMTValue(ISDS_1) &= MaxNodBiInsMTValue(MaxNetInsDTDCCFP(OM\_ON,"SEV\_ON"),"SEV\_ON")\\ &+ MaxNodBiInsMTValue(MaxNetInsDTDCCFP(OM\_QC,"SEV\_ON"),"SEV\_ON")\\ &+ MaxNodBiInsMTValue(MaxNetInsDTDCCFP(DSPS\_ON,"SEV\_ON"),"SEV\_ON")\\ &= MaxNodBiInsMTValue(r_{1,1},"SEV\_ON")\\ &+ MaxNodBiInsMTValue(r_{2,1},"SEV\_ON")\\ &+ MaxNodBiInsMTValue(r_{4,1},"SEV\_ON")\\ &= 5+0+5=10\end{aligned}$$

Similarly, the *MaxNodBiInsMTValue* of other ISDSs can be calculated as:

$$MaxNodBiInsMTValue(ISDS_2)=5+0+0=5 \quad MaxNodBiInsMTValue(ISDS_3)=5+1+1=7$$
$$MaxNodBiInsMTValue(ISDS_4)=5+5+0=10 \quad MaxNodBiInsMTValue(ISDS_5)=0+5+0=5$$
$$MaxNodBiInsMTValue(ISDS_6)=1+5+0=6 \quad MaxNodBiInsMTValue(ISDS_7)=1+1=2$$

We now need to choose the ISDS which has the maximum value of the above function. We refer to this ISDS as $ISDS_{max}$. As calculated above, $ISDS_1$ and $ISDS_4$ have the highest value of 10 messages. We arbitrarily choose $ISDS_1$ as $ISDS_{max}$.

Step 3

Step 3 of the algorithm is to schedule the SDs of $ISDS_{max}$, chosen in the Step 2, so that all maximum stress messages execute at the same time. First, the latest start time among the selected DCCFPs of all SDs in $ISDS_{max}$ should be calculated.

$$\begin{aligned}DCCFPsLatestStartTime &= \max_{\forall SD_i \in ISDS_{max}=\{OM\_ON,DSPS\_ON\}}\left(\min_{\forall m \in MaxNodBiInsMTMsgs(MaxNodBiInsMTDCCFP(SD_i,"SEV\_ON"),"SEV\_ON")}(m.start)\right)\\ &= max\left(\min_{\forall m \in MaxNodBiInsMTMsgs(r_{1,1},"SEV\_ON")}(m.start),\min_{\forall m \in MaxNodBiInsMTMsgs(r_{2,1},"SEV\_ON")}(m.start),\min_{\forall m \in MaxNodBiInsMTMsgs(r_{4,1},"SEV\_ON")}(m.start)\right)\\ &= max(A_2.start,null,D_2.start)=max(100ms,null,100ms)=100ms\end{aligned}$$

Now, we use *DCCFPsLatestStartTime* to schedule those SDs of $ISDS_{max}$, which have a DCCFP with bidirectional message traffic to/from node *SEV_ON* (Step 3.2 of Algorithm x). As $ISDS_{max}=\{OM\_ON,OM\_QC,DSPS\_ON\}$, we need to schedule SDs *OM_ON*, *OM_QC* and *DSPS_ON*. Stress test schedule is an ordered set of tuples $STS_i=(r_{i\max},ar_{i\max})$ where $r_{i\max}$ is the maximum DCCFP of SD_i, calculated using $r_{i\max}=MaxNodBiInsMTDCFP(SD_i,nod)$, and $ar_{i\max}$ is $r_{i\max}$'s start time and is equal to:

$$ar_{i\max}=DCFPsLatestStartTime-\min(MaxNodBiInsMTMsgs(r_{i\max},nod).start)$$

If SD_i is not a member of the selected $ISDS_{max}$ or it does not have a DCCFP going through node "SEV_ON", its tuple in the stress test schedule will be null. Therefore, the stress test schedule for the current test objective will be:

$$StressTest\ Schedule_{TestElement3}=<(r_{1,1},ar_{1,1}),null,null,(r_{4,1},ar_{4,1}),null,null>$$

where $ar_{1,1}$ and $ar_{4,4}$ can be calculated as:

$$\begin{aligned}ar_{1,1} &= DCFPsLatestStartTime-\min(MaxNodBiInsMTMsgs(r_{1,1},"SEV\_ON").start)\\ &= 100ms-100ms=0ms\end{aligned}$$

and

$$\begin{aligned}ar_{4,1} &= DCFPsLatestStartTime-\min(MaxNodBiInsMTMsgs(r_{4,1},"SEV\_ON").start)\\ &= 100ms-100ms=0ms\end{aligned}$$

Therefore:

$$StressTest\ Schedule_{TestElement3}=<(r_{1,1},0ms),null,null,(r_{4,1},0ms),null,null>$$

The stress test schedule indicates that in order to *instant* stress test node "SEV_ON" in terms of message traffic, we have to execute both DCCFPs $r_{1,1}$ of SD *OM_ON* and $r_{4,4}$ of SD *DSPS_ON* in time=0ms.

12.4.9 Derivation of Test Cases

We derived three sets of test requirements for the three test objectives in the previous sections. We derive here the corresponding test cases.

12.4.9.1 Test Objective 1

For the test objective 1, we derived the following stress test requirement (schedule):

$$StressTest\ Schedule_{TestElement1} = < (r_{1,1}, 100ms), null, null, (r_{4,1}, 0ms), null, null >$$

The stress test schedule indicates that in order to stress test network *Canada* in terms of data traffic in an instant time, we have to execute DCCFPs $r_{1,1}$ of SD *OM_ON* and $r_{4,4}$ of SD *DSPS_ON* in time=100ms and time=0ms, respectively. Since both SDs *OM_ON* and *DSPS_ON* have only one DCCFP (and CCFP) each, input values and conditions are not relevant to the design of test cases, only the schedule is.

12.4.9.2 Test Objective 2

The test requirement of this test objective was chosen in Section 12.4.8.2 to be the maximum stress DCCFPS of $CSDFP_1$ as:

$$BuildDCCFPS(CSDFP_1, MaxNodInPerDT, "SEV\_CA1") = \begin{pmatrix} r_{0,0}\begin{pmatrix} r_{1,1} \\ r_{2,1} \end{pmatrix} \\ \begin{pmatrix} r_{4,1} \\ r_{5,1} \end{pmatrix} \end{pmatrix}$$

Since the current test objective is an interval one, and the maximum stress requirement has been selected as the above DCCFPS, it can be repeated as many times as desired to perform an interval stress test. In other words, the $DCCFPS_{max}$ can be repeated an arbitrary k number of times.

$$DCCFPS_{\max} = \begin{pmatrix} r_{0,0}\begin{pmatrix} r_{1,1} \\ r_{2,1} \end{pmatrix} \\ \begin{pmatrix} r_{4,1} \\ r_{5,1} \end{pmatrix} \end{pmatrix}^k$$

Again, since $r_{0,0}$, $r_{1,1}$, $r_{2,1}$, $r_{4,1}$, and $r_{5,1}$ are the only DCCPFs of their corresponding SDs, input values and conditions are not relevant to the design of test cases. We only need to make sure SDs *OM_ON*, *OM_QC*, *DSPS_ON* and *DSPS_QC* start at the same time to satisfy the current test requirement.

12.4.9.3 Test Objective 3

As the test requirement for the test objective 3, we derived the following stress test schedule:

$$StressTest\ Schedule_{TestElement3} = < (r_{1,1}, 0ms), null, null, (r_{4,1}, 0ms), null, null >$$

The stress test schedule indicates that in order to *instant* stress test node *SEV_ON* in terms of message traffic, we must start executing both DCCFPs $r_{1,1}$ of SD *OM_ON* and $r_{4,4}$ of SD *DSPS_ON* at time=0ms. Therefore, test case for this test requirement is simply to run SDs *OM_ON* and *DSPS_ON* at the same time.

12.5 Stress Test Architecture

An overview of the SCAPS stress test architecture is shown in Figure 132. The sequence of high-level steps to be performed by a tester to run a complete stress test procedure is shown.

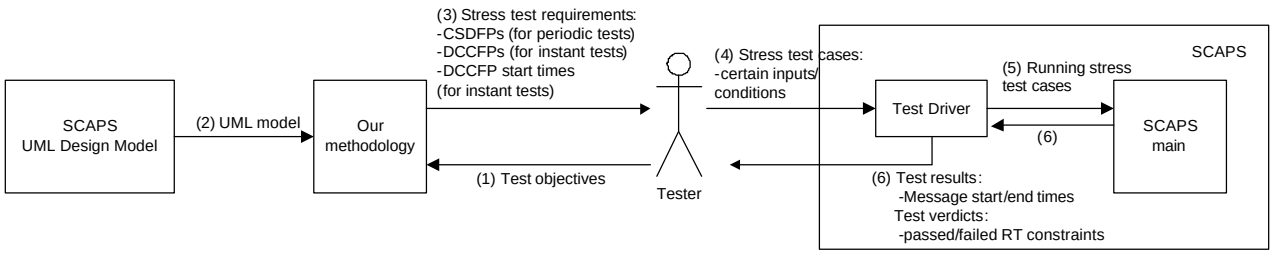


Figure 132-Overview of SCAPS Stress Test Architecture.

The steps are briefly explained below.

(1) 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) Tester devises appropriate test case for the test requirements. Note that this step is currently done manually by the tester. 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 system. 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 a test object 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 are negligible when compared to the SCAPS message durations (several hundreds of milliseconds, as it can be seen in the SCAPS SDs in Figure 110-Figure 115). 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[1]. This included features such as: easy selection of any subset of test cases and flexibility in scheduling DCCFPs.

(6) Test results are gathered from the system. They include: start/end times of distributed messages and test verdicts on real-time constraints, which specify whether each real-time constraint has been adhered to in a particular run. Test results are both logged in files and also displayed live in a text box to the tester. A high level of observability[2] 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.

[1] 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 [126].

[2] Observability is a measure for how well internal states of a system can be inferred by knowledge of its external outputs.

12.6 Running Stress Test Cases

As shown in the SCAPS stress test architecture in Figure 132, 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[1] 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 (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.

12.7 Test Results

Results of running the stress test cases, as derived in the previous sections, are reported and discussed in this section. 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 descried in Sections 12.7.2-12.7.4. We discuss in Section 12.7.1 how we derive OPTCs.

In the presentation of the test results, we compare the statistical start and end times of distributed messages and also determine if a stress test case causes a RT-constraint violation which is not observed while running OPTCs. 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 reported in this section, we analyze and discuss the MIOD-level soft and hard RT constraints described in Section 12.3.4.5. SD-level constraints can be defined in a similar way and the corresponding test results can also be analyzed.

12.7.1 Baseline of Comparisons

We define here what 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 [71] 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 116) and CCFGs (Figure 119 to Figure 124), we model the operational profile to be the probabilities in which the true and false edges of conditions are taken. More precisely, we focus on the decision nodes in the CCFGs of SDs *OC* and *PRNF*, Figure 121 and Figure 124, respectively. These two CCFGs are the only CCFGs of SCAPS where the control flow might actually change.

The two decision nodes in *CCFG(OC)* check for overload status in load data of provinces of Ontario and Quebec. In case of overload status in any of the provinces, a new load policy is generated and is sent to the

[1] 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.

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 12, where probabilities of per 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 12-An operational profile for SCAPS.

For example, Figure 133 shows a part of *CCFG(OC)* with decision node outgoing edges 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.

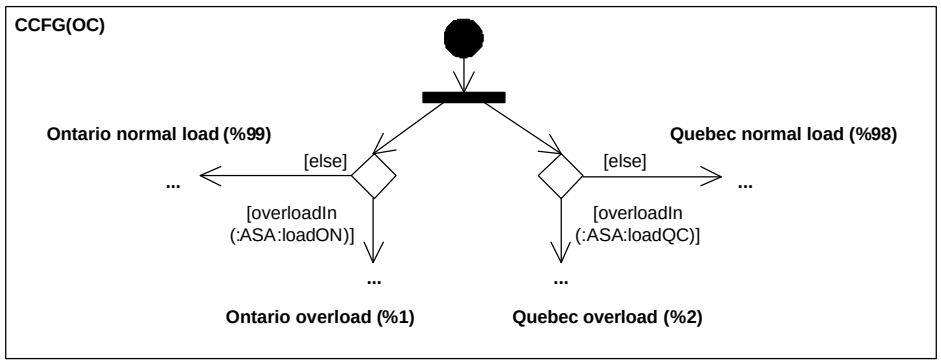


Figure 133-Part of *CCFG(OC)*, annotated with probabilities of paths after decision nodes.

Using the operational profile in Table 12, we can derive the probabilities of different DCCFPs in *OC* and *PRNF*. When probabilities of taking edges after decision nodes are given, the probability of any DCCFP can be calculated. For example DCCFP $r_{3,1}$ of SD *OC* corresponds to taking "Ontario overload" and "Quebec overload" edges of the decision node in *CCFG(OC)*, Figure 133. Using the operational profile in Table 12, 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 13.

SD	DCCFP	Probability
OC	$r_{3,1}$	%0.02
	$r_{3,2}$	%0.98
	$r_{3,3}$	%1.98
	$r_{3,4}$	%9702
PRNF	$r_{6,1}$	%0.00125
	$r_{6,2}$	~%0.0049
	$r_{6,3}$	~%0.0024
	$r_{6,4}$	%0.9925

Table 13-Probabilities of taking DCCFPs of SDs *OC* and *PRNF* according to the operational profile given in Table 12.

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 a 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 12) can be used to select CCFPs for each of the SD in the above CSDFP. DCCFPs probabilities (Table 13) can then be used to randomly select a DCCFP for each SD in a selected CSDFP.

12.7.2 Test Objective 1

The test requirements corresponding to the test objective 1 (Section 12.4.8.1) is to trigger DCCFP $r_{1,1}$ from SD *OM_ON* and $r_{4,4}$ from SD *DSPS_ON*. Referring to the SCAPS MIOD (Figure 116), we can see that *SRTC1* and *SRTC2* are visited by triggering these two DCCFPs. Therefore, we can say that test objective 1 is associated with *SRTC1* and *SRTC2* (Figure 116), and thus we report here how the duration[1] of the soft RT constraint *SRTC1* is affected when running test cases corresponding to test objective 1. Our experiments showed that *SRTC2* has a similar behavior.

In order to determine if stress testing makes a difference in the durations of *SRTC1* when compared the results with OPT test cases, we measured the executions of 500 randomly selected OPT test cases. We then ran 500 test cases corresponding to test objective 1 and collected the duration of *SRTC1* across all these runs. The comparison between Operational Profile Test (OPT) and Stress Test (ST) cases is depicted by the two execution time distributions in Figure 134. The x-axis is the test type and the y-axis is execution time. The quantiles and the histograms of the two distributions are depicted.

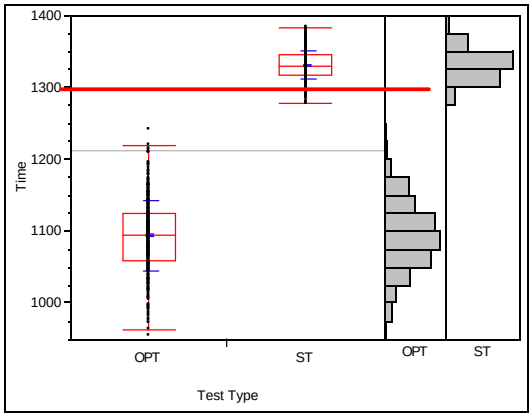


Figure 134-Execution times distributions of test suites corresponding to SRT constraint *SRTC1* by running operational profile test (OPT) and stress test (ST) cases corresponding to test objective 1.

[1] By "duration" of a RT constraint, we mean the time difference between the arrival times of the start and end events of a RT constraint.

Level	Min.	10%	25%	Median	75%	90%	Max.
OPT	953	1029	1059	1094	1125	1156	1241
ST	1276	1305	1317	1329	1344	1358	1382

Table 14-Quantiles of the distribution in Figure 134.

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 134. However, all OPT test executions satisfy *SRTC1* whereas *SRTC1* is violated in almost 96.4% (482/500) of stress test cases.

12.7.3 Test Objective 2

As derived in Section 12.4.9.2, the stress test requirement corresponding to test objective 2 was:

$$DCCFPS_{\max} = \left(r_{0,0} \begin{pmatrix} r_{1,1} \\ r_{2,1} \end{pmatrix} \begin{pmatrix} r_{4,1} \\ r_{5,1} \end{pmatrix} \right)^{k}$$

By examining the corresponding SDs of each of the DCCFPs of this DCCFPS with the RT constraints in the SCAPS MIOD (Figure 116), we can see that only *SRTC1* and *SRTC2* are visited when executing test objective 2. Therefore, we monitor the duration of these two constraints when executing OPTCs and stress test executions. The comparison of time distributions for *SRTC1* is shown in Figure 135.

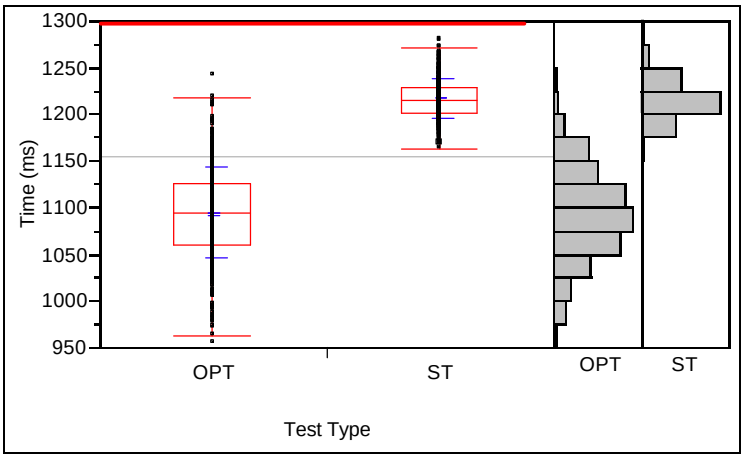


Figure 135- Execution times distributions for constraint *SRTC1* by running operational profile test (OPT) and stress test (ST) cases corresponding to test objective 2.

As it can be seen, there is a difference in the *SRTC1* time distributions between OPT and ST test cases corresponding to test objective 2. However, none of the time distributions indicate a RT constraint violation, as they are both below the 1,300 ms deadline. This result can be easily explained as the test case for test objective 2 is just a typical DCCFPS of the system, which is executed when neither overloaded situation nor a separated power grid is found and SCAPS continues (in a loop) to monitor the national power grid. Recall that, during the preliminary testing of SCAPS, we made sure that all RT constraints held in typical execution scenarios.

However, as the ST distribution shows significantly higher values (e.g., average) that the OPT distribution, we can conclude that stress testing has been successful in running the system under stress conditions. However, this was not enough to trigger a network-traffic or RT fault. This result suggests that not all stress test strategies may expose network-traffic or RT faults in a SUT, under specific settings for RT constraint values and network infrastructure (capacity, buffer sizes, etc.).

12.7.4 Test Objective 3

As derived in Section 12.4.9.3, the stress test requirement corresponding to test objective 3 was:

$$StressTest\ Schedule_{TestElement3} =< (r_{1,1}, 0ms), null, null, (r_{4,1}, 0ms), null, null >$$

By examining the corresponding SDs of each of the DCCFPs of this DCCFPS and the RT constraints in the SCAPS MIOD (Figure 116), we can determine that only *SRTC1* and *SRTC2* are visited when executing test objective 3. Therefore, we monitor the duration of these two constraints while executing OPT and ST test cases. The comparison is shown in Figure 136 and we can see, once again, that *SRTC1* is violated in most of ST test executions.

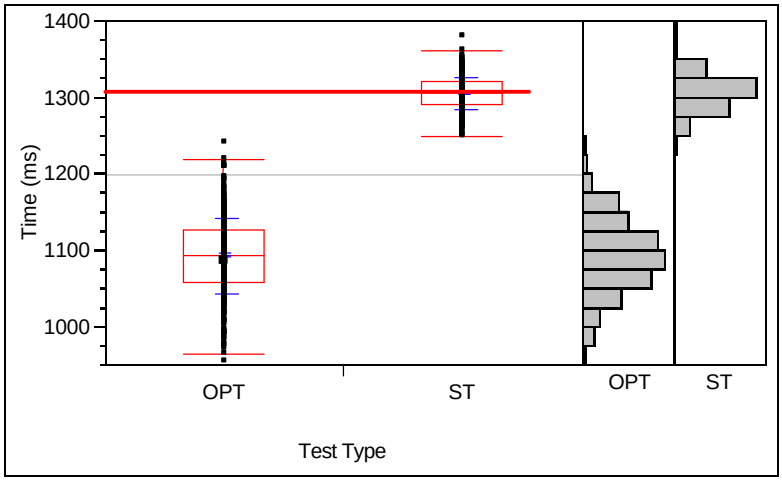


Figure 136- Execution times distributions of test suites corresponding to SRT constraint *SRTC1* by running operational profile tests (OPT) and stress tests (ST) corresponding to test objective 3.

12.7.5 Conclusions

In this chapter, using the specification of a real-world power distribution system, we designed and implemented a system and described how the stress test cases were derived and executed using our methodology. We also reported the results of applying our stress test methodology on this system and discussed its effectiveness in detecting violations of real-time constraints when compared to test cases based on an operational profile. The results are promising as they suggest that our stress test cases can help significantly increase the probability of exhibiting network traffic-related faults in distributed systems.

Chapter 13

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

13.1 Conclusions

A model-driven, stress test methodology aimed at increasing chances of discovering faults related to network 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 network traffic they entail. This data 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 the 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 network traffic-related faults in distributed systems.

13.2 Open Questions

The open questions we are now working on are: (1) How can we account for data flow and parameters in the SD sequential constraint modeling?, (2) How can we account for the variation in the data traffic value of a distributed message during its execution?. We also need to perform further, larger scale investigations of the Genetic Algorithm-based stress test technique in terms of its capacity to reveal distributed faults.

13.3 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 3.2.1) 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.

The UML Testing Profile [127] 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 [127]. 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 specifying testing 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 has been 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 [128].

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.

It would also be important to develop a Stress-Test based Performance Engineering (STPE) approach which can assist testers and system analysts in fixing distribution-related faults. Following STPE, the designer would use 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.

Risk assessment / fault analysis of distributed-type faults, the investigation of *QoS faults* and 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 in its required QoS requirement. We also intend to stress test more complex distributed systems using our methodology and perform more empirical investigations of its effectiveness.

ACKNOWLEDGMENTS

This work was in part supported by Siemens Corporate Research, Princeton, NJ and the Canada research chair in Software Quality Engineering.

REFERENCES

[1] M. Wall, "GAlib: A C++ Library of Genetic Algorithm Components," v2.4, Document Revision B, Massachusetts Institute of Technology 1996.

[2] 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.

[3] 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.

[4] E. Weyuker and F. I. Vokolos, "Experience with Performance testing of Software Systems: Issues, an Approach and Case Study," *IEEE Trans. of Soft. Eng.*, vol. 26, no. 12, pp. 1147-1156, 2000.

[5] R. Kuhn, "Sources of Failure in the Public Switched Telephone Network," *IEEE Computer*, vol. 30, no. 4, pp. 31-36, 1997.

[6] Object Management Group (OMG), "Unified Modeling Language Specification (v1.3)," 1999.

[7] Object Management Group (OMG), "Unified Modeling Language Specification (v1.5)," 2003.

[8] Object Management Group (OMG), "UML 2.0 Superstructure Final Adopted specification," http://www.omg.org/docs/ptc/03-08-02.pdf Sept. 2003.

[9] T. Pender, *UML Bible*: Wiley, Sept. 2003.

[10] Object Management Group (OMG), "UML Profile for Schedulability, Performance, and Time (v1.0)," 2003.

[11] C. S. D. Yang, "Identifying Potentially Load Sensitive Code Regions for Stress Testing," 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.

[12] J. Zhang and S. C. Cheung, "Automated Test Case Generation for the Stress Testing of Multimedia Systems," *Software Practice & Experience*, vol. 32, no. 15, pp. 1411-1435, 2002.

[13] A. Avritzer and E. J. Weyuker, "The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software," *IEEE Trans. on Software Eng*, vol. 21, no. 9, pp. 705-716, 1995.

[14] L. C. Briand, Y. Labiche, and M. Shousha, "Automating Stress Testing for Real-Time Systems Using Genetic Algorithms," Technical Report SCE-03-23, Carleton University Sept. 2003.

[15] A. Avritzer and B. Larson, "Load Testing Software Using Deterministic State Testing," International Symposium on Software Testing and Analysis (ISSTA), pp. 82-88, Cambridge, MA, 1993.

[16] L. C. Briand, Y. Labiche, and M. Shousha, "Automating Stress Testing for Real-Time Systems Using Genetic Algorithms," Proc. of Genetic and Evolutionary Computation Conference, Search-based Software Engineering Track, pp. 1021-1028, 2005.

[17] W. Brauer, W. Reisig, and G. R. (eds.), "Petri nets, central models and their properties," in *Advances in Petri Nets 1986, Part I. Proceedings of an Advanced Course, Bad Honnef, Lecture Notes in Computer Science*, vol. 254. Berlin: Springer, 1987.

[18] J. F. Allen, "Maintaining Knowledge about Temporal Intervals," *Communications of the ACM*, vol. 26, no. 11, pp. 832-843, 1983.

[19] R. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*: Addison-Wesley, 1999.

[20] I. P. Paltor and J. Lilius, "Digital Sound Recorder: a Case Study on Designing Embedded Systems Using the UML Notation," Turku Centre for Computer Science, Finland TUCS Technical Report No. 234, 1999.

[21] B. Douglass, *Doing Hard Time, Developing Real-Time Systems with UML Objects, Frameworks, and Patterns*: Addison Wesley, 1999.

[22] D. Herzberg, "UML-RT as a Candidate for Modeling Embedded Real-Time Systems in the Telecommunication Domain," 2nd International Conference on the Unified Modeling Language (UML'99), pp. 331-338, Fort Collins, Colorado, USA, 1999.

[23] L. Kabous and W. Neber, "Modeling Hard Real Time Systems with UML: The OOHARTS Approach," 2nd International Conference on the Unified Modeling Language (UML'99), pp. 339-355, Fort Collins, Colorado, USA, 1999.

[24] A. Lanusse, S. Gerard, and F. Terrier, "Real-Time Modeling with UML: The ACCORD Approach," 1st International Conference on the Unified Modeling Language (UML'98), pp. 319-335, Mulhouse, France, 1998.

[25] J. Hakansson, L. Mokrushin, P. Pettersson, and W. Yi, "An Analysis Tool for UML Models with SPT Annotations," Presented at SVERTS2004, Lisbon, Portugal, October 2004.

[26] S. Bernardi, S. Donatelli, and J. Merseguer, "From UML sequence diagrams and statecharts to analysable petri net models," the third international workshop on Software and performance (WOSP), pp. 35-45, Rome, Italy, 2002.

[27] C. M. Woodside and D. C. Petriu, "Capabilities of the UML Profile for Schedulability Performance and Time (SPT)," 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.

[28] D. C. Petriu, "Performance Analysis Based on the UML SPT Profile," tutorial given at QEST'2004, Enschede, The Netherlands, September 2004.

[29] D. C. Petriu and C. M. Woodside, "Extending the UML Profile for Schedulability Performance and Time (SPT) for component-based systems," 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.

[30] B. P. Douglass, "Rhapsody 5.0: Breakthroughs in Software and Systems Engineering," I-Logix Corp. whitepaper 2003.

[31] A. Avizienis, J.-C. Laprie, and B. Randell, "Fundamental concepts of dependability," LAAS (Laboratory for Analysis and Architecture of Systems) 01-145, April 2001.

[32] Y. Huang, P. Jalote, and C. Kintala, "Two Techniques for Transient Software Error Recovery," *Lecture Notes in Computer Science (LNCS)*, vol. 774, pp. 159-170, 1994.

[33] J. Gray, "Why do Computers Stop and What Can be Done About it?," Proc. of 5th Symposium on Reliability in Distributed Software and Database Systems, pp. 3-12, Los Angeles, California, USA, 1986.

[34] J. Gray, "A Census of Tandem System Availability Between 1985 and 1990," *IEEE Trans. on Reliability*, vol. 39, pp. 409-418, 1990.

[35] M. Sullivan and R. Chillarege, "Software Defects and Their Impact on System Availability - A Study of Field Failures in Operating Systems," Proc. 21st IEEE Intl. Symposium on Fault-Tolerant Computing, pp. 2-9, 1991.

[36] R. Chillarege, S. Biyani, and J. Rosenthal, "Measurement of Failure Rate in Widely Distributed Software," Proc. of 25th IEEE Intl. Symposium on Fault Tolerant Computing, pp. 424-433, Pasadena, CA, USA, July 1995.

[37] I. Lee, "Software Dependability in the Operational Phase," in *Department of Electrical and Computer Engineering*. Urbana-Champaign, IL: University of Illinois, 1995.

[38] I. Lee and R. K. Iyer, "Software Dependability in the Tandem GUARDIAN System," *IEEE Trans. on Software Engineering*, vol. 21, no. 5, pp. 455-467, May 1995.

[39] A. S. Tanenbaum, *Computer Networks*, Fourth ed: Prentice Hall, 2003.

[40] A. Ganesh, N. O'Connell, and D. Wischik, *Big Queues*: Springer Publication, 2004.

[41] B. P. Douglass, *Real Time UML: Advances in the UML for Real-Time Systems*, 3rd ed: Addison-Wesley Professional, 2004.

[42] J. M. Bacon, *Concurrent Systems: Operating systems, database and distributed systems, an integrated approach*, Second ed: Addison Wesley, 1997.

[43] D. Hovemeyer and W. Pugh, "Finding Concurrency Bugs in Java," In Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs, St. John's, Newfoundland, Canada, July 25-26, 2004.

[44] Y. Ben-Asher, Y. Eytani, and E. Farchi, "Heuristics for finding concurrent bugs," In International Parallel and Distributed Processing Symposium, IPDPS 2003, PADTAD Workshop, 2003.

[45] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded Java Program Test Generation," Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande, pp. 181, Palo Alto, California, United States, 2001.

[46] S. D. Stoller, "Testing Concurrent Java Programs using Randomized Scheduling," Proceedings of the Second Workshop on Runtime Verification (RV), July 2002.

[47] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*: Addison-Wesley, 2000.

[48] Object Management Group (OMG), "UML 2.0 Superstructure Final Adopted specification," 2003.

[49] V. Garousi, L. Briand, and Y. Labiche, "Control Flow Analysis of UML 2.0 Sequence Diagrams," Proc. (to appear) of the European Conf. on Model Driven Architecture-Foundations and Applications, 2005.

[50] OMG, "UML 2.0 Superstructure Final Adopted specification," Object Management Group, www.omg.org Sept. 2003.

[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] A. Abdurazik and J. Offutt, "Using UML collaboration diagrams for static checking and test generation," Proceedings of the International Conference on the Unified Modeling Language, pp. 383-395, York, UK, 2-6th October, 2000.

[54] L. Briand and Y. Labiche, "A UML-based Approach to System Testing," *Journal of Software and Systems Modeling*, vol. 1, no. 1, pp. 10-42, 2002.

[55] F. Fraikin and T. Leonhardt, "SeDiTeC-testing based on sequence diagrams," In International Conference on Automated Software Engineering, pp. 261-266, Edinburgh, Scotland, 23–27 September 2002.

[56] Y. Wu, M.-H. Chen, and J. Offutt, "UML-based Integration Testing for Component-Based Software," In International Conference on COTS-Based Software Systems, 2003.

[57] 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.

[58] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. GilChrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development - The Fusion Method*: Prentice Hall, 1994.

[59] I. Lanese and U. Montanari, "A Graphical Fusion Calculus," Proceedings of the Workshop of the COMETA Project on Computational Metamodels, pp. 199-215, 2004.

[60] I. Lanese and U. Montanari, "Mapping Fusion and Synchronized Hyperedge Replacement into Logic Programming," Seminar on Foundations of Global Computing, 2005.

[61] R. J. A. Buhr, "Use Case Maps as Architectural Entities for Complex Systems," *IEEE Transactions on Software Engineering*, vol. 24, no. 12, December 1998.

[62] J. F. Allen, "Maintaining knowledge about temporal intervals," *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, 1983.

[63] C. Nebut, F. Fleurey, Y. L. Traon, and J.-M. Jézéquel, "Requirements by Contracts allow Automated System Testing," th International Symposium on Software Reliability Engineering (ISSRE), Denver, Colorado, November 17 - 21, 2003.

[64] T. Pender, *UML Bible*: Wiley, 2003.

[65] OMG, "UML 2.0 Superstructure Final Adopted specification," 2003.

[66] OMG, "UML Profile for Schedulability, Performance, and Time (v1.0)," 2003.

[67] S. Muchnick, *Advanced Compiler Design and Implementation*, First ed: Morgan Kaufmann, 1997.

[68] E. W. Weisstein, "Strongly Connected Component," *From MathWorld--A Wolfram Web Resource.* *http://mathworld.wolfram.com/StronglyConnectedComponent.html*, 2005.

[69] R. E. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM J. Comput.*, vol. 1, no. 146-160, 1972.

[70] Sun Microsystems, "Trail: Learning the Java Language," in *http://java.sun.com/docs/books/tutorial/java/nutsandbolts/datatypes.html*, 2005.

[71] M. S. Gittens, "The Extended Operational Profile Model for Usage-Based Software Testing," Doctoral Thesis - University of Western Ontario, 2004.

[72] M. J. Atallah, *Handbook of Algorithms and Theory of Computation*: CRC Press, 1999.

[73] 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.

[74] R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms*: Wiley-Interscience, 1998.

[75] J. Lahtinen, P. M. Silander, and H. Tirri, "Empirical Comparison of Stochastic Algorithms," Proceedings of the Nordic Workshop on Genetic Algorithms and their Applications, pp. 45-60, 1996.

[76] P. Chardaire, A. Kapsalis, J. W. Mann, V. J. Rayward-Smith, and G. D. Smith, "Applications of Genetic Algorithms in Telecommunications," Proc. Applications of Neural Networks to Telecommunications, pp. 290-299, 1995.

[77] S. W. Mahfoud and D. E. Goldberg, "Parallel Recombinative Simulated Annealing: A Genetic Algorithm," *Parallel Computing*, vol. 21, no. 1, 1995.

[78] S. Y. Mahfouz, "Design Optimization of Structural Steel Work," Ph.D. Thesis, Dept. of Civil and Environmental Eng., University of Bradford, UK, 1999.

[79] K. De Jong, "Learning with Genetic Algorithms: An overview," *Machine Learning*, no. 2, pp. 121-138, 1988.

[80] J. J. Grefenstette and H. G. Cobb, "Genetic Algorithms for Tracking Changing Environments," Proc. of the 5th Int. Conf. on Genetic Alg, pp. 523-530, 1993.

[81] 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," Proceedings of the third international conference on Genetic algorithms, pp. 51-60, 1989.

[82] A. E. Eiben, P. E. Raue, and Z. Ruttkay, "Solving Constraint Satisfaction Problems using Genetic Algorithms," Proc. IEEE World Conference on Evolutionary Computing, pp. 542-547, 1994.

[83] M. A. Pawlowsky, "Crossover Operators," *Handbook of Genetic Algorithms Applications,*, vol. 1, pp. 101-114, 1995.

[84] M. A. Pawlowsky, "Crossover Operators," Practical Handbook of Genetic Algorithms Applications, L. Chambers Ed., pp. 101-114, 1995.

[85] T. Back, "Towards a Practice of Autonomous Systems," Proc. European Conference on Artificial Life, pp. 263-271, 1992.

[86] H. Mühlenbein, "Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization," Proceedings of the third international conference on Genetic algorithms, pp. 416-421, 1989.

[87] J. E. S. a. T. C. Fogarty, "Adaptively Parameterized Evolutionary Systems: Self Adaptive Recombination and Mutation in a Genetic Algorithm," *The International Conference on Parallel Problem Solving From Nature*, pp. 441-450, 1996.

[88] S. J. Louis and G. J. E. Rawlins, "Predicting Convergence Time for Genetic Algorithms," Computer Science Department, Indiana University, Technical Report 370, 1993.

[89] S. Mackay, E. Wright, and J. Park, *Practical Data Communications for Instrumentation and Control*: Newnes, June, 2003.

[90] A. Daneels and W. Salter, "What is SCADA?," Int. Conf. on Accelerator and Large Experimental Physics Control Systems, 1999.

[91] S. C. Bhatia, "Indusirial Scada: Scada Control Systems In Integrated Steel Plants," Proceedings of Power Quality '98, Santa Clara, California, 1998.

[92] J. Brunton, G. Digby, and A. Doherty, "Design and Operational Philosophy for a Metro Power Network SCADA System," Fourth International Conference on Power System Control and Management, pp. 176-180, London, UK, 1996.

[93] Y. Ebata, H. Hayashi, Y. Hasegawa, S. Komatsu, and K. Suzuki, "Development of the Intranet-based SCADA (supervisory control and data acquisition system) for power system," IEEE Power Engineering Society Winter Meeting, pp. 1656-1661, 2000.

[94] T. Seki, T. Tsuchiya, T. Tanaka, H. Watanabe, and T. Seki, "Network Integrated Supervisory Control for Power Systems based on Distributed Objects," Proceedings of the 2000 ACM symposium on Applied computing, pp. 620-626, Como, Italy, 2000.

[95] M. Mavrin, V. Koroman, and B. Borovic, "SCADA in Hydropower Plants," Proceedings of the IEEE International Symposium on Computer Aided Control System Design, Hawai'i, USA, August 1999.

[96] E.-K. Chan and H. Ebenhoh, "The Implementation and Evolution of a SCADA System for a Large Distribution Network," *Transactions on Power Systems*, vol. 7, no. 1, pp. 320-326, 1992.

[97] D. Trung, "Modern SCADA Systems for Oil Pipelines," Petroleum and Chemical Industry Conference, pp. 299-305, Denver, CO, USA, 1995.

[98] 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.

[99] J. A. How, J. W. Farthing, and V. Schmidt, "Trends in Computing Systems for Large Fusion Experiments," Proceedings of 22nd Symposium on Fusion Technology (SOFT), Helsinki, Finland, 2002.

[100] 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," IEEE Power Engineering Society Winter Meeting, pp. 695-700, Jan. 2002.

[101] R. Wakizono, T. Kawamura, T. Tsuchiya, T. Hatanaka, and T. Tanaka, "Object-oriented Database Management System for Process Control Systems-Development and Evaluation," Proc. of the ACM Symp. on Applied Computing, pp. 204-209, 1999.

[102] K. C. Thramboulidis, "Towards a UML-based Engineering Support System," 9th IEEE Mediterranean Conference on Control and Automation, MED'01, Croatia, 2001.

[103] K. Thramboulidis, "Development of Distributed Industrial Control Applications: The CORFU Framework," 4th IEEE International Workshop on Factory Communication Systems, Vasteras, Sweden, August 2002.

[104] T. Brown, A. Pasetti, W. Pree, T. A. Henzinger, and C. M. Kirsch, "A Reusable and Platform-independent Framework for Distributed Control Systems," Proc. of the Digital Avionics Systems Conference, pp. 1-11, 2001.

[105] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A Time-triggered Language for Embedded Programming," *Lecture Notes in Computer Science*, vol. 2211, pp. 166-184, 2001.

[106] A. Pasetti, "A Software Framework for Satellite Control Systems – Methodology and Development," in *PhD Dissertation, University of Konstanz*, Feb. 2001.

[107] H. Brand, D. Beck, E. Gaul, W. Geithner, S. Götte, T. Kühl, K. Poppensieker, M. Roth, and U. Thiemer, "The PHELIX Control System Based on UML Design Level Programming in LabVIEW," Proceedings of ninth International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS), Gyeongju, Korea, 2003.

[108] R. Jamal and H. Pichlik, *LabVIEW Applications and Solutions*: Prentice Hall, 1999.

[109] E. W. Gaul, "PHELIX (Petawatt High Energy Laser for heavy Ion eXperiments)," GSI Scientific Report 2003.

[110] M. Ivey, A. Akhil, D. Robinson, K. Stamber, and J. Stamp, "Accommodating Uncertainty in Planning and Operations," Transmission Reliability Program, Office of Power Technologies, U.S. Department of Energy 1999.

[111] 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, Oct 2003.

[112] ABB Co., "ABB Group Annual Report," http://www.abb.com/Global/Clabb/CLABB155.NSF/viewunid/2DDB4104B522E26A04256C3000527EEB/$file/ABB_TECH_E-Annual2000.pdf 2000.

[113] 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, no. 5, 2005.

[114] The Liberty Consulting Group, "Nova Scotia Power Inc. Power Outage Review," Nova Scotia Utility and Review Board, http://www.nspower.ca/AboutUs/RegulatoryAffairs/Nov2004/DOCS/Liberty/LibertyIR-51_60.pdf 2005.

[115] Z. Constantinescu, P. Petrovic, A. Pedersen, D. Federici, and J. Campos, "QADPZ (Quite Advanced Distributed Parallel Zystem)," in *http://qadpz.sourceforge.net*, 2003.

[116] A. Sauvé, C. Matthews-Dickson, and O. Peterson, "Real–Time Distributed Factory Automation System," Department of Systems and Computer Engineering, Carleton University, A report submitted in partial fulfillment of the requirements of the 94.498 Engineering Project 2003.

[117] Y. Ebata, H. Hayashi, Y. Hasegawa, S. Komatsu, and K. Suzuki, "Development of the Intranet-based SCADA (supervisory control anddata acquisition system) for power system," IEEE Power Engineering Society Winter Meeting, pp. 1656-1661, 2000.

[118] European Information Society Technologies (IST), "COACH (Component Based Open Source Architecture for Distributed Telecom Applications," in *http://coach.objectweb.org*, 2003.

[119] US military, "The Joint Interoperability Test Command," in *http://jitc.fhu.disa.mil/*, 2005.

[120] "CitectSCADA," in *http://www.citect.com/products/citectscada*, 2005.

[121] BWI Co., "ElipseSCADA," in *http://www.bwi.com/proot/2775*, 2004.

[122] N. Toshida, M. Uesugi, Y. Nakata, M. Nomoto, and T. Uchida, "Open Distributed EMS/SCADA System," *Hitachi Review*, vol. 47, no. 5, pp. 208-213, 1998.

[123] 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," International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies (NPIC/HMIT), pp. 629-633, Washington DC, USA, 2000.

[124] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns, and Java*, 2nd Edition ed: Prentice Hall, 2003.

[125] 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," Proc. of Int. Conf. and Exhibition on Electricity Distribution, 2001.

[126] Wikipedia, "Definition of Controllability," in *http://en.wikipedia.org/wiki/Controllability*, 2005.

[127] Object Management Group (OMG), "UML 2.0 Testing Profile Specification," 2003.

[128] V. Garousi, L. Briand, and Y. Labiche, "A Unified Approach for Predictability Analysis of Real-Time Systems using UML-based Control Flow Information," 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.

[129] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*: Addison-Wesley, 1989.

[130] T.-P. Hong, H.-S. Wang, and W.-C. Chen, "Simultaneously Applying Multiple Mutation Operators in Genetic Algorithms," *J. Heuristics*, vol. 6, no. 4, pp. 439-455, 2000.

[131] E. S. H. Hou, N. Ansari, and H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 2, pp. 113-120, 1994.

[132] Wikipedia, "Definition of NP-hard," in *http://en.wikipedia.org/wiki/NP_hard*, 2005.

APPENDIX A- GENETIC ALGORITHMS OVERVIEW

In [1, 129-131], the authors describe GAs as a means of solving complex optimization problems that are often NP-hard[1] [131] 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*. Figure 137 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.

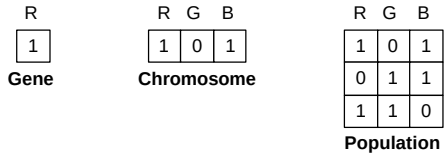


Figure 137-GA chromosome terminology.

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 137 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 138 through *single point* crossover[2].

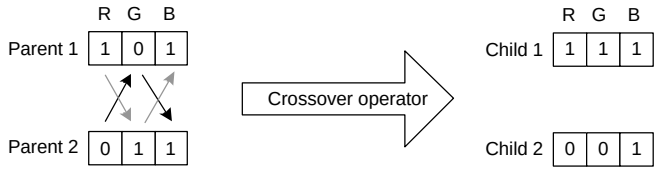


Figure 138-Illustration of crossover operator (*single point* crossover).

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

---

[1] 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 $L = H$. 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* [132].

[2] Single-point crossover is on type of crossover operators. There are other types such as multi-point crossover.

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 139, the red gene is mutated, resulting in a chromosome with RGB values 010.

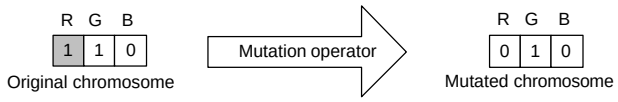


Figure 139-Illustration of mutation operator.

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 [1]. By adopting the GA process concept from [74], we can draw an activity diagram for the process as shown in Figure 140.

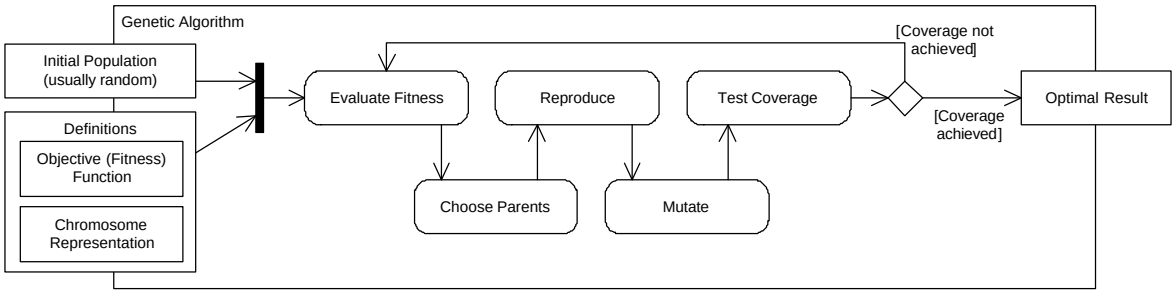


Figure 140-Activity diagram of the most general form of genetic algorithms (concept from [74]).

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.