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

Vahid Garousi, Lionel C. Briand, and Yvan Labiche
{vahid | briand | labiche}@sce.carleton.ca
Software Quality Engineering Laboratory (SQUALL)
http://squall.sce.carleton.ca
Department of Systems and Computer Engineering, Carleton University
1125 Colonel By Drive, Ottawa, ON K1S5B6, Canada

Abstract. This report presents a model-driven, stress test methodology aimed at increasing chances of discovering faults related to network traffic in Distributed Real-Time Systems (DRTS). The technique uses the UML 2.0 model of the distributed system under test, augmented with timing information, and is based on an analysis of the control flow in sequence diagrams. It yields stress test requirements that are made of specific control flow paths along with time values indicating when to trigger them. The technique considers different types of arrival patterns (e.g., periodic) for real-time events (common to DRTSs), and generates test requirements which comply with such timing constraints. Though different variants of our stress testing technique already exist (that stress different aspects of a distributed system), they share a large amount of common concepts and we therefore focus here on one variant that is designed to stress test the system at a time instant when data traffic on a network is maximal. Our technique uses Genetic Algorithms to find test requirements which lead to maximum possible traffic-aware stress in a system under test. Using a real-world DRTS specification, we design and implement a prototype DRTS and describe, for that particular system, how the stress test cases are derived and executed using our methodology. The stress test results indicate that the technique is significantly more effective at detecting network traffic-related faults when compared to test cases based on an operational profile.

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

1 INTRODUCTION

Distributed Real-Time Systems (DRTS) are becoming more important to our everyday life. Examples include command and control systems, aircraft aviation systems, robotics, and nuclear power plan systems [56]. However, the development and testing of such systems is difficult and takes more time than for systems without real-time constraints or distribution [59]. Furthermore, based on an analysis of sources of failures in the United States Public Switched Telephone Network (PSTN) [30], it is reported that in the 1992-1994 time period, although only 6% of the outages were overloads, they led to 44% of the PSTN's service downtime. 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 do not happen frequently, the failure resulting from them can be quite expensive.

Therefore the high-level motivation for our work can be stated as follows: because DRTS are by nature concurrent and are real-time, there is a need for methodologies and tools for testing and debugging DRTS under stress conditions such as heavy user loads and intense network traffic. These systems should be tested under stress before being deployed in the field in order to assess their robustness to distribution-specific problems. In this work, our focus is on network traffic, one of the fundamental factors affecting the behavior of DRTS, though we will see that our methodology can be easily tailored to other aspects.

Distributed nodes of a DRTS regularly need to communicate with each other to perform system functionality. Network communications are not always successful and on time as problems such as congestion, transmission errors, or delays might occur. On the other hand, many real-time and safety-critical systems have hard deadlines for many of their operations, where if the deadlines are not met, serious or even catastrophic consequences will happen. Furthermore, a DRTS might behave well with normal network traffic loads (e.g., in terms of amount of data, number of requests), but abnormal and/or faulty behavior (e.g., violation of real-time constraints) might result from poor and unreliable communication if many network messages or high loads of data are concurrently transmitted over a particular network or towards a particular node. This is the type of problems that our test methodology purports to uncover.

Our overall approach to testing is model-driven [5]. Since 1997, UML has become the de facto standard for modeling object-oriented software for nearly 70 percent of IT industry [46]. The new version of UML, version 2.0 [42] offers an improved modeling language compared to UML 1.x versions. Some of the high level improvements are: enhanced architecture modeling and extensibility, support for component-based development, and model management [46]. As we expect UML to be increasingly used for DRTS, it is therefore important to develop automatable UML model-driven, stress test techniques.

Proposing that UML design models for a DRTS be in the form of Sequence Diagrams (SD) annotated with timing information, and the systems' network topology be given in a specific modeling format, we devise a technique to derive test requirement to stress the DRTS with respect to network traffic in a way that will likely reveal robustness problems. Note that, for a DRTS where several concurrent objects are running on each distributed node and objects communicate frequently with each other, the number of all possible object interaction interleavings on a network is extremely large[1]. Testing all those interleavings is in general not feasible. We thus 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.

The current work is an extended version of the work in [23], where we considered distributed systems in which external or internal events did *not* exhibit arrival patterns (e.g., periods and bounded). The technique in the current work takes into account different types of events arrival patterns that are common in DRTSs. Such patterns impose constraints on the time instant when interactions between distributed objects can take place. We make use of specifically-tailored Genetic Algorithms (a much simpler technique was used in [23]) to automatically generate test requirements which comply with such timing constraints and lead to high traffic-aware stress in a SUT.

The remainder of this article is structured as follows. Related works are discussed in Section 2. An overview of our stress test methodology is described in Section 3. Input system models are described in Section 4. Section 5 discusses how a stress test model is built to support automation. The use of the stress test model to derive test requirements is described in Section 6. Our prototype tool, referred to as *GARUS (GA-based test Requirement tool for real-time distribUted Systems)* and its empirical analysis are presented in Section 7. The results of applying the methodology to a case study system is described in Section 8 which shows the applicability and assesses the effectiveness of the methodology in revealing faults related to network traffic. Finally, Section 9 concludes the article and discusses some of the future research directions.

2 Related Works

No existing work seems to directly address the automated derivation of test requirements from UML models for performance stress testing of DRTS from the perspective of maximizing the chance of exhibiting network traffic faults. In general, there have been relatively few works [3, 7, 23, 62, 63] on systematic generation of stress and load test suites for software systems.

---

[1] A network interaction interleaving is a possible sequence of network interactions among a subset of objects on a subset of nodes.

Authors in [3] 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 is modeled by Markov chains can be considered a limitation of this work. Furthermore, testing based on an operational profile (representing typical use) can hardly be expected to stress a system.

Briand et al. [7] 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 helps identify the worst-case scenario of task executions, this is not always the case because of the assumptions made by schedulability theory regarding aperiodic tasks. The authors develop a methodology that helps identify performance scenarios that can lead to performance failures in a system.

Yang proposes a technique [62] to identify potentially load sensitive code regions and 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.

Zhang et al. [63] describe a procedure, with a similar goal to ours, for automating stress test case generation in multimedia systems. The authors consider a SUT to be a multimedia system consisting of a group of servers and clients connected through a network. 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, specifically 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 coupled with timing constraints. A specific flavor of temporal logic [1] was used to model temporal constraints. The following are some of the limitations of their technique: (1) It cannot be easily generalized to generate test cases to stress test other kinds of resources, such as network traffic, as this would require important changes in the test model; (2) The resource utilization (CPU) of media objects is assumed to be constant over time, although such utilization would likely depend on the requests the server receives for example; (3) Although the objective is similar to ours, i.e., maximizing resource usage at a given time instant, no variation of the technique is proposed or even mentioned to stress test over a specific period of time. A system may only exhibit failures if stress testing is prolonged for a period of time; (4) In practice, the use of Petri Nets and temporal logic can be an impediment to usage.

In this article, we build on a traffic-aware stress testing technique for distributed systems we presented in [23]. An important aspect of real-time systems taken into account in the current work, which was left out in [23], is the arrival patterns for events (e.g., periods) triggering SDs. Such patterns impose constraints on the time instant when interactions between distributed objects can take place, and thus on the derivation of (stress) test requirements. The stress test technique in this work uses Genetic Algorithms (GA) to find test requirements which comply with such timing constraints and lead to high traffic-aware stress in a SUT.

There is an important body of work (e.g., [28, 44, 54, 55, 58]) that uses evolutionary algorithms (such as GAs) for test case generation, which is commonly referred to as *Evolutionary Testing (ET)*. ET uses meta-heuristic

search-based techniques[1] to find good quality test data. Test data quality is often defined by a test adequacy criterion (typically defined in terms of the program's predicates) built into a fitness function. This function determines the fitness of candidate test data, which in turn, drives the search implemented by an optimization technique. Reported techniques in [28, 44, 55, 58] aim at generating adequate test data for branch coverage and other white-box testing criteria for a program under test [6]. Reported fitness functions essentially measure how close a candidate test input is to executing the desired (target) Control Flow Path (CFP). Generating test data using ET has been shown to be successful, but its effectiveness is significantly reduced in the presence of programming constructs which make the definition of an effective fitness function problematic, e.g., unstructured control flow (in which loops have many entry and exit points) affects the ability to determine how alike are the traversed and target paths [6]. ET techniques are also used for black-box testing. For instance, Tracy et al. [54] use a genetic algorithm to derive test cases from pre and post-conditions. They transform those predicates into disjunctive normal form and make each conjunct contribute to the final fitness value. The fitness function rewards values that satisfy the pre-condition of a subprogram and result in a violation of its post-condition. Since any particular test input either satisfies this criterion or not, the authors also introduce the notions of better and *worse* values to represent values that *nearly* satisfy the criterion or are *long away* from satisfying the criterion, respectively (this is similar to the aforementioned measure of how close an input is to executing a specific CFP).

Though the focus of ET techniques has not been so far on load, performance or stress testing, the methodology reported in this article is an evolutionary stress testing technique which searches among model-based CFPs in a SUT to maximize a fitness function, but the CFPs are identified from models rather than code. Another difference is that our fitness function (Section 6.5.5) is based on the amount of traffic a CFP entails instead of how close a candidate test input drives execution to traversing the desired (target) CFP. Furthermore, compared to ET techniques, our methodology takes into account a different set of constraints (Section 6.5.2), which are specific to DRTSs. Two of such constraints we consider are: (1) sequential constraints between SDs which imply that executing an arbitrary sequence of SDs in a SUT might not be always valid or possible, e.g., the *withdraw* SD of a banking system can not be executed before *login*; (2) SDs arrival patterns which impose constraints on the time instant when interactions between distributed objects can take place, e.g., a periodic event may not be allowed to be triggered in arbitrary time instances which do not belong to its periodic domain. Yet another difference is that our work derives stress test requirements given a set of stress test objectives (e.g., a network to be stress tested in a time instance), while most existing ET techniques focus on deriving a test case (input data) given a test requirement (e.g., a CFP).

3 AN OVERVIEW OF OUR METHODOLOGY

An overview of our model-based stress test methodology is presented using an activity diagram in Figure 1. A UML model of a SUT, following specific but realistic requirements, is used as 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.

Note the distinction made in Figure 1 using a color coding scheme (refer to the legend) between the contributions of the work in [23] and the current article. The stress testing technique in [23] is referred to as *Time-Shifting Stress Test Technique (TSSTT)*, which uses only four elements of the TM. The technique in the current article is referred to as *Genetic Algorithm-based Stress Test Technique (GASTT)*, and uses all five elements of the TM. The Arrival Pattern Model (Section 5.5) incorporates the arrival pattern constraints of events in a SUT and enables GASTT (Section 6) to derive test requirement complying with such constraints. Test activities with a crossed gray background (deriving test cases from test requirements and test execution

---

[1] Typically genetic algorithms and simulated annealing have been used, but evolutionary testing requires only that the technique used is characterized by some fitness (or cost) function, for which the search seeks to find an optimal or near-optimal solution [6].

by the tester) are not addressed in this report but we will discuss those aspects in the context of our case study (Section 8).

At a high level, the goal of our stress test technique is to choose the maximum number of SDs (to create an amount of traffic) which can realistically be run concurrently, according to the business logic of a SUT, and schedule them such that their maximum traffic messages run at the same time. The detailed steps of Figure 1 are described in the next sections:

- Specification of the input system models (Section 4)
- Specification and construction of the test model (Section 5)
- Derivation of stress test requirements using Genetic Algorithms (Section 6)

Stress test parameters (objectives) in Figure 1 specify the variant of the stress test technique to be applied and the values for the parameters of that stress test strategy. We have 16 such variants in our methodology, which share a common framework and many common concepts [22]. Each strategy is specified and named according to four attributes: (1) a stress test *location* (a network or a node); (2) a stress test *direction* (applies only to a node test location-*In* for towards, *Out* for from, or *Bi* for bidirectional traffic); (3) a test *duration* (a time instant, *Ins*, or a time interval, *Int*); and (4) a stress test *type* (*DT* for maximizing amount of data traffic and *MT* for maximizing number of messages). For example, the stress test strategy we focus on in this article is named *StressNetInsDT* which is designed to stress test the system at a time instant (attribute duration-*Ins*) when data traffic (attribute type-*DT*) on a network (attribute location-*Net*) is maximal. For this stress test strategy, the tester should provide the name of the network under stress test (the network for which our methodology will derive stress test requirements such that the instant data traffic is maximized).

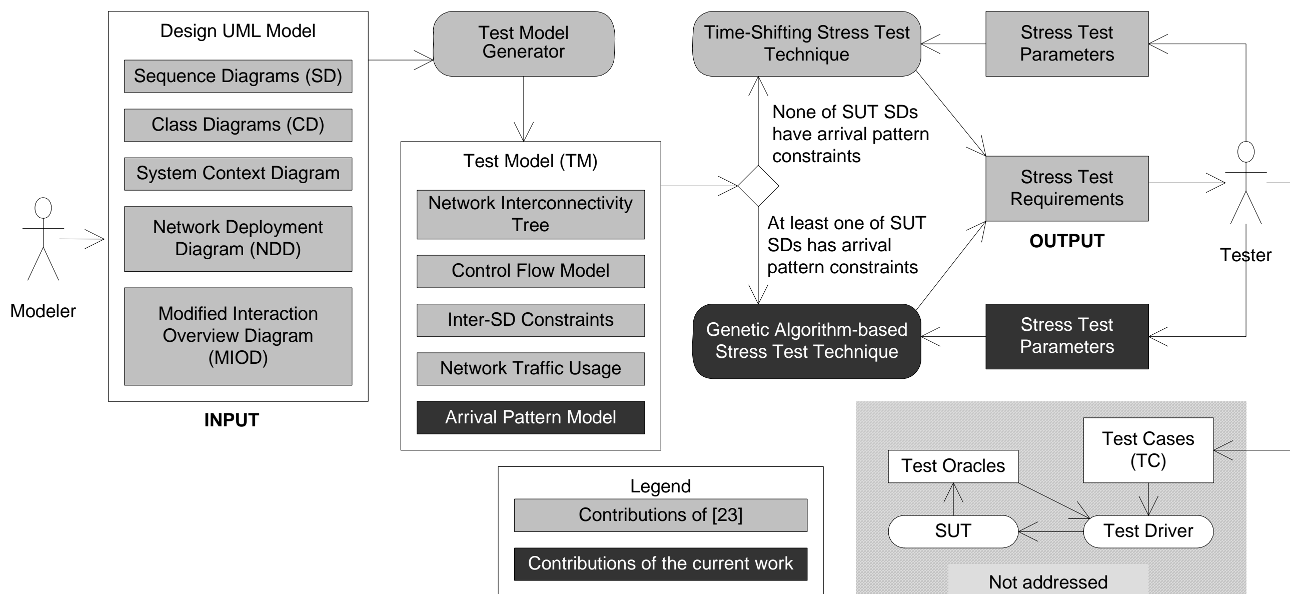


Figure 1. An overview of our model-based stress test methodology.

4 Input System Models

The assumed input system models for the stress test methodologies in this article and [23] are almost the same, except that the current work requires the arrival pattern of SDs to be modeled using stereotypes from the UML Profile for Schedulability, Performance, and Time (UML-SPT) [43] in SDs. Thus, we present in Section 4.1 only an overview of the assumed input system models. Interested reader can refer to [23] and [22] for further details. Section 4.2 discusses how arrival pattern information can be modeled in SDs.

4.1 An Overview of the Input System Models

The input model consists of a number of UML diagrams. Some of them are standard in mainstream development methodologies (class diagram, sequence diagrams, and system context diagram [24]). The other two, further described in the next subsections, are needed to describe the distributed architecture of the SUT (Network Deployment Diagram) and sequential constraints among SDs, i.e., their respective use cases, (Modified Interaction Overview Diagram).

4.1.1 Network Deployment

The structure of the distributed architecture of a SUT as we need it to be described is formalized in Figure 2 as a metamodel. Such network information is paramount as one of our objectives is to stress, not only nodes in a network, but also (sub-)networks. An example of a distributed architecture is depicted in Figure 3-(a) which shows networks in a hierarchical structure (each network can have many subnets and only one supernet), nodes belonging to networks, and objects distributed on nodes, e.g., $node_1$ hosts three objects ($o_{1,1}$, $o_{1,2}$, and $o_{1,3}$).

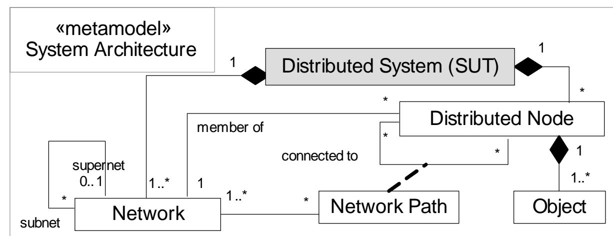


Figure 2. Metamodel for distributed architectures.

Each node can be connected to other nodes through several network paths. A path is defined as a sequence of networks. For example, $node_1$ is connected to $node_3$ through the network path <$Network_1$, *SystemNetwork*, $Network_2$> in Figure 3-(a). In the current work, we consider that there is only one path between two nodes, rather than several paths. This simplifying assumption is not too simplistic though since many (proprietary) SUT networks (e.g., a distributed controller system of a factory) are not as complex (e.g., in terms of topology) as the World Wide Web. The main reason is that we want to evaluate whether our approach is feasible under this assumption and leads to interesting results before considering multiple paths. Considering multiple paths would increase the complexity of our network traffic usage model (Section 5.4) since this would require a detailed analysis of the routing policy used in the network of the SUT.

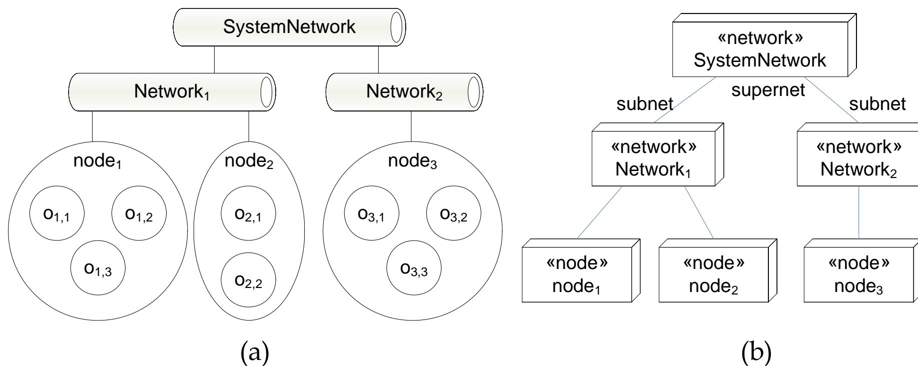


Figure 3. (a): An example distributed architecture. (b): An example Network Deployment Diagram (NDD).

Modeling a hierarchical set of networks and their inter-connectivity is not directly addressed in the UML 2.0 specification [42]. We therefore extend UML 2.0 deployment diagrams by adding two stereotypes to the node notation: *«network»* and *«node»*. We thus identify the type of an entity as a network or a node.

Furthermore, association roles stereotyped with *supernet* and *subnet* are used to model the containment relationships between super and sub-networks. As an example, the architecture in Figure 3-(a) is modeled by the Network Deployment Diagram (NDD) in Figure 3-(b).

4.1.2 Modified Interaction Overview

The name Modified Interaction Overview Diagram (MIOD) comes from the UML 2.0's Interaction Overview Diagram (IOD) [42]. To model which actor can trigger a particular SD, we modify IODs to include activity partitions: one partition per actor. A MIOD is used to model sequential and conditional constraints between SDs (inter-SD constraints): activities (i.e., nodes in the diagram) are SDs and edges depict those sequential constraints. Standard activity diagram decision nodes are used to model conditional constraints between SDs. There exist alternative representations (e.g., [10, 14, 40]). However, as we discuss in [23], MIODs suit best our needs for modeling sequential and conditional constraints among SDs in the context of UML-based development.

Taking sequential and conditional constraints into account is important while defining stress tests since executing an arbitrary sequence of SDs in a SUT might not be always valid or possible. The 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. An example MIOD is shown Figure 4, where SDs *SD1* and *SD2* are triggered by *actor1* and *SD3* by *actor2*. The MIOD specifies the sequential and conditional constraints among SDs, e.g., *SD1* and *SD2* should be executed and condition *c2* should hold before *SD3* can be executed.

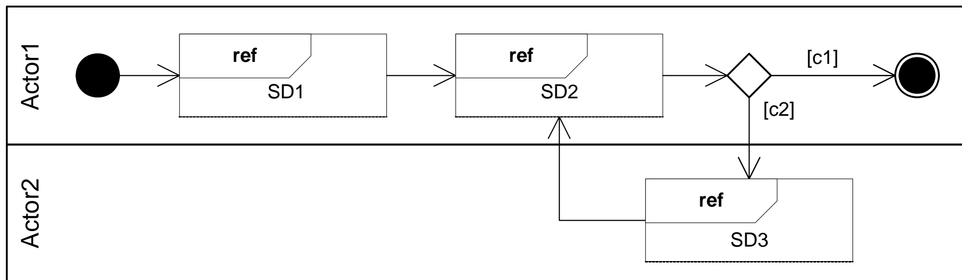


Figure 4-An example MIOD.

4.2 SDs with Arrival Pattern Information

Our technique assumes that the arrival pattern[1] information of SDs is given using the *RTArrivalPattern* tagged-value which is a modeling construct in the *TimeModel* package of the UML Profile for Schedulability, Performance, and Time (UML-SPT) [43]. As an example, the UML 2.0 SD in Figure 5 shows the temperature data update process for a simplified chemical reactor system, where a sensor controller is getting the two temperature values from two sensors (deployed on nodes n_{s1} and n_{s2}), and then sends the data to be updated in the sensors database (on n_{cs}). The timing information of messages has been modeled using the UML-SPT. For example, «*RTstimulus*» denotes that the first message is a RT stimulus with an execution duration of less than 10 milliseconds (ms) and an arrival pattern specified by the *RTArrivalPattern* tagged-value: a periodic event with a period value of 100 ms. *RTstart* and *RTend* tagged-values specify the start and end time instances of a message. The time basis (t=0) in the UML-SPT is assumed to be the execution start time of a SD.

[1] Arrival-pattern constraints relate to timing of SDs. The time instant when a SD can start running might be constrained in a SUT. Each SD might be allowed to execute only in some particular time instants.

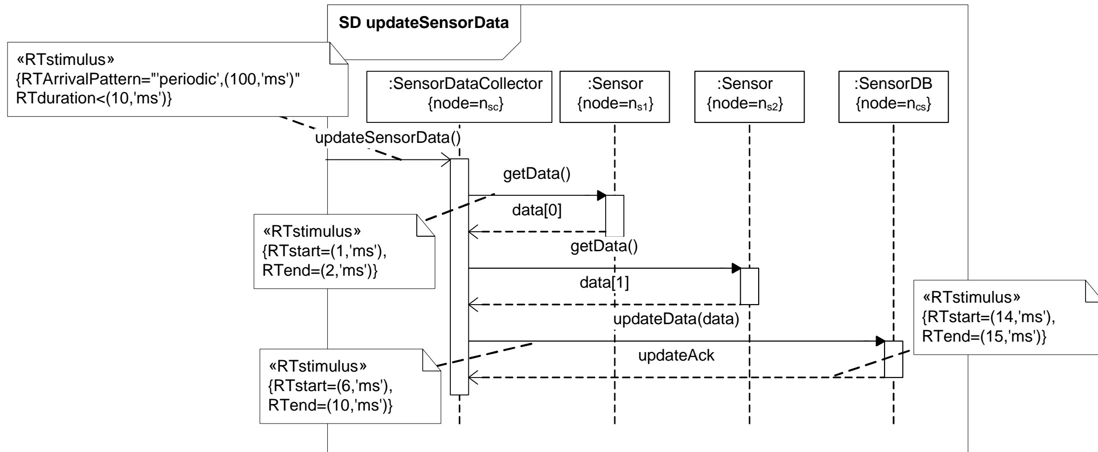


Figure 5-Example of time modeling using the UML-SPT profile.

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 5 and prompt corrective actions should be carried out if the temperature is higher than a pre-specified threshold.

5 BUILDING THE TEST MODELS

We build a Test Model (TM) which includes the following elements: (1) Control flow model, (2) Network interconnectivity tree, (3) Network traffic usage patterns, (4) Inter-SD constraints model, and (5) Arrival patterns model. These models are needed to facilitate the automated derivation of test requirements. The activity diagram in Figure 6 illustrates the relationships among these models and input UML models, as well as five distinct activities responsible for the construction of test models, e.g., the control flow analysis activity builds the control flow model from an analysis of sequence and class diagrams.

The following subsections describe how the TM is built. Four of the five elements UML models composing the TM are discussed in [23] and, due to space constraints, we present in Sections 5.1-5.3 only a brief overview of those models and their construction. We devote more space to the description of the Network Traffic Usage Model (Section 5.4) and the Arrival Patterns Model (Section 5.5).

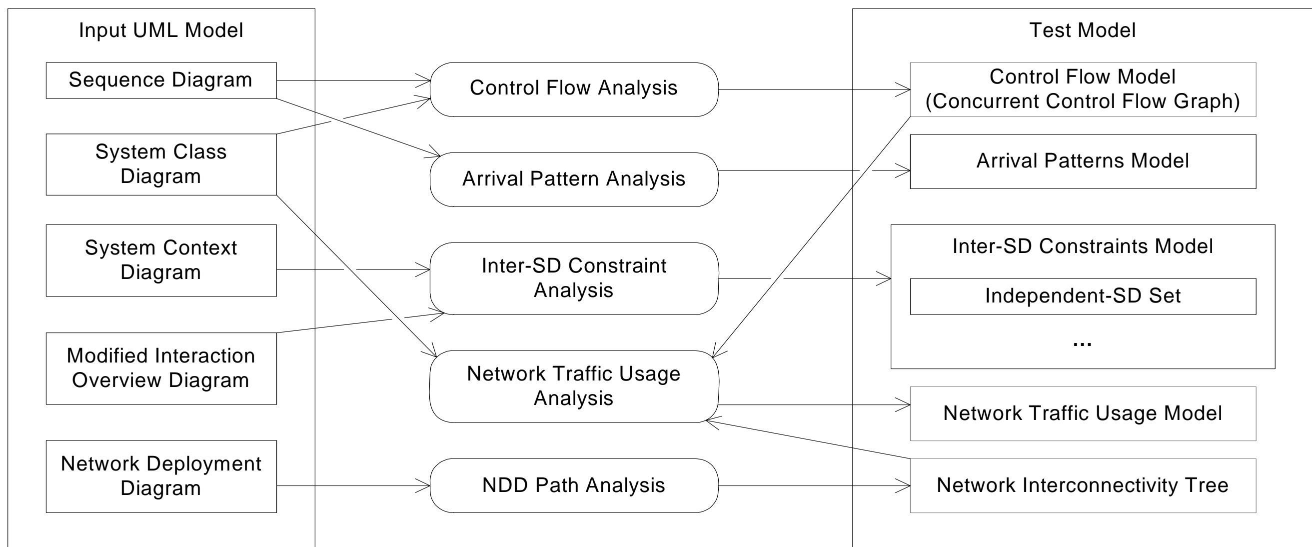


Figure 6. An overview of how test models are built from input UML models.

5.1 Control Flow Model

In UML 2.0 [42], 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 [21], asynchronous messages and parallel combined fragments entail concurrency inside SDs. Additionally, in a SD of a distributed system, 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) thus entailing network traffic. Since network traffic usage varies with CFPs (e.g., varying number of distributed messages transmitting data of varying sizes), a comprehensive model-based stress testing should take into account the differences among CFPs in a SD.

In [23], we used the Model-based Control Flow Analysis (MBCFA) technique presented in [21], which was formalized using meta-modeling and consistency-rules in the Object Constraint Language (OCL) [41]. We also introduced *Concurrent Control Flow Graphs* (CCFG) as a means to analyze the concurrent control flow of SDs, due for instance to asynchronous messages, and the associated notion of *Concurrent Control Flow Path* (CCFP), i.e., a path in a CCFG.

5.2 Inter-Sequence Diagram Constraints Model

Recall from Section 4.1.2 that taking sequential and conditional constraints among SDs in a SUT into account is important while defining stress tests since executing an arbitrary sequence of SDs in a SUT might not be always valid or possible. A MIOD is used to model sequential and conditional constraints (inter-SD constraints) between SDs. The goal of our stress test technique is to choose the maximum number of SDs (to create maximum possible traffic) which can realistically be run concurrently, according to the MIOD, and schedule them such that their maximum traffic messages run at the same time.

To comply with inter-SD constraints while considering the maximum number of SDs, we introduce the concept of *Independent SD Set* (ISDS). Two SDs are *independent* if there is no path (inter-SD constraints) between them in the MIOD: e.g., Figure 7-(a) shows the MIOD of a power distribution controller system we use as a case study in Section 8, in which SDs *A* and *B* are independent. (More details about this MIOD, such as actual SD names and the semantics of stereotype «HRT» are provided in Section 8.) An ISDS is a largest (maximal) set of SDs, in which any two SDs are independent, thus enabling all the SDs in the set to run concurrently. A MIOD can lead to several ISDSs and, as discussed in Section 6, the ISDS with maximum traffic (among all the ISDSs for a given MIOD) will be chosen to generate stress test requirements.

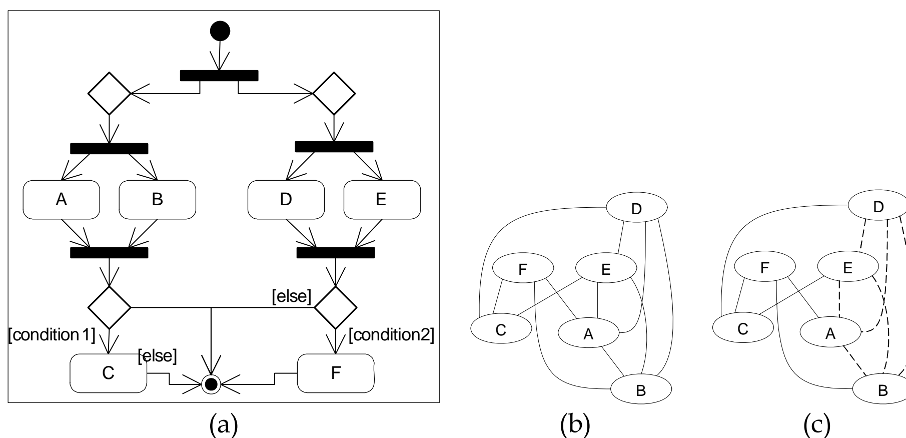


Figure 7. (a): The MIOD of our case study system. (b) and (c): Deriving Independent SD Sets of the MIOD.

To derive the set of ISDSs of a MIOD, we use a graph-based approach in which we first build a graph, e.g., Figure 7-(b), where nodes are SDs and there is an edge between two nodes if and only if the two

corresponding SDs are independent. Finding the ISDSs can then be formulated as a graph problem. More specifically, every maximal-complete subgraph in this graph is an ISDS. Standard graph algorithms can then be used to find those maximal-complete subgraphs. For the MIOD in Figure 7, four ISDSs are identified (e.g., $ISDS_1$ is illustrated in Figure 7-(b)):

$$ISDS_1 = \{ A, B, D, E \} \quad ISDS_2 = \{ A, B, F \}$$
$$ISDS_3 = \{ C, D, E \} \quad ISDS_4 = \{ C, F \}$$

5.3 Network Interconnectivity Tree

A Network Interconnectivity Tree (NIT) is built from a NDD (Section 4.1.1). The root of the tree is always the entire system network while system networks and nodes are its children. The motivation for NITs is to easily identify the subset of nodes and networks that are relevant for deriving stress test cases and the network paths between any two given nodes: e.g., when stress testing a specific network in a DRTS, we must identify the messages, exchanged by nodes, that are transmitted through that network.

To identify the network path between any two given nodes, we define the network path function $getNetworkPath(n_s,n_r)$, where n_s and n_r are two nodes, which returns the network path that messages sent from n_s to n_r would follow. (An algorithm for this function can be found in [22].) For example, the derivation of the network path between $node_1$ (the sender) and $node_3$ (the receiver) in Figure 3-(a) is formally represented as:

$$getNetworkPath(node_1, node_3) = \langle Network_1, SystemNetwork, Network_2 \rangle$$

5.4 Network Traffic Usage Model

A network traffic usage model describes the extent to which messages, and thus CCFPs entail traffic on a network. An estimate of network traffic usage for each message and CCFP is required in order to derive appropriate stress test requirements to stress test a SUT with respect to network traffic. We present in this section a resource usage analysis (RUA) technique to measure traffic usage for CCFPs.

In order to analyze the traffic usage of a CCFP, we need to analyze the traffic usage entailed by its messages. Only *distributed* messages (those sent between two different nodes) in SDs are of interest here since they are the only ones entailing network traffic. A *Distributed CCFP (DCCFP)* is a CCFP where only distributed messages are modeled. To measure the traffic entailed by a distributed message, we compute the data sizes of the parameters of a call message or the return values of a reply message. For a distributed signal message, we consider the size of the signal object (sum of the attributes' size) as the size of the signal message[1]. We define the data size of an object to be the summation of sizes (in bytes) of the attributes in its class. Admittedly, other measures (perhaps more accurate) of network traffic can be considered. We however consider our measurement as a reasonable and practical surrogate for network traffic.

In order to precisely define how we perform traffic usage analysis of CCFPs, we formally define SD messages. Similar to the tabular representation of messages, proposed by UML 2.0 [42], each message annotated with timing information (using the UML-SPT profile [43]) 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 of 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.

[1] In UML 2.0, in the case of a message of type signal, the arguments of the message must correspond to the attributes of the signal class. The data carried by a signal message is represented as attributes of the signal instance.

- *receiver* denotes the receiver of the message and is itself a tuple of the same form as *sender*.
- *methodOrSignalName* is the name of the method on the message or the signal class name in case of a signal on the message.
- *parameterList* is the list of parameters for call messages. *parameterList* is a sequence of the form <$(p_1, C_1, in/out), ..., (p_n, C_n, in/out)$>, where p_i is the *i*-th parameter of class type C_i and *in/out* defines the kind of the parameter. For example if the call message is $m(o_1{:}C_1, o_2{:}C_2)$, then the ordered parameters set will be <$(o_1, C_1, in), (o_2, C_2, in)$>. If the method call has no parameter, this set is 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 for a reply message. We show *returnList* in the form of a sequence <$(var_1{=}val_1,C_1), \ldots, (var_n{=}val_n,C_n)$>, where val_i is the return value 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 of each message in the UML metamodel can be used to distinguish signal and call messages, the metamodel does not provide a

To formalize our network traffic usage model, we define a *Network Traffic Usage (NTU)* function (Equation 1), which estimates the amount of traffic entailed by a distributed message. A dash (-) symbol indicates that a field can take any arbitrary value. NTU is a function from the set of messages to real values (data traffic).

The data traffic (DT) value depends on the type of the message. For a signal message (function *SignalDT* is used), DT is equal to the data sizes of all the attributes of the signal class referred by the message. For a call message (function *CallDT* is used), DT is the sum of data sizes of all the attributes of each parameter. For a reply message (function *ReplyDT* is used), DT is the sum of data sizes of all attributes of each member of the return list. Data size of the data type of an attribute is extracted from the specification of the target programming language as specified by the user.

$$NTU : Message \rightarrow Real$$

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

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

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

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

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

Equation 1. Network Traffic Usage (NTU) function.

As an example, suppose we want to measure the traffic usage of a call message with two parameters of type A and one of class type B, respectively, where classes *A* and *B* are defined in the class diagram of Figure 8-(a). Using these class specifications, we can estimate the size of the message to be 5.8KB, as illustrated in Figure 8-(b), assuming the target programming language is Java (the size of a *char* and a *long* variable are two and eight bytes, respectively).

A
-attribute1 : long[100] -attribute2 : long[500]

B
-attribute1 : long[100] -attribute2 : char[100]

(a)

NTU(msg) = CallDT(msg)
= dataSize(A) + dataSize(B)
= (8×(100+500)) + (8×100+2×100)
= 5.8KB (kilobytes)

(b)

Figure 8. (a): Two classes with data fields. (b): An example of computation of NTU.

Using NTU, let us now define *Network Traffic Usage Pattern (NTUP)* as a function from the set of DCCFPs, networks, and time domain to real values (usage pattern values). The usage pattern of a DCCFP ρ on a network *net* at a particular time instant *t* is the sum of NTU values of the subset of the DCCFPs' messages whose start/end time interval includes *t* and that go through *net* (using *getNetworkPath()* defined in Section 5.3). *Dur()* denotes the time duration of a message and since a message can span over several time units, our definition for the data traffic value of a message at a given time unit is its total data size divided by its duration, which yields the average message traffic per time unit.

$$NTUP : DCCFP \times Network \times Time \rightarrow Real$$

$$NTUP(\rho, net, t) = \begin{cases} \sum_{msg \in MSG} NTU(msg_i) / Dur(msg_i) & ;\text{where } MSG = \\ & \left\{ \begin{array}{l} msg_i \mid msg_i \in \rho \wedge \\ msg_i.start \le t \le msg_i.end \wedge \\ net \in getNetworkPath(msg_i.sender.node, msg_i.receiver.node) \end{array} \right\} \\ 0 & ;\text{otherwise} \end{cases}$$

Equation 2. Network Traffic Usage Pattern (NTUP) function.

5.5 Arrival Pattern Model

An *Arrival Pattern Model (APM)* is built based on SDs' Arrival Pattern (AP) information. We first describe in Section 5.5.1 why we need an APM by explaining the impacts of arrival patterns on the stress testing. Specifically, an APM will help our stress test requirement derivation process (Section 6) to derive valid test requirements, i.e., test requirements which comply with SD's APs. Types of arrival patterns we consider in this work are discussed in Section 5.5.2. The analysis of arrival patterns to derive an APM is described in Section 5.5.3. Section 5.5.4 presents the concept of *Accepted Time Set*, our APM, which is used by our stress test technique.

5.5.1 Impact of Arrival Patterns

We discuss in this section the impacts of SD arrival patterns on the test requirements derivation process and thus motivate the need for an Arrival Pattern Model (APM). Arrival Patterns (Section 4.2), modeled in UML using the UML-SPT profile, specify constraints on the start times of messages in SDs, and thus on the start times of SDs and thus as well as on the start times of DCCFPs. Arrival Patterns therefore impact the test requirements generation process by limiting the search scope from unlimited time instants to limited intervals for the start times of DCCFPs.

The impacts can be better visualized by the example of Figure 9. Let us first consider a simple search heuristic (used by our earlier work in [23]), to be used when there is no arrival pattern: Figure 9-(a). The heuristic searches among all the ISDSs and finds the one with maximum instant stress. Then the SDs of the selected ISDS are scheduled, i.e., their start time is determined, to yield the maximum stress. The scheduling is done so as the maximum stress message of different SDs start concurrently. In Figure 9-(a), showing the selected ISDS with three SDs SD_1, SD_2, and SD_3, the heuristic determined that the three SDs can start at the same time to yield maximum stress.

On the other hand, if the same SDs have APs, time intervals, referred to as *AP regions*, are specified during which SDs can start executing: for instance, Figure 9-(b). As it can be seen, there are three AP regions for SD_1, one AP region for SD_2, and three AP regions for SD_3. Due to such time constraints, SDs can not be scheduled freely in any arbitrary time instants. The heuristics to find maximum possible stress while respecting APs, in this case, will be to search among the AP regions of every SD and find a time instant when the summation of entailed traffic values by DCCFPs from all the SDs is maximized. One of such possible schedules (among an infinite number of them) is shown in Figure 9-(b).

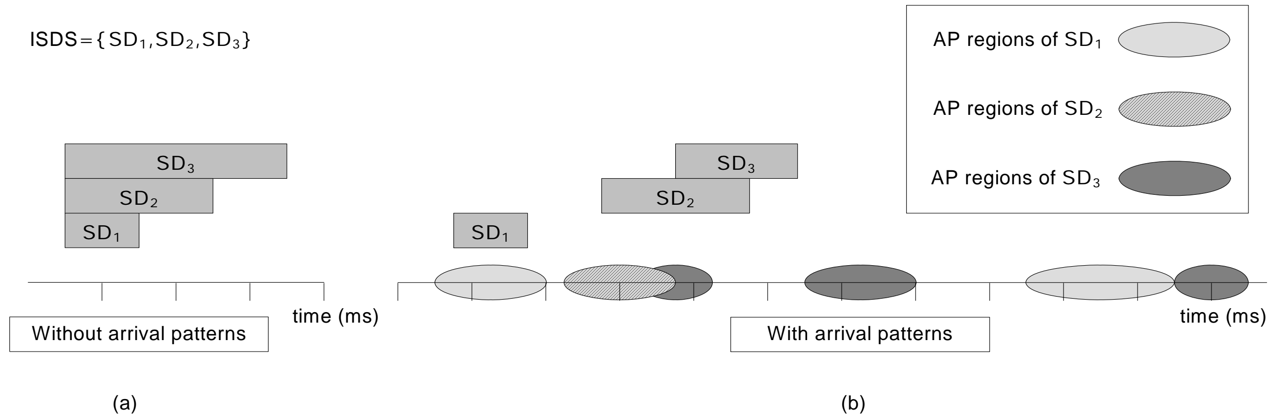


Figure 9-Impact of arrival patterns on the derivation of test requirements

5.5.2 Types of Arrival Patterns

We assume that SD APs are modeled using the UML-SPT profile's *RTarrivalPattern* tagged-value [43] (e.g., Figure 5). We provide next an overview on the five types of APs in the UML-SPT profile:

- Bounded: An AP where the iter-arrival time of two consecutive arrivals is bounded by minimum and a maximum arrival times.
- Bursty: In this AP, a maximum number of events can occur during a specific interval.
- Irregular: An ordered list of time values represents successive arrival times.
- Periodic: Arrival times comply with a period and a deviation value.
- Unbounded: An AP specified by a *Probability Distribution Function*. The types of supported distributions are: bernoulli, binomial, exponential, gamma, geometric, histogram, normal (Gaussian), poisson, and uniform.

5.5.3 Analysis of Arrival Patterns

In order to study APs and devise a stress test strategy to account for them when generating stress test requirements, the timing characteristics of APs should be analyzed. Furthermore, given an arrival time, we should be able to determine if it satisfies an AP, i.e., whether the arrival time is legal given the AP.

The pseudo-code of function *IsAPCSatisfied()* shown in Figure 10 determines if a DCCFP arrival time satisfies an AP. The AP can be any of the following: {*'bounded', 'bursty', 'irregular', 'periodic', 'unbounded'*}. The pseudo-code is described in detail next.

If the *AP* is *bounded*, *IsAPCSatisfied()* returns true if the arrival time is inside the time intervals specified by the bounded pattern. Such a pattern is identified by a *minimal* and a *maximal interval time* (*MinIAT*, *MaxIAT*). We assume that *MinIAT* and *MaxIAT* of a bounded AP can not be equal. If the two values are equal, the arrival pattern is equivalent to a periodic one. For example, a bounded AP where *MinIAT=MaxIAT=3ms*, is indeed a periodic arrival pattern with *period=3ms*. Consider a bounded AP with *MinIAT=4ms* and *MaxIAT=5ms*. The gray eclipses in the timing diagram in Figure 11 depict the *Accepted Time Intervals (ATI)* of the AP, i.e., the time intervals where an AP is satisfied.

Function IsAPCSatisfied(*arrivalTime*, *AP*)
AP∈{*'bounded', 'bursty', 'irregular', 'periodic', 'unbounded'*}
1 **Switch** *AP* {

```
2    'bounded':
3         If arrivalTime is in one of the intervals of the bounded pattern, then Return True
4         Else Return False
5    'bursty': Return True
6    'irregular':
7         If arrivalTime is one of the time values in the AP list, then Return True
8         Else Return False
9    'periodic':
10        If there exists an arbitrary integer k such that arrivalTime∈[kp-d… kp+d], where p
          and d are the period and the derivation values of the AP: then Return True
11        Else Return False
12 }
```

Figure 10- Pseudo-code to check if the arrival pattern *AP* is satisfied by an arrival time.

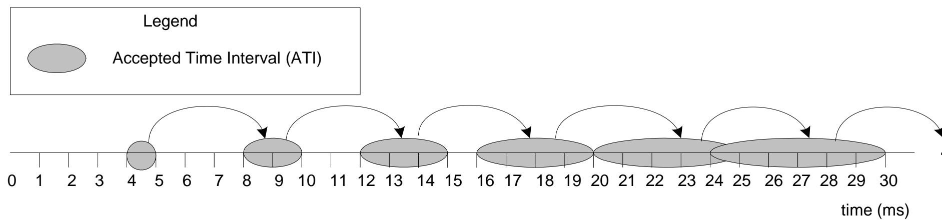


Figure 11-Accepted Time Intervals (ATI) of a *bounded* arrival pattern (*'bounded', (4, ms), (5, ms)*), i.e. *MinIAT=4ms, MaxIAT=5ms.*

Note that the ATIs of a bounded AP denote all *possible* arrival times, regardless of actual arrival times in a specific scenario. The curved arrows in Figure 11 denote how an ATI is derived from the previous one. For the AP discussed above, assuming that the AP 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, a value between 4 and 5 ms will cause the next arrival time to be in the range [8…10ms]. Therefore, the second ATI is [8…10ms]. The next ATIs are [12…15ms], [16…20ms], [20…25ms], [24…30ms] and so on. Since a busty AP only constrains the number of arrivals in a specific time interval, any 'single' arrival at any arbitrary time instance thus satisfies this AP. Similar analysis for other APs (explaining the rest of the pseudo-code in Figure 10) can be found in [22].

5.5.4 Accepted Time Sets

To better formulate our GASTT technique (Section 6), we define the concept of *Accepted Time Set (ATS)* for each SD as the set of time instances or time intervals when a SD is allowed to be triggered, according to its AP. An ATS can be derived from the AP of the corresponding SD. The ATS metamodel in Figure 12-(a) formalizes the fundamental concepts.

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, whereas all the other arrival patterns deal with time intervals. The mutual exclusion between ATIs and ATPs is shown by two OCL invariants (*hasATInoATP* and *hasATPnoATI*) in Figure 12-(a). Each ATI has a start time and an end time of type *RTtimeValue* (from the UML-SPT), denoting the start and end times of an interval. ATP is of type *RTtimeValue* too. The end time of an ATI can be null, which denotes an ATI which has no upper bound (this is further justified below).

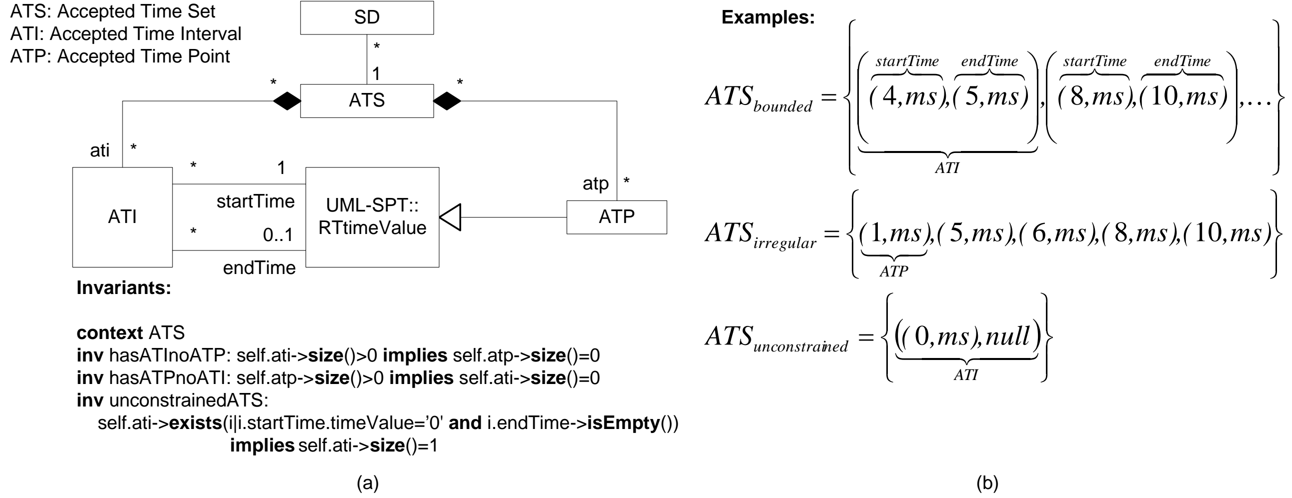


Figure 12-(a): Accepted Time Set (ATS) metamodel. (b): Three instances of the metamodel.

Three ATS examples are illustrated in Figure 12-(b), which comply with the metamodel in Figure 12-(a). $ATS_{bounded}$ and $ATS_{irregular}$ are the ATSs corresponding to a bounded and an irregular arrival pattern. $ATS_{unconstrained}$ is an ATS for SDs which do not have any arrival pattern, i.e., can be triggered any time.

Our convention to represent an unconstrained ATS is to leave the end time of its only interval as *null*: it is unconstrained so no upper bound can be defined. Such an ATS has only one ATI from time 0 to ∞. This constraint has been formalized by the third OCL invariant (*unconstrainedATS*) in Figure 12-(a). Note that one could need to consider other kinds of constraints such as the following, that we refer to as *partly-constrained* ATS: $ATS_{partly\text{-}constrained} = \{((0,ms),(3,ms)),((5,ms),null)\}$; where the corresponding SD can be triggered in all times, except interval]3ms…5ms[. In such an ATS, there is at least one ATI where the end time is null. However, modeling arrival patterns which lead to partly-constrained ATSs is not currently possible using the UML-SPT. Since we assumed the UML-SPT as the modeling language to model arrival patterns in this work, we assume that there will not be any SD with a partly-constrained ATS.

Our GA-based algorithm in Section 6 will require computing the intersection of the ATSs of two SD. This will enable our algorithm to generate GA individuals (test requirements) with high stress values. Therefore, we define an intersection operator (∩) for any pair of ATSs: Equation 3. For brevity, *startTime* and *endTime* have been replaced by *s* and *e*.

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

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

Equation 3-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 12-(a), $(1,ms) \in ATS_{irregular}$.

The output of the formula is the union of three sets: (a) common ATPs (in the case the two ATSs contain only ATPs), (b) common ATPs in ATIs (in case one ATS contains only ATIs and the other contains only ATPs), and (c) common ATIs (in case the two ATSs contain only ATIs). In case (a), the result is the set of ATPs (atp∈ATP means that atp is an ATP) that belong to both ATSs ats_1 and ats_2. 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 12-(b), $(1,ms) \in ATS_{irregular}$. In case (b), the result is the set of ATPs in one ATS (e.g., ats_1) for which there exists an ATI in the other ATS (e.g., ats_2), such that the (ATP) time point is inside the (ATI) time interval. The formula uses a (in-range) operator $\angle$ to compare a time point (i.e., an ATP) and a time interval (i.e., an ATI): $\forall\ atp \in ATP, ati \in ATI : ati.startTime \le atp \le ati.endTime \Leftrightarrow atp \angle ati$. In case (c), the result is the set of overlapping time intervals. The rationale for finding overlapping (common) intervals of two ATSs is illustrated in Figure 13.

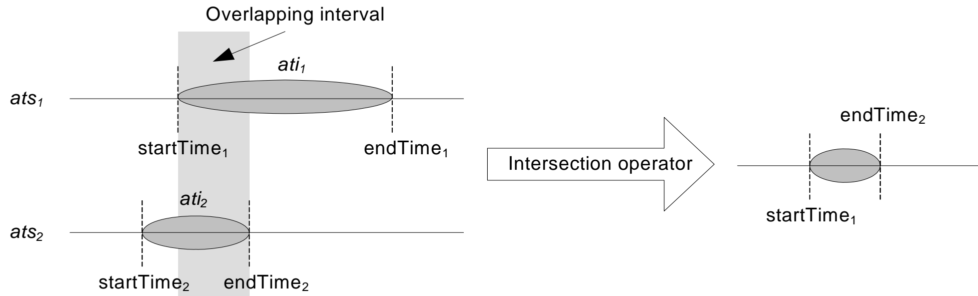


Figure 13- Illustrating the overlap of two ATSs' intervals.

Note that the union of the above three sets is allowed in the current context from the set theory point of view, since as the metamodel in Figure 12-(a) shows, ATS is a *hybrid* set of two element types: ATI and ATP. Therefore, a set of type ATIs together with another set of type ATP can be the operands of a union operator, yielding an ATS. Two examples, showing how intersections of two ATSs can be calculated using Equation 3, are illustrated in Figure 14: between an ATS made of ATIs and an ATS made of ATPs (upper part of the figure); between two ATSs made of ATIs (lower part of the diagram).

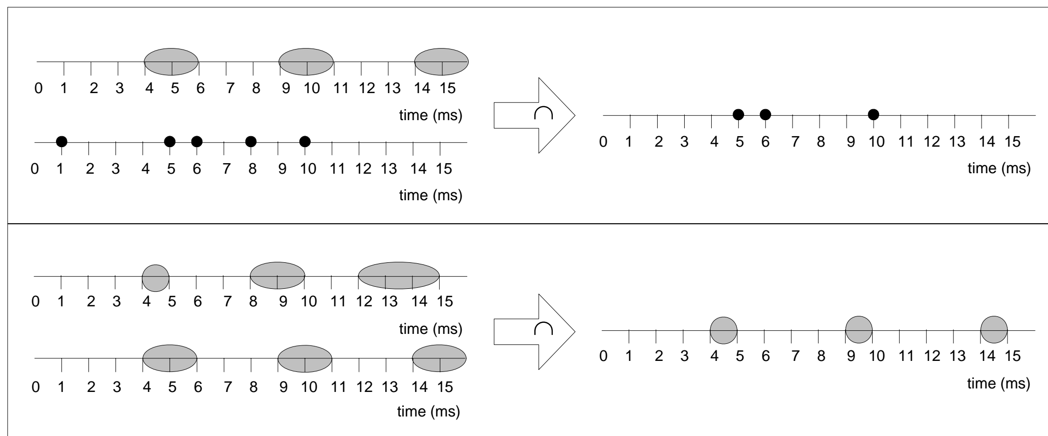


Figure 14-Example intersections of two ATSs.

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

6 Using Genetic Algorithms To Derive Stress Test Requirements

This section describes how stress test requirements are derived from our test model. The heuristics of our stress test technique are described in Section 6.1. Section 0 formulates the stress test generation problem as an optimization problem. The output stress test requirements format is presented in Section 0. Our choice of the optimization technique (genetic algorithms) to solve the stress test generation optimization problem is discussed in Section 6.4. The genetic algorithm formulation to our problem is presented in Section 6.5.

6.1 Stress Test Heuristics

Our previous stress testing technique, TSSTT [23], can be applied only when SDs do not have AP constraints, i.e., their scheduling is not constrained. Therefore, given a specific network to stress test, we identify a message (or a set of messages) in a DCCFP of a SD which imposes maximum traffic on the network [23]. We refer to such messages as *maximum stressing messages*. Then, using the start times of the maximum stressing messages selected in each DCCFP, the selected set of DCCFPs are scheduled in such a way that the maximum stressing messages can all be sent concurrently. In other words, the SDs' DCCFPs are scheduled by shifting their NTUP functions along the time axis. Such a heuristic is illustrated in Figure 15-(a) and (b). Figure (a) shows the NTUPs for three DCCFPs of a given ISDS (the ISDS contains three SDs and each SD have one DCCFP). The corresponding maximum stressing messages (marked with vertical lines) execute at different time instants. Figure (b) shows how the three DCCFPs are scheduled to lead to maximum stress at a given time instant: $DCCFP_3$ is triggered first, then $DCCFP_2$ is triggered a bit later, followed by $DCCFP_1$.

When SDs have AP constraints, DCCFPs executions can no longer be freely shifted along the time axis. Each SD's DCCFP can only be scheduled in time instances inside the SD's Accepted Time Set (ATS) (Section 5.5.4). The difference between the stress test heuristics of the current work with that of TSSTT [23] can be better emphasized by comparing the two illustrations in Figure 15-(b) and (c). As opposed to Figure (b), Figure (c) shows the periods of time (ATSs) during which it is legal to trigger the three DCCFPs. Each of the three DCCFPs have different APs (the color-coded ellipses denote the ATS of each SD).

In short, our stress test heuristic in the current work is to look for SD schedules such that each SD start time is inside its ATS. Only SDs (i.e., their respective DCCFPs) that are members of an ISDS are considered in order to ensure we comply with inter-SD constraints. For each such schedule, the entailed traffic (stress) is the maximum combined traffic (over all involved DCCFPs). Note that it is not always possible to achieve the maximum possible stress (which could be entailed by concurrent execution of the SDs under study) when considering AP constraints due to constraints in SD start times, e.g., marked by a vertical line in Figure 15-(c). Considering that ATSs of different SDs in a SUT can be in general very different than each other, we can also see in Figure 15-(c) that the optimization (search) algorithm needed to derive test requirements in this context will likely have a complex set of constraints to satisfy and is expected not to be as simple as the one in [23].

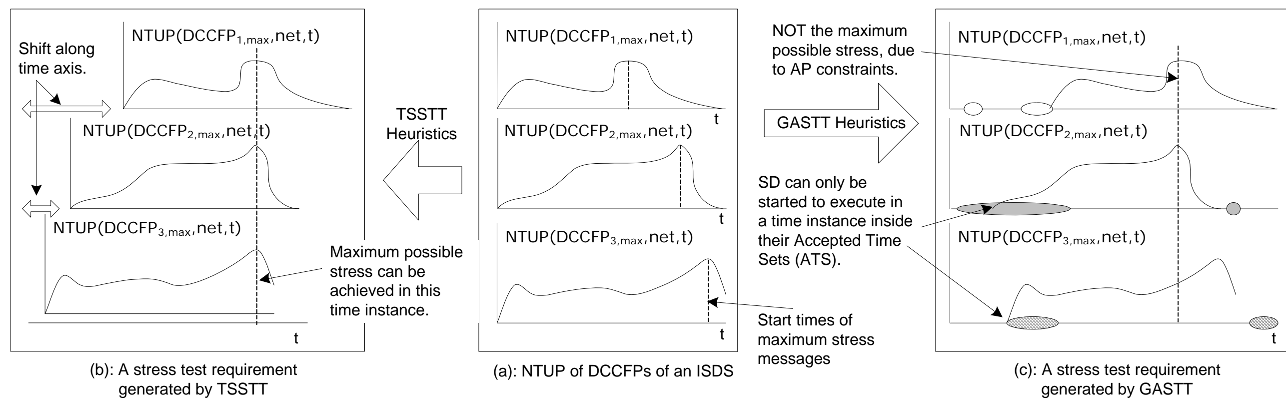


Figure 15. An example comparing the heuristics of TSSTT [23] and GASTT (the current work).

6.2 Formulation as an Optimization Problem

The stress test heuristics defined above is an optimization problem, since it tries to find the maximum stress messages given a set of constraints. In order to solve this optimization problem, we formulate it formally as shown in Figure 16.

Objective Function: Maximize the traffic on a specified network

Variables:

- A subset of DCCFPs
- Schedule to run the selected DCCFPs

Constraints:

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

Figure 16. Formulating the problem of generating stress test requirements as an optimization problem.

Note that multiple concurrent invocations of a SD might be allowed in a system, e.g., a SD which is triggered by five sensors concurrently. Therefore multiple DCCFP instances of such a SD can be executed to maximize stress during testing. Our technique derives the number of multiple invocations of a SD from the information specified in a system context diagram [24], i.e., a diagram specifying actors interacting with the system and their expected numbers at run-time. For example, if five instances of an actor can trigger a SD, it implies that five instances of the SD (i.e., one of its corresponding DCCFPs) can run concurrently.

6.3 Output Stress Test Requirements

Assuming that a SUT has n SDs (SD_1, …, SD_n), a test requirement will be a schedule of a selected set of DCCFPs in the form of: <(ρ_{1max}, $\alpha\rho_{1max}$), …, (ρ_{nmax}, $\alpha\rho_{nmax}$)>, where for the i-th entry of the sequence, ρ_{imax} is a DCCFP in the DCCFP set of SD_i, $DCCFP(SD_i)$, that entails the maximum traffic over the selected network. $\alpha\rho_{imax}$ is the start time of ρ_{imax}, i.e., the time to trigger ρ_{imax}. Intuitively, if none of the DCCFPs of SD_i has any message going through the selected network, it means that that SD_i does not have any traffic on the network and hence it will not be included in the test requirements. In such a case, the i-th entry is null.

6.4 Choice of the Optimization Technique: Genetic Algorithms

For the test requirement generation problem at hand, which is an optimization scheduling problem, using Linear Programming (LP) is impossible as the constraint regions of several ATSs altogether (their unions) can generally be non-linear (disconnected in the context of ATSs). To better explain such a non-linearity, suppose an n-dimensional space where n ATSs (corresponding to n SDs) are specified, where each ATS can be disconnected (e.g., the ATS of a bounded or a periodic AP). In such a case, the acceptable search space of the problem is the union of all those ATSs. Due to the disconnectivity of each ATS, the search space resulting from their union will also be non-linear, thus making the entire problem unsolvable by LP. 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 Non-Linear Programming (NLP) technique is thus needed that alleviates this problem by exploring multiple parts of the non-linear problem space.

However, due to the disconnected nature of ATSs and also the unbounded number of possible schedules for each SD in our problem, we expect to face one of the major common challenges in NLP: "local optima". Algorithms that propose to overcome this difficulty are termed "Global optimization techniques" [27], also known as meta-heuristic methods. They continually search for better solutions by altering a set of current solutions. Furthermore, meta-heuristic methods are usually more scalable and flexible [52] than other NLP techniques (e.g., branch-and-bound) for complex problems like ours.

Genetic Algorithms (GA) and Simulated Annealing (SA) are two of the commonly used global optimization techniques. Some studies, such as [31] indicate that SA outperforms GAs, while others, such as [13] suggest that GAs produce solutions equivalent or superior to SA. Most researchers, however, seem to agree that because GAs maintain a population of possible solutions, they have a better chance of locating the global optimum compared to SA and Taboo Search (TS) which proceed one solution at a time [34, 35].

Furthermore, because SAs maintain only one solution at a time, good solutions may be discarded and never regained if cooling occurs too quickly. Similarly, TS may miss the optimum solutions. Alternatively, steady state GAs, one of the variations of GAs, accept newly generated solutions only if they are fitter than previous solutions. Furthermore, GAs lend themselves to parallelism, as they manipulate whole populations: computations for different parts of the population can be dispatched to different processors. SA, on the other hand, cannot easily run on multiple processors because only one solution is constantly manipulated [34]. Hence, we adopt GA as our optimization technique methodology.

6.5 Tailoring Genetic Algorithm to Derive Instant Stress Test Requirements

We use a GA to solve the optimization problem of finding DCCFPs and their triggering times such that instant traffic on a network or a node is maximized. This section describes how we tailored the different components of the GA to this problem. We define a chromosome representation in Section 6.5.1. Constraints defining legal chromosomes are formulated in Section 6.5.2. Derivation of the initial GA population is discussed in Section 6.5.3. The concept of a time search range which is needed in our GA for the initialization process as well as the operators is discussed in Section 6.5.4. The objective (fitness) function is described in Section 6.5.5. GA operators (crossover and mutation) are finally presented in Section 6.5.6.

6.5.1 Chromosome

Chromosomes define a group of solutions to be optimized. Their representation and length must be precisely defined and justified [26]. Recall we need to optimize the selection of SDs' DCCFPs and their schedule, i.e., their start times. Thus, we need to encode both DCCFP identifiers and their arrival times in a chromosome. A gene can be depicted as a pair ($\rho_{i,selected}$, $\alpha\rho_{i,selected}$), where $\rho_{i,selected}$ is a selected DCCFP of SD_i, and $\alpha\rho_{i,selected}$ is the start time of $\rho_{i,selected}$. Together, the pair represents a schedule of a specific DCCFP. If no DCCFP is selected from a SD (because the SD does not have traffic over a particular network, for example), the gene is denoted as *null*. This is to ensure that the number of genes in each chromosome remains constant as this facilitates the definition of mutation/cross-over operators and fitness function.

We formalize the concepts we employ in a metamodel which is depicted in Figure 17-(a). Such a metamodel also constitutes a starting point for the design of our tool (Section 7). A *Chromosome* is composed of a sequence of *Gene* instances, specifically as many genes as SDs in the system. 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 6.5.6.

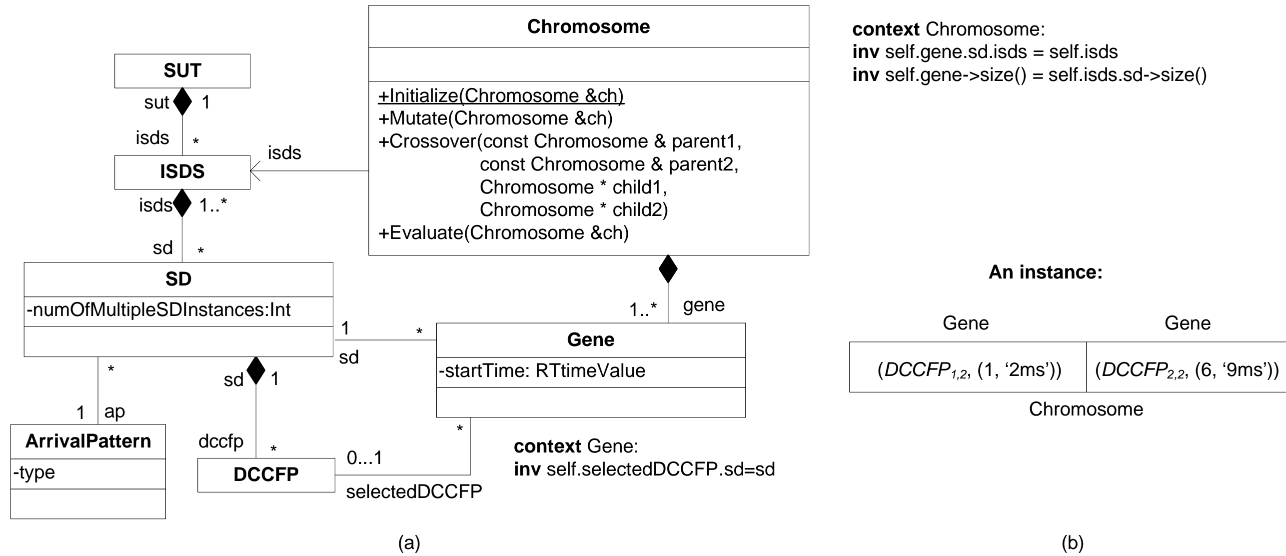


Figure 17-(a): Metamodel of chromosomes and genes in our GA algorithm. (b): Part of an instance of the metamodel.

Each *Gene* is associated with a SD. Furthermore, it has an association (*selectedDCCFP*) to zero (if no DCCFP is chosen) or one DCCFP. A Gene has an attribute *startTime*, of type *RTtimeValue* (defined in the UML-SPT), which is the time value to trigger *selectedDCCFP*. Each DCCFP belongs to a SD, whereas each SD can have several DCCFPs. Attribute *numOfMultipleSDInstances* is the number of multiple SD instances which are allowed to be triggered concurrently. Each SD can be a member of several ISDSs, and an ISDS can have one or more SDs. Arrival pattern information of SDs is stored in instances of a class *ArrivalPattern* (attributes of such a class can be easily defined based on the discussions in Section 5.5.2, such as type and AP parameters). A SUT (model) has one or more ISDSs. Finally, to specify the well-formedness criteria of the above metamodel, we have defined two and one invariants for the *Chromosome* and *Gene* classes, respectively. For example, the SD instance owning the *selectedDCCFP* of a *Gene* should be the same SD instance refer by *Gene*. Recall that we are maximizing traffic at a given instant and what matters is thus the number of SDs that can be triggered concurrently. We therefore do not need to model sequential and conditional constraints.

An example of a chromosome and a gene is illustrated in Figure 17-(b), which complies with the metamodel in Figure 17-(a). The chromosome is composed of two genes, since it is assumed that the SUT has two SDs: SD_1 and SD_2. $DCCFP_{1,2}$ and $DCCFP_{2,2}$ are selected DCCFPs of SD_1 and SD_2, respectively. The genes indicate that the DCCFPs' start times are 2 ms and 9 ms, respectively.

6.5.2 Constraints

Inter-SD and arrival pattern constraints should be satisfied when generating new chromosomes from parents. Otherwise GA *backtracking* procedures [26] should be used. Backtracking, however, has its drawbacks: it is time consuming and some GA tools incorporate backtracking while others do not. To allow for generality, we assume no backtracking methodology is available. Therefore, we have to ensure that the GA operators always produce chromosomes which satisfy the GA's constraints. In order to do so, we formally express inter-SD and arrival pattern constraints based on our metamodel.

6.5.2.1 Constraint #1: Inter-SD constraints

We incorporated inter-SD constraints in ISDSs (Section 5.2). A set of DCCFPs are allowed to execute concurrently in a SUT only if their corresponding SDs are members of an ISDS. As discussed in Section 6.5.1, each chromosome is a sequence of genes, where each gene is associated with zero or one DCCFP. Therefore, a chromosome satisfies Constraint #1 only if the SDs of DCCFPs corresponding to its genes are members of a

same ISDS. In other words, each chromosome corresponds to only one ISDS. We can formulate the above constraint as a class invariant on class *Chromosome* (Figure 17-(a)) as presented in Figure 18.

```
context Chromosome
inv: self.gene.selectedDCCFP.sd.isds->asset()->size()=1
```

Figure 18- Constraint #1 of the GA (an OCL expression).

6.5.2.2 Constraint #2: Arrival pattern constraints

Given a chromosome, the OCL post-condition in Figure 19 determines if the chromosome (the scheduling of its genes) satisfies the Arrival Pattern Constraints (APC) of SDs. The function *IsAPCSatisfiedByAChromosome(c :Chromosome)* returns true if all genes of the chromosome satisfy the APCs. The OCL post-condition makes use of function *IsAPCSatisfied(startTime, AP)*, defined in Section 5.5.3.

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

Figure 19- Constraint #2 of the GA (an OCL function).

6.5.3 Initial Population

Determining the population size of a GA is challenging [2]. A small population size will cause the GA to quickly converge on a local minimum because it insufficiently samples the search space. A large population, on the other hand, causes the GA to run longer in search for an optimal solution. Haupt and Haupt in [26] list a variety of works that suggests adequate population sizes. The authors reveal that the work of De Jong [17] suggests a population size ranging from 50 to 100 chromosomes. Grefenstette et al. [25] recommend a range between 30 and 80, while Schaffer and his colleagues [47] suggest a smaller population size, between 20 and 30. We choose 80 as the population size as it is consistent with most of experimental results.

The GA initial population generation process should ensure that the two constraints of Section 6.5.2 are met. The pseudo-code to generate the initial set of chromosomes is presented in Figure 20. As indicated by the constraint #1, each chromosome corresponds to an ISDS. Therefore, line 1 of the pseudo-code chooses a random ISDS and the initialization algorithm continues with the selected ISDS to create an initial chromosome. Note that to generate our GA's initial population, *CreateAChromosome()* is invoked 80 times.

```
Function CreateAChromosome(): Chromosome
c: Chromosome
1  ISDS=a random ISDS
2  For all SDi∈ISDS
3      c.genei.selectedDCCFP = a random DCCFP from SDi
4  For all SDi∉ISDS
5      c.genei=null
6  Intersection=ATS(SD1) ∩ ATS(SD2) ∩… ∩ATS(SDi), where
   SDj,j=1…i∈ISDS
7  If Intersection≠{}
8      Choose a random time instance tschedule in Intersection
9          For all c.genei ≠null
10         c.genei.startTime= tschedule
11 Else
12     For all c. genei ≠null
13         c.genei.startTime= A random time instance ti in ATS(SDi)
14 Return c
```

Figure 20-Pseudo-code to generate chromosomes of the GA's initial population.

For each SD in the ISDS selected in line 1, 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, and thus higher stress test values are achieved earlier in the GA evolution, i.e., being as close as possible to the optimal solution as early as possible in the search. 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 5.5.4. The intersection of two ATSs is an ATS that contains all the time instances and time intervals that are common to the two ATSs. 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 (line 8). All DCCFPs of the genes are then scheduled to this time instance (lines 10-11). 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 12-13).

When selecting a random time instance for a gene, we need a range to select of time values to select from. When calculating an intersection of ATSs, we also need a range of time values, especially when some ATSs are unbounded. This range is discussed in Section 6.5.4 below.

Following the algorithm in Figure 20, we ensure the initial population of chromosomes complies with both constraints of Section 6.5.2. Note that the above algorithm does not ensure that all the ISDSs are represented in the initial population. However, after creating an initial population of randomly-selected ISDS and during the GA process, one of our GA's mutation operators (Section 6.5.6.2) will mutate an entire chromosome by assigning another, randomly-selected ISDS, to the chromosome. That operator allows the search to investigate different ISDSs.

6.5.4 Determining a Maximum Search Time

One important issue in our GA design is the range of the random numbers chosen from the ATS of a SD with an arrival pattern. As discussed in Section 5.5.4, the number of ATIs or ATPs in some types of APs (e.g. periodic, bounded) can be infinite. Therefore, choosing a random value from such an ATS can yield very large values, thus creating implementation problems.

Another direct impact of such unboundedness on our GA is that it would significantly decrease the probability that all (or a subset) of start times of DCCFPs (corresponding to the genes of a chromosome) overlap or be close to each other. If the maximum range when generating a set of random numbers is infinity, the probability that all (or a subset) of the generated numbers are relatively close to each other is very small. Thus, to eliminate such problems, we introduce a *Maximum Search Time*. This maximum search time is essentially an integer value (in time units) which enforces an upper bound on the selection of random values for start times of DCCFPs, chosen from an ATS. The GA maximum search time will be used in our GA operators (Section 6.5.6) to limit the maximum ranges of generated random time values.

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

The impact of MST on exercising the time domain is illustrated in Figure 21 using an example, where the ATSs of three APs (a periodic, a bounded and a bursty one) are depicted. Four maximum search times (MST_i) have been arbitrarily chosen. The search range specified by MST_1 (*Search Range*$_1$) is not a *suitable* one since only time values in the first ATI of the bounded ATS will be chosen thus preventing the GA from searching all possible start times in the ATS range of the depicted bounded AP. This will limit the search space, thus reducing the chances of finding the most stressful situations. Following a similar reasoning, the search range specified by MST_2 (*Search Range*$_2$) is not a suitable one either. MST_3 and MST_4 specify ranges in which a complete search over the possible ATS values can be performed. Comparing the last two, the latter does not provide any advantage in terms of completeness of the search range over the former, while at the same time causing a slower convergence of the GA. Therefore, MST_3 is a preferable maximum search time over MST_4. Note that the ATS of the bursty AP in Figure 21 does not play any role in determining a suitable maximum search time, since by having an unrestricted ATS, regardless of the choice of such a MST, any start time can be chosen for a bursty AP.

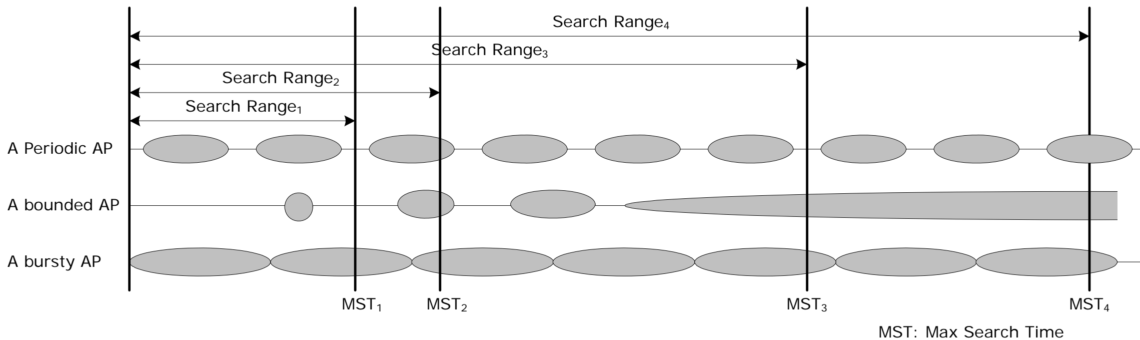


Figure 21-Impact of maximum search time on exercising the time domain.

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

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

If all APs are irregular, then a suitable MST ($MST_{suitable}$) will be the maximum of all latest irregular arrival times in all ATSs. For example, the ATSs of three irregular APs are depicted in Figure 22. A $MST_{suitable}$ will be the last arrival time of the third AP (as depicted), which has the maximum time value. This maximum search time will allow the GA to effectively search in the time domain, considering all possible start times from all APs.

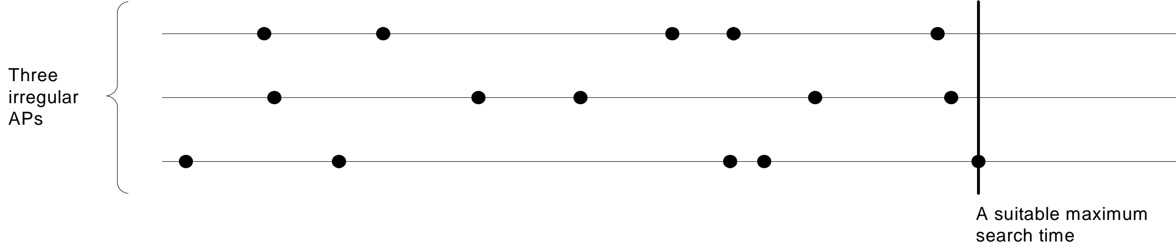


Figure 22- The ATSs of three irregular APs.

If APs are infinite, the occurrence and intersection of different ATIs in the APs should be taken into account. Since only periodic, irregular, and bounded APs have discrete ATSs (Section 5.5.4), we only consider them in finding a suitable maximum search time. Unrestricted APs (bursty and unbounded) do not impose any restrictions on the selection of a suitable maximum search time, since any time value is acceptable by a bursty or an unbounded AP.

We present in Table 1 a set of heuristics to identify a $MST_{suitable}$ based on a given set of periodic, irregular, and bounded APs. In the heuristics, *ap.type* denotes the type of an AP, e.g., *'bounded'*, *'periodic'*.

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

Table 1-A set of heuristics to identify a suitable MST ($MST_{suitable}$).

Heuristic #1 denotes that a $MST_{suitable}$ should be greater than the maximum value among all maximum ATPs of irregular APs. The rationale beyond this heuristic is the same as the case when all APs of a TM are irregular (discussed above). *ap.maxATP* denotes the maximum ATP of an irregular AP and can be calculated using the formula in Equation 4.

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

Equation 4-A formula to calculate the maximum ATP (*maxATP*) of an irregular AP.

Heuristic #2 is meant to provide a full search coverage on all bounded APs simultaneously. Recall from Section 5.5.4 that every bounded ATS has an ATI whose end time is unbounded. Furthermore, all time instances after the start time of such an ATI are accepted arrival times. If the MST value is chosen to be greater than all such start times among all bounded APs, the GA will be able to have a full search coverage on all of the APs, and thus, maximizing the chances of finding a test schedule with high stress value. To

better formalize heuristic #2, we refer to such a start time as the bounded AP's *Unbounded Range Starting Point (URSP)*. The URSP of a bounded AP can be calculated using the formula in Equation 5, given the ATIs of the AP.

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

Equation 5-A formula to calculate the *Unbounded Range Starting Point (URSP)* of a bounded AP, given the ATIs of the AP.

For example, the URSP of the bounded APs in Figure 23 are denoted as $URSP_i$. If the minimum and maximum inter-arrival times (*minIAT* and *maxIAT*) of a bounded AP are given, the formula in Equation 6 can be used to calculate the value of the URSP. The proof of this formula is given in Appendix A.

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

Equation 6-A formula to calculate the *Unbounded Range Starting Point (URSP)* of a bounded AP, given the minimum and maximum inter-arrival times (*minIAT* and *maxIAT*) of the AP.

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

A $MST_{suitable}$ value should be calculated by considering all three heuristics in Table 1, i.e., a $MST_{suitable}$ is equal to the maximum value among the three right-hand side in the three ≥ inequalities. To better explain the above set of heuristics, an example with three irregular, three periodic, and three bounded APs is shown in Figure 23 and the process of deriving a $MST_{suitable}$ for this particular example is described next.

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

The URSP of each bounded AP has been calculated using the formula in Equation 6 based on the $minIAT_i$ and $maxIAT_i$ of each AP, and is denoted as $URSP_i$. The maximum value among all $URSP_i$'s is referred to as $URSP_{max}$. Heuristic #2 denotes that a $MST_{suitable}$ should be greater than this value.

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

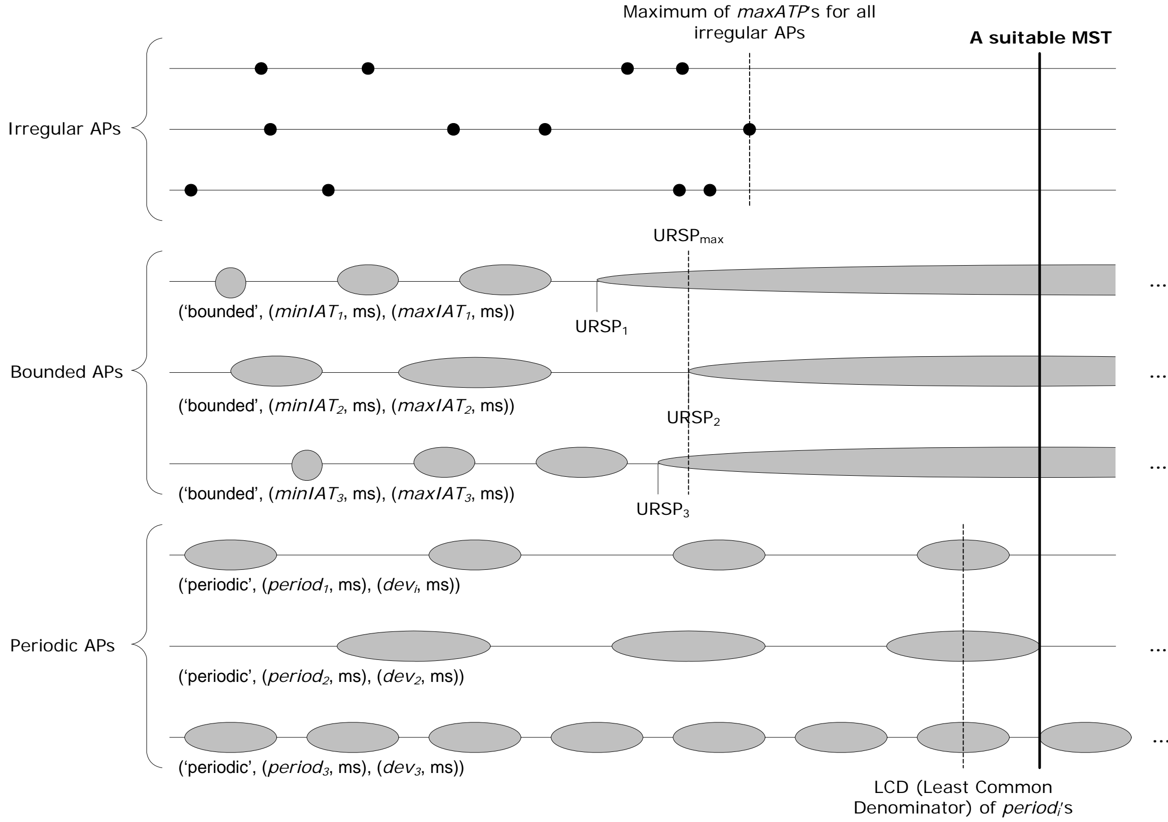


Figure 23-Illustration showing the heuristic of choosing a suitable maximum search time.

6.5.5 Objective (Fitness) Function

Optimization problems aim at searching for a solution within the search space of the problem such that an objective function is minimized or maximized [2]. In other words, the objective function can aim at either minimizing the fitness of chromosomes or maximizing them. The objective function of a GA measures the fitness of a chromosome. Recall from Section 0 that our optimization problem is defined as follows: *What selection and what schedule of DCCFPs maximize the traffic on a specified network or node (at a specified time instant)?*

Recall from Section 5.5 that we apply our GA-based technique to find stress test requirements which stress a SUT in a time instant. Therefore, let us refer to the objective function in this section as *Instant Stress Test Objective Function (ISTOF)*. The *ISTOF* should measure the maximum instant traffic entailed by a schedule of DCCFPs, specified by a chromosome. Using the network traffic usage model in Section 5.4, we define ISTOF in Equation 7.

$$ISTOF : Chromosome \rightarrow \mathrm{Re}\,al$$

$$\forall c \in Chromosome : ISTOF(c) = \max_{\forall t \in SearchRange} \sum_{\forall g \in Genes(c)} NTUP(g.selectedDCCFP, net, t) \times g.sd.numOfMultipleSDInstances$$

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

Equation 7- Instant Stress Test Objective Function (ISTOF).

The first line of Equation 7 indicates that the input domain and range of ISTOF are chromosomes and real numbers. *Length(dccfp)* is a function to calculate the time duration of a DCCFP (modeled in the corresponding SD using UML-SPT tagged-values). *Genes(c)* returns the set of not null genes of the chromosome *c*. *net* is the given network to stress test. *NTUP* is the traffic usage function (Section 5.4) to

measure the instant data traffic in a network. The value of *NTUP* is multiplied by the SD's *numOfMultipleSDInstances* value. When multiple instances of a DCCFP are triggered at the same time, the entailed traffic at each time instant is proportional to the number of instances.

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

To better illustrate the idea behind ISTOF, let us discuss how ISTOF for the chromosome in Figure 17-(b) is calculated. The calculation process is shown in Figure 24. The chromosome contains two genes, which correspond to $DCCFP_{1,2}$ and $DCCFP_{2,2}$. The search range is [2ms, 20ms]: 2 is the start time of the earliest DCCFP, namely $DCCFP_{1,2}$; 20 is the start time (9) of the other DCCFP, $DCCFP_{2,2}$, plus its length (11). ISTOF sums the *NetInsDT* values in this range and finds the maximum value: bottom right of Figure 24. The output value of ISTOF is 110 KB.

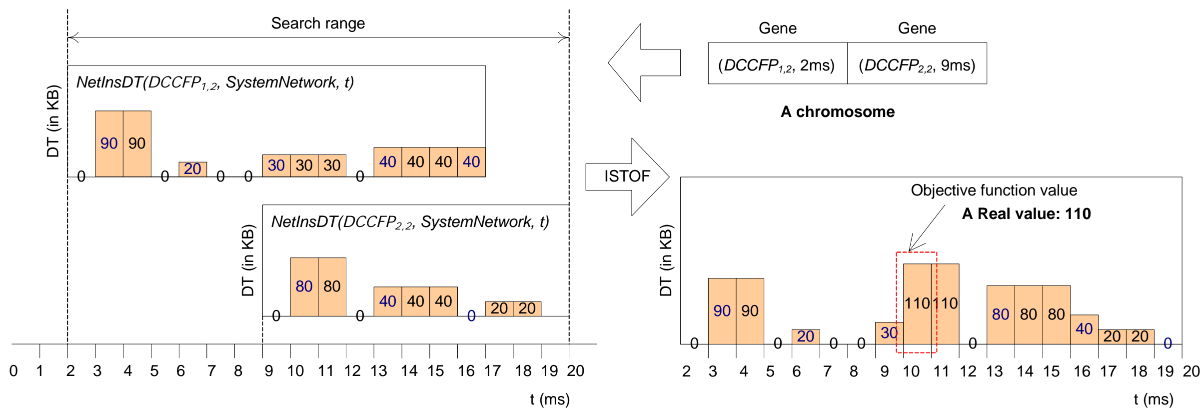


Figure 24-Computing the Instant Stress Test Objective Function (ISTOF) value of a chromosome.

6.5.6 Operators

Operators enable GAs to explore a solution space [26] and must therefore be formulated in such a way that they efficiently and exhaustively explore it. If the application of an operator yields a chromosome which violates at least one of the GA's constraints, the operation is repeated to generate another chromosome. This is an alternative to GA backtracking and is done inside each operator, i.e., each operator generates temporary children first and checks if they do not violate any constraints (Section 6.5.2). If the temporary children satisfy all the constraints, they are returned as the results of the operator. Otherwise, the operation is repeated. Furthermore, operators should be formulated such that they can explore the entire solution space. We define the crossover and mutation operators next.

6.5.6.1 Crossover Operator

Crossover operators aim at passing on desirable traits or genes from generation to generation [26]. Varieties of crossover operators exist, such as sexual, asexual and multi-parent. The former uses two parents to pass traits to two resulting children. Asexual crossover involves only one parent. Multi-parent crossover combines the genetic makeup of three or more parents when producing offsprings. Different GA applications call for different types of crossover operators. We employ the most common of these operators: sexual crossover.

The general idea behind sexual crossover is to divide both parent chromosomes into two or more fragments and create two new children by mixing the fragments [26]. In our application, since each gene corresponds to a SD, we consider the sexual crossover's 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 [45]) is (n-1)-point, and is denoted *nPointCrossover*. In our application, the mixing of the fragments is additionally subject to a number of constraints (Section 6.5.2): A newly generated chromosome should satisfy the inter-SD and arrival pattern constraints. We ensure this by designing the GA operators in a way that they would never generate an offspring violating a constraint.

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

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

Let us consider the example in Figure 25 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 the SUT has two ISDSs, $ISDS_1$ and $ISDS_2$ such that $ISDS_1$={SD_1, SD_4, SD_5} and $ISDS_2$={SD_1, SD_3, SD_4}, and DCCFP $\rho_{i,x}$ belongs to SD_i. Parent 1 has genes corresponding to DCCFPs in {SD_1, SD_4, SD_5}$\subset ISDS_1$. Parent 2's genes are DCCFPs in {SD_1, SD_3, SD_4}$\subset ISDS_2$. The results of applying *nPointCrossover* and *nPointProbCrossover* are shown in Figure 25 (b) and (c) respectively. In *nPointCrossover*, the fragments of Parent 1 and Parent 2 are alternately interchanged (Figure 25-(b)): Child1 (resp. Child2) receives the first, third and fifth genes from Parent1 (resp. Parent2) and the second and fourth genes from Parent2 (resp. Parent1). Using the same example for *nPointProbCrossover*, one possible outcome appears in Figure 25-(c). Bold genes indicate the fragments interchanged by *nPointProbCrossover*. Three of the four generated children (all except Child 2 in Figure 25-(c)) conform to constraint #1, i.e., the SDs corresponding to the genes of each child belong to one ISDS ($ISDS_1$ or $ISDS_2$), as well as constraint #2. Since Child 2 in Figure 25-(c) violates constraint #1, the two temporary children (Child 1 and 2 in Figure 25-(c)) are abandoned, and this particular execution of *nPointProbCrossover* is repeated.

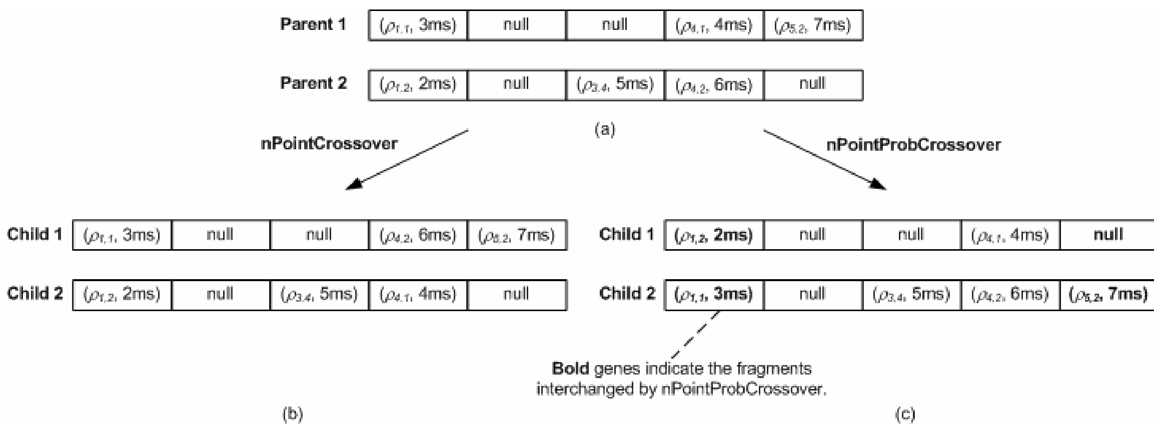


Figure 25-Two example uses of the crossover operators.

The advantages of *nPointProbCrossover* are twofold. It introduces further randomness in the crossover operation. By doing so, it allows further exploration of the solution space. However, *nPointProbCrossover* has

its disadvantages: the resulting children may be replicas of the parents, with no alteration occurring. This is never the case with *nPointCrossover*; resulting children are always genetically distinct from their parents.

Crossover rates are critical. A crossover rate is the percentage of chromosomes in a population being selected for a crossover operation. If the crossover rate is too high, desirable genes will not be able to accumulate within a single chromosome whereas if the rate is too low, the search space will not be fully explored [26]. De Jong [17] concluded that a desirable crossover rate should be about 60%. Grefenstette et al. [25] built on De Jong's work and found that the crossover rate should range between 45% and 95%. Consistent with the findings of De Jong and Grefenstette, we apply a crossover rate of 70%.

6.5.6.2 Mutation Operator

Mutation aims at altering the population to ensure that the GA avoids being caught in local optima. The process of mutation proceeds as follows: a gene (or a chromosome) is randomly chosen for mutation, the gene (or the chromosome) is mutated, and the resulting chromosome is evaluated for its new fitness. We define three mutation operators that (1) mutate a non-null gene (a gene with an already assigned DCCFP) in a chromosome by altering its DCCFP, (2) mutate the start time of a non-null gene, or (3) mutate the entire chromosome by assigning another, randomly-selected ISDS to it (i.e., assign to each gene of the chromosome a randomly-selected DCCFP from the corresponding ISDS's SDs, and start times from the ATSs of that ISDS's SDs, in a way similar to the creation of the chromosomes of the initial population). The mutation operators are referred to as *DCCFPMutation*, *startTimeMutation*, and *ISDSMutation,* respectively.

The idea behind the *DCCFPMutation* operator is to allow the search to investigate different DCCFPs. The idea behind the *startTimeMutation* operator is to move DCCFP executions along the time axis. This is done in such a way that the constraints we defined on the chromosomes are met (Section 6.5.2). The purpose of the *ISDSMutation* operator is to increase the population of genes related to an ISDS, thus increasing population variability. This is expected to lead to a better search, especially when the number of ISDSs is close to (or above) the selected population size (Section 6.5.3). In that case, the initial population created by the algorithm in Section 6.5.3 will have, on average, only one, a few, or even no chromosome corresponding to an ISDS. In that case, different combinations of DCCFPs and their triggering times inside an ISDS may not thus be thoroughly searched. Our initial experiments with the GA were not using the *ISDSMutation* operator and revealed that this operator was crucial to converge towards high fitness values.

Since the mutation operators alter non-null genes only, they do not change the set of SDs corresponding to a chromosome, thus ensuring that constraint #1 is satisfied (the set of SDs will still belong to the same ISDS). However, start times are changed by the mutation operator *startTimeMutation*, resulting in a possible violation of constraint #2. The output of the *DCCFPMutation* operator will always adhere to constraint #2, since the start times are unchanged by the operator. One way of making sure that a generated chromosome by the *startTimeMutation* operator satisfies the arrival pattern constraints is to set the new start times to a random value in the range of accepted arrival time values of a SD, i.e., Accepted Time Sets (ATS) – (Section 5.5.3). 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.

A mutation rate is the percentage of chromosomes in a population being selected for mutation. Throughout the GA literature, various mutation rates have been used. If the rates are too high, too many good genes of a chromosome are mutated and the GA will stall in converging [26]. Back [4] enumerates some of the more common mutation rates used. The author states that De Jong [17] suggests a mutation rate of 0.001, Grefenstette [25] suggests a rate of 0.01, while Schaffer et al. [47] formulated the expression $1.75/\lambda\sqrt{length}$ (where λ denotes the population size and *length* is the length of chromosomes) for the mutation rate. Mühlenbein [38] suggests a mutation rate defined by $1/length$. Smith and Fogarty [49] show that, of the common mutation rates, those that take the length of the chromosomes and the population size into consideration perform significantly better than those that do not. Based on these findings, we apply for all

the three mutation operators the mutation rate suggested by Schaffer et al.: $1.75/\lambda\sqrt{length}$. Once mutation is decided, using this mutation score, the three mutation operators are applied with the same probability.

7 AUTOMATION AND ITS EMPIRICAL ANALYSIS

Section 7.1 provides an overview of a prototype tool that was implemented to support the application of our genetic algorithm-based stress test technique: *GARUS (GA-based test Requirement tool for real-time distribUted Systems)*. A carefully designed empirical study, using this tool, is then presented in Section 7.2 to validate the design choices of our GA.

7.1 Tool Description

GARUS (GA-based test Requirement tool for real-time distribUted Systems) is our prototype tool for deriving stress test requirements. Section 7.1.1 presents the class diagram of GARUS. A functional overview of GARUS is described in Section 7.1.2. The input/output file formats are presented in Section 7.1.3 and Section 7.1.4, respectively.

7.1.1 Class Diagram

The simplified class diagram of GARUS is shown in Figure 26. The classes in the class diagram are grouped in three packages: *TestModelGenerator*, *TestModel* and *GA*.

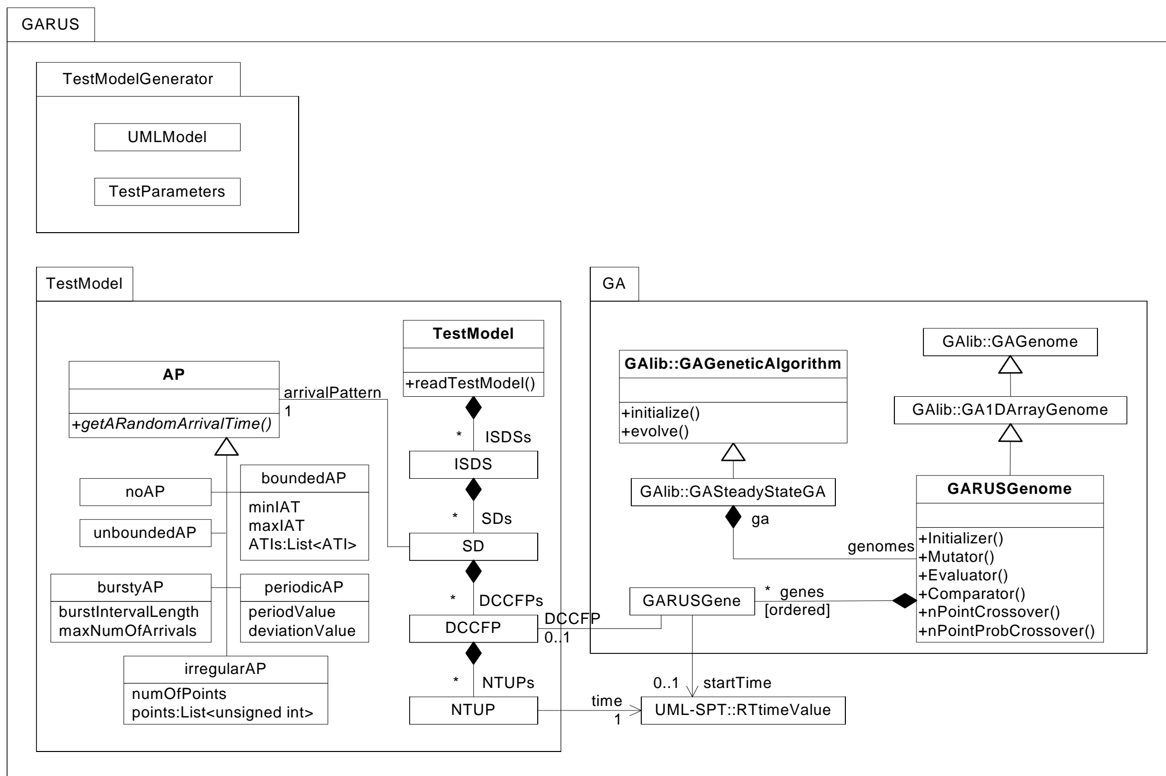


Figure 26-Simplified class diagram of GARUS.

To simplify the implementation of GARUS, we assume that a TM has already been built from a given UML model and a set of test parameters by a test model generator (the *TestModelGenerator* package). The TM is also assumed to be filtered by the given set of test parameters. For example, if test parameters are for a *StressTestNetInsDT* test strategy over a network *net*, all DCCFPs in the CFM and network usage pattern parts of a TM are assumed to have been filtered by that particular network. Thus, we would ideally have a package in GARUS that handles this. However, to simplify

the implementation of GARUS, the package is currently bypassed (the filtered TM is built manually from a UML model).

The classes in the *TestModel* package store information about the test model of a SUT. The *GA* package includes the GA domain-specific classes, which solve the optimization problem and derive stress test requirements.

One object of class *TestModel* and one object of class *GASteadyState GA* are instantiated at runtime for a SUT. The connection between the two packages is achieved via class DCCFP (in the *TestModel* package) and class *GARUSGene* (in the *GA* package).

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 5.5.2). 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 5.5.4) according to the type of arrival pattern.

7.1.2 Functional Overview

A functional overview of GARUS is presented using an activity diagram in Figure 27. 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 6.5.3). Note that object *ga* has a collection of chromosomes of type *GARUSGenome,* and each object of type *GARUSGenome* has an ordered set of genes of type *GARUSGene* (refer to the class diagram in Figure 26). Furthermore, *ga*'s parameters (e.g. mutation rate) are set to the values as discussed in Section 6.5.

GARUS then evolves *ga* using the overloaded GA mutator and crossover operators (Section 6.5.6). When the evolution of *ga* finishes, the best individual (accessible by `ga.statistics().bestIndividual()`) is saved in the output file (see Section 7.1.4).

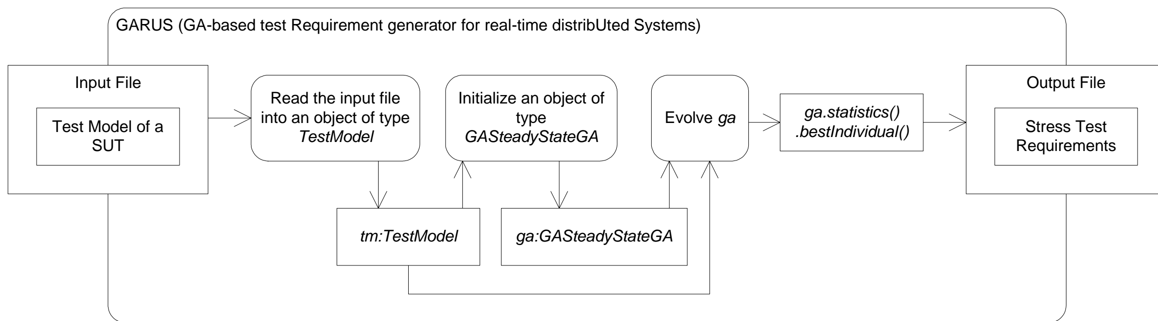


Figure 27-Overview activity diagram of GARUS.

7.1.3 Input File Format

The input file provided to GARUS contains the test model (TM) of a SUT. As it was shown in Figure 6, a TM consists a CFM (including DCCFPs), inter-SD constraint (ISDSs) and distributed traffic usage patterns.

Referring to Figure 6, stress test parameters are also part of the input. As discussed in Section 4.1, 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 Section 5.4, a test model can be filtered based on different attributes discussed in distributed traffic usage analysis (e.g. location, direction, and period).

The input file is in a format to accommodate a *filtered* TM. For example, if test parameters are for a *StressTestNetInsDT* test strategy over a network *net*, all DCCFPs in the CFM and network usage pattern parts of a TM are assumed to have been filtered by that particular network. The input file format consists of several blocks, each specifying different elements of a TM. GARUS input file format is shown using the BNF in Figure 28.

The input file format can be best described using an example. An example input file is shown in Figure 29. Different blocks are separated with a gray highlight. The TM starts with a block of two ISDSs *ISDS0* and *ISDS1* (*ISDSsBlock* in Figure 28). For example, *ISDS0* consists of three SDs: *SD0*, *SD1*, and *SD2*.

The second block of the input file shows SDs (*SDsBlock* in Figure 28). 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*.

$$inputFileFormat ::= ISDSsBlock\ \ SDsBlock\ \ SDAPsBlock\ \ DCCFPsBlock$$
$$ISDSsBlock ::= nISDSs\ \ ISDS_1 \ldots ISDS_{nISDSs}$$
$$ISDS_i ::= ISDSName_i\ \ nSDsInISDS_i\ \ SDName_1 \ldots SDName_{nSDsInISDS_i}$$
$$SDsBlock ::= nSDs\ \ SD_1 \ldots SD_{nSDs}$$
$$SD_i ::= SDName_i\ nMultipleInstances_i\ \ nDCCFPsInSD_i\ \ DCCFPName_1 \ldots DCCFPName_{nDCCFPsInSD_i}$$
$$SDAPsBlock ::= SDAP_1 \ldots SDAP_{nSDs}$$
$$SDAP_i ::= SDName_i\ \ APType_i\ \ APParameters_i$$
$$APType_i ::= no_arrival_pattern \,|\, periodic \,|\, bounded \,|\, irregular \,|\, bursty \,|\, unbounded$$
$$APParameters_i ::= \begin{cases} \in & \text{;if } APType_i \in \{no_arrival_pattern, bursty, unbounded\} \\ periodValue_i\ \ deviationValue_i & \text{;if } APType_i = periodic \\ minIAT_i\ \ maxIAT_i & \text{;if } APType_i = bounded \\ nArrivalPointsInAP_i\ \ APoint_1 \ldots APoint_{nArrivalPointsInAP_i} & \text{;if } APType_i = irregular \end{cases}$$
$$DCCFPsBlock ::= \underbrace{DCCFP_1 \ldots DCCFP_{nDCCFPs}}_{nDCCFPs = \sum_{\forall SD_i} nDCCFPsInSD_i}$$
$$DCCFP_i ::= DCCFPName_i\ \ nDTUPPsInDCCFP_i\ \ DTUPP_1 \ldots DTUPP_{nNTUPPsInDCCFP_i}$$
$$DTUPP_i ::= (\ \ time_i\ \ value_i\ \ )$$
$$GAMaxSearchTime$$

Figure 28-GARUS input file format.

The third block shows SD Arrival Pattern (AP) - (*SDAPsBlock* in Figure 28). 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 Section 5.5 for further details.

The next block in an input file is the *DCCFPsBlock*. The number of DCCFPs in a *DCCFPsBlock*, is equal to the sum of DCCFPs of all SDs, specified in the *SDsBlock*. For example, in the example input file in Figure 29, 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 5.4, 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 30 are: (1, 90), (3, 40), (4, 40), (8, 30), and (12, 50). For example, in the input file in Figure 29, 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.

```
2
ISDS0 3 SD0 SD1 SD2
ISDS1 4 SD0 SD2 SD3 SD4
5
SD0 1 5 p01 p02 p03 p04 p05
SD1 1 3 p11 p12 p13
SD2 1 2 p21 p22
SD3 1 1 p31
SD4 1 4 p41 p42 p43 p44
SD0 periodic 5 0
SD1 periodic 4 2
SD2 bounded 4 5
SD3 no_arrival_pattern
SD4 irregular 5 2 3 6 8 9
p01 5 ( 2 10 ) ( 3 5 ) ( 6 7 ) ( 12 20 ) ( 15 9 )
p02 2 ( 1 5 ) ( 4 20 )
p03 3 ( 3 5 ) ( 5 10 ) ( 6 7 )
p04 2 ( 3 9 ) ( 6 35 )
p05 1 ( 5 40 )
p11 2 ( 4 4 ) ( 7 3.4 )
p12 3 ( 1 1 ) ( 2 9 ) ( 5 6 )
p13 5 ( 2 3 ) ( 5 4 ) ( 7 1 ) ( 9 6 ) ( 11 20 )
p21 1 ( 4 30 )
p22 4 ( 2 20 ) ( 3 10 ) ( 7 15 ) ( 9 30 )
p31 3 ( 3 3 ) ( 5 9 ) ( 7 20 )
p41 2 ( 4 20 ) ( 7 4 )
p42 6 ( 2 3 ) ( 5 6 ) ( 8 8 ) ( 10 1 ) ( 12 9 ) ( 15 10 )
p43 5 ( 4 2 ) ( 6 7 ) ( 10 5 ) ( 12 3 ) ( 15 2 )
p44 2 ( 4 32 ) ( 6 10 )

25
```

Figure 29-An example input file of GARUS.

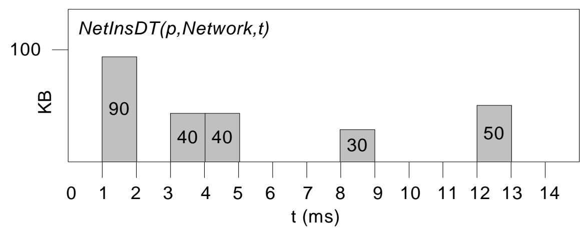


Figure 30-An example NTUP of a DCCFP.

The last piece of information in the input file is a Real value, referred to as *GAMaxSearchTime* (Section 6.5.4). This value (in time units) specifies the range (from zero) in which the GA tries to search for a best result. The initialization and the mutation operators use this maximum search value to select a random start time for the DCCFP of a gene. For example, the *GAMaxSearchTime* in the input file in Figure 29 is 25 time units. Therefore, the GA operators in GARUS will choose random seeding times for DCCFPs in the range of [0…25] time units. The higher the *GAMaxSearchTime* value, the more broad the GA’s search range. However, we expect that higher *GAMaxSearchTime* values deteriorates our GA’s performance in converging, since the higher the *GATimeSearchRange* value, the less probable that multiple DCCFPs overlap with each other. A suitable *GAMaxSearchTime* can be calculated using the two heuristics we presented in Section 6.5.4.

However, to allow variability of choices for *GAMaxSearchTime* in our experimentation, we assume that such a time instance has been calculated by a tester and is given in the input file.

7.1.4 Output File Format

GARUS exports the stress test requirements to an output file, whose name is specified in the command line. If no output file name is given by the user, the output is simply printed on the screen. Furthermore, the output file also contains standard GAlib statistics, including the numbers of selections, crossovers, mutations, replacements and genome evaluations since initialization, as well as min, max, mean, and standard deviation of each generation. The main output is the stress test requirements, while GA statistics are just informative values for debugging purposes. The format of stress test requirements in an output file is shown in Figure 31-(a). An example set of stress test requirements is presented in Figure 31-(b), which is generated by GARUS for the input file in Figure 29.

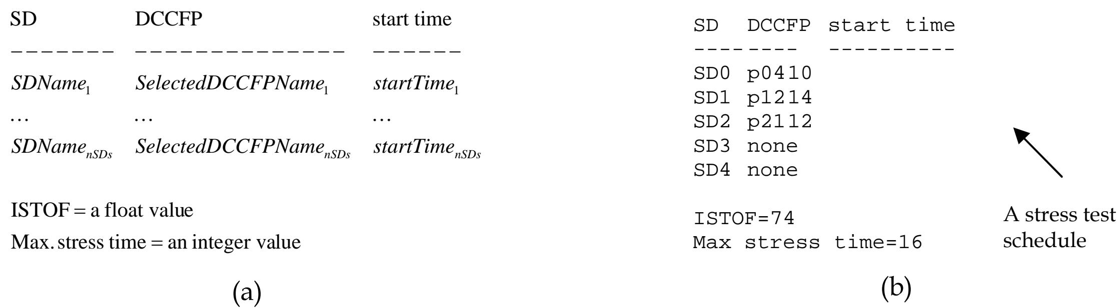


Figure 31-(a): Stress test requirements format in GARUS output file. (b): An example.

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 6.3 for the formalized representation of a stress test requirement. For example, Figure 31-(b) indicates that *p04* of *SD0*, *p12* of *SD1*, and *p21* of *SD2* should be triggered at start times 10, 14 and 12 unit of time, respectively. No DCCFP has been specified to be triggered for *SD3* and *SD4*. This is because a set of stress test requirements corresponds to an ISDS in a SUT, and as shown in Figure 29, the SUT we used for these results has two ISDSs and *SD0*, *SD1* and *SD2* are members of one of them. In other words, triggering all SDs *SD0* …*SD4* is not allowed in this SUT. Note that GARUS never schedules a DCCFP in a start time which is not allowed according to SDs' arrival patterns.

7.2 An Empirical Analysis to Validate Test Requirements Generated by GARUS

Along with a stress test requirement, GARUS also generates a maximum traffic value and a maximum traffic time. The maximum traffic value is in fact the objective function value of the GA's best individual at the completion of the evolution process. The objective function was described in Section 6.5.5, and was referred to as *Instant Stress Test Objective Function (ISTOF)*. The maximum traffic time is the time instant when the maximum traffic happens. For example the ISTOF value and maximum traffic time for the SUT specified by the input file in Figure 29 are 74 (unit of traffic, e.g. KB) and 16 (unit of time, e.g. ms), respectively.

Test requirements generated by GARUS can be validated according to at least six criteria:

1. *Satisfaction of ATSs by start times of DCCFPs in the generated stress test requirements* (Section 7.2.1): As explained in Section 6.5, each chromosome (including the final best chromosome) should satisfy this constraint, i.e., the start times of each DCCFP in the final best chromosome of the GA should be inside the Accepted Time Set (ATS) of its corresponding SD.
2. *Checking ISTOF values* (Section 7.2.2): As a heuristic, GAs do not guarantee to yield optimum results, and checking that the ISTOF value of the final best chromosome is the maximum possible traffic value among all interleavings is a NP-hard problem. It is, therefore, not possible to fully check how

optimal GA results are. However, simple checks can be done to determine if, for example, GARUS has been able to choose the DCCFP with maximum traffic value among all DCCFPs in a SD.

3. *Repeatability of GA results across multiple runs* (Section 7.2.3)*:* It is important to assess how stable and reliable the results of the GA will be. To do so, the GA is executed a large number of times and we assess the variability of the average or best chromosome's fitness value.
4. *Convergence efficiency across generations towards a maximum* (Section 7.2.4): In order to assess the design of the selected mutation and cross-over operators, as well as the chosen chromosome representation, it is useful to look at the speed of convergence towards a maximum fitness plateau [32]. 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 of fitness values. A maximum fitness plateau is reached when the standard deviation of the fitness values equals 0.
5. *Impacts of variations in test model size (scalability of the GA)* - (Section 7.2.6): Assessing how The GA performance and its repeatability are affected with different test model sizes.
6. *Impacts of variations in parameters other than test model size* - (Sections 7.2.7-7.2.9): Assessing how the GA performance and its repeatability are affected when it is applied to different test models with different properties. In the current work, we investigate the impacts of variations in arrival pattern types (Section 7.2.7), arrival pattern parameters (such as periodic arrival pattern period and deviation, and bounded arrival pattern minimum and maximum inter-arrival time values) (Section 7.2.8), and GA maximum search time (Section 7.2.9) on the GA results and on its repeatability aspect, respectively.

Using the above six criteria, we analyze the stress test requirements generated by running GARUS on a set of *experimental* test models, which were designed to test the repeatability and scalability aspects of our GA. We discuss in Section 7.2.5 how we designed the set of the experimental test models, which will be used in the rest of this chapter as a test-bed for our experiments and validations.

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

We check whether the start times of the DCCFPs in the generated stress test requirements satisfy the ATSs of the corresponding SDs. In order to investigate this, we first derive the ATSs of the SDs in the test model corresponding to the input file in Figure 29. Consistent with discussions in Section 5.5.4, they are shown in Figure 32.

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 4 has been depicted in Figure 32. 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, 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 32, 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 6.5.3) and the allowability property of our mutation operator (Section 6.5.6.2). The start time of each DCCFP is always chosen from the ATS of its corresponding SD. This is achieved by building the ATS of each SD according to its type of AP when GARUS initializes a test model. Then, when a random start time is to be chosen for a DCCFP, method *getARandomArrivalTime()*, which is associated with a SD is invoked on an object from a subclass of the abstract class AP. Refer to Figure 26 for details.

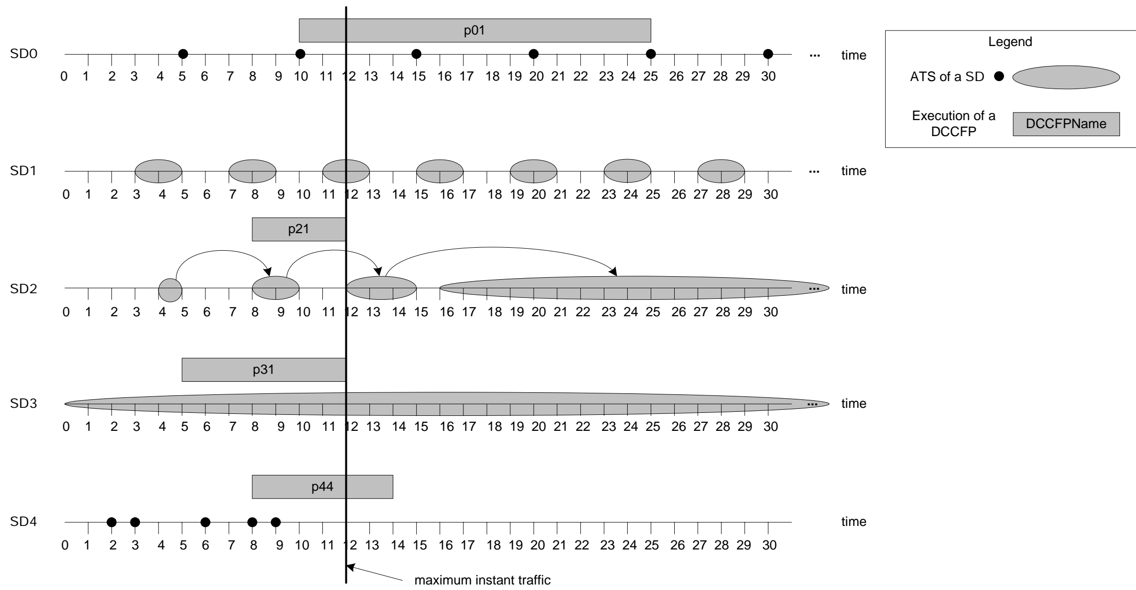


Figure 32-ATSs of the SDs in the TM in Figure 29**, and a stress test schedule generated by GARUS.**

7.2.2 Checking the extent to which ISTOF is maximized

As a test to check if GARUS is able to choose the DCCFP with maximum traffic value among all DCCFPs of a SD, we artificially modify NTUPs of the DCCFPs in the test model of Figure 29 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 33.

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.

```
--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 33-Modified DCCFPs of the test model in Figure 29.

We executed GARUS 10 times with this TM, and the 10 schedules generated by GARUS had the format described in Table 2, where *x* stands for values which changed across different runs. As expected, DCCFPs *p21*, *p31*, and *p42* were present in all 10 stress test schedules, thus suggesting that GARUS selects the correct DCFFPs. On the other hand, different DCCFPs from *SD0* were reported in the output schedules. This can be

explained as *SD0*'s ATS contains specific time points (5, 10, 15, and so on) and *p03* (the modified DCCFP) will therefore not be able to have an effect on the maximum possible instant traffic (at time=16 or 17) since its modified NTUP point is at time=5.

SD	DCCFP	Start Time
SD0	x	x
SD1	none	
SD2	p21	x
SD3	p31	x
SD4	p42	x

ISTOF=1500 or 1520

Max stress time=16 or 17

Table 2-Output format of 10 schedules generated by GARUS.

The reason why *p12* (from *SD1*) is not selected in any of the outputs across different runs is that a set of DCCFPs are generated by GARUS as a stress test schedule only if the SDs corresponding to the DCCFPs belong to one ISDS. The set of SDs {*SD0, SD1, SD2, SD3, SD4*} does not belong to an ISDS. Furthermore, among all ISDSs (*ISDS0*={*SD0*, *SD1*, *SD2*} and *ISDS1*={*SD0*, *SD2*, *SD3*, *SD4*}) of the test model, the maximum instant traffic of *ISDS1* has a larger value than that of *ISDS0,* thus not letting *SD1* (and all of its DCCFPs) play a role in the output stress test schedules.

7.2.3 Repeatability of GA Results across Multiple Runs

Since GAs are heuristics, their performance and outputs can vary across multiple runs. We refer to such a GA property as *repeatability* of GA results. This property is investigated by analyzing maximum ISTOF values as they are the fitness values of chromosomes in our GA. We furthermore analyze maximum stress time values as such a time value denotes the time instance when a stress situation actually occurs.

Figure 34-(a) depicts the distributions of maximum ISTOF and stress time values for 1000 runs of test model corresponding to the input file in Figure 29. From the ISTOF distribution, we can see that the maximum fitness values for most of the runs are between 60 and 72 units of traffic. Descriptive statistics of the fitness values are shown in Table 3. Average and median values are very close, thus indicating that the distribution is almost symmetric.

Min	Max	Average	Median	Standard Deviation
50	92	66.672	65	6.4

Table 3-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 34-(a) that with a few runs a chromosome with a fitness value close to the maximum would likely be identified. Since each run lasts a few seconds, perhaps a few minutes for very large examples, relying on multiple runs to generate a stress test requirement should not be a practical problem.

Corresponding portions of max stress time values for the most frequent maximum ISTOF value (72 units of traffic) have been highlighted in black in Figure 34-(b). As we can see, these maximum stress time values are scattered across the time scale (e.g., from 10 to 60 units of time). This highlights that a single ISTOF value (maximum stress traffic) can happen in different time instances, thus suggesting the search landscape for the GA is rather complex for this type of problem. Thus, a strategy to further explore would be for testing to cover all (or a subset of) such test requirements with maximum ISTOF values in different time instances. Indeed, although their ISTOF value are the same, a SUT's reaction to different test requirements might vary,

since a different DCCFP (and hence set of messages) in a different stress time instance may be trigerred. This might in turn lead to uncovering different RT faults in the SUT.

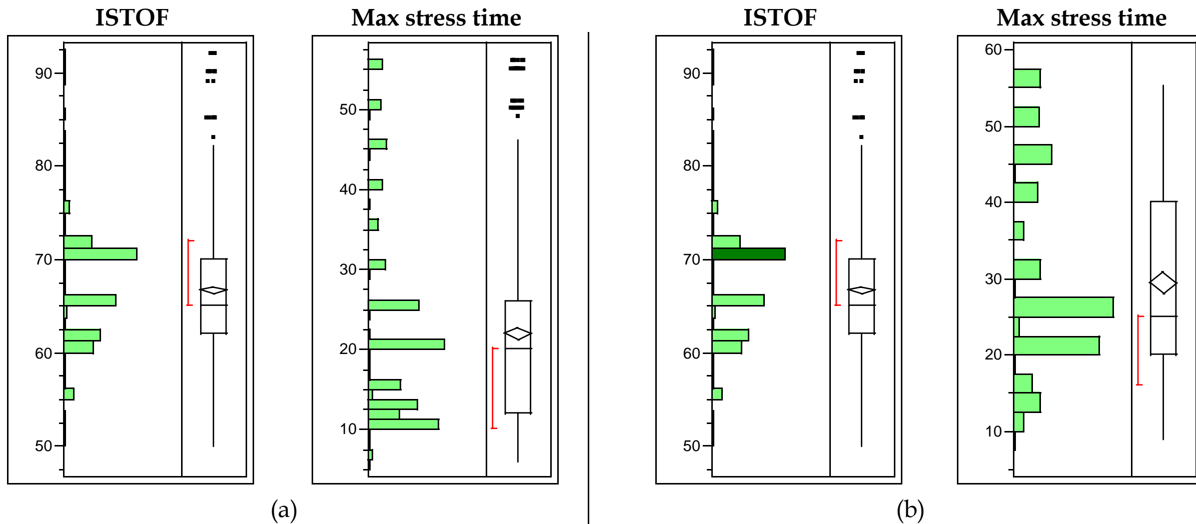


Figure 34-(a): Histogram of maximum ISTOF and stress time values for 1000 runs of test model corresponding to the input file in Figure 29. (b): Corresponding max stress time values for one of the frequent maximum ISTOF values (72 units of traffic).

7.2.4 Convergence Efficiency across Generations

Another interesting property of the GA is the number of generations required to reach a stable maximum fitness plateau. The distribution of these generation numbers over 1000 runs of test model corresponding to the input file in Figure 29 is shown in Figure 35, 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 [26] but is however likely to be dependent on the number and complexity of SDs as well as their ATSs.

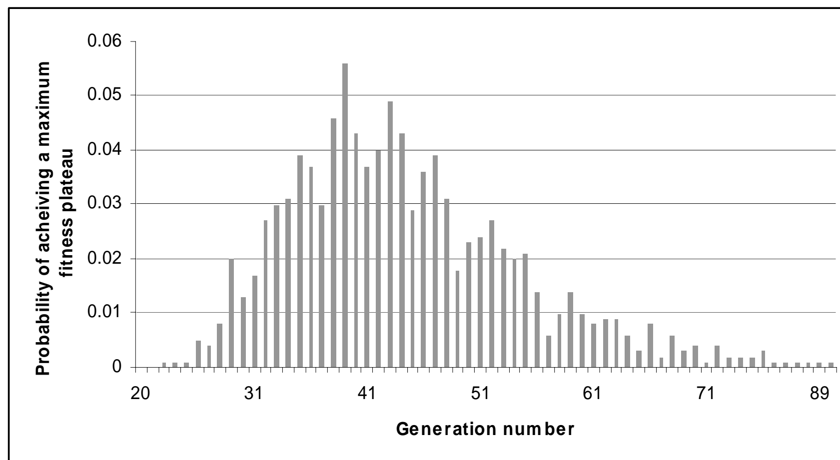


Figure 35-Histogram of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of the test model corresponding to the input file in Figure 29 by GARUS.

From the initial to the final populations in the test model corresponding to the input file in Figure 29, the maximum fitness values typically increase by about 80%, which can be considered a large improvement. So, though we cannot guarantee that a GA found the global maximum, we clearly generate test requirements that will significantly stress the system.

The variability in the objective function and start times as well as detailed information for the first five runs of the test model corresponding to the input file in Figure 29 are reported in Table 4. We can clearly see from the table that in a single run when the generation number increases, the population converges (i.e. deviation value decreases). For example, in the outputs reports in Table 4, the value of the *deviation*[1] column decreases continuously from 8.95 at generation #0 to 0.00 in generation #80. Also notice the convergence of minimum and mean values towards the maximum (ISTOF) values when the generation number increases.

Run #	Generation #	Mean	Max (ISTOF)	Min	Deviation	Best individual		
1	0	36.74	55	30	8.95	SD	DCCFP	start time
	10	44.47	58	38	7.04	----	----	----------
	20	52.46	61	41	6.44	SD0	p05	25
	30	61.14	66	55	5.86	SD1	none	
	40	67.23	71	61	4.90	SD2	p22	21
	50	71.43	79	70	4.21	SD3	p31	23
	60	74.62	85	70	3.01	SD4	p42	2
	70	82.03	88	72	2.95			
	80	90.00	90	90	0.00	ISTOF=90		
	90	90.00	90	90	0.00	Max stress time=30		
	100	90.00	90	90	0.00			
2	0	36.45	58	30	8.82	SD	DCCFP	start time
	10	43.84	60	36	7.13	----	----	----------
	20	51.23	65	41	6.97	SD0	p01	10
	30	57.82	66	50	5.45	SD1	none	
	40	64.70	73	59	5.27	SD2	p21	8
	50	72.52	76	62	5.04	SD3	p31	5
	60	80.50	82	80	3.39	SD4	p44	8
	70	81.34	84	80	3.78			
	80	83.78	86	80	2.58	ISTOF=92		
	90	91.64	92	80	2.05	Max stress time=12		
	100	92.00	92	92	0.00			
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		

[1] *Deviation* of a population in GAlib is defined as the standard deviation of individuals' objective scores [57].

							SD	DCCFP	start time
	100	75.00	75	75	0.00				
4	0	37.03	53	30	8.94		SD	DCCFP	start time
	10	45.37	58	37	8.14		----	----	----------
	20	55.14	60	43	7.21		SD0	p05	15
	30	66.63	69	52	7.08		SD1	none	
	40	73.29	78	70	6.22		SD2	p22	18
	50	79.02	80	72	2.62		SD3	p31	13
	60	80.00	80	80	0.00		SD4	p43	9
	70	80.00	80	80	0.00				
	80	80.00	80	80	0.00		ISTOF=80		
	90	80.00	80	80	0.00		Max stress time=20		
	100	80.00	80	80	0.00				
5	0	37.54	55	30	8.44		SD	DCCFP	start time
	10	45.60	58	39	7.50		----	----	----------
	20	54.09	64	48	6.93		SD0	p05	5
	30	61.67	66	52	6.32		SD1	none	
	40	68.42	69	65	2.52		SD2	p21	12
	50	70.37	71	70	0.78		SD3	p31	11
	60	71.14	72	70	0.99		SD4	p44	6
	70	72.00	72	72	0.00				
	80	72.00	72	72	0.00		ISTOF=72		
	90	72.00	72	72	0.00		Max stress time=10		
	100	72.00	72	72	0.00				

Table 4-Summary of GARUS results for five runs.

7.2.5 Our Strategy for Investigating Variability/Scalability

To assess and validate test requirements generated by GARUS, we design a set of different test models (referred to as *experimental* test models), and execute GARUS with each of them as input. Note that these test models are in the input file format, described in Section 7.1.3. To ensure variability in the experimental test models, a set of experimental test models were designed based on the following criteria:

- Test models with variations in test model sizes (Section 7.2.5.3)
- Test models with variations in SD arrival patterns (Section 7.2.5.4)
- Test models with variations in the GA's maximum search time (Section 7.2.5.5)

Since most of the input files containing the test models are large in size, we do not list them in this article. As an example, the contents of one input file corresponding to one of the test model in this section are shown in Appendix B. We discuss next a set of *variability parameters*, we used in our experiments to incorporate variability in different test models based on the above criteria.

7.2.5.1 Variability Parameters

The variability parameters used in our experiments are listed in Table 5, along with their explanations. The parameters are grouped into three groups corresponding to the above three criteria: (1) Test model size, (2) SD arrival patterns, and (3) Maximum search time.

We have defined eight parameters under the group of 'test model size' to incorporate variability in different sizes perspectives in experimental TMs. Each parameters in this group correspond to a sizes perspective, e.g., number of ISDSs, number of SDs, and minimum number of DCCFPs per SD. For example, a large TM might have many ISDSs (by setting large values for *nISDSs*), while another large TM can have many

DCCFPs per SD (by setting large values for *minnDCCFPs* and *maxnDCCFPs*). Parameters prefixed with *min* and *max* serve as statistical means, which enable us to incorporate statistically-controlled randomness into the sizes of our experimental TMs. For example, we can control the minimum and maximum number of DCCFPs per SD in a TM by *minnDCCFPs* and *maxnDCCFPs* parameters. Such a statistical range for number of SDs per ISDSs, DCCFPs per SD, and NTUPPs per DCCFP also conforms to real-world models, where for example, there are not variant number of DCCFPs per SDs in a TM. Our experimental TMs with variations in TM sizes based on the variability parameters are presented in Section 7.2.5.3.

Parameter Group	Parameter	Parameter Explanation
Test model size	*nISDSs*	# of ISDSs
	nSDs	# of SDs in TM
	minISDSsize	min. # of SDs per ISDS
	maxISDSsize	max. # of SDs per ISDS
	minnDCCFPs	min. # of DCCFPs per SD
	maxnDCCFPs	max. # of DCCFPs per SD
	minnNTUPPs	min. # of NTUPPs per DCCFP
	maxnNTUPPs	min. # of NTUPPs per DCCFP
SD arrival patterns (all parameter values are in time units, except *minnAPirregularPoints* and *maxnAPirregularPoints*)	*APtype*	type of AP
	minAPperiodicPeriod	min. period value
	maxAPperiodicPeriod	max. period value
	minAPperiodicDeviation	min. deviation value
	maxAPperiodicDeviation	max. deviation value
	minAPboundedMinIAT	min. value for min. inter-arrival time
	maxAPboundedMinIAT	max. value for min. inter-arrival time
	minAPboundedMaxIATafterMin	min. distance between *maxIAT* and *minIAT*
	maxAPboundedMaxIATafterMin	max. distance between *maxIAT* and *minIAT*
	minnAPirregularPoints	min. # for irregular points
	maxnAPirregularPoints	max. # for irregular points
	minAPirregularPointsValue	min. time value for irregular points
	maxAPirregularPointsValue	max. time value for irregular points
Maximum search time	*MaximumSearchTime*	GA maximum search time

Table 5- Variability parameters for experimental test models.

The second group of the variability parameters (SD arrival patterns) is designed to incorporate variability in different SD arrival pattern properties of our experimental TMs. All parameter values in this group are in time units, except *minnAPirregularPoints* and *maxnAPirregularPoints*. The first parameter in this group (*APtype*) determines the type of APs to be assigned for the SD of an experimental TM. *APtype* is of type enumeration with possible values of:

- *no_arrival_pattern*: All SDs of the TM will have no APs, i.e., any arrival time in test schedules is accepted for all SDs.
- *periodic*: All SDs of the TM will have periodic APs. Each AP's parameters are set to the four **APperiodic** variability parameters.
- *bounded*: All SDs of the TM will have bounded APs. Each AP's parameters are set to the four **APbounded** variability parameters.
- *irregular*: All SDs of the TM will have irregular APs. Each AP's parameters are set to the four **APirregular** variability parameters.
- *mixed*: Different SDs of a TM have different arrival patterns. The type of AP for a SD is chosen from (*no_arrival_pattern*, *periodic*, *bounded*, or *irregular*) with equal probabilities

Parameters with *APperiodic, APbounded* and *APirregular* substring are specific to periodic, bounded and irregular APs, respectively, and specify the range of values to be set for specific parameters of these AP. For example, *minAPperiodicPeriod*, *maxAPperiodicPeriod*, *minAPperiodicDeviation*, and *maxAPperiodicDeviation* specify min./max. period values, and min./max. deviation values for periodic APs of SDs. If they are set to 5ms, 10 ms, 2ms, and 3ms respectively, the following APs might be generated in an experimental TMs: (*'periodic', (6, ms), (2, ms)*), (*'periodic', (8, ms), (3, ms)*), (*'periodic', (10, ms), (3, ms)*), and (*'periodic', (6, ms), (3, ms)*). Our experimental TMs with variations in SD arrival patterns time based on the parameters in the second group are presented in Section 7.2.5.4.

The third group of the variability parameters (maximum search time) has only one parameter (*MaximumSearchTime*) which is designed to incorporate variability in GA maximum search time (Section 6.5.4) of our experimental TMs. Our experimental TMs with variations in maximum search time based on the *MaximumSearchTime* parameters are presented in Section 7.2.5.5.

7.2.5.2 Random Test Model Generator

To facilitate the generation of experimental TMs based on the variability parameters (Section 7.2.5.1), we developed a random test model generator (*RandTMGen*) in C++. A simplified activity diagram of *RandTMGen* is shown in Figure 36, where the value for variability parameters are provided as input, and an experimental TM is generated (in input file format of Figure 28).

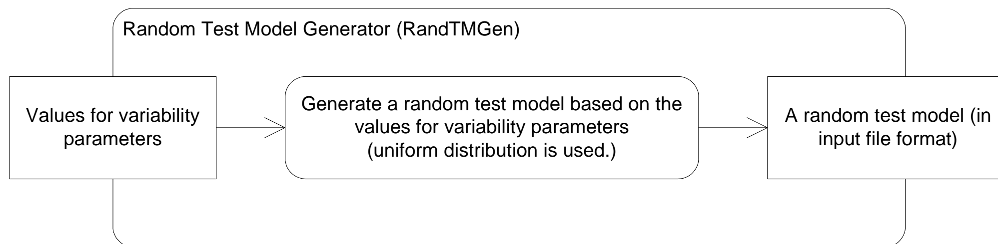


Figure 36-Simplified activity diagram of our random test model generator.

For the pair of parameters specifying min./max. of a feature, e.g. *minnDCCFPs* and *maxnDCCFPs*, *RandTMGen* uses uniform distribution to generate a value in the range of [*minnDCCFPs, maxnDCCFPs*]. This is a design decision, which can be modified easily, i.e., any other distribution can be used to generate a random value in [*minnDCCFPs, maxnDCCFPs*]. As an example on how *RandTMGen* generate a random value in a range, assume *minnDCCFPs*=3 and *maxnDCCFPs*=8. In such a case, all the integer values in the range of [3, 8] are chosen by an equal probability to be set for the number of DCCFPs per a SD in a TM.

In the following three subsections, we describe the parameters provided to *RandTMGen* to generate a set of experimental test models to ensure variability in test models as well as the scalability of GARUS.

7.2.5.3 Test Models with Variations in Sizes

In order to investigate the performance and size scalability of our GA (implemented in GARUS), six test models with different sizes were generated using *RandTMGen* as reported in Table 6.

The SDs of the each of the above test models were assigned *arbitrary (mixed)* arrival patterns (*no_arrival_pattern*, *periodic*, *bounded*, or *irregular*) with equal probabilities. Depending on the selected arrival pattern type for a SD, the AP variability parameters were set as follows.

- Periodic arrival pattern
    - *minAPperiodicPeriod*=5
    - *maxAPperiodicPeriod*=10
    - *minAPperiodicDeviation*=0
    - *maxAPperiodicDeviation*=3

- Bounded arrival pattern
    - *minAPboundedMinIAT*=2
    - *maxAPboundedMinIAT*=4
    - *minAPboundedMaxIATafterMin*=2
    - *maxAPboundedMaxIATafterMin*=5
- Irregular arrival pattern
    - *minnAPirregularPoints*=5
    - *maxnAPirregularPoints*=15
    - *minAPirregularPointsValue*=1
    - *maxAPirregularPointsValue*=30

Test Models / Parameters	*tm1* (small) **Figure 29**	*tm2* (many ISDSs)	*tm3* (many SDs in TM)	*tm4* (many SDs per ISDS)	*tm5* (many DCCFPs per SD)	*tm6* (many NTUPs per DCCFP)
nISDSs	2	**100**	10	10	10	2
nSDs	5	50	**200**	50	10	5
minISDSsize	3	2	2	**20**	2	2
maxISDSsize	4	5	5	**30**	5	5
minnDCCFPs	1	1	2	1	**10**	1
maxnDCCFPs	5	3	5	3	**50**	5
minnNTUPPs	2	1	1	1	1	**50**
maxnNTUPPs	6	10	10	10	10	**100**

Table 6-Experimental test models with different sizes.

Note that the above value for the AP variability parameters are chosen to be typical values, as our main intention in designing TMs *tm1*,…, *tm6* is to experiment our GA's behavior and scalability aspect with reaction to different TM sizes.

7.2.5.4 Test Models with Variations in SD Arrival Patterns

To investigate the effect of variations in SD arrival patterns in generated test requirements by GARUS, 12 test models were generated using *RandTMGen* based on two variation strategies as reported in Table 7. The two AP-related variation strategies we followed when generating TMs in this section were:

1. *Different-AP-Types*: Comparing generated test requirements for TMs with different AP types
2. *Same-AP-Different-Parameters:* Comparing generated test requirements for TMs with a same AP type, but different AP variability parameters (e.g. *maxAPperiodicPeriod*, *maxAPboundedMinIAT* and *minnAPirregularPoints*).

A dash "-" in a cell of Table 7 indicates that the parameter (corresponding to the cell) does not apply to the corresponding TM. For example, all SDs in *tm8* are supposed to be periodic. Therefore, only periodic AP parameters apply to this TM. The following TM-size parameters were used for the test models in Table 7. As these parameters denote, the size of TMs *tm7, …, tm18* have been chosen to be relatively medium (a typical setting) as our main intention in designing these TMs is to experiment our GA's behavior and output results with reaction to variations in SD arrival patterns.

- *nISDSs*=10
- *minISDSsize*=5
- *maxISDSsize*=10
- *nSDs*=20
- *minnDCCFPs*=2
- *maxnDCCFPs*=5
- *minnNTUPPs*=1
- *maxnNTUPPs*=10

	AP-related variation strategies											
	Different-AP-Types					*Same-AP-Different-Parameters*						
Test Models / **Parameters**	*tm7*	*tm8*	*tm9*	*tm10*	*tm11*	*tm12*	*tm13*	*tm14*	*tm15*	*tm16*	*tm17*	*tm18*
APtype	*no_arrival_pattern*	*periodic*	*bounded*	*irregular*	*mixed*	*periodic*	*periodic*	*periodic*	*bounded*	*bounded*	*irregular*	*irregular*
minAPperiodicPeriod	-	5	-	-	5	**5**	5	**20**	-	-	-	-
maxAPperiodicPeriod	-	10	-	-	10	**5**	5	**25**	-	-	-	-
minAPperiodicDeviation	-	0	-	-	0	1	**5**	1	-	-	-	-
maxAPperiodicDeviation	-	3	-	-	3	5	**5**	2	-	-	-	-
minAPboundedMinIAT	-	-	2	-	2	-	-	-	**50**	2	-	-
maxAPboundedMinIAT	-	-	4	-	4	-	-	-	**60**	4	-	-
minAPboundedMaxIATafterMin	-	-	2	-	2	-	-	-	2	**60**	-	-
maxAPboundedMaxIATafterMin	-	-	5	-	5	-	-	-	5	**70**	-	-
minnAPirregularPoints	-	-	-	5	5	-	-	-	-	-	**50**	5
maxnAPirregularPoints	-	-	-	15	15	-	-	-	-	-	**100**	15
minAPirregularPointsValue	-	-	-	1	1	-	-	-	-	-	1	**100**
maxAPirregularPointsValue	-	-	-	30	30	-	-	-	-	-	30	**200**

Table 7-Experimental test models with variations in SD arrival patterns.

We discuss next our rationale of designing each TM based on the two above variation strategies (*Different-APs* and *Same-AP-Different-Parameters*).

- *Different-AP-Types*: The four TMs *tm7… tm11* are intended to investigate the impacts of variations in AP types on repeatability and convergence efficiency of the GA outputs. Each of these TMs have different criteria to assign APs to SDs, as discussed in Section 7.2.5.1.
- *Same-AP-Different-Parameters:* Sets of two or more TMs where SDs in each TM have the same AP, but different AP parameters. For example, all SDs in *tm12*, *tm13* and *tm14* have periodic APs, but have different values for *minAPperiodicPeriod*, *maxAPperiodicPeriod*, *minAPperiodicDeviation* or *maxAPperiodicDeviation*. *tm15* and *tm16* are our pair of experimental test models with bounded APs. TMs *tm17* and *tm18* have SDs with irregular APs. Such a variation strategy will enable us to study the impacts of variations in AP parameters on repeatability and convergence efficiency of the GA outputs.

7.2.5.5 Test Models with Variations in the GA Maximum Search Time

To investigate the effect of variations in maximum search time (Section 6.5.4) on GARUS test requirements by GARUS, we created the following two test models using our random test model generator. As an example, the contents of the input file corresponding to *tm20* is shown in Appendix B.

- *tm19*: SUT components (SDs, DCCFPs, ISDSs, etc.) are the same as *tm1*, but the *MaximumSearchTime* is 5 time units instead of 50 in *tm1*.
- *tm20*: SUT components (SDs, DCCFPs, ISDSs, etc.) are the same as *tm1*, but the *MaximumSearchTime* is 150 time units instead of 50 in *tm1*.

7.2.6 Impacts of Test Model Size (Scalability of the GA)

We investigate how the GA performance and its repeatability are affected when it is applied to test models with different sizes. We study scalability by analyzing the impact of variations in test model size on the following metrics:

- Execution time
- Repeatability of maximum ISTOF values
- Repeatability of maximum stress time values
- Convergence efficiency across generations

7.2.6.1 Impact on Execution Time

To investigate the impact of test model size on the execution time of our GA, the average, minimum and maximum execution times over all the 1000 runs, by running GARUS with *tm1*…*tm6* on an 863MHz Intel Pentium III processor with 512MB DRAM memory are reported in Table 8. As we can see in the table, minimums and maximums of the statistics in Table 8 for each test model are relatively close to the corresponding average value. Therefore, we use the average values to discuss the impacts of test model size on execution time of our GA. To better illustrate the differences, the average values are depicted in Figure 37.

Test Model	Min	Max	Average	Median	Standard Deviation
tm1 (small)	46	125	58	62	11.34
tm2 (many ISDSs)	743	1,150	1,089	375	44.79
tm3 (many SDs in TM)	1,015	2,219	1,240	1,171	199.10
tm4 (many SDs per ISDS)	734	1,641	809	782	97.61
tm5 (many DCCFPs per SD)	141	597	263	250	62.83
tm6 (many NTUPs per DCCFP)	734	1,375	820	797	74.74

Table 8-Execution time statistics of 1000 runs of *tm1*…*tm6*.

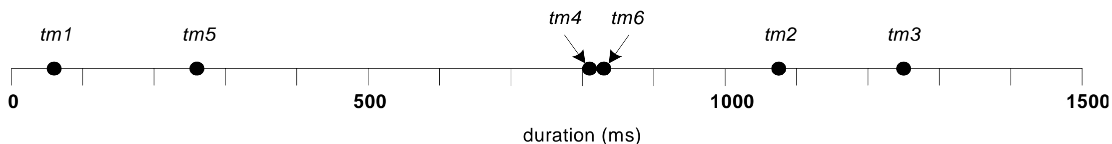


Figure 37-Visualization of the average values in Table 8.

Average duration of the GA run of test model *tm1* (58 ms) is the smallest among all. This is expected since TM *tm1* has the smallest size in terms of test model components (ISDSs, SDs, and DCFFPs). *tm3* has the highest average execution time among the six TM runs. Durations of *tm2*, *tm6*, *tm4*, and *tm5* are next in decreasing order. Based on the above order of execution values, we can make the following observations:

- The execution time of the GA is strongly sensitive to an increase in number of SDs in a TM. The more SDs in a TM, the longer a single run of our GA takes (e.g. *tm3*). This can be explained as the number of genes per chromosome in our GA is the same as the number of SDs in a TM. Thus, as the execution results indicate, the execution time of our GA sharply increases when the number of genes per chromosome increases. Such an increase impacts all functional components of the GA, the two operators (Section 6.5.6) and the fitness evaluator (Section 6.5.5).
- As expected, the execution time of our GA is also highly dependent on the number of ISDSs (e.g. *tm2*). As the number of ISDSs increases, the size of initial population grows, and so does the number of the mutations and crossovers applied in each generation. The number of times the operators are applied is determined by the mutation and crossover rates and the size of initial population.

- The execution time of our GA is also dependent on an increase in number of SDs per ISDS (e.g. *tm4*), as well as an increase in number of NTUPs per DCCFP (e.g. *tm6*). As the number of SDs per ISDS increases, the number of non-null genes per chromosome will increase. This will, in turn, lead to more mutations and crossovers and an increase in computation for the fitness evaluator. Similarly, an increase in number of NTUPs per DCCFP will lead to an increase in fitness computation time.
- The execution time of our GA is not as dependent on an increase in number of DCCFPs per SD (e.g. *tm5*), when compared to other TM components. This can be explained as there will not be any change in chromosome size, nor in the initial population in that case. Even the frequency of mutations and crossovers will not change. For example, as the mutation operator chooses a random DCCFP among all DCCFPs of a SD, there will be no effect in terms of execution time if the number of DCCFPs per SD increases. The small difference between average durations of *tm5* and *tm1* in Figure 37 is due to the fact that *tm5's* number of SDs is slightly more than that of *tm1*.

7.2.6.2 Impact on Repeatability of Maximum ISTOF Values

To investigate the impact of test model size on the repeatability of maximum ISTOF values generated by our GA, Figure 38 depicts the histograms of maximum ISTOF values for 1000 runs on each of the test models *tm1,…,tm6*. The corresponding descriptive statistics are shown in Table 9. Average and median values of all distributions are very close, thus indicating that the distributions are almost symmetric.

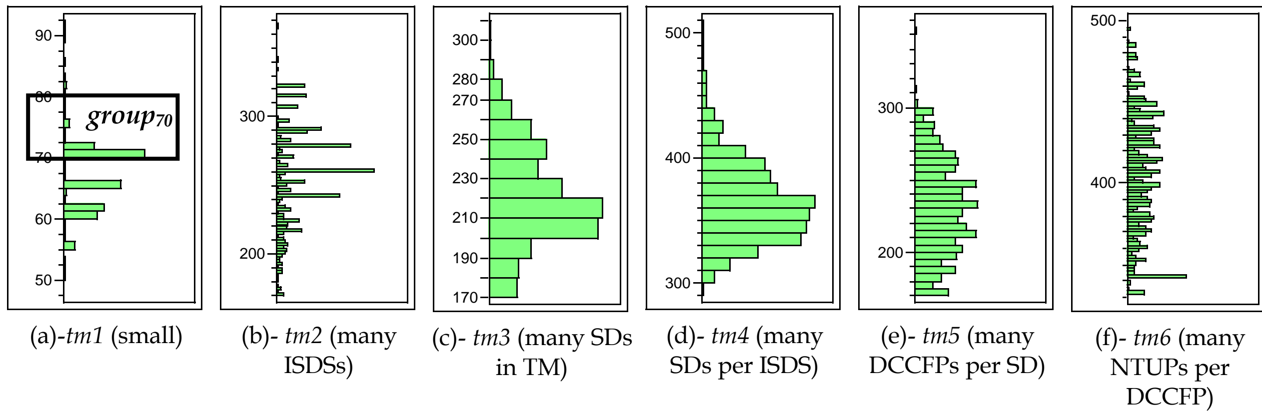
We can see from the ISTOF distributions that the maximum fitness values for most of *tm1* runs, for example, are between 60 and 74 units of traffic. Referring to Figure 38, the variations in fitness values across runs is expected when using genetic algorithms on complex optimization problems. However, though the variation above is not negligible, one would expect based on Figure 38 that with a few runs, a chromosome with a fitness value close to the maximum would likely be identified. To discuss the practical implications of multiple runs of GARUS to get a sub-maximum result (stress test requirement) for *tm1*, we can perform an analysis by using the probability distributions of maximum ISTOF values in the histogram of Figure 38-(a).

In the distributions of maximum ISTOF values for , as it can be easily seen in Figure 38-(a), 1000 runs of GARUS has generated mainly three groups of outputs: values between 70 and 80 units of traffic ($group_{70}$ in Figure 38-(a)), [60,70] and [50,60]. Obviously, the goal of using GARUS is to find stress test requirements which have the highest possible ISTOF values. Thus, the strategy is to run GARUS for multiple times and choose a test requirement with the highest ISTOF value across all runs.

The practical implication of multiple runs to achieve a test requirement with the highest ISTOF value is to predict the minimum number of times GARUS should be executed to yield an output with an ISTOF value in $group_{70}$ in Figure 38-(a). By looking into the raw data of the distribution (a), we found out that 425 (of 1000) values in the histogram belong to $group_{70}$. Thus, in a sample population of 1000 GARUS outputs, the probabilities that an output belongs to $group_{70}$ is $p(group_{70})$=0.425. Thus, to predict the minimum number of times GARUS should be executed to yield an output with an ISTOF value in $group_{70}$, we can use the following probability formula:

$$p(\text{a test requirement with an ISTOF value in } group_{70} \text{ is yielded in a series of } n \text{ runs of GARUS})=$$
$$1-\left(1-p(group_{70})\right)^{n-1}p(group_{70})=1-(0.575)^{n-1}(0.425)$$

The above probability function is depicted in Figure 39, for n=1…15. Figure 39-(b) depicts a zoom-out of the curve in Figure 39-(a) for n=0…40. For values of n higher than 15, the value of the above function is very close to 1, meaning that one can get a stress test requirement with a ISTOF value in the range [70, 80] in 15 runs. Since each run lasts a few seconds (Section 7.2.6.1), perhaps a few minutes for very large examples, relying on multiple runs to generate a stress test requirement should not be a practical problem.



(a)-*tm1* (small) (b)- *tm2* (many ISDSs) (c)- *tm3* (many SDs in TM) (d)- *tm4* (many SDs per ISDS) (e)- *tm5* (many DCCFPs per SD) (f)- *tm6* (many NTUPs per DCCFP)

Figure 38-Histograms of maximum ISTOF values (y-axis) for 1000 runs of each test model. The y-axis values are in traffic units.

Test Model	Min	Max	Average	Median	Standard Deviation
tm1 (small)	65	112	81.66	81	7.0
tm2 (many ISDSs)	171	367	255	260	36.9
tm3 (many SDs in TM)	171	306	220	217	25.2
tm4 (many SDs per ISDS)	299	502	364	360	32.9
tm5 (many DCCFPs per SD)	171	352	230	231	32.2
tm6 (many NTUPs per DCCFP)	333	494	404	406	36.8

Table 9-Descriptive statistics of the maximum ISTOF values for each test model over 1000 runs. Values are in units of data traffic.

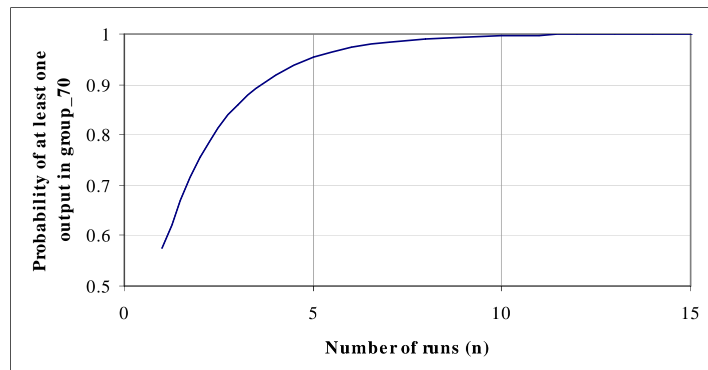


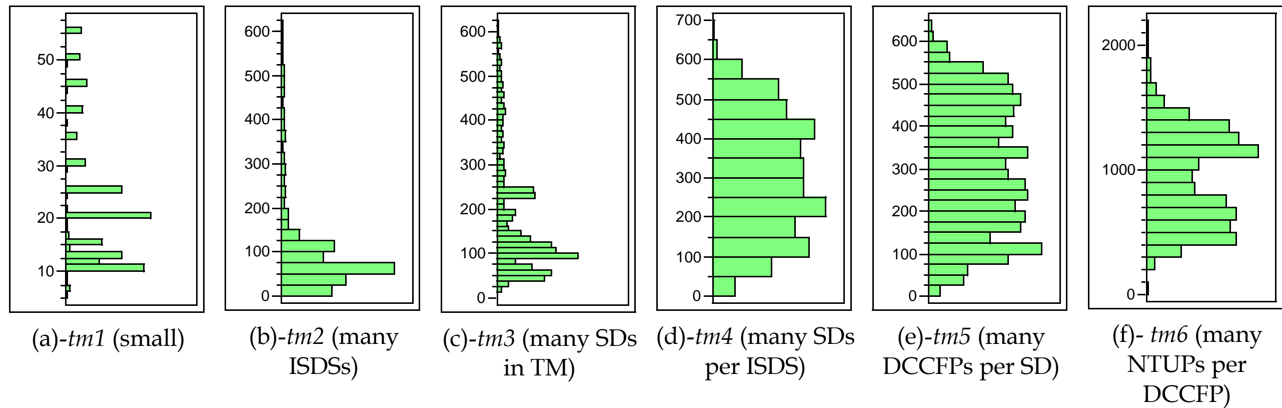
Figure 39- Probability of the event that at least one test requirement with an ISTOF value in $group_{70}$ is yielded in a series of *n* runs of GARUS.

We discuss two main observations based on the results shown in Figure 38.

- In all of the histograms, despite the fact that the results correspond to 1000 runs of different test models which were designed to test the scalability and repeatability properties of our GA, the maximum ISTOF values of the outputs produced by the GA lie in rather small intervals. For example as shown in Figure 38-(f), the maximum ISTOF values generated for *tm6* range in [330…500] units of traffic.
- In terms of scalability, histograms in (b), (c), (d), (e), and (f) suggest that the GA can reach a maximum plateau when the size of a specific component (SD, ISDS, DCCFP, etc) of a given TM is very large.

7.2.6.3 Impact on Repeatability of Maximum Stress Time Values

To investigate the impact of test model size on the repeatability of maximum stress time values generated by our GA, Figure 40 depicts the histograms of maximum stress time values for 1000 runs on each of test models *tm1, …, tm6*. The corresponding descriptive statistics are shown in Table 10. We discuss four main observations based on the results shown in Figure 40.



(a)-*tm1* (small) (b)-*tm2* (many ISDSs) (c)-*tm3* (many SDs in TM) (d)-*tm4* (many SDs per ISDS) (e)-*tm5* (many DCCFPs per SD) (f)- *tm6* (many NTUPs per DCCFP)

Figure 40-Histograms of maximum stress time values for 1000 runs of each test model. The y-axis values are in time units.

Test Model	Min	Max	Average	Median	Standard Deviation
tm1 (small)	6	56	22	20	12
tm2 (many ISDSs)	11	602	103	61	108.8
tm3 (many SDs in TM)	16	618	180	123	135.2
tm4 (many SDs per ISDS)	7	655	301	296	147.0
tm5 (many DCCFPs per SD)	19	626	298	289	149.0
tm6 (many NTUPs per DCCFP)	87	2128	937	933	373

Table 10-Descriptive statistics of the maximum stress time values for each test model over 1000 runs. Values are in time units.

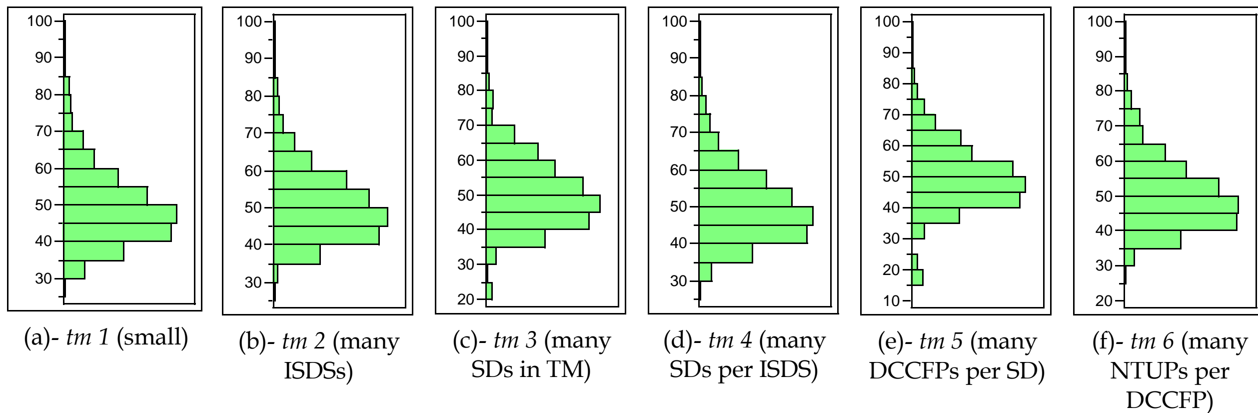
- Average and median values of distributions in (d), (e), and (f) are quite close, thus indicating that the distributions are almost symmetric. Conversely, distributions in (a), (b), and (c) are not symmetric. This reveals that for *tm4* (d), *tm5* (e), and *tm6* (f), the GA might produce maximum stress time values with peak values only at some points. For *tm1* (a), *tm2* (b), and *tm3* (c), the GA generated stress time values with low standard deviation. Furthermore, these distributions are skewed towards the minimums. Thus, one has to run the GA many number of times to get a value close to the maximums.
- Distributions in (d) and (e) are quite flat. This can be explained as for *tm4* (d) *tm5* (e), the number of alternatives to yield a best chromosome by the GA is higher than the others. Furthermore, large sets of best chromosomes can yield different maximum stress time values due to the random start times selected from the legal start times of DCCFPs.
- The standard deviation value for *tm1* (a) is relatively smaller than the other distributions because of the smaller range of maximum ISTOF values in *tm1* results.
- The standard deviation of distribution (f), *tm6*, is much higher than the five others. This can be explained by the higher number of NTUPs per DCCFP (50-100) in *tm6*, compared to the other TMs (1-10). This will, in turn, result in a wider range in the time domain for the GA to search in and find best individuals. As it can be seen in Figure 40, the range of minimum and maximum values in

distribution (f), [87, 2128], is much larger than the other five distributions, which is also resulted from the aforementioned property of *tm6*.

7.2.6.4 Impact on Convergence Efficiency across Generations

Another interesting property of the GA to look at is the number of generations required to reach a stable maximum fitness plateau. To investigate the impact of test model size on convergence efficiency across generations in the GA, Figure 41 depicts the histograms of the generation numbers required to reach a stable maximum fitness plateau over 1000 runs of test models *tm1, …, tm6*. The corresponding descriptive statistics are shown in Table 11. On the average, 49-50 generations of the GA were required to converge to the final result (a stress test requirement) for *tm1, …, tm6*. We therefore see that the TM size does not affect the convergence efficiency across generations in our GA.

The variations around this average in different TMs are limited and no more 100 generations will be required. This number is in line with the experiments reported in the GA literature [26]. As we can see, test model size does not have an impact on convergence efficiency across generations, and the GA is able to reach a stable maximum fitness plateau after about 50 generations on average, independent of test model size.



(a)- *tm 1* (small) (b)- *tm 2* (many ISDSs) (c)- *tm 3* (many SDs in TM) (d)- *tm 4* (many SDs per ISDS) (e)- *tm 5* (many DCCFPs per SD) (f)- *tm 6* (many NTUPs per DCCFP)

Figure 41- Histograms of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of each test model.

Test Model	Min	Max	Average	Median	Standard Deviation
tm1 (small)	27	98	49	47	10.78
tm2 (many ISDSs)	28	96	50	49	10.44
tm3 (many SDs in TM)	23	97	50	49	10.84
tm4 (many SDs per ISDS)	25	98	49	49	10.63
tm5 (many DCCFPs per SD)	17	96	49	49	11.77
tm6 (many NTUPs per DCCFP)	27	98	50	48	10.74

Table 11-Minimum, maximum and average values of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of each test model.

7.2.7 Impacts of Arrival Pattern Types

The impact of variations in arrival pattern types are investigated by running GARUS with test models *tm7…tm11*. The results are reported in the following four subsections.

- Impact on Execution Time
- Impact on Repeatability of Maximum ISTOF Values
- Impact on Repeatability of Maximum Stress Time Values

- Impact on Convergence Efficiency across Generations

7.2.7.1 Impact on Execution Time

We computed the average, minimum and maximum execution times over all the 1000 runs, by running GARUS with test models *tm7…tm11* on an 863MHz Intel Pentium III processor with 512MB DRAM memory (Table 12). Minimums and maximums of the statistics in Table 12 for each test model run are relatively close to the corresponding average value. Therefore, we use the average values to discuss the impacts of variations in arrival patterns on execution time. To better illustrate the differences, the average values are depicted in Figure 42.

Recall from Section 7.2.5.4 that test models *tm7…tm11* have been designed such that they all have the same number of SDs, CCFPs, and NTUPPs (same TM size). Based on the values depicted in Figure 42, the average execution times of the test models *tm7…tm11* with the same test model size, but different arrival patterns for SDs, are relatively close to each other (within 100 ms). This indicates that execution time is not strongly dependent on SD arrival patterns in a test model. Furthermore, as we discuss below, the difference in execution times are mainly due to the implementation details of a method of class *AP* in GARUS.

Test Model	Min	Max	Average	Median	Standard Deviation
tm7 (all SDs with no arrival patterns)	234	593	296	281	51.27
tm8 (all SDs with periodic arrival patterns)	234	625	290	266	50.47
tm9 (all SDs with bounded arrival patterns)	250	594	295	281	53.78
tm10 (all SDs with irregular arrival patterns)	156	344	186	172	29.12
tm11 (SDs with arbitrary arrival patterns)	217	245	231	224	61.23

Table 12-Execution time statistics of 1000 runs of *tm7…tm11*.

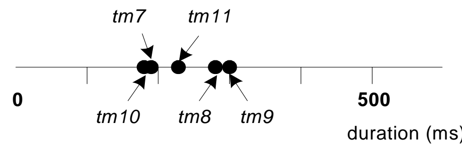


Figure 42-Visualization of the average values in Table 12.

The execution times of two of these test models (*tm8* and *tm9*), are slightly higher than those of *tm7* and *tm10*. The difference between the two TM groups (*tm8* and *tm9* versus *tm7* and *tm10*) can be explained by an implementation detail of GARUS. Function `getARandomArrivalTime`, a member function of class *AP* (Figure 26), is overridden in each of *AP*'s subclasses. The time complexity of this function in *noAP* and *irregularAP* classes is *O(1)*, i.e., choosing a random value from a range or an array, respectively. However, the implementation of the function in *periodicAP* and *boundedAP* classes required some extra considerations (related to the ATSs of periodic and bounded APs), and thus the time complexities of the function are not constant anymore, but dependent on the specific arrival pattern parameters.

The execution time of *tm11*, in which each SD can have an arbitrary arrival pattern, is placed somehow close to the average value of the other four TMs (*tm7*, *tm8, tm9,* and *tm10*). This is as predicted since the APs of SDs in *tm11* are a mix of APs in the other four, thus leading to such an impact in its average execution time.

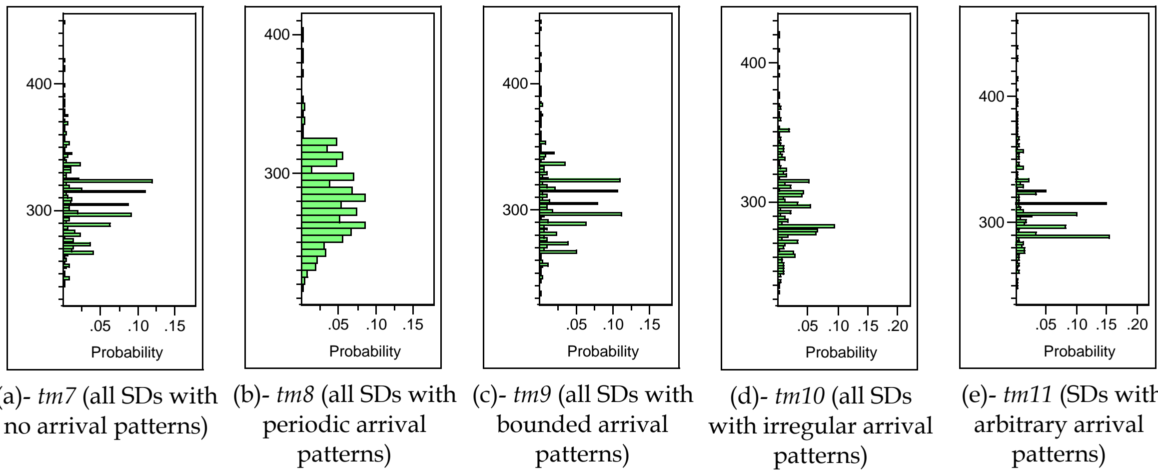
7.2.7.2 Impact on Repeatability of Maximum ISTOF Values

To investigate the impact of variations in arrival pattern types on the repeatability of maximum ISTOF values, Figure 43 depicts the histograms of maximum ISTOF values for 1000 runs on each of the test models *tm7,…, tm11*. The corresponding descriptive statistics are shown in Table 13.

As we can see, there are no big differences among the five distributions. The histogram of maximum ISTOF values for *tm8* is the only noticeable difference, in which the distribution is flatter than the four others (with

more peaks and valleys). This is perhaps due to the specific ATS properties of periodic arrival patterns (the arrival pattern type of SDs in *tm8*).

Another observation is that the histograms in Figure 43-(a) and Figure 43-(c) are quite similar. Recall from Section 5.5.4 that the ATS of a bounded arrival pattern covers the entire time domain except few time intervals close to zero. Therefore, if the common unconstrained time intervals of a set of bounded arrival patterns are considered, they resemble a set of SD with no arrival patterns. The effect of such a property, thus, shows itself in the two histograms.



(a)- *tm7* (all SDs with no arrival patterns) (b)- *tm8* (all SDs with periodic arrival patterns) (c)- *tm9* (all SDs with bounded arrival patterns) (d)- *tm10* (all SDs with irregular arrival patterns) (e)- *tm11* (SDs with arbitrary arrival patterns)

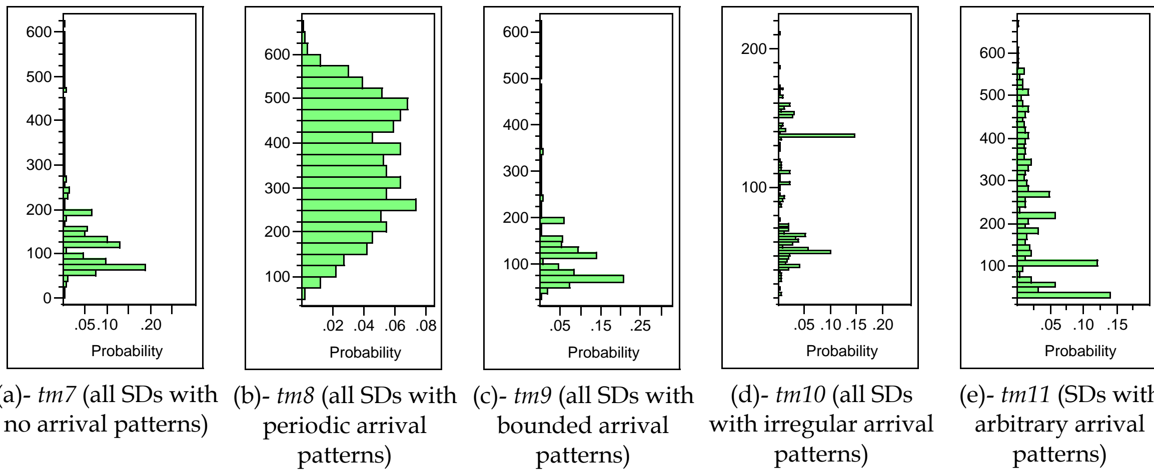
Figure 43-Histograms of maximum ISTOF values for 1000 runs of each test model. The y-axis values are in traffic units.

Test Model	Min	Max	Average	Median	Standard Deviation
tm7 (all SDs with no arrival patterns)	240	448	305	304	27.7
tm8 (all SDs with periodic arrival patterns)	216	404	279	279	27.0
tm9 (all SDs with bounded arrival patterns)	232	448	306	304	28.8
tm10 (all SDs with irregular arrival patterns)	234	420	294	289	28.41
tm11 (SDs with arbitrary arrival patterns)	248	459	309	307	25.72

Table 13-Descriptive statistics of the maximum ISTOF values for each test model over 1000 runs. Values are in units of data traffic.

7.2.7.3 Impact on Repeatability of Maximum Stress Time Values

To investigate the impact of variations in arrival pattern types on the repeatability of maximum stress time values, Figure 44 depicts the histograms of maximum stress time values for 1000 runs on each of test models *tm7*,…, *tm11*. The corresponding descriptive statistics are shown in Table 14.



(a)- *tm7* (all SDs with no arrival patterns) (b)- *tm8* (all SDs with periodic arrival patterns) (c)- *tm9* (all SDs with bounded arrival patterns) (d)- *tm10* (all SDs with irregular arrival patterns) (e)- *tm11* (SDs with arbitrary arrival patterns)

Figure 44-Histograms of maximum stress time values for 1000 runs of each test model. The y-axis values are in time units.

Test Model	**Min**	**Max**	**Average**	**Median**	**Standard Deviation**
tm7 (all SDs with no arrival patterns)	12	622	140	119	105.7
tm8 (all SDs with periodic arrival patterns)	58	655	347	346	129.8
tm9 (all SDs with bounded arrival patterns)	33	618	137	118	105.0
tm10 (all SDs with irregular arrival patterns)	22	211	89	66	43.0
tm11 (SDs with arbitrary arrival patterns)	29	669	214	184	157.04

Table 14-Descriptive statistics of the maximum stress time values for each test model over 1000 runs. Values are in time units.

We attempt below to interpret the distributions shown in Figure 44.

- Distributions (a) and (c) are skewed towards their minimum values. For example, the mode of (a) is around 70 units of traffic which is closer to 0 (the minimum) than 620 (the maximum). This can be explained based on the early start times of NTUPs in DCCFPs of the fittest GA individual generated for *tm7* and *tm9*. Since the ATS of *tm7* is unconstrained and the one for *tm9* has only few constrained ATIs, the GA chooses the common start times of maximum stressing DCCFPs as early as possible.
- The distribution in (b), corresponding to *tm8,* is flatter than the others. This can be explained based on the specific ATS properties of periodic arrival patterns (the arrival pattern type of SDs in *tm8*). The intersection of several periodic ATSs will be a *discrete* unbounded ATS (refer to Figure 45 for an example). Therefore, given a TM with periodic SDs, the GA might converge to any of the common ATPs in the intersection of all periodic ATSs.

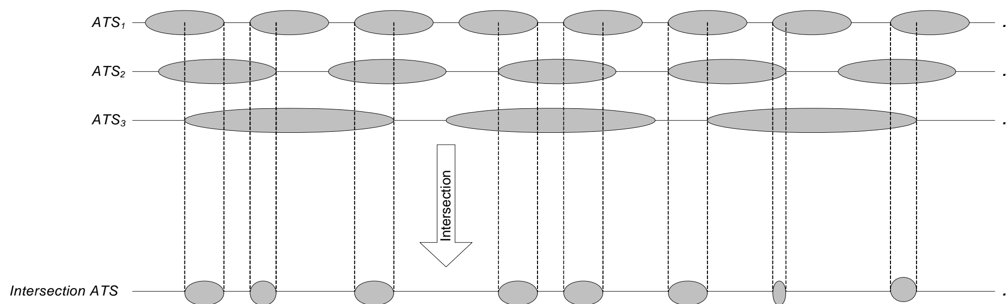
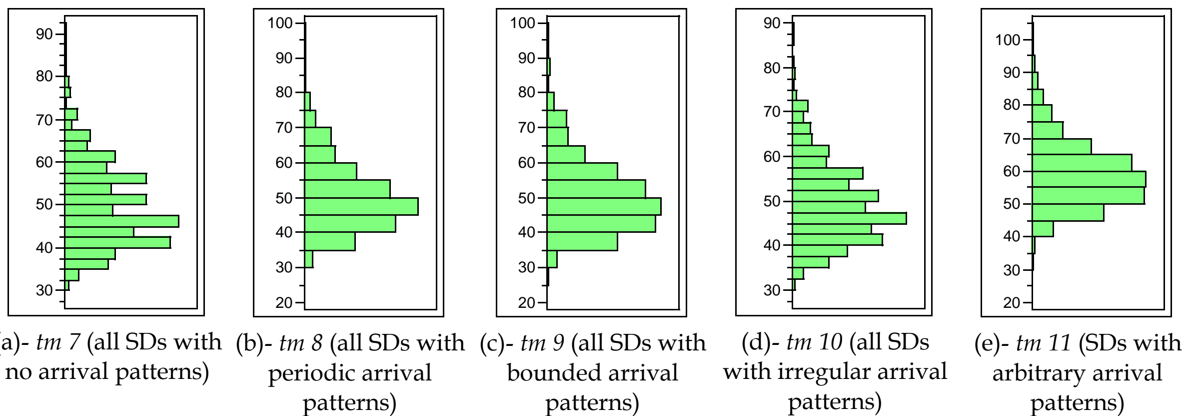


Figure 45- The intersection of several periodic ATSs is a *discrete* unbounded ATS.

- Another observation is that the histograms in (a) and (c) are quite similar. Recall from Section 5.5.4 that the ATS of a bounded arrival pattern covers the entire time domain except few time intervals close to zero. Therefore, if the common unconstrained time intervals of a set of bounded arrival patterns are considered, they resemble a set of SD with no arrival patterns. Such similarity is visible in the two histograms.

7.2.7.4 Impact on Convergence Efficiency across Generations

Regarding the impact of arrival patterns on convergence efficiency across generations in the GA, Figure 46 depicts the histograms of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of test models *tm7,…, tm11*. The corresponding descriptive statistics are shown in Table 15. It is interesting to see that, on average, 49-50 generations were required to converge to the final result (a stress test requirement) across all TMs: *tm7,…, tm11*.



(a)- *tm 7* (all SDs with no arrival patterns) (b)- *tm 8* (all SDs with periodic arrival patterns) (c)- *tm 9* (all SDs with bounded arrival patterns) (d)- *tm 10* (all SDs with irregular arrival patterns) (e)- *tm 11* (SDs with arbitrary arrival patterns)

Figure 46- Histograms of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of each test model.

Test Model	Min	Max	Average	Median	Standard Deviation
tm7 (all SDs with no arrival patterns)	30	90	50	48	9.97
tm8 (all SDs with periodic arrival patterns)	28	97	50	49	10.36
tm9 (all SDs with bounded arrival patterns)	30	98	50	49	10.86
tm10 (all SDs with irregular arrival patterns)	27	88	49	48	9.35
tm11 (SDs with arbitrary arrival patterns)	28	96	58	57	11.58

Table 15-Minimum, maximum and average values of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of each test model.

The standard deviations variations of the distributions are limited and no more 100 generations are required in all cases. Therefore, variations in arrival pattern types do not have a significant impact on convergence efficiency across generations, and the GA is able to reach a stable maximum fitness plateau after about 50 generations on average, independent of any arrival pattern types.

7.2.8 Impacts of Arrival Pattern Parameters

The impact of variations in arrival pattern parameters were investigated by running GARUS on experimental test models in which all SDs were periodic (*tm8, tm12, tm13,* and *tm14*), bounded (*tm9, tm15* and *tm16,* or irregular (*tm10, tm17* and *tm18*) (Table 7). The combinations of TMs to investigate in this section were chosen to study the impacts of arrival pattern *parameters* such the period and deviations values of a periodic AP, and minimum/maximum inter-arrival values of a bounded AP. The results are reported in the following four subsections.

- Impact on Execution Time

- Impact on Repeatability of Maximum ISTOF Values
- Impact on Repeatability of Maximum Stress Time Values
- Impact on Convergence Efficiency across Generations

7.2.8.1 Impact on Execution Time

We computed the average, minimum and maximum execution times over all the 1000 runs, by running GARUS on test models *tm8, tm9, tm10, tm12,…tm18* on an 863MHz Intel Pentium III processor with 512MB DRAM memory (Table 16).
Minimums and maximums of the statistics in Table 16 for each test model run are relatively close to the corresponding average value. Therefore, we use the average values to discuss the impacts of variations in arrival patterns on execution time. Based on the execution values in Table 16, we can make the following observations:

- The GA execution takes longer time for higher numbers of periodic ATIs in a specific maximum search time. The distribution mean and median values for execution times of *tm12* (AP period value=5) and *tm13* (AP period value=5) are higher than those of *tm8* (AP period values from 5 to 10) and *tm14* (AP period value from 20 to 25). This is explained by an implementation detail of GARUS. Function `getARandomArrivalTime`, a member function of class *AP* (**Figure 26**), is overridden in each of *AP*'s subclasses. The explanations can be made similar to the discussions in Section 7.2.7.1. As the period value of an AP increases, the number of periodic ATIs in a specific maximum search time decreases. This, in turn, reduces the GA's execution time.
- Increasing the minimum inter-arrival time range (from [2,4] in *tm9* to [50,60] in *tm15*) in bounded APs has increased the average (from 590 in *tm9* to 667 in *tm15*) and median execution times (from 562 in *tm9* to 657 in *tm15*). This is explained by formula $k = \lceil minIAT/(maxIAT - minIAT) \rceil$ (proved in Appendix C), where *k* is the number of ATIs for a bounded AP. Due to the implementation detail in member function `getARandomArrivalTime` of class *AP* (Section 7.2.7.1), increasing the number of ATIs in a bounded AP will increase execution time. Note that the denominator value in the above formula is the same in both *tm9* and *tm15*, by using the *minAPboundedMaxIATafterMin* parameter (Table 5), while the nominator value is changed.
- Increasing the different between minimum and maximum inter-arrival time range (from [2,5] in *tm9* to [60,70] in *tm16*) in bounded APs has decreased the average (from 667 in *tm9*to 503 in *tm16*) and median execution times (from 657 in *tm9*to 494 in *tm16*). This is explained again by the above formula and the implementation detail in member function `getARandomArrivalTime` of class *AP* (Section 7.2.7.1), whereas decreasing the number of ATIs in a bounded AP will decrease execution time. Note that the nominator value in the above formula is the same in both *tm9* and *tm16*, while the denominator value is changed.

Test Model Group	Test Model	Min	Max	Average	Median	Standard Deviation
periodic	*tm8*	468	1,250	580	532	100.94
	tm12	625	1,359	746	703	123.13
	tm13	640	1,547	758	703	125.11
	tm14	393	1,109	519	556	101.55
bounded	*tm9*	500	1,188	590	562	50.56
	tm15	625	890	667	657	22.92
	tm16	447	853	503	494	35.50
irregular	*tm10*	312	688	372	344	58.24
	tm17	453	1235	582	531	97.77
	tm18	500	984	557	532	64.4

Table 16-Execution time statistics of 1000 runs of *tm8, tm9, tm10, tm12,…tm18*.

7.2.8.2 Impact on Repeatability of Maximum ISTOF Values

To investigate the impact of variations in arrival pattern parameters on the repeatability of maximum ISTOF values, Figure 47 depicts the histograms of maximum ISTOF values for 1000 runs on each of the test models *tm8, tm9, tm10, tm12,…tm18*. The corresponding descriptive statistics are shown in Table 17. We discuss three main observations based on the results shown in Figure 47.

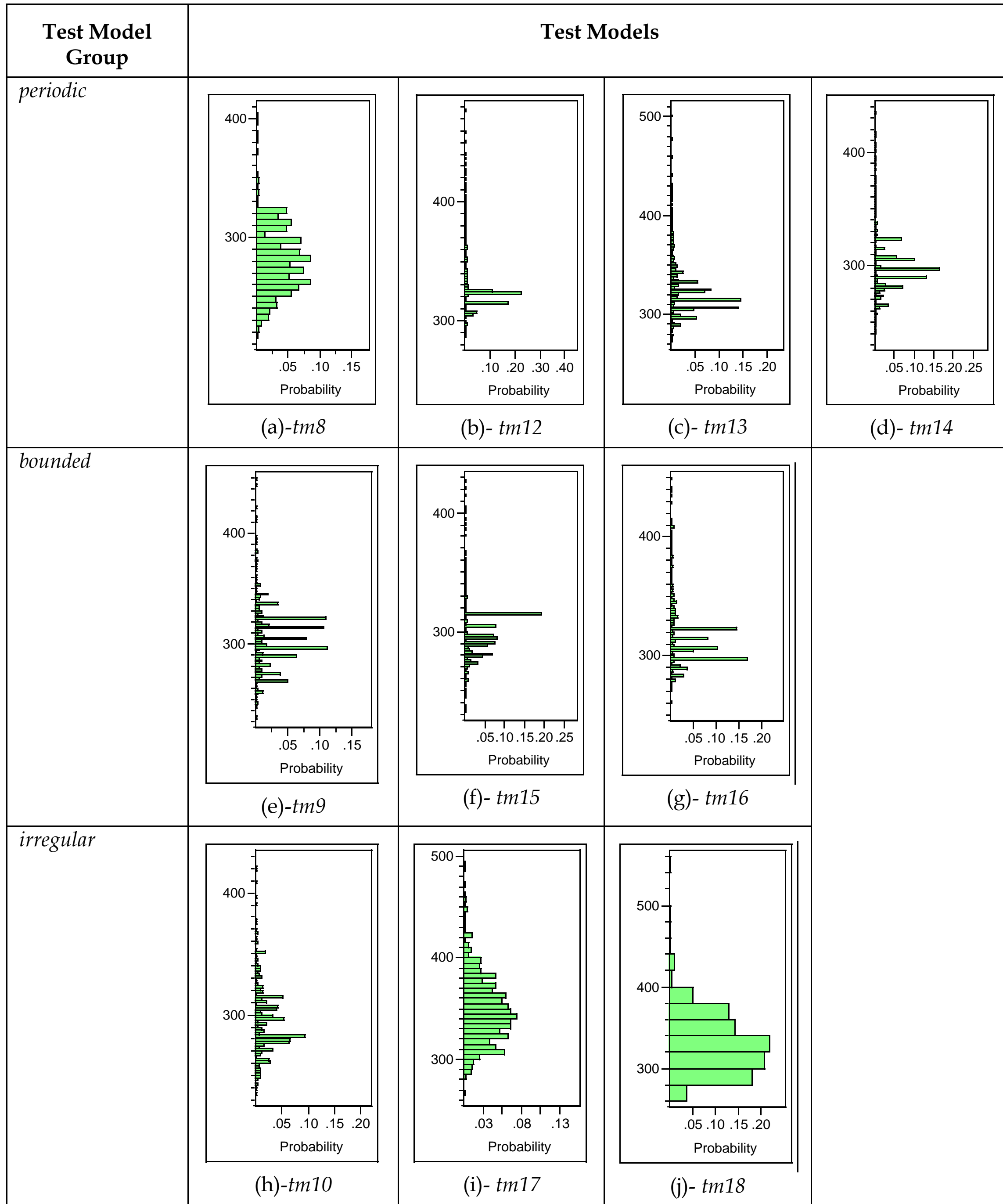


Figure 47-Histograms of maximum ISTOF values for 1000 runs of each test model. The y-axis values are in traffic units.

Test Model Group	Test Model	Min	Max	Average	Median	Standard Deviation
periodic	*tm8*	216	404	279	279	27.0
	tm12	273	476	323	317	26.65
	tm13	287	500	331	323	27.8
	tm14	240	435	298	296	25.21
bounded	*tm9*	232	448	306	304	28.8
	tm15	249	507	306	297	22.29
	tm16	260	449	317	310	28.34
irregular	*tm10*	234	420	294	289	28.41
	tm17	268	490	348	345	33.58
	tm18	262	556	329	330	35.01

Table 17-Descriptive statistics of the distributions in Figure 47.

- Among TMs with all periodic APs (*tm8, tm12, tm13,* and *tm14*), *tm13* has the highest maximum ISTOF value (500). This is because the APs in *tm13* have both period and deviation values of 5, which means that the corresponding ATSs are virtually unconstrained, i.e., all time instants are accepted. This, in turn, lets the GA search the entire time domain and find the best possible stress test schedule. The ATSs of other three TMs (*tm8, tm12,* and *tm14*) are constrained and do not include any time instant in the time domain.
- Increasing the number of arrival points (from [5,15] in *tm10* to [50,100] in *tm17*) in irregular APs has increased the highest maximum ISTOF value (from 420 in *tm10* to 490 in *tm17*). This is because as the number of arrival points increases, the number of possible stress test requirements increases, and so does the chances of finding a stress test requirement with the highest stress value.
- Increasing the location of arrival points (from [1,30] in *tm10* to [100,200] in *tm18*) in irregular APs does not have a significant impact on the distribution of maximum ISTOF values. This is because such a change will only shift (in time domain) the time instant when stress traffic will be entailed (maximum stress time).

7.2.8.3 Impact on Repeatability of Maximum Stress Time Values

To investigate the impact of variations in arrival pattern types on the repeatability of maximum stress time values, Figure 48 depicts the histograms of maximum stress time values for 1000 runs on each of test models *tm8, tm9, tm10, tm12,…tm18*. The corresponding descriptive statistics are shown in Table 18. We discuss three main observations based on the results shown in Figure 48.

- The difference between the mode and other values of the distribution (d) is slightly more than such a difference in distributions (a), (b) or (c). This can be explained as *tm14*, corresponding to the distribution (d), has relatively high period values ([20-25]) compared to the other three TMs. This, in turn, affects the GA by providing less chances of finding overlaps between different ATSs in a TM. Because of this, the number of such overlaps are maximized in fewer instants in the time domain in *tm14* compared to the other three.
- Among TMs with bounded AP, distributions (e) and (g) are skewed towards their minimum values, while the values in (f) are distributed almost evenly across the distribution's range, i.e., the distribution is not skewed neither towards its minimum nor it maximum value. This can be explained by the different values of the *Unbounded Range Starting Point (URSP),* Section 6.5.4., in the bounded APs of the above three TMs. As shown in Appendix C, the URSP of a bounded AP can be calculated by: $URSP = \lceil minIAT/(maxIAT - minIAT) \rceil . minIAT$. Therefore, the URSP value of *tm15* is higher than those of *tm9* and *tm16*, since the range of *minIAT* in *tm15* is higher than those in *tm9* and *tm16*, while the denominator of the above formula are in the same range. For example, considering a bounded AP in *tm15* with *minIAT=55* and *maxIAT=58*, the URSP will be 1045 units of time. This value is even higher than the maximum search time (500 units of time) set in all the above three TMs.

Therefore, there will not be any unconstrained ATI in the ATSs of *tm15*, and thus the GA will not be able to do an unconstrained search for stress test requirements in time domain. Such a search is possible in *tm9* and *tm16* because their URSPs will be inside the specified maximum search time. For example, one example URSP for a SD AP in *tm9* with *minIAT=3* and *maxIAT=5* is *6* units of time, which is < 500.

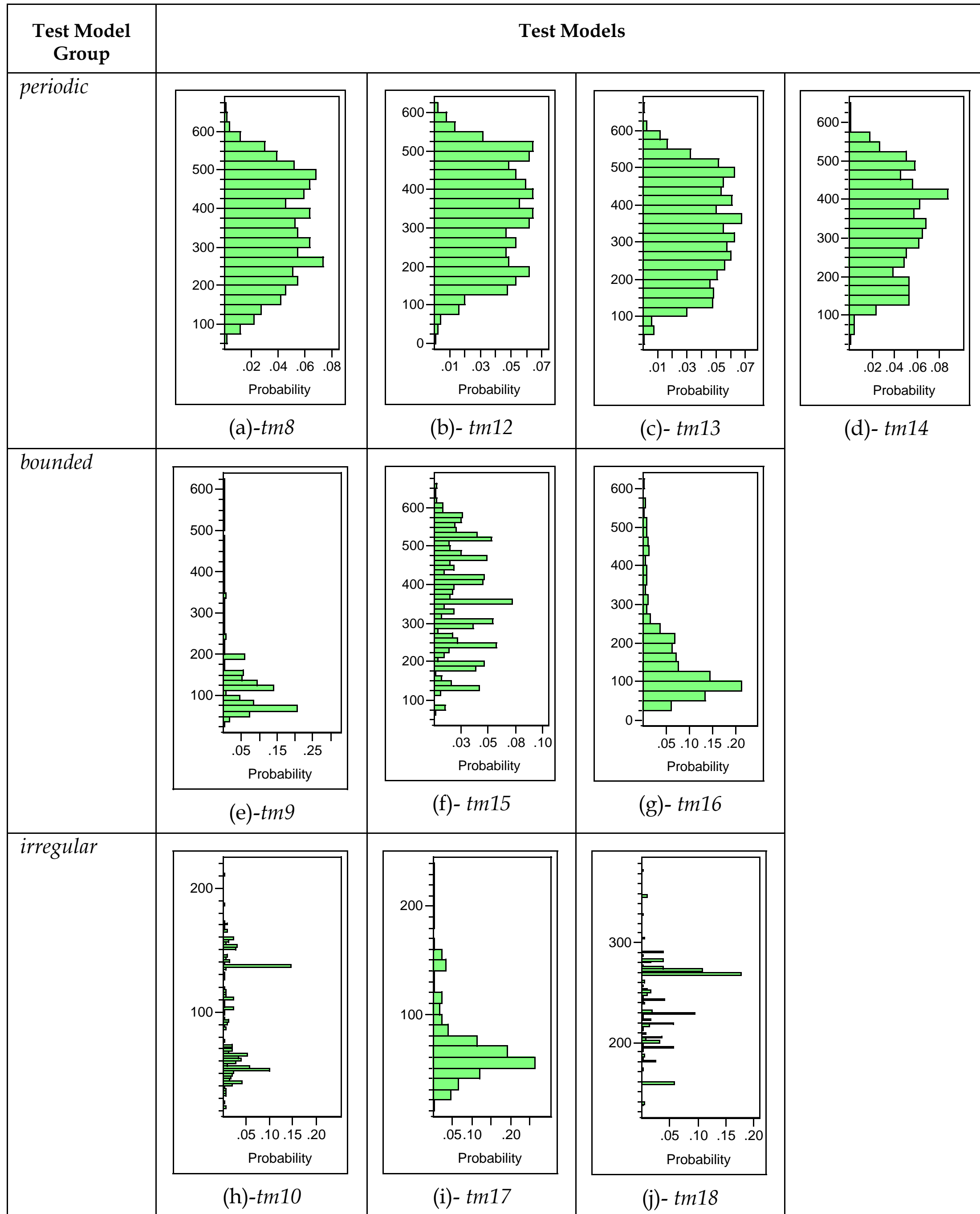


Figure 48- Histograms of maximum stress time values for 1000 runs of each test model. The y-axis values are in time units

Test Model Group	Test Model	Min	Max	Average	Median	Standard Deviation
periodic	*tm8*	58	655	347	346	129.8
	tm12	18	613	332	337	131.54
	tm13	49	664	334	334	131.15
	tm14	41	631	333	337	125.3
bounded	*tm9*	33	618	137	118	105.0
	tm15	72	657	366	362	139.32
	tm16	26	615	151	115	109.11
irregular	*tm10*	22	211	89	66	43.0
	tm17	14	235	67	59	32.54
	tm18	136	372	239	242	40.39

Table 18-Descriptive statistics of the distributions in Figure 48.

- As discussed in the previous subsection, increasing the location of arrival points (from [1,30] in *tm10* to [100,200] in *tm18*) in irregular APs does not have a significant impact on the distribution of maximum ISTOF values. However, as we can see by comparing distributions (h), (i) and (j), such an increase shifts (in time domain) the time instant when stress traffic will be entailed (maximum stress time). As we can see, both average and median values in distribution (j), corresponding to *tm18*, are higher than those in distribution (h), corresponding to *tm10*.

7.2.8.4 Impact on Convergence Efficiency across Generations

Regarding the impact of arrival patterns on convergence efficiency across generations in the GA, Figure 49 depicts the histograms of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of test models *tm8, tm9, tm10, tm12,…tm18*. The corresponding descriptive statistics are shown in Table 19. On average, 49-59 generations were required to converge to the final result (a stress test requirement) across all TMs: *tm8, tm9, tm10, tm12,…tm18*. No more 100 generations are required in all cases.

As we can see, variations in arrival pattern parameters has a slight impact on convergence efficiency across generations. We discuss such impacts individually for each of the three test model groups:

- TMs with periodic APs (*tm8, tm12, tm13,* and *tm14*): The results for *tm13* converge relatively faster than the other three (almost the same), as the minimum, average and medians denote. This can be explained as the APs in *tm13* are periodic with both period and deviation values of 5. This AP resembles to a unconstrained ATS, in which all time instant are accepted. Therefore, the GA will have better chances to find “fit” individuals earlier. In the other three, more applications of the GA operators are required to converge the population.
- TMs with bounded APs (*tm9, tm15* and *tm16*): As the corresponding minimum, average and medians denote, the results for *tm16* converge relatively faster than the other two (almost the same). This can be explained by the URSP formula for bounded APs ($URSP = \lceil minIAT / (maxIAT - minIAT) \rceil . minIAT$). By calculating the range of URSPs based on the given ranges of *minIAT* and *maxIAT*, *tm16* has the lowest value of URSP among the above set of TMs with bounded APs (*tm9, tm15* and *tm16*). By having a smaller value for URSP, our GA can search the time domain (up to maximum search time) to a greater extent, thus, yielding a faster convergence.
- TMs with irregular APs (*tm10, tm17* and *tm18*): As the corresponding minimum, average and medians denote, the results for *tm17* converge relatively faster than the other two. This can be explained by the higher number of irregular arrival points in *tm17,* i.e., the range of [50,100] compared to [5,10] in *tm10*and *tm18*. By having a higher number of irregular arrival points, our GA can search the time domain (up to maximum search time) to a greater extent, thus, yielding a faster convergence.

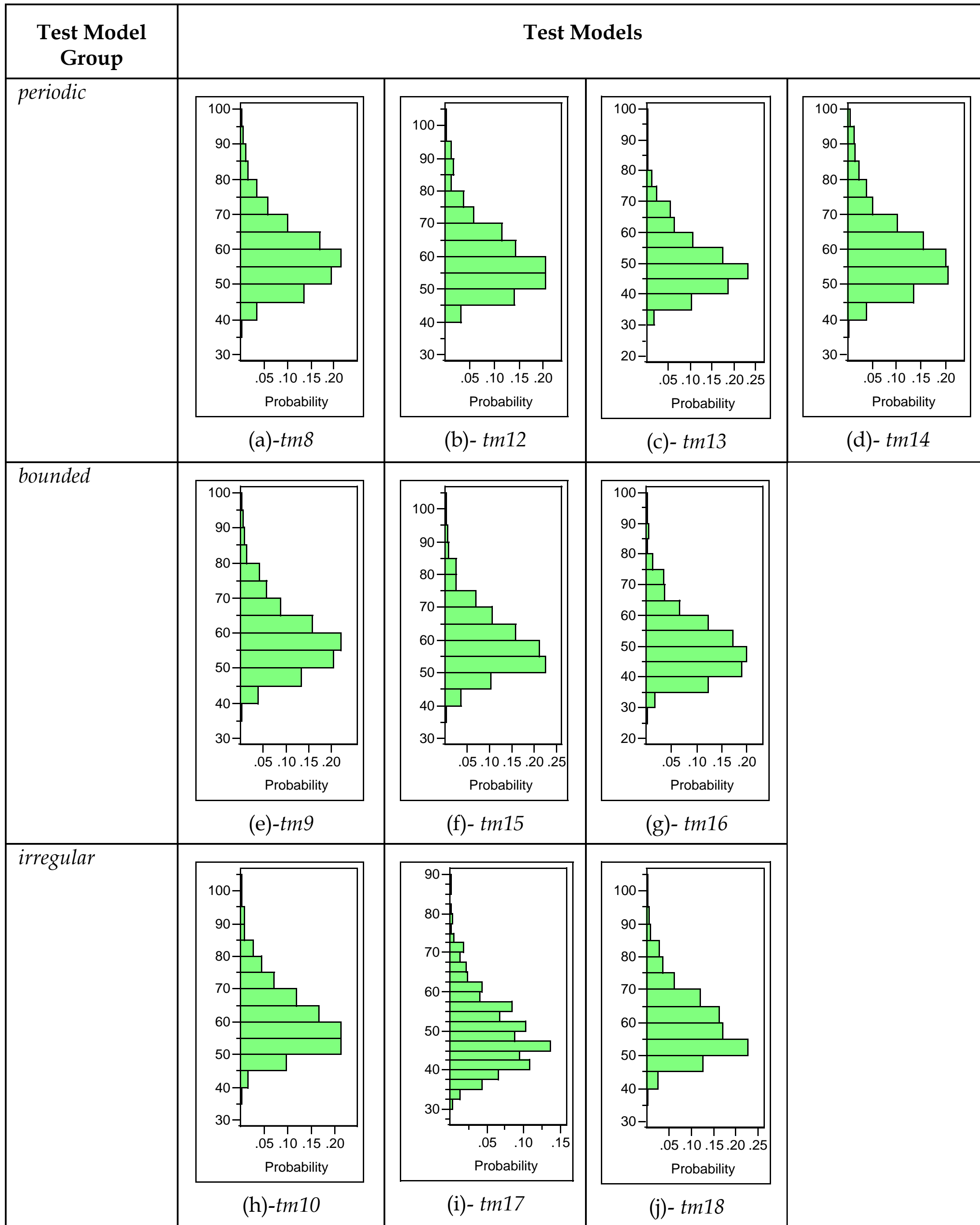


Figure 49- Histograms of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of each test model.

Test Model Group	Test Model	Min	Max	Average	Median	Standard Deviation
periodic	*tm8*	42	99	57	56	10.45
	tm12	41	99	58	57	10.94
	tm13	28	97	50	49	10.81
	tm14	43	99	58	57	11.70
bounded	*tm9*	39	97	58	57	11.0
	tm15	40	99	58	57	10.86
	tm16	30	98	50	49	10.48
irregular	*tm10*	44	99	59	58	11.27
	tm17	27	88	49	48	9.35
	tm18	36	96	59	57	11.09

Table 19-Descriptive statistics of the distributions in Figure 49.

7.2.9 Impact of Maximum Search Time

We report in this section the impact of variations in GA maximum search time on execution time, repeatability of outputs (maximum ISTOF and stress time values), and also maximum plateau generation numbers. Maximum search time is the range of the random numbers chosen from the ATS of a SD with arrival pattern (Section 6.5.4).

We first compare GA results for TMs *tm1*, *tm19* and *tm20* in Figure 50. As described in Section 7.2.5.5, *tm19* and *tm20* have the same SUT components (SDs, DCCFPs, ISDSs, etc.) as *tm1*, but the *GATimeSearchRange* value for *tm19* and *tm20* are 5 and 150 time units, respectively, instead of 50 in *tm1*. Therefore, comparing GA results for *tm1*, *tm19* and *tm20* should reveal the impact of maximum search time on all variables of interest. The corresponding descriptive statistics are shown in Table 20.

There are 12 graphs (3 rows in 4 columns) in Figure 50. Three rows correspond to different maximum search time, while columns relate to GA variables (execution time, maximum ISTOF values, maximum stress time values, and maximum plateau generation number).

In terms of execution time, variations in maximum search time do not have an impact. Across 1000 runs, all three TMs (*tm1*, *tm19* and *tm20*) show execution times in the range [45 ms, 130 ms]. Since a change in maximum search time only changes the range in which a random time from an ATS is selected, it is not surprising that there is no effect on the workload of different GA components.

As the maximum search time increases across the three test models (5 in *tm19* to 50 in *tm1* and 150 in *tm20*), the maximum of maximum ISTOF values across 1000 runs of a TM increases, i.e. 82 traffic units for *tm19*, 91 traffic units for *tm1* and 110 traffic units for *tm20.* This can be explained by an increase in the size of GA's time search range (in ATSs) from *tm19* to *tm1* and *tm20*. From another perspective, the difference between the maximum and minimum of maximum ISTOF values also increases with the maximum search time. The differences between the maximum and minimum of maximum ISTOF values for *tm19*, *tm1* and *tm20* are 20 (82-62), 41 (91-50), and 69 (110-41) respectively. This can also be explained by the increase in the size of GA's time search range (in ATSs) from *tm19* to *tm1* and *tm20*.

Distribution Group	Test Model	Min	Max	Average	Median	Standard Deviation
Execution time	*tm1*	46	125	58	62	11.34
	tm19	46	125	65	62	16.86
	tm20	46	125	69	63	17.50
Maximum ISTOF values	*tm1*	65	112	81	81	7.0
	tm19	60	82	73	72	4.67
	tm20	42	112	61	62	6.79
Maximum stress time values	*tm1*	6	56	22	20	12
	tm19	6	11	8	9	1.54
	tm20	9	156	31	13	37.56
Max Plateau Generation #	*tm1*	27	98	49	47	10.78
	tm19	26	94	48	46	10.52
	tm20	27	99	48	47	10.69

Table 20-Descriptive statistics of the distributions in Figure 50.

In terms of maximum stress time values, similar patterns to maximum ISTOF values can be observed among the three distributions in column 'maximum stress time values' of Figure 50. In terms of maximum plateau generation number, we can see that the increase in maximum search time slightly *delays* convergence across generations, i.e., the maximum plateau generation number in *tm19* runs is reached at 91, while it is 100 for both *tm1* and *tm20* runs.

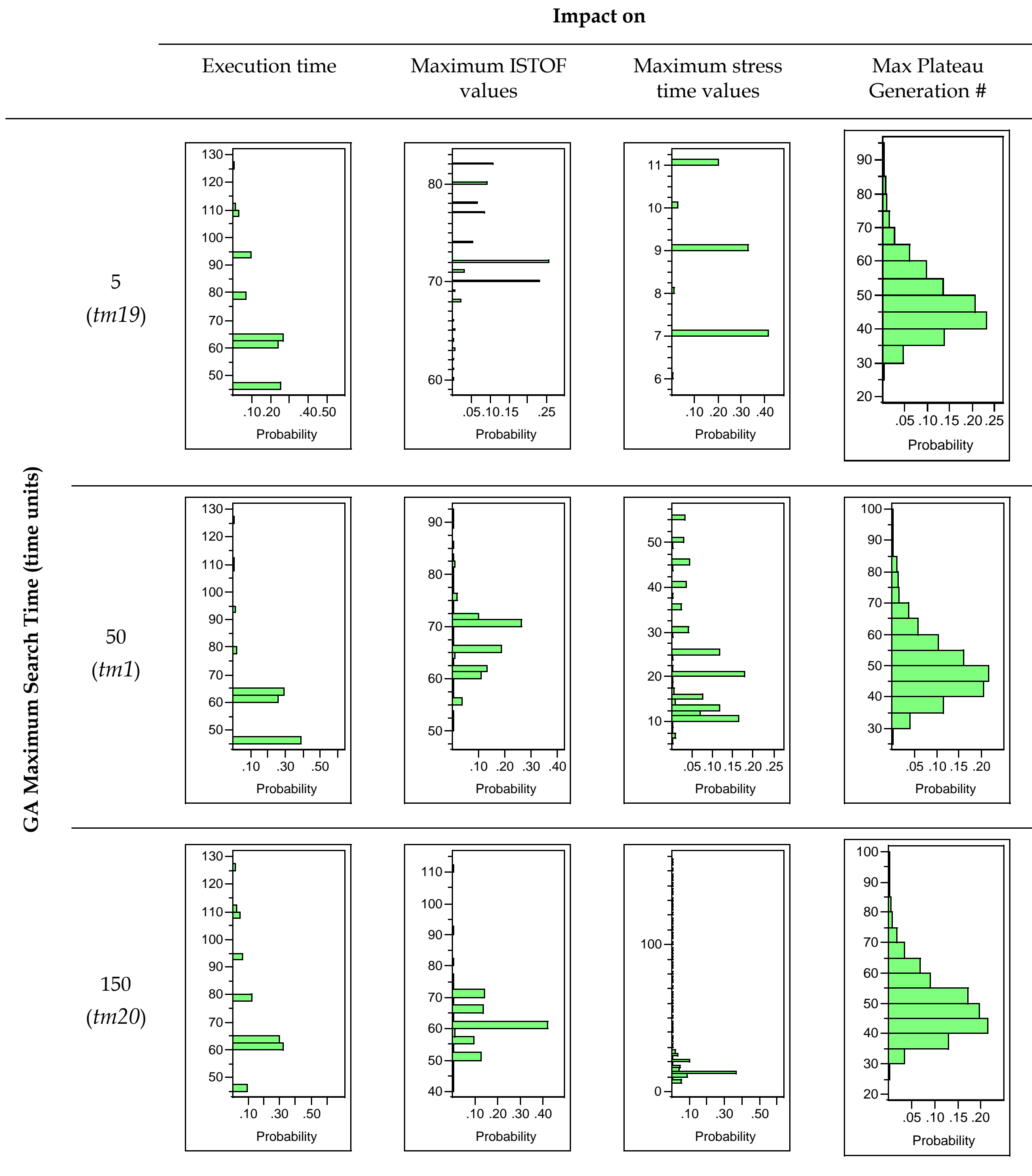


Figure 50- Impact of variations in maximum search time on the GA's behavior and outputs.

8 CASE STUDY

This section presents a case study based on an actual Distributed Real Time System (DTRS). We describe in Section 8.1 the SUT we chose for our case study. The stress test results are presented in Section 8.4.

8.1 System Under Test

Our stress test methodology can be used to stress test distributed systems, with an emphasis on safety-critical and data-intensive systems. *Distributed Control Systems (DCS)* [33] and *Supervisory Control and Data Acquisition (SCADA) Systems* [16] are two kinds of such systems.

We surveyed numerous existing systems (e.g. [11, 15, 19, 20]) to choose a suitable case study. Selection criteria were that it should be possible to run a system on a standard hardware/software platform, the design model and source code of the system should be available, and also the system should be accessible for use. Since no public domain systems met all the above requirements, we decided to analyze, design and build a prototype system based on a real-world specification.

SCAPS (our prototype system) is a SCADA-based power system (e.g. [50]) which controls the power distribution grid across a nation consisting of several provinces, in which are cities and regions. Each city and region has several local power distribution grids, each with a Tele-Control unit (TC), which gathers the grid data and can also be controlled remotely. There is a nation-wide central server, and each province has one central server that gathers the SCADA data from TCs from all over the province and sends them to the central server. The central sever performs the following real-time data-intensive safety-critical functions as part of the *Power Application Software* [53]: (1) Overload monitoring and control, (2) Detection of separated power systems and (3) Power restoration after network failure.

We designed SCAPS so that it meets all the suitability criteria for a case study. The UML model was defined and the system was implemented using Borland Delphi, which is a well-known Integrated Development Environment for Rapid Application Development. Further details can be found in [22].

We used our GARUS tool (Section 7) with the SCAPS UML model (Section 8.1.4) as input to derive stress test requirements maximizing instant data traffic on the SCAPS nation-wide network (*Canada*). We then derived the corresponding stress test cases for those requirements by finding the specific inputs/conditions which drive the test execution through the specific CFPs in the stress test cases. Furthermore, in our test execution, we scheduled those CFPs (their corresponding SDs) to be executed in the specific time instances as were determined by the stress test requirements. The test requirements included executing SDs *D* and *E* presented in Figure 7. The two SDs are parts of the power application software model discussed above. There are time constraints defined on these SDs' executions and our goal is to assess whether stress testing can help detect violations of these constraints.

Because none of the systems we surveyed meets the requirements, we decided to analyze, design, and build our own prototype system by using the ideas and concepts from existing distributed system technologies.

Section 8.1.1 presents an overview on SCADA-based power systems. Our design of a *SCAda-based Power System* (SCAPS) is described in Section 8.1.2. In Section 8.1.3, we discuss how and why SCAPS meets our case study requirements. We present the SCAPS' UML design model in Section 8.1.4. Relevant implementation issues are presented in Section 8.1.5. Section 8.1.6 provides a brief description of SCAPS' hardware and configuration.

8.1.1 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 [18]. A new class of SCADA systems that is called

open distributed systems [53] 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-standards, 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 [29, 37] and distribution grids [9, 12, 19, 48, 51]. 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 SCADA systems are usually available. Figure 51 shows a typical SCADA model of a power distribution system [19].

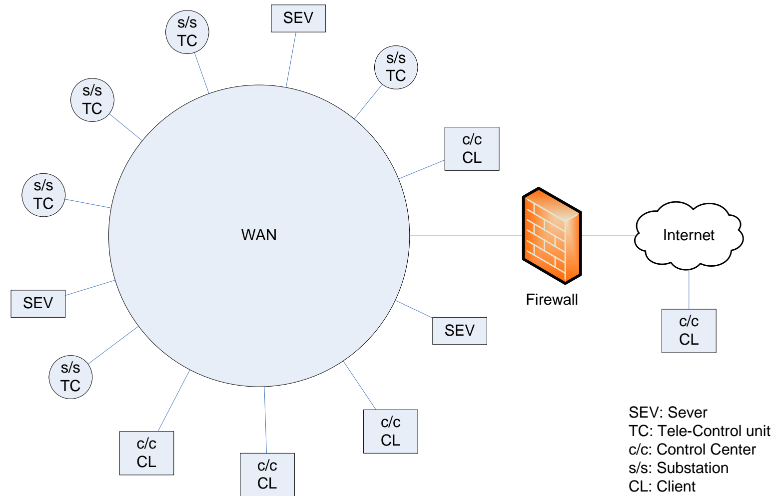


Figure 51-Power systems SCADA model [19].

The model consists of TCs (Te1e-Control units) that send data 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 (e.g., a city or a geographical region) of the power system to servers. Multicast communication based on IP is applied to the communication between TCs 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.

The communication model between tele-control units (TCs) and servers (SEVs) in a SCADA system [19] is shown in Figure 52. As we can see in this figure, power systems usually have a hierarchical operational organization [48].

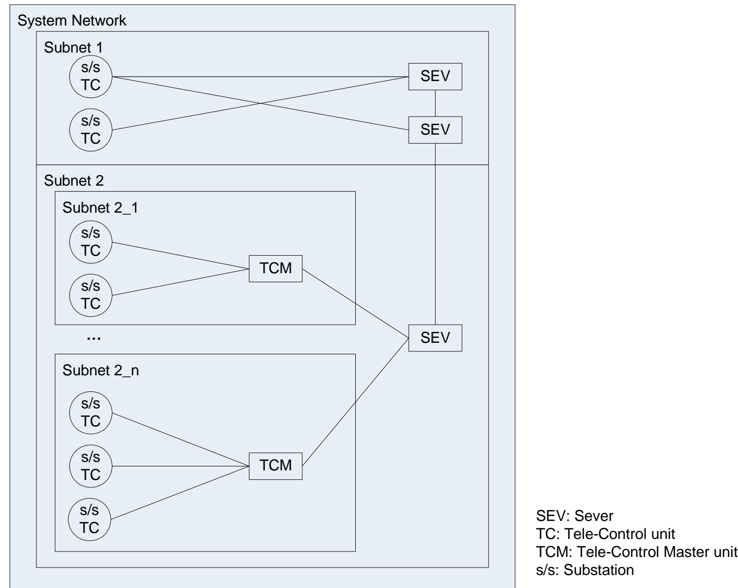


Figure 52- Communication model between tele-control units and servers in a SCADA system [19].

A typical operational organization of power systems is shown in Figure 53. 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 Graph (NIG).

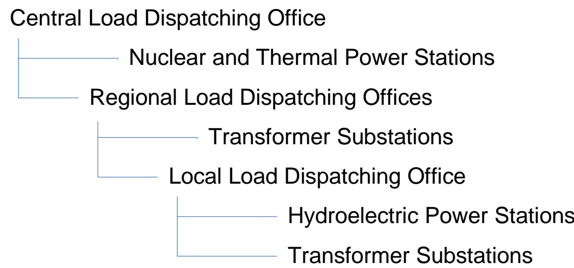


Figure 53-A typical operational organization for power systems [48].

8.1.2 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*: This functionality provides emergency strategies to prevent network disruption just after a network fault and later presents strategies and switching operation of breakers and disconnectors to restore power while keeping network's reliability.

It should be noted that we only focus on the real-time data-intensive safety-critical functions of the SCAPS here. Therefore, our stress test technique will be more effective in revealing faults if it is applied to such functions (use-cases) of a SUT. The above three functions are typical functions performed by SCADA power systems [19, 53], and will be shown in a use case diagram (Section 8.1.4), 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 [19, 53]:

- *State estimation*: Estimates most likely numerical data set to represent current network
- *Load forecasting*: Anticipates hourly total loads (24 points) for 1-7 days ahead based on the weather forecast, type of day, etc. utilizing historical data about weather and load.
- *Power flow control*: Supports operators activities by providing effective power flow control by evaluating network reliability for each several-minute time period for the next several hours, considering anticipated total load, network configuration, load flow, and contingencies.
- *Economical load dispatching*: Controls generator outputs economically according to demand considering the dynamic characteristics of boiler controller of thermal power generators while keeping ability to respond quickly to sudden load changes.
- *Unit commitment of generator*: A suitable schedule for starting/stopping the generators for the next 1-7 days is made using dynamic programming.

8.1.3 SCAPS Meets the Case-Study Requirements

To justify our decision, we discuss below how SCAPS meets all our requirements:

- *Requirement 1*: SCAPS is distributed, hard real-time, and safety-critical.
- *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 8.1.1). 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.

8.1.4 Partial UML Model

Consistent with the SCAPS specification in Section 8.1.2, 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. 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)

8.1.4.1 Use-Case Diagram

The SCAPS use-case diagram is shown in Figure 54. An ASA (Automatic System Agent) actor triggers Overload Monitoring (OM) and Detection of Separated Power System (DSPS) use-cases. As it will be modeled in SCAPS Modified Interaction Overview Diagram (Section 8.1.4.5), ASA will have a reactive behavior by continuously triggering those two SDs in a non-ending loop until the system is stopped.

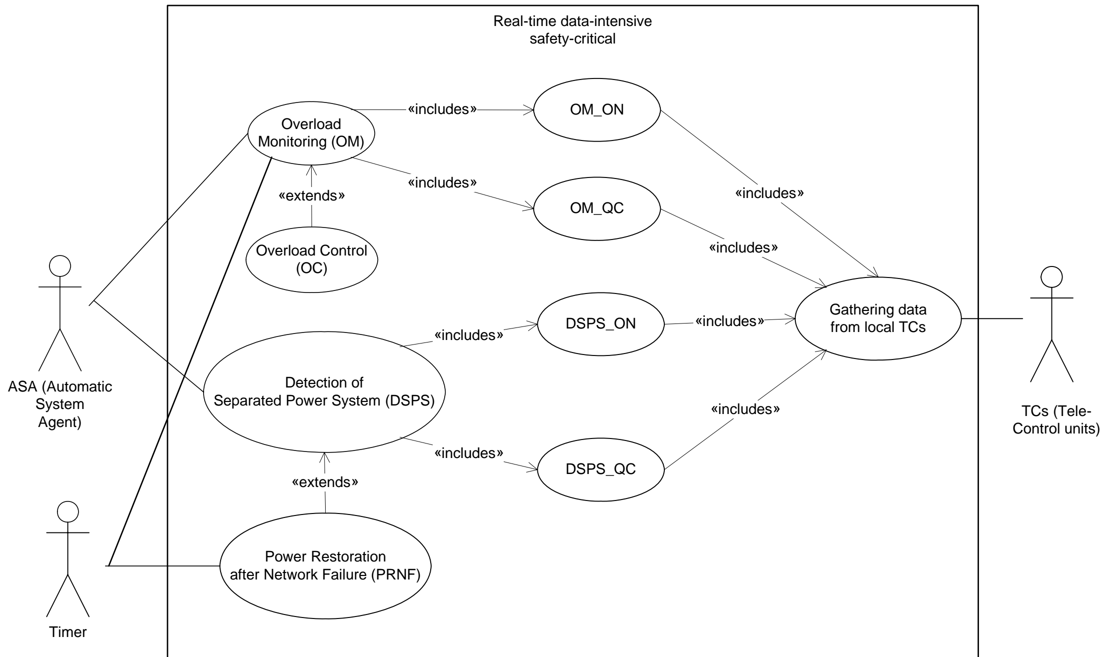


Figure 54- SCAPS use-case diagram.

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.

8.1.4.2 Network Deployment Diagram

The Network Deployment Diagram (NDD) of SCAPS is shown in Figure 55.

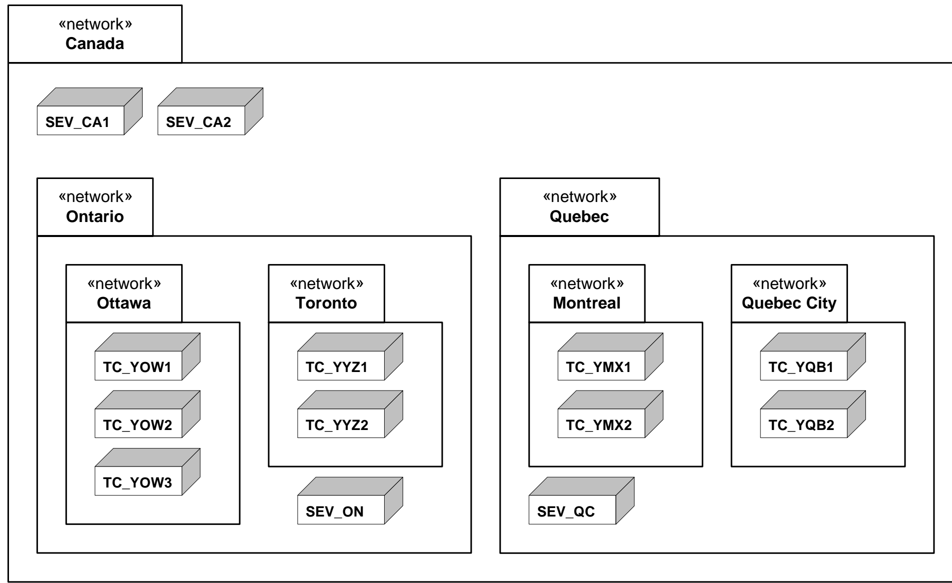


Figure 55- SCAPS network deployment diagram.

The networks for the provinces of Ontario and Quebec are shown in the NDD. Only two cities are considered in each of these two provinces. Three TCs (Tele-Control units) are considered for the city of Ottawa, while other cities have two TCs. There is one server (*SEV_ON* and *SEV_QC*) in each of the provinces. There are two servers at the national level: *SEV_CA1* is the main server. *SEV_CA2* is the backup server, i.e., it starts to operate whenever the main server fails.

8.1.4.3 Class Diagram

Part of the SCAPS class diagram which is required to illustrate the case study is shown in Figure 56. The classes are grouped in two groups: entity and control classes [8, 24]. Entity classes are those which are used either as parameters (by inheriting from *SetFuncParameter*) or return values (by inheriting from *QueryFuncResult*) of methods of control classes. Control classes are those from which active control objects will be instantiated and are the participating objects in SDs. All entity classes are data-intensive, since grid and load data of power systems for each region (or city) usually contain huge amounts of data [29, 37]. Furthermore, since there are two main groups of use-cases (overload and separated grid handlers), we group entity classes by two abstract classes *GridData* and *LoadData*. *LoadStatus* and *GridStatus* are the results of function *query* in class *TC* and *queryONData* and *queryQCData* in class *ProvController*. *LoadPolicy* and *GridStructure* are the parameters of set functions *setNewLoadPolicy* and *setNewGridStructure* in class *TC,* respectively. For brevity, usage dependencies among classes have not been shown in the class diagram, e.g. from *ProvController* to *QueryFuncResult*.

Tele-Control (TC) unit objects will be instantiated from class *TC*. Objects of class *ProvController* and *ASA* will be deployed on provincial (*SEV_ON* and *SEV_QC*) and national servers (the main server *SEV_CA1* and the backup *SEV_CA2*), respectively.

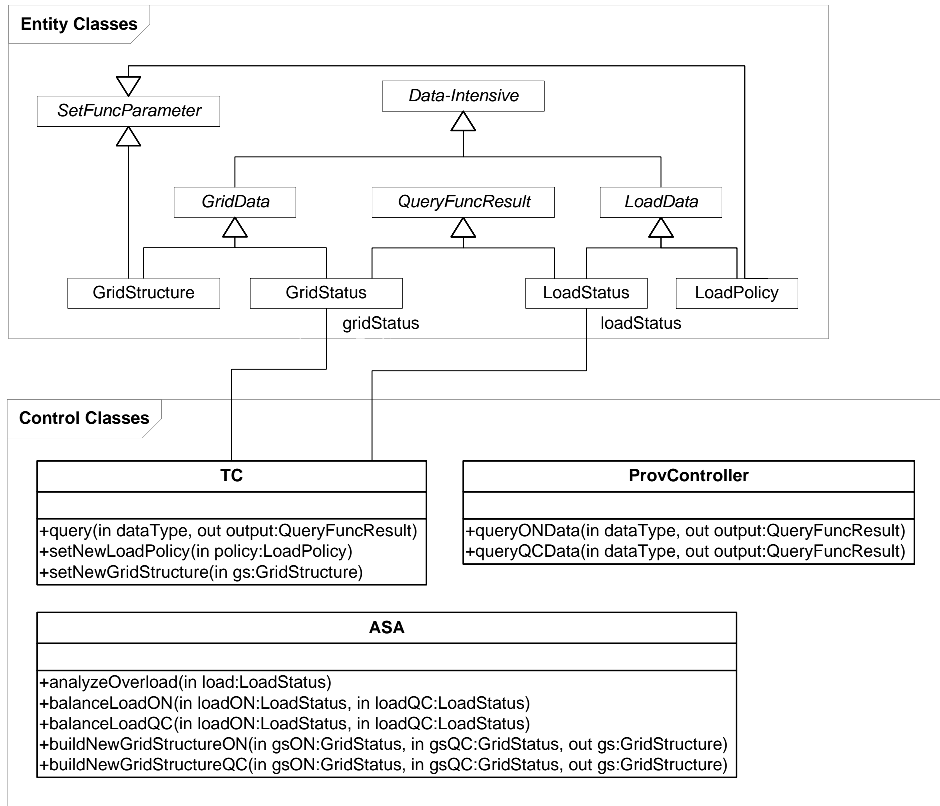


Figure 56-SCAPS partial class diagram.

8.1.4.4 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 functionalities of the system. In doing so, we tried to emulate as closely as possible the behavior of real SCADA-based power systems.

More precisely, we designed the SDs in ways that the simplifications did not impact the types of faults (e.g., RT faults) targeted by our stress test technique. We incorporated enough messages and alternatives in SDs to allow the generation of non-trivial stress test requirements. Since we designed SCAPS as a hard RT system, we therefore modeled the RT constraint using the UML SPT profile [43].

Eight SDs are presented in Figure 57-Figure 62. They correspond to use-cases in the SCAPS use-case diagram (Figure 54). SDs *OM_ON* and *OM_QC* in Figure 57 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*.

A realistic periodic arrival pattern value must be larger than the execution duration of the SD it is assigned to. This is because an invocation of the SD should complete execution before it is re-executed (due to a new event according to its arrival pattern). For example, as we measured, the duration of SD *OM_ON* is on average 1300 ms. We assume a periodic arrival pattern value of, say, 2400 ms for it. Similarly, since the durations of *DSPS_ON* and *DSPS_QC* are 1300 and 1100 ms, periodic arrival patterns with period and deviation values of 1700, 200 ms, and 1400, 200 ms are assigned to them, respectively.

To account for time delays in real-world, a deviation of 100 ms is considered for this periodic arrival pattern. Since SCAPS is a reactive system, we assign periodic arrival pattern to its SDs. Reactive systems usually check for environment changes periodically and take appropriate actions.

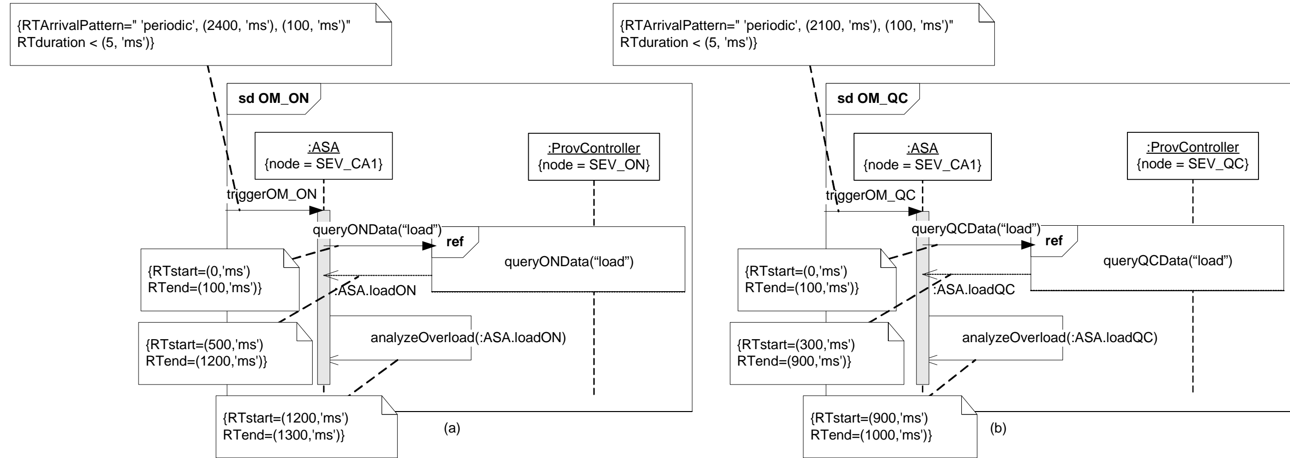


Figure 57- SDs *OM_ON* and *OM_QC* (Overload Monitoring).

The two SDs in Figure 58 (*queryONData(dataType)*) and Figure 59 (*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 55), 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 58 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.

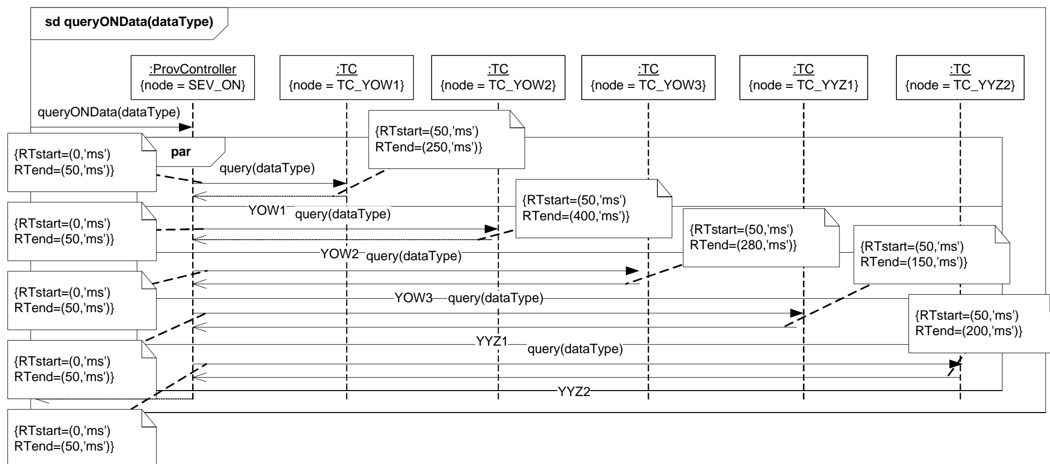


Figure 58-SD *queryONData(dataType).*

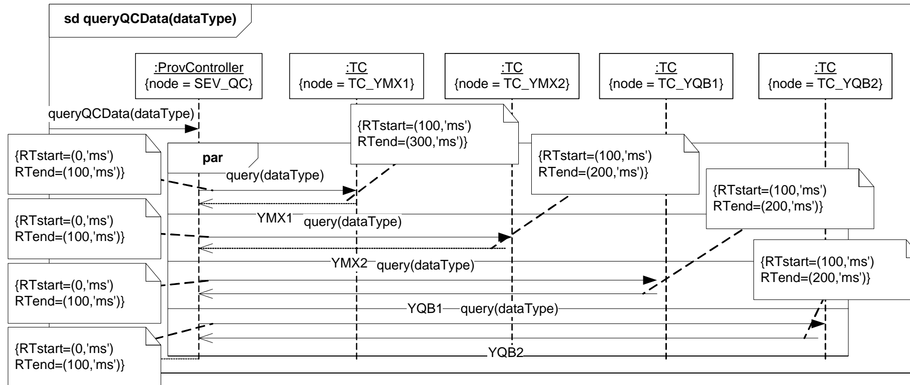


Figure 59-SD *queryQCData(dataType).*

OC (Overload Control) SD (Figure 60) checks if there is an overload situation in any of the two provinces (using *overloadIn()* as a condition). If this is the case in any of the two provinces, a new power distribution load policy is generated by an object of type *ASA* and it is sent to the respective provincial controller (using *setNewLoadPolicy()*).

Similar to the *OM_ON* and *OM_QC* SDs, *DSPS_ON* and *DSPS_QC* SDs (Figure 61) fetch grid connectivity data from the provincial controllers and check whether there is any separated power system (using *detectSeparatedPS()*).

Similar to the *OC* SD (Figure 60), *PRNF* (Power Restoration after Network Failure) SD (Figure 62) checks if there is any separated power system in any of the two provinces (using *anySeparationIn()* as a condition). If this is the case in any of the two provinces, a new power grid structure is generated by an object of type *ASA* and it is sent to the respective provincial controller (*setNewGridStructure()*).

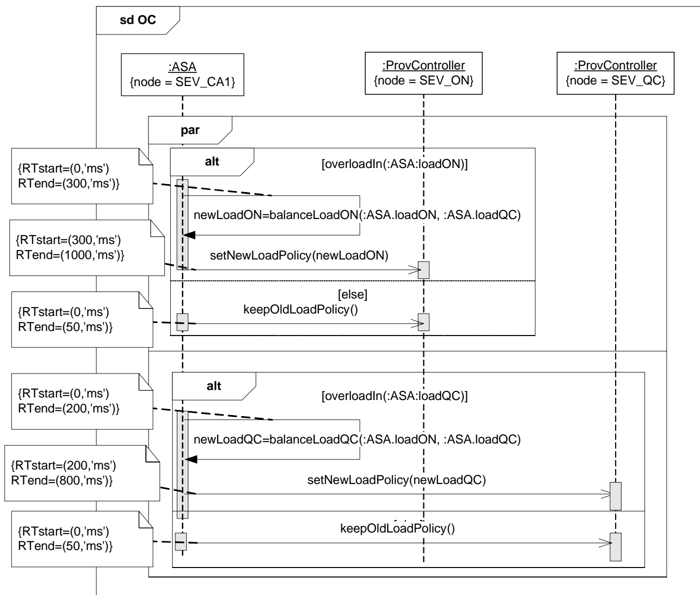


Figure 60- SD *OC* (Overload Control).

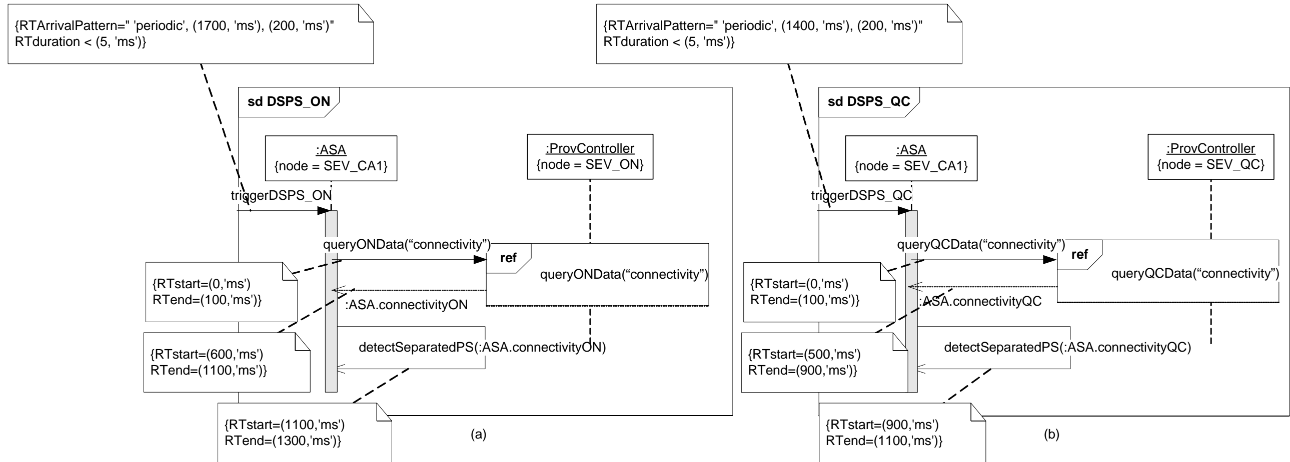


Figure 61-SD *DSPS_ON* and *DSPS_QC* (Detection of Separated Power System).

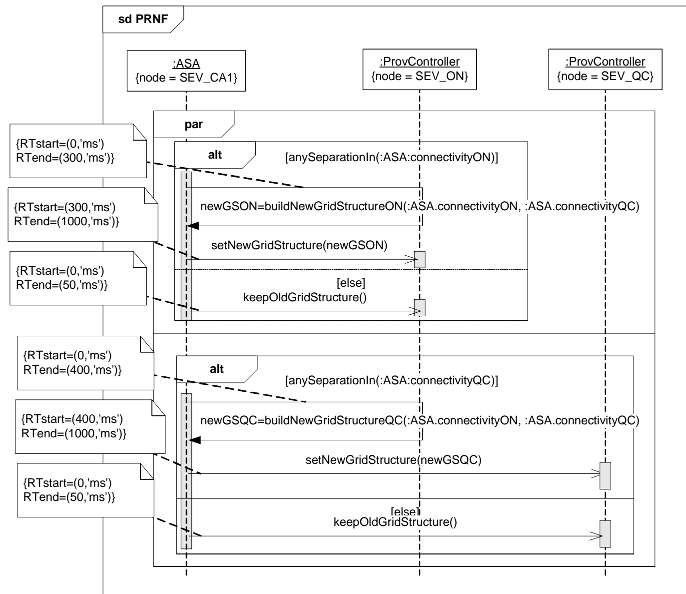


Figure 62-SD *PRNF* (Power Restoration after Network Failure).

8.1.4.5 Modified Interaction Overview Diagram

The MIOD of SCAPS is shown in Figure 63. As denoted in the SCADA-based power systems literature (e.g. [9, 12, 19, 48, 51]), such systems have both soft and hard RT constraints.

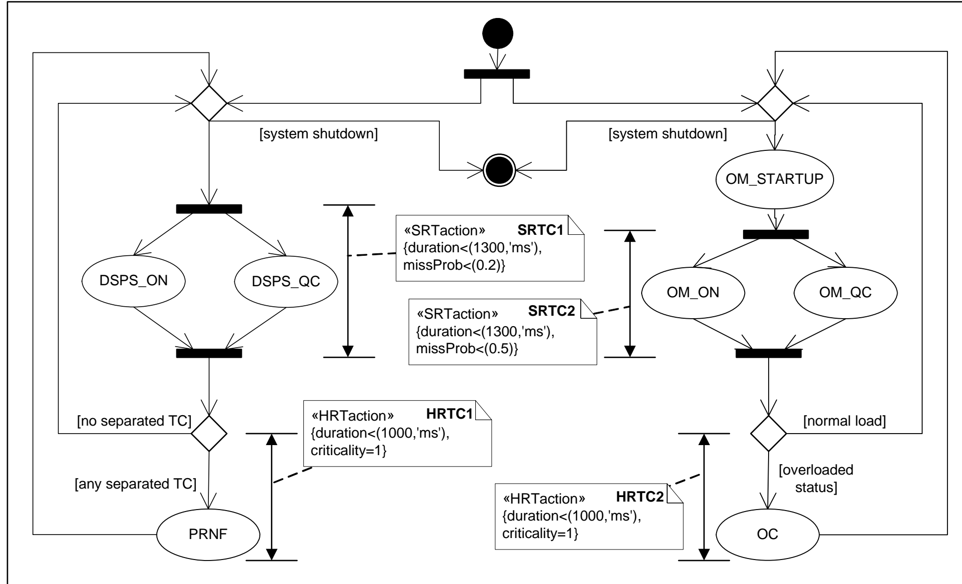


Figure 63-SCAPS Modified Interaction Overview Diagram (MIOD).

In order to model hard and soft RT constraints, we propose an extension to the *RTaction* stereotype of the UML-SPT referred to as *HRT* (Hard RT Constraints) and *SRT* (Soft RT Constraints). Furthermore, in order to model the statistical threshold probability up to which SRT constraints can be missed, we consider a tagged value referred to as *missProb* for SRT constraints. Similarly, we consider a tagged value referred to as *criticality* for HRT constraints. Criticality is a real number in the range [0..1] indicating the degree to which the consequences of missing a hard deadline are unacceptable: the closer to one the criticality of a HRT constraint, the more severe the consequences of missing it. For example, if violating a HRT constraint may cause life-threatening situations, it would be better to set criticality to 1. Conversely, if for example the cost of violating a HRT constraint is just an increase in the temperature of a water hydro plant (which will not immediately lead to catastrophic results), then this constraint would have a lesser value of criticality. *HRTaction* and *SRTaction* stereotypes are presented in Table 21 and Table 22, which are similar to the representation used in the UML-SPT [84].

Stereotype	Base Class	Tags
SRTaction	Message	RTduration
	MessageSequence	RTmissProb
	Action	
	ActionSequence	

Table 21-A stereotype to model SRT constraints.

Stereotype	Base Class	Tags
HRTaction	Message	RTduration
	MessageSequence	RTcriticality
	Action	
	ActionSequence	

Table 22-A stereotype to model HRT constraints.

Table 21 and Table 22 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 [84]. The «SRT» and «HRT» stereotypes have two associated tagged values each, which are defined in Table 23.

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

Table 23-Tagged values of SRT and HRT stereotypes.

Table 23 defines the type of each tag. An *RTduration* tagged value is an instance of the *RTtimeValue* data type (Section 4.2.2.4 of [84]). *RTmissProb* and *RTcriticality* are real value in the range of [0…1]. Each tag also has a multiplicity indicating how many individual values can be assigned to each tag. A lower bound of zero implies that the tagged value is optional.

«SRTaction» and «HRTaction» stereotypes can be used either in a SD or a MIOD. In the former case, the RT constraint is applied to a *Message* or a *MessageSequence,* while in the latter, the constraint is applied to an *Action*, or an *ActionSequence* (since MIOD is a subtype of activity diagrams).

We consider four MIOD-level RT constraints for SCAPS. Figure 63 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. The constraints are labeled (bold face text) to make it easier to refer to them later, and are explained below.

1. SRT constraints
    a. $SRTC_1$: Detection of separated power systems (concurrent runs of *DSPS_ON* and *DSPS_QC*) should be done in less than 1300 ms, with an acceptable missing probability of 0.2 (20%). In other words, this constraint must not be missed in more than 20% of the runs.
    b. $SRTC_2$: Overload monitoring (concurrent runs of *OM_ON* and *OM_QC*) should complete within less than 1300 ms from its start time. We set the acceptable missing probability of this SRT constraint to 0.5.
2. HRT constraints
    a. $HRTC_1$: As soon as a separated power system is detected, the power restoration policy (*PRNF* SD) should be executed in less than 1000 ms. We assign criticality=1 to this constraint.
    b. $HRTC_2$: As soon as an overload situation is detected, overload control policy (*OC* SD) should be executed in less than 1000 ms. We assign criticality[1]=1 to this constraint. As discussed above, 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).

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

We developed a Delphi application for SCAPS. The application asks the user for the node on which it is to run, e.g., *SEV_CA1*, *SEV_ON*, and *TC_YOW1*. Afterwards, the business logic of the application changes accordingly. For example, if *SEV_CA1* is chosen, the application switches to the national server node,

---

[1] As defined by UML SPT profile [43], criticality determines the extent to which the consequences of missing a hard deadline are unacceptable.

[2] www.borland.com/delphi

waiting for connections from provincial nodes. When different copies of the application on different nodes have been deployed and all nodes connections are ready, the system then starts functioning. A screenshot of the main screen of SCAPS is shown in Figure 64, where the application is running as a *SEV_CA1* node and has just accepted a connection from the *TC_YOW1* node.

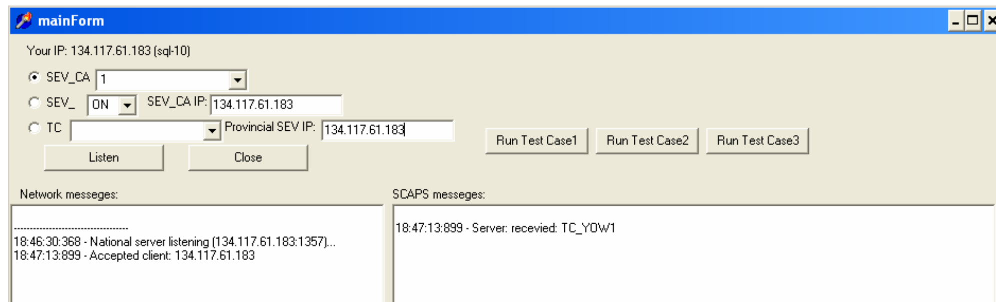


Figure 64-A screenshot of the main screen of SCAPS.

We had to account for the limitation of our research center's hardware/software platforms when implementing the system in such a way to preserve the realism of our case study. The parts of the system for which we had to incorporate stubs to emulate behavior were: (1) dedicated power distribution hardware such as load and connectivity meters and sensors, which are parts of the TC actors (refer to the SCAPS use-case diagram in Figure 54), 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 56).

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 stress testing a SUT 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 the edges specified by a test case. In terms of returned values by stubs for the dedicated power distribution hardware, for example function *query()* of class *TC*, they only return a random large data object.

The implementation of stubs for complex functionalities of the power application software was similar to that of the dedicated power distribution hardware. The results generated by such functions were not really needed in our context to execute test cases. However, we had to make sure the durations of such functions were as close as possible to real world situations. We made realistic assumptions in such cases using the power systems literature [9, 12, 19, 48, 51], e.g., we assumed that function *analyzeOverload* of class *ASA* takes 100 ms to run (refer to the SDs *OM_ON* in Figure 57). 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.

8.1.6 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.

8.2 Stress Test Architecture

An overview of the SCAPS stress test architecture is shown in Figure 65. The sequence of high-level steps to be performed by a tester to run a complete stress test procedure is shown. The steps are briefly explained below.

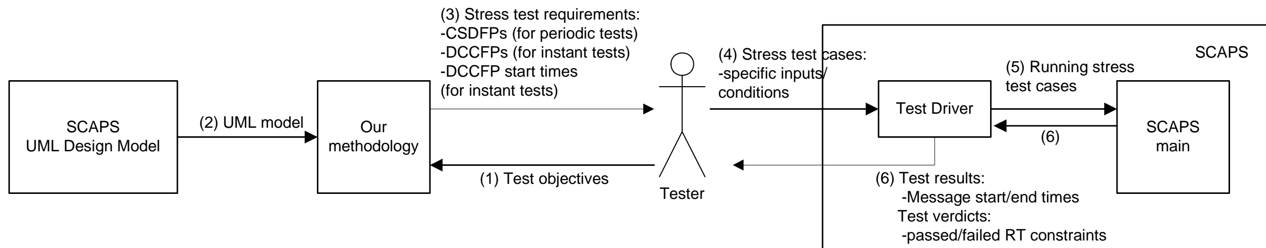


Figure 65-Overview of SCAPS Stress Test Architecture.

1. The tester feeds the test objectives to the methodology. For example, we considered three test objectives in our case study.
2. The methodology uses the SCAPS UML model as input.
3. The methodology uses the SCAPS UML model to generate test requirements for the given test objectives and returns the test requirements to the tester. Note that this step is completely automated.
4. The tester devises appropriate test case for the test requirements. Note that this step is currently done manually by the tester. The tester feeds the test cases into a test driver which is responsible for running the test cases.
5. The test driver runs the generated test cases by feeding them into the SUT. Note that we have made the test driver a component of the SCAPS system in our current implementation. Embedding the test driver inside SCAPS helped us simplify the actual test environment and test executions. It also enabled us to reduce the probe effects (due to monitoring) as much as possible. The probe effects resulting from the test driver were negligible since the test driver only feeds specific test cases and monitors the system. Feeding test cases consisted in setting the attributes of an instance of a test class (in the test driver) to specific values and starting the system. The resulting probe effect in this case was then the time to set specific variables to specific values, which is in the range of several milliseconds, which is negligible when compared to the SCAPS message durations (several hundreds of milliseconds, as it can be seen in the SCAPS SDs in Figure 57-Figure 62). 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 flexibility in scheduling DCCFPs (via a scheduler in the test driver).
6. Test results are gathered from the SUT. They include: start/end times of distributed messages and test verdicts on real-time constraints, which indicate whether each real-time constraint has been adhered to in a particular run. Test results are both logged in 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-

[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 [60].

[2] Observability is a measure of how well the internal state of a system can be inferred by knowledge of its external outputs [61].

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.

8.3 Building the Stress Test Model for SCAPS

Using the given UML design model in Section 8.1.4, we first build the test model required by our test technique.

8.3.1 Network Interconnectivity Tree

The Network Interconnectivity Graph (NIG) of SCAPS can be derived from the Network Deployment Diagram (NDD) in Figure 55. The NIG is shown in Figure 66.

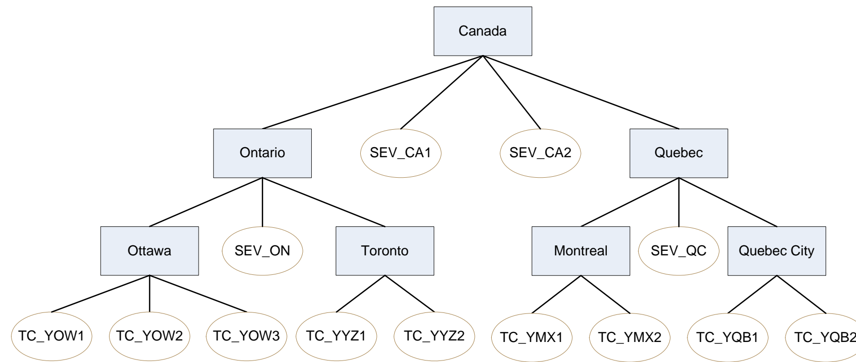


Figure 66- SCAPS Network Interconnectivity Graph (NIG).

8.3.2 Control Flow Analysis of SDs

Recall from Section 5.1 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 8.1.4.4. CCFGs shown in Figure 67 to Figure 72 correspond to SDs in Figure 57 to Figure 62. 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.

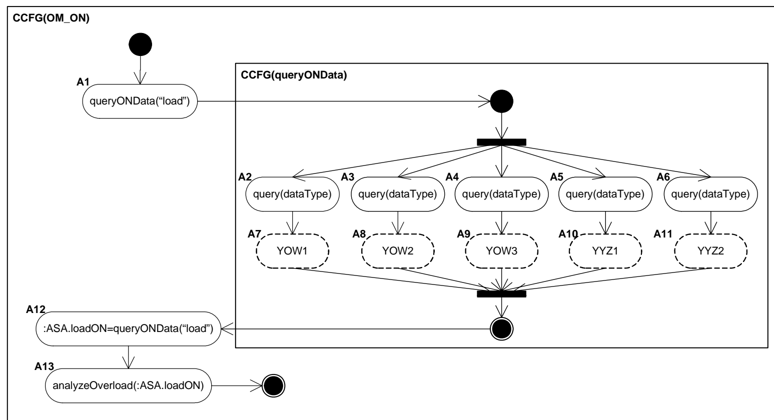


Figure 67-CCFG(OM_ON).

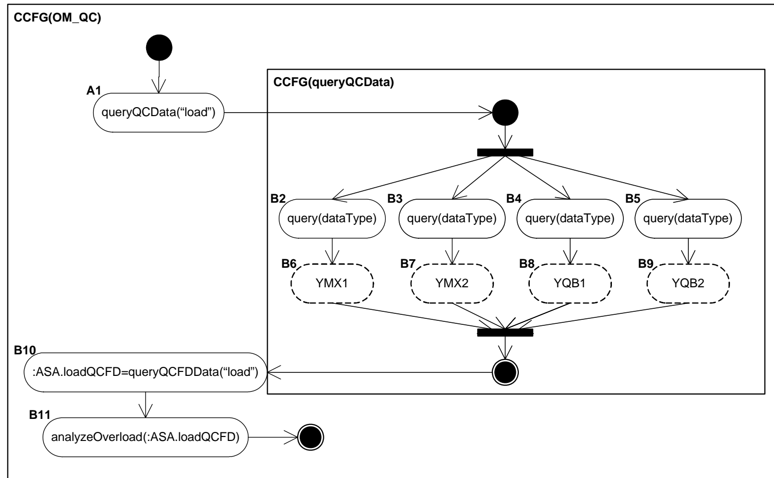


Figure 68-CCFG(OM_QC).

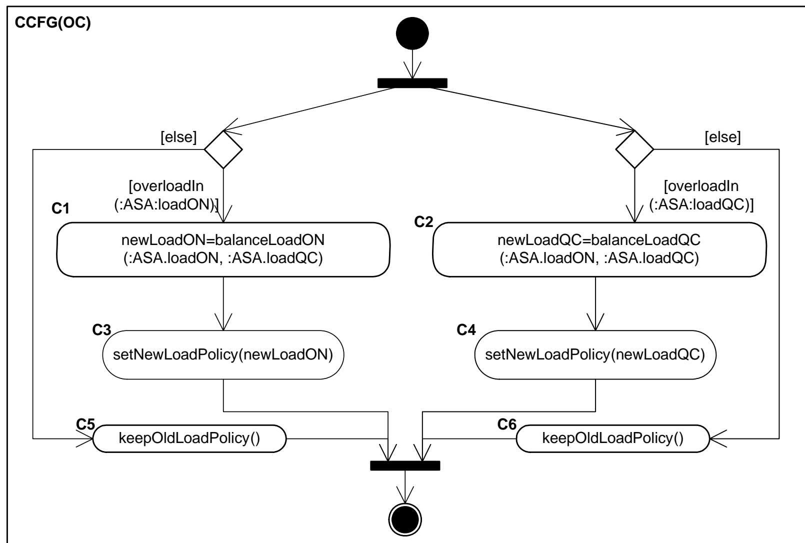


Figure 69-CCFG(OC).

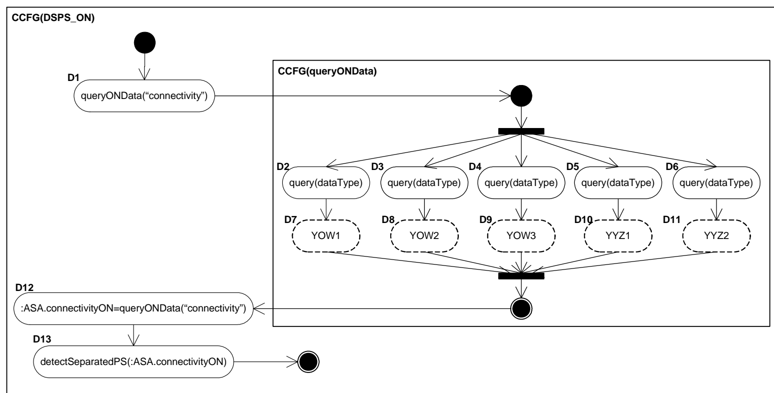


Figure 70-CCFG(DSPS_ON).

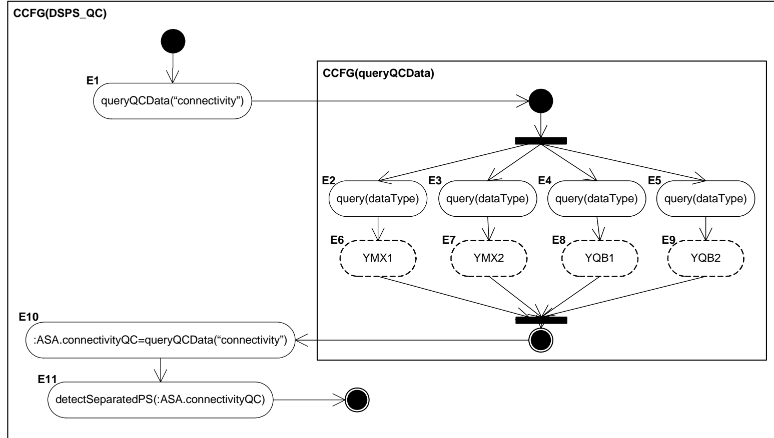


Figure 71-CCFG(DSPS_QC).

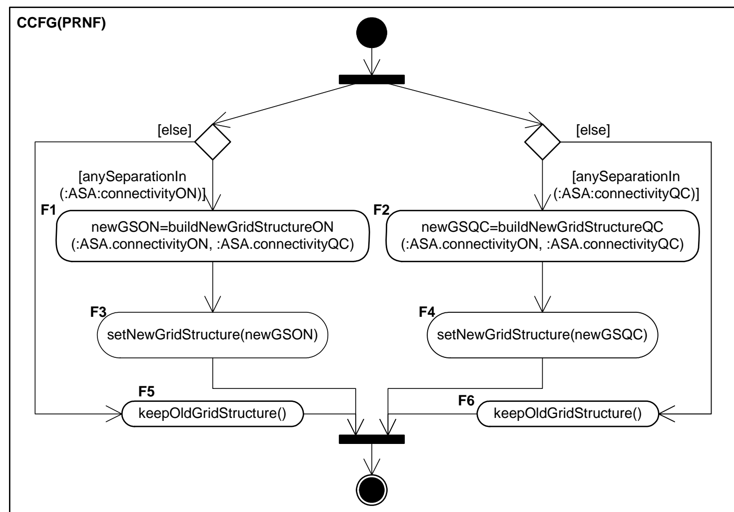


Figure 72-CCFG(PRNF).

8.3.3 Derivation of Distributed Concurrent Control Flow Paths

Using the technique presented in [21], the CCFPs and DCCFPs are derived from the CCFGs shown in Figure 67 to Figure 72, and are shown in Figure 73Figure 74. To ease future references, we assign SD_i and $\rho_{i,j}$ indices to SDs and the DCCFPs of each SD, respectively. Let us assign $\rho_{0,0}$ to the only CCFP of SD *OM_STARTUP*, which does not contain any distributed message.

$$CCFP(\underbrace{OM_ON}_{SD_1}) = \left\{ A_1 \begin{pmatrix} A_2A_7 \\ A_3A_8 \\ A_4A_9 \\ A_5A_{10} \\ A_6A_{11} \end{pmatrix} A_{12}A_{13} \right\} \Rightarrow DCCFP(OM_ON) = \left\{ \underbrace{A_1 \begin{pmatrix} A_2A_7 \\ A_3A_8 \\ A_4A_9 \\ A_5A_{10} \\ A_6A_{11} \end{pmatrix}}_{\rho_{1,1}} A_{12} \right\}$$

$$CCFP(\underbrace{OM_QC}_{SD_2}) = \left\{ B_1 \begin{pmatrix} B_2B_6 \\ B_3B_7 \\ B_4B_8 \\ B_5B_9 \end{pmatrix} B_{10}B_{11} \right\} \Rightarrow DCCFP(OM_QC) = \left\{ \underbrace{B_1 \begin{pmatrix} B_2B_6 \\ B_3B_7 \\ B_4B_8 \\ B_5B_9 \end{pmatrix}}_{\rho_{2,1}} B_{10} \right\}$$

$$CCFP(\underbrace{QC}_{SD_3}) = \left\{ \begin{pmatrix} C_1C_3 \\ C_2C_4 \end{pmatrix}, \begin{pmatrix} C_5 \\ C_6 \end{pmatrix} \right\} \Rightarrow DCCFP(OC) = \left\{ \underbrace{\begin{pmatrix} C_3 \\ C_4 \end{pmatrix}}_{\rho_{3,1}}, \underbrace{\begin{pmatrix} C_3 \\ C_6 \end{pmatrix}}_{\rho_{3,2}}, \underbrace{\begin{pmatrix} C_5 \\ C_4 \end{pmatrix}}_{\rho_{3,3}}, \underbrace{\begin{pmatrix} C_5 \\ C_6 \end{pmatrix}}_{\rho_{3,4}} \right\}$$

$$CCFP(\underbrace{DSPS_ON}_{SD_4}) = \left\{ D_1 \begin{pmatrix} D_2D_7 \\ D_3D_8 \\ D_4D_9 \\ D_5D_{10} \\ D_6D_{11} \end{pmatrix} D_{12}D_{13} \right\} \Rightarrow DCCFP(DSPS_ON) = \left\{ \underbrace{D_1 \begin{pmatrix} D_2D_7 \\ D_3D_8 \\ D_4D_9 \\ D_5D_{10} \\ D_6D_{11} \end{pmatrix}}_{\rho_{4,1}} D_{12} \right\}$$

$$CCFP(\underbrace{DSPS_QC}_{SD_5}) = \left\{ E_1 \begin{pmatrix} E_2E_6 \\ E_3E_7 \\ E_4E_8 \\ E_5E_9 \end{pmatrix} E_{10}E_{11} \right\} \Rightarrow DCCFP(DSPS_QC) = \left\{ \underbrace{E_1 \begin{pmatrix} E_2E_6 \\ E_3E_7 \\ E_4E_8 \\ E_5E_9 \end{pmatrix}}_{\rho_{5,1}} E_{10} \right\}$$

$$CCFP(\underbrace{PRNF}_{SD_6}) = \left\{ \begin{pmatrix} F_1F_3 \\ F_2F_4 \end{pmatrix}, \begin{pmatrix} F_5 \\ F_6 \end{pmatrix} \right\} \Rightarrow DCCFP(PRNF) = \left\{ \underbrace{\begin{pmatrix} F_3 \\ F_4 \end{pmatrix}}_{\rho_{6,1}}, \underbrace{\begin{pmatrix} F_3 \\ F_6 \end{pmatrix}}_{\rho_{6,2}}, \underbrace{\begin{pmatrix} F_5 \\ F_4 \end{pmatrix}}_{\rho_{6,3}}, \underbrace{\begin{pmatrix} F_5 \\ F_6 \end{pmatrix}}_{\rho_{6,4}} \right\}$$

Figure 73-CCFP and DCCFP sets of SDs in SCAPS.

8.3.4 Derivation of Independent-SD Sets

Using the method in Section 5.2 and the SCAPS MIOD (Figure 63), we derive SCAPS Independent-SD Sets (ISDSs). We need to first derive the Independent-SDs Graph (ISDG) corresponding to the MIOD. Using the algorithm in Section 5.2, the ISDG shown in Figure 75 is derived from the MIOD of Figure 63. Note that we do not include SD *OM_STARTUP* in this ISDG, since it does not have any distributed messages.

$$\underbrace{CCFP(OM_ON)}_{SD_1} = \left\{ A_1 \begin{pmatrix} A_2A_7 \\ A_3A_8 \\ A_4A_9 \\ A_5A_{10} \\ A_6A_{11} \end{pmatrix} A_{12}A_{13} \right\} \Rightarrow DCCFP(OM_ON) = \left\{ \underbrace{A_1 \begin{pmatrix} A_2A_7 \\ A_3A_8 \\ A_4A_9 \\ A_5A_{10} \\ A_6A_{11} \end{pmatrix} A_{12}}_{\rho_{1,1}} \right\}$$

$$\underbrace{CCFP(OM_QC)}_{SD_2} = \left\{ B_1 \begin{pmatrix} B_2B_6 \\ B_3B_7 \\ B_4B_8 \\ B_5B_9 \end{pmatrix} B_{10}B_{11} \right\} \Rightarrow DCCFP(OM_QC) = \left\{ \underbrace{B_1 \begin{pmatrix} B_2B_6 \\ B_3B_7 \\ B_4B_8 \\ B_5B_9 \end{pmatrix} B_{10}}_{\rho_{2,1}} \right\}$$

$$CCFP(\underbrace{QC}_{SD_3}) = \left\{ \begin{pmatrix} C_1C_3 \\ C_2C_4 \end{pmatrix}, \begin{pmatrix} C_5 \\ C_6 \end{pmatrix} \right\} \Rightarrow DCCFP(OC) = \left\{ \underbrace{\begin{pmatrix} C_3 \\ C_4 \end{pmatrix}}_{\rho_{3,1}}, \underbrace{\begin{pmatrix} C_3 \\ C_6 \end{pmatrix}}_{\rho_{3,2}}, \underbrace{\begin{pmatrix} C_5 \\ C_4 \end{pmatrix}}_{\rho_{3,3}}, \underbrace{\begin{pmatrix} C_5 \\ C_6 \end{pmatrix}}_{\rho_{3,4}} \right\}$$

$$\underbrace{CCFP(DSPS_ON)}_{SD_4} = \left\{ D_1 \begin{pmatrix} D_2D_7 \\ D_3D_8 \\ D_4D_9 \\ D_5D_{10} \\ D_6D_{11} \end{pmatrix} D_{12}D_{13} \right\} \Rightarrow DCCFP(DSPS_ON) = \left\{ \underbrace{D_1 \begin{pmatrix} D_2D_7 \\ D_3D_8 \\ D_4D_9 \\ D_5D_{10} \\ D_6D_{11} \end{pmatrix} D_{12}}_{\rho_{4,1}} \right\}$$

$$\underbrace{CCFP(DSPS_QC)}_{SD_5} = \left\{ E_1 \begin{pmatrix} E_2E_6 \\ E_3E_7 \\ E_4E_8 \\ E_5E_9 \end{pmatrix} E_{10}E_{11} \right\} \Rightarrow DCCFP(DSPS_QC) = \left\{ \underbrace{E_1 \begin{pmatrix} E_2E_6 \\ E_3E_7 \\ E_4E_8 \\ E_5E_9 \end{pmatrix} E_{10}}_{\rho_{5,1}} \right\}$$

$$CCFP(\underbrace{PRNF}_{SD_6}) = \left\{ \begin{pmatrix} F_1F_3 \\ F_2F_4 \end{pmatrix}, \begin{pmatrix} F_5 \\ F_6 \end{pmatrix} \right\} \Rightarrow DCCFP(PRNF) = \left\{ \underbrace{\begin{pmatrix} F_3 \\ F_4 \end{pmatrix}}_{\rho_{6,1}}, \underbrace{\begin{pmatrix} F_3 \\ F_6 \end{pmatrix}}_{\rho_{6,2}}, \underbrace{\begin{pmatrix} F_5 \\ F_4 \end{pmatrix}}_{\rho_{6,3}}, \underbrace{\begin{pmatrix} F_5 \\ F_6 \end{pmatrix}}_{\rho_{6,4}} \right\}$$

Figure 74-CCFP and DCCFP sets of SDs in SCAPS.

As discussed in Section 5.2, this needs finding maximal-complete subgraph in a graph. By finding the maximal-complete subgraph of the ISDG in Figure 75, the Independent SD Sets of SCAPS can be derived. SCAPS has seven ISDSs:

$$ISDS_1 = \{ OM_ON, OM_QC, DSPS_ON, DSPS_QC \} \quad ISDS_2 = \{ OM_ON, OM_QC, PRNF \}$$
$$ISDS_3 = \{ DSPS_ON, DSPS_QC, OC \} \quad ISDS_4 = \{ OC, PRNF \}$$

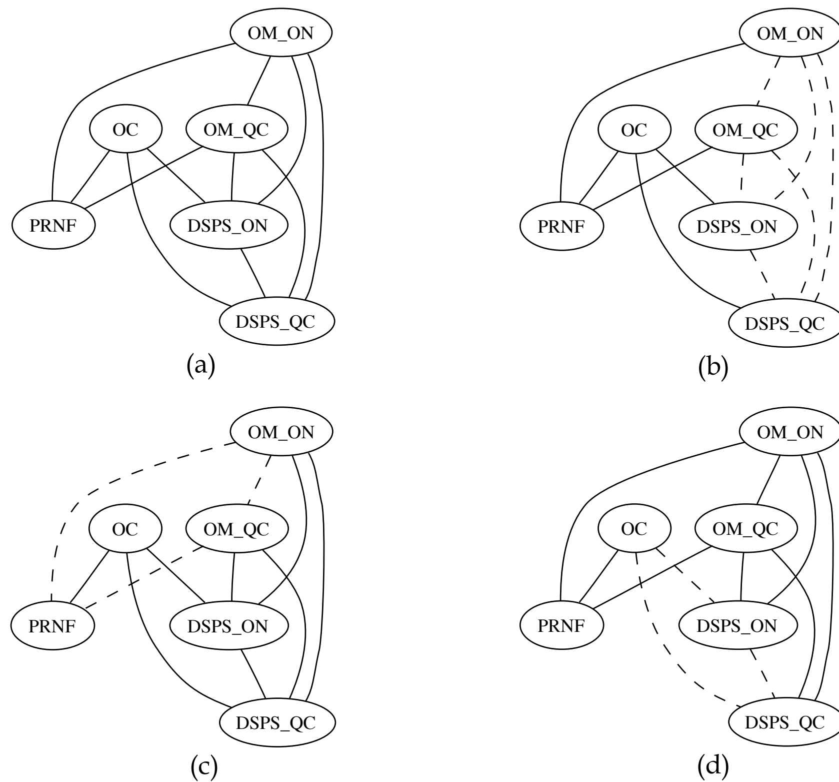


Figure 75-(a):Independent-SDs Graph (ISDG) corresponding to the MIOD of Figure 63. (b), (c) and (d): Three of the maximal-complete subgraphs of the ISDG (shown with dashed edges), yielding three ISDSs.

8.3.5 Data Size of Messages

Note that, for brevity, we do not discuss the data structure of the entity data classes in SCAPS (Figure 56). But according to the literature on SCADA-based power systems [9, 12, 19, 48, 51], 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 24 as the mean data sizes of the entity data classes in Figure 56. These values are realistic size estimates of real grid and load values according to the literature on SCADA-based power systems [36]. For example, an instance of the load object of the power distribution grid of a city includes the load values of the different hubs and components of the grid. This value can vary depending on the size of the city as well as the complexity of the distribution grid. We assume the data size to be in the order of several mega-bytes, which is reasonable assumption based on what is reported in the literature.

Note that we assume the data sizes in Table 24 to be representative for instances of all TCs. However, as different TCs are deployed in different cities/regions, the load or grid status data can vary to a large extent. This can be easily accounted for by extending data sub-classes and calculating the corresponding data sizes. For example, sub-classes like *OttawaLoadStatus* and *TorontoLoadStatus* (with different data fields sizes) can be derived from the class *LoadStatus* in Figure 56.

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

Table 24-Mean data sizes of the entity data classes of SCAPS.

8.3.6 Using GARUS to Derive Stress Test Requirements

In this section, we use GARUS to derive stress test requirements for the above test objective. The derivation of GARUS input files for the test objective is described in Section 8.3.6.1. The GA execution and the repeatability of its results are discussed in Section 8.3.6.2. The stress test requirements generated by GARUS are presented in Section 8.3.6.3.

8.3.6.1 Input File

Recall from Section 7.1.3 that an input file passed to GARUS contains the test model of a SUT. We furthermore assumed that the test model in an input file has already been *filtered* according to the test parameters of a test objective (e.g. stress location, stress direction). 'Filtered' in a sense that, for example, given a set of test parameters, only those SDs messages are included in a TM, which comply with the criteria specified in the test parameters. For example, SD messages going through a specified network are inculded. We use the SCAPS test model (Section 8.3) to build an input file (Figure 76) corresponding to the above test objective. This input file is based on the format presented in Section 7.1.3.

```
--ISDSs
4
ISDS1 4 OM_ON OM_QC DSPS_ON DSPS_QC
ISDS2 3 OM_ON OM_QC PRNF
ISDS3 3 DSPS_ON DSPS_QC OC
ISDS4 2 PRNF OC
--SDs
6
OM_ON 1 1 p11
OM_QC 1 1 p21
OC 1 4 p31 p32 p33 p34
DSPS_ON 1 1 p41
DSPS_QC 1 1 p51
PRNF 1 4 p61 p62 p63 p64
--SD_Arrival_Patterns
OM_ON periodic 24 1
OM_QC periodic 21 1
OC no_arrival_pattern
DSPS_ON periodic 17 2
DSPS_QC periodic 14 2
PRNF no_arrival_pattern
--DCCFPs
p11 7 ( 5 2.8 ) ( 6 2.8 ) ( 7 2.8 ) ( 8 2.8 ) ( 9 2.8 ) ( 10 2.8 ) ( 11 2.8 )
p21 6 ( 3 2.66 ) ( 4 2.66 ) ( 5 2.66 ) ( 6 2.66 ) ( 7 2.66 ) ( 8 2.66 )
p31 8 ( 2 0.33 ) ( 3 0.61 ) ( 4 0.61 ) ( 5 0.61 ) ( 6 0.61 ) ( 7 0.61 ) ( 8 0.33 ) ( 9 0.33 )
p32 7  ( 3 0.28 ) ( 4 0.28 ) ( 5 0.28 ) ( 6 0.28 ) ( 7 0.28 ) ( 8 0.28 ) ( 9 0.28 )
p33 6 ( 2 0.33 ) ( 3 0.33 ) ( 4 0.33 ) ( 5 0.33 ) ( 6 0.33 ) ( 7 0.33 )
p34 0
p41 5 ( 6 3 ) ( 7 3 ) ( 8 3 ) ( 9 3 ) ( 10 3 )
p51 4 ( 5 3 ) ( 6 3 ) ( 7 3 ) ( 8 3 )
p61 7 ( 3 0.14 ) ( 4 0.3 ) ( 5 0.3 ) ( 6 0.3 ) ( 7 0.3 ) ( 8 0.3 ) ( 9 0.3 )
p62 7 ( 3 0.14 ) ( 4 0.14 ) ( 5 0.14 ) ( 6 0.14 ) ( 7 0.14 ) ( 8 0.14 ) ( 9 0.14 )
p63 6 ( 4 0.16 ) ( 5 0.16 ) ( 6 0.16 ) ( 7 0.16 ) ( 8 0.16 ) ( 9 0.16 )
p64 0
--GASearchTimeRange
200
```

Figure 76- Input File containing SCAPS Test Model for a GASTT Test Objective.

The first block (--ISDSs) of the input file lists SCAPS' four ISDSs. SD data descriptions (--SDs) then follow. We explained earlier that only one copy of each SD will be triggered in SCAPS at a single time instance (as denoted the parameters set to '1' after SDs' names in *--SDs* block). Numbers and names of DCCFPs for each SD (e.g. DCCFP *p11* for *OM_ON*) are taken from Section 8.3.3. Arrival pattern data are extracted from SDs in Section 8.1.4. Note that, for brevity, the time unit is assumed to be 100 ms. For example, the first DTUPP of DCCFP *p11*, states that it entails 2.8 units of traffic in time instant 500 ms (5x100ms). Finally, the GA time search range has been set to 200 time units (20,000 ms). This value was selected using the heuristics in Section 6.5.4.

Our experimentation of different values for the GA time search range also confirmed that 200 time units is a suitable value. To explain how we come to this conclusion, a timing diagram showing the relationship between ATSs of SCAPS SDs and their possible execution durations, shown as data series *d(sd_name)*, in 50 time units is shown in Figure 77.

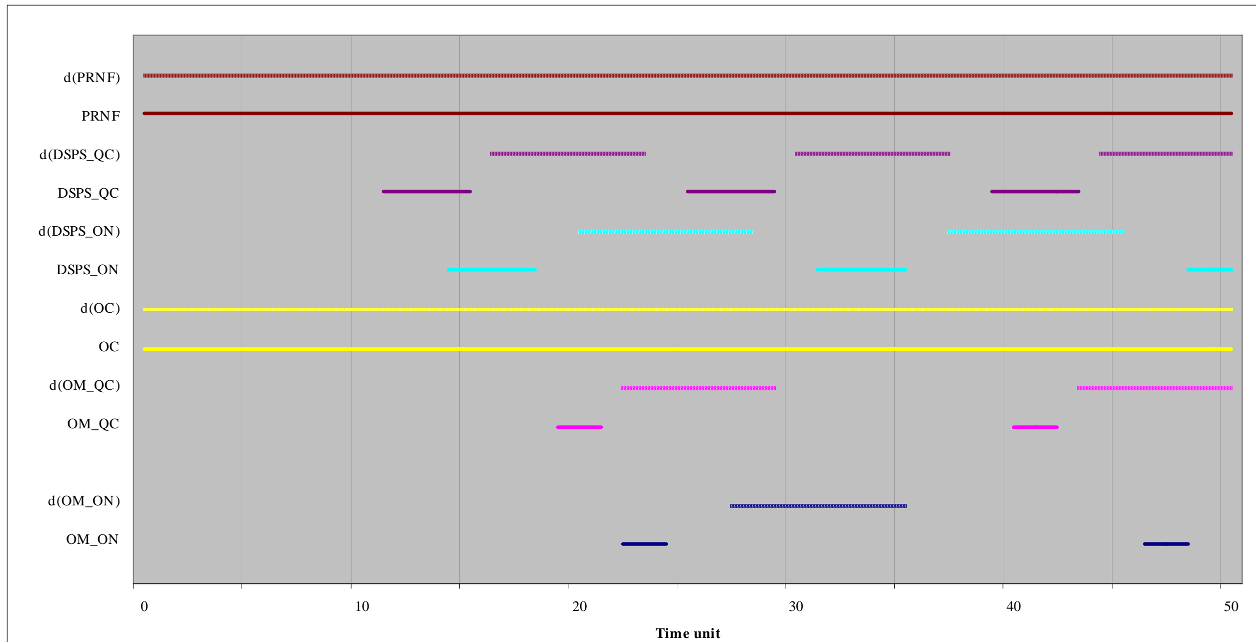


Figure 77-Relationship between ATSs of SCAPS SDs and their execution durations, *d(sd_name)*, to each other in 50 time units.

For example, as the period and deviation values of the periodic AP for SD *OM_ON* are 24 and 1 time units, its ATS has been depicted as ATIs: [23, 25], [47, 49] and etc. The execution duration of a SD is the longest interval made of the difference between the minimum and maximum timing values of a DCCFP of the SD. For example, referring to the input file in Figure 76, the minimum and maximum timing values in the DCCFP *p11* of SD *OM_ON* are 5 and 11 time units. Therefore, considering an ATI such as [23, 25] in which an execution of SD *OM_ON* can be started, the distributed messages of such an execution can take place in interval of [23+5=28, 25+11=36], as depicted in Figure 77. The rationale of analyzing the ATSs and execution durations of SDs is to find a suitable GA time search range. Since SDs *OC* and *PRNF* have no arrival patterns, they can be triggered in any time instance.

As we can easily see in Figure 77, there will not be any chance for our GA algorithm (GASTT) to find a time instance when the execution durations of all six SDs overlap, and thus generating a stress test schedule to entail maximum traffic on the network. This is because the intersection of all execution durations in Figure 77 is simply null. This is why 50 time units is not a suitable GA time search range. We now discuss how the overlapping between ATSs and SD execution durations change by increasing the search range from 50 to, say, 200 time units (Figure 78).

When the search range is increased to 200 time units, the situation changes and as it can be calculated (and seen in Figure 78, the six execution durations will overlap in time instances: 106-107, 128-130, and 173-176. Therefore, there will be chances of overlapped executions of SDs in the random schedules generated during GASTT's operation. This will, in turn, enable GASTT to generate GA individuals which have higher fitness values (entailed traffic values). Increasing the search range to values more than 200 will not increase GASTT's chances in generating better individuals (better stress test requirements), since as long as there is overlapping time among the six execution durations, there are chances to find such test requirements. However, an increase in the search range will deteriorate our GA's performance, since it will take longer time for the GA to converge to a maximum plateau. This is because the selection of random start times for DCCFPs will be sparser (compared to when the range is 200) and GA has to iterate through more generations to settle on a stable maximum plateau (Section 6.5.4). Refer to Sections 7.2.9 for the impacts of variations in GA maximum search time on our GA's performance.

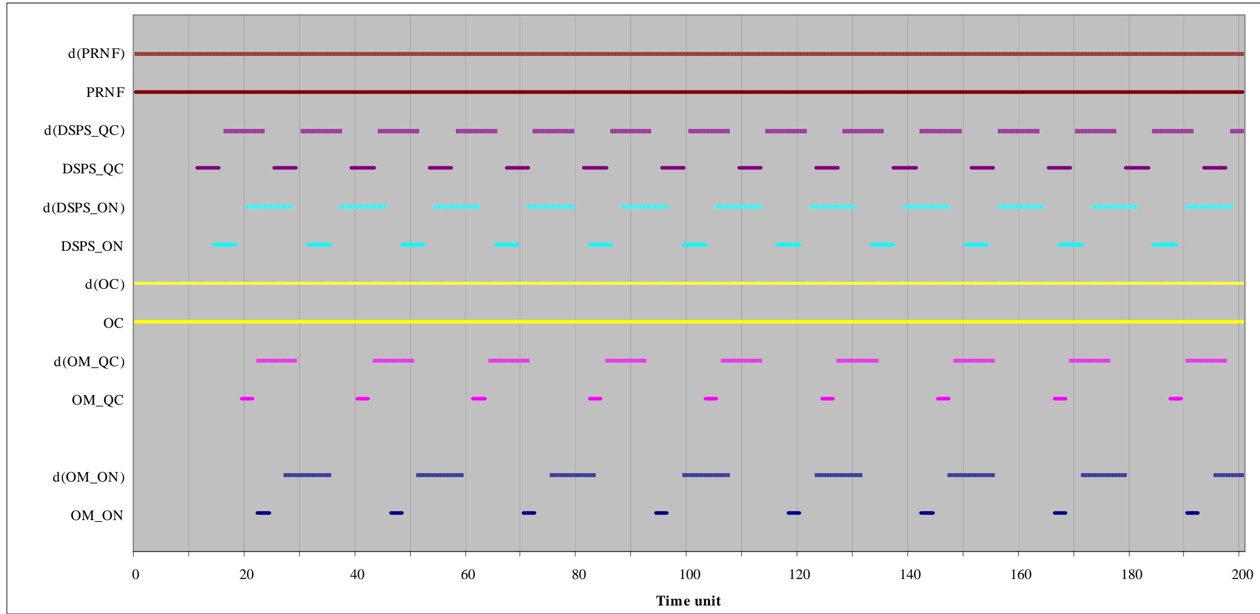
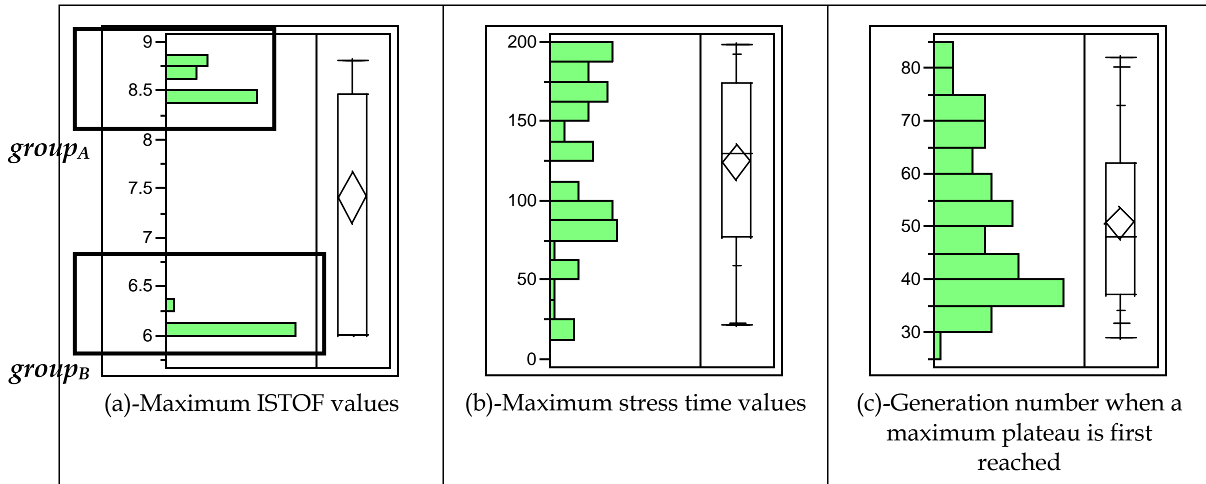


Figure 78--Relationship between ATSs of SCAPS SDs and their execution durations, *d(sd_name)*, to each other in 200 time units.

8.3.6.2 GA Execution and the Repeatability of Results

Similar to discussions in Section 7.2, since GARUS is based on GAs, the stability and repeatability of results across multiple runs need to be investigated as GAs are by definition a heuristic. Therefore, GARUS was executed 100 times with the above input test file. Histograms of maximum ISTOF values, maximum stress time values, and maximum plateau generation number are depicted in Figure 79.



(a)-Maximum ISTOF values

(b)-Maximum stress time values

(c)-Generation number when a maximum plateau is first reached

Figure 79-Histograms of 100 GARUS Outputs for a SCAPS Test Objective.

Each execution had the duration of 326 ms on average. Therefore, running GARUS for 100 times was not a practical problem from a time standpoint. As we can see in Figure 79-(a), 100 runs of the test model generated five different ISTOF values. The corresponding maximum stress time values are spanned in the range of [21…198] time units (each time unit=100 ms). As the histogram in Figure 79-(c) shows, GARUS was able to converge to a maximum plateau in 29 to 82 generations (48 on average) across the 100 runs.

We now discuss the practical implications of multiple runs of GARUS to get stable results. For the particular case study in this chapter, as it can be easily seen in Figure 79-(a), 100 runs of GARUS has generated mainly two groups of outputs: a group with maximum ISTOF values of between 8.25 and 9 units of traffic ($group_A$ in Figure 79-(a)), and the one with values between 6 and 6.5 ($group_B$ in Figure 79-(a)). Obviously, the goal of using GARUS is to find stress test requirements which have the highest possible ISTOF values. Thus, the strategy is to run GARUS for multiple times and choose a test requirement with the highest ISTOF value across all runs.

The practical implication of multiple runs to achieve a test requirement with the highest ISTOF value is to predict the minimum number of times GARUS should be executed to yield an output with an ISTOF value in $group_A$ in Figure 79-(a). Such an analysis can be performed by using the probability distributions of the above two groups of maximum ISTOF values in the histogram of Figure 79-(a). 54 and 46 (of 100) values in the histogram belong to $group_A$ and $group_B$, respectively. Thus, in a sample population of 100 GARUS outputs, the probabilities that an output belongs to $group_A$ or $group_B$ are $p(group_A)$=0.54 and $p(group_B)$=0.46, respectively.

The two outcomes of an output being in $group_A$ or $group_B$ are two *mutually-exclusive* events[1]. Thus, to predict the minimum number of times GARUS should be executed to yield an output with an ISTOF value in $group_A$, we can use the following probability formula for two independent events:

> For two mutually-exclusive events *A* and *B* with probabilities p_A and p_B, the probability that *A* occurs at least once in a series of *n* samples (runs) is:
>
> $$p(\text{event A occurs at least once in a series of n samples})=1- {p_B}^{n-1} p_A$$

Substituting $group_A$ and $group_B$ probabilities in the above formula will yield us:

$$p(\text{a test requirement with an ISTOF value in } group_A \text{ is yielded in a series of n runs of GARUS})=$$
$$1 - p(\,group_B\,)^{n-1}\, p(\,group_A\,) = 1-(\,0.46\,)^{n-1}(\,0.54\,)$$

The above probability function is drawn in Figure 80. The probability values (y-axis) are shown in linear scale in Figure 80-(a). Figure 80-(b) depicts a zoom-out of the curve in Figure 80-(a) for *n*=0…10.

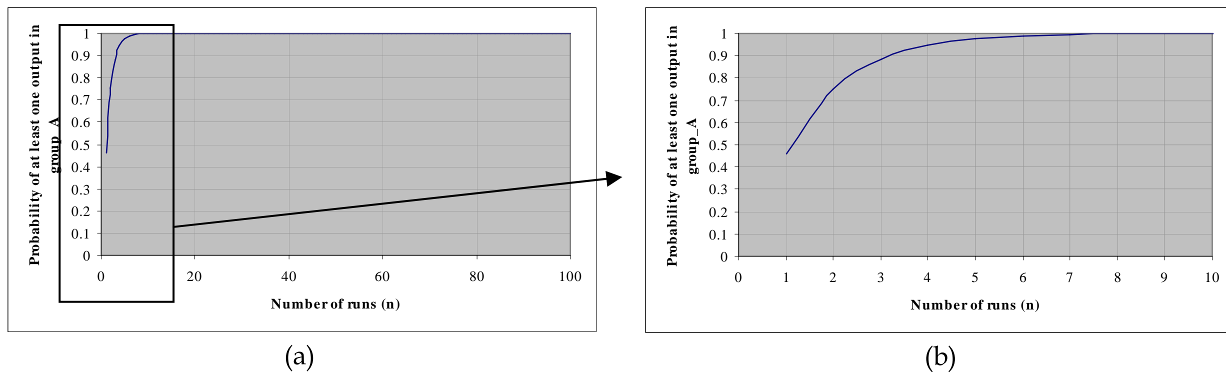


Figure 80- Probability of the event that at least one test requirement with an ISTOF value in *group_A* is yielded in a series of *n* runs of GARUS for SCAPS.

As it can be easily seen in Figure 80, the probability values increases exponentially by increasing number of runs (*n*) and converges to 1 very quickly in values above *n*=6. Therefore, it can be said that one can achieve a good stress test requirement (with a value in $group_A$) by running GARUS with SCAPS test model for about 6

---

[1] In probability theory, two events A and B are said to be mutually-exclusive if event A happens, then event B cannot, or vice-versa.

times and selecting the best output (with highest ISTOF value). Similar probabilistic analysis can be done when using GARUS to generate test requirements for other SUTs.

8.3.6.3 Stress Test Requirements

As discussed above, 100 runs of the test model by GARUS generated five different ISTOF values (Figure 79-(a)). Five of the test requirements reported by GARUS for those five different ISTOF values are shown in Figure 81.

```
SD DCCFP start time
---- ---- ----------
OM_ON none
OM_QC none
OC p34148
DSPS_ON  p41155
DSPS_QC  p51156
PRNF  none
```

(a)- *TR1* (ISTOF=6)

```
SD DCCFP start time
---- ---- ----------
OM_ON none
OM_QC none
OC p32149
DSPS_ON  p41151
DSPS_QC  p51152
PRNF  none
```

(b)- *TR2* (ISTOF=6.28)

```
SD DCCFP start time
---- ---- ----------
OM_ON p1123
OM_QC p2121
OC none
DSPS_ON  p4119
DSPS_QC  p5112
PRNF  none
```

(c)- *TR3* (ISTOF=8.46)

```
SD DCCFP start time
---- ---- ----------
OM_ON p11119
OM_QC p21188
OC none
DSPS_ON  p41185
DSPS_QC  p51183
PRNF  none
```

(a)- *TR4* (ISTOF=8.66)

```
SD DCCFP start time
---- ---- ----------
OM_ON p1171
OM_QC p2185
OC none
DSPS_ON  p4170
DSPS_QC  p5168
PRNF  none
```

(b)- *TR5* (ISTOF=8.8)

Figure 81-Five Different Test Requirements (TR) generated by GARUS for a SCAPS Test Objective.

As we can see, some of the test requirements (TR) generated by GARUS have different combinations of DCCFPs (e.g. *TR1*, *TR2* and *TR3*), while some have the same combinations of DCCFPs, but with different schedules. For example, in *TR3*, *TR4* and *TR5,* DCCFPs *p11*, *p21*, *p41* and *p51* have been selected with different start times.

Since our stress test goal is to maximize the amount of traffic, we choose *TR5*, and stress test SCAPS with its corresponding test case. The test requirement is to trigger DCCFPs *p11*, *p21*, *p41* and *p51* from SDs *OM_ON*, *OM_QC*, *DSPS_ON* and *DSPS_QC* in time units 71, 85, 70 and 68 respectively.

8.4 Stress Test Results

In this section we compare the durations for SDs *D* and *E* presented in Figure 7 when running Operational Profile Tests (OPT) and Stress Tests (ST). We considered OPTs to be a useful baseline of comparison as test cases are derived from the operational profile of SCAPS and therefore represent a “typical” situation in which the system can be exercised. This is a common testing practice to assess a system based on its expected usage in the field [39]. To derive operational profile test cases, we took into account SCAPS business logic in the context of SCADA-based power systems. For example, overload and power failure situations are expected to be fairly rare in a power grid [53].

Recall that we also modeled a HRT constraint in the MIOD of SCAPS in Figure 7. It specifies the maximum acceptable value for the durations of SDs *D* and *E*: they should be less than 1,300 ms (milliseconds). Figure 82 shows the observed values of this duration by running 500 Operational Profile Tests (OPT) and 500 Stress Tests (ST). The x-axis shows the test type and the Y-axis the duration in milliseconds. The quantile regions and the histograms of the two distributions are also depicted, and reported in Table 25.

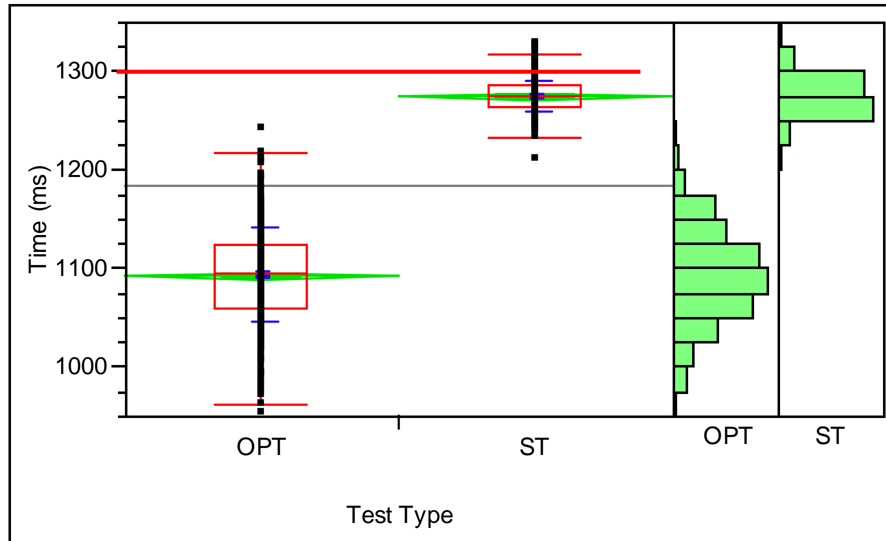


Figure 82. Duration of a hard RT constraint by running Operational Profile Tests (OPT) and Stress Tests (ST).

Table 25. Quantiles of the distribution in Figure 82. Values are in milliseconds.

Level	Min.	10%	25%	Median	75%	90%	Max.
OPT	953	1029	1059	1094	1125	1156	1241
ST	1211	1254	1263	1274	1285	1295	1327

Due to the indeterminism of distributed environments (different message transmission times due, to, among many reasons, different delay times in network links and routers, and different load situations in nodes or networks), the duration of distributed messages can be different across different executions, hence the variance in the distributions of Figure 82. The 1,300 ms deadline (HRT constraint) is illustrated by a horizontal bold line in Figure 82 and all OPT test executions satisfy it. In contrast, it is violated in almost 7.8% (39/500) of ST stress test cases. Furthermore, the differences in average and median value between OPT and ST distributions are large too, illustrating the ability of ST test cases to stress the system.

9 Conclusions and Future Works

This report presents a model-driven, stress test methodology aimed at increasing chances of discovering faults related to network traffic in distributed systems. The technique uses a UML 2.0 model of a system, augmented with timing information, as an input model. Such input model was carefully defined so as to be adequate for our objectives but also as practical as possible from the standpoint of modelers. Our input model includes, in addition to the standard class and sequence diagrams, (1) a System Context diagram that describes actors interacting with the system under test and their expected numbers at run-time, (2) a Network Deployment Diagram (following the UML deployment diagram notation) that describes the distributed architecture in terms of system nodes and networks, and (3) a Modified Interaction Overview Diagram (following the UML 2.0 interaction overview diagram notation) that describes execution constraints between sequence diagrams. Our stress testing methodology relies on a careful identification of the control flow 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 to the largest extent possible.

The current work is an extended version of the work in [23], where we considered distributed systems in which external or internal events do *not* exhibit arrival patterns (e.g., periods). The technique in the current work takes into account different types of arrival patterns for events that are common in DRTSs. Such patterns impose constraints on the time instant when interactions between distributed objects can take place.

Tool support was developed based on a specifically tailored genetic algorithm (GA) to automatically generate test requirements based on the above input model. GAs being heuristics, a careful analysis showed that the tool was able to converge efficiently towards test requirements that significantly stressed the system and do so relatively consistently across different executions (repeatability).

Using the specification of a real-world distributed system as a case study, we designed and implemented a prototype distributed system and reported the results of applying our stress test methodology to it. We discussed its effectiveness in detecting violation of a hard real-time constraint when compared to test cases based on an operational profile of the system usage. Our first results are promising as they clearly show our stress test technique is able to identify constraint violations whereas operational profile test cases are not. This suggests that our stress test cases can help increase the probability of exhibiting network traffic-related faults in distributed systems.

Some of our future works include: (1) Experimenting with the other variants of stress testing techniques; (2) Generalizing the methodology to other distributed-type faults, such as distributed unavailability of networks and nodes, and other resources such as CPU, memory and database.

ACKNOWLEDGEMENTS

This work was in part supported by Siemens Corporate Research, Princeton, NJ, and a Canada Research Chair (CRC) grant. Lionel Briand and Yvan Labiche were further supported by NSERC operational grants.

REFERENCES

[1] J. F. Allen, "Maintaining Knowledge about Temporal Intervals," *Communications of the ACM*, vol. 26, no. 11, pp. 832-843, 1983.

[2] M. J. Atallah, *Handbook of Algorithms and Theory of Computation*: CRC (Chemical Rubber Company) Press, 1999.

[3] A. Avritzer and E. J. Weyuker, "The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software," *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 705-716, 1995.

[4] T. Back, "Towards a Practice of Autonomous Systems," Proceeding of European Conference on Artificial Life, pp. 263-271, 1992.

[5] R. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*: Addison-Wesley, 1999.

[6] J. P. Bowen, K. Bogdanov, J. Clark, M. Harman, R. Hierons, and P. Krause, "FORTEST: Formal Methods and Testing," Proc. of Int. Computer Software and Applications Conf., pp. 91-101, 2002.

[7] L. C. Briand, Y. Labiche, and M. Shousha, "Using Genetic Algorithms for Early Schedulability Analysis and Stress Testing in Real-Time Systems," *Forthcoming in the Journal of Genetic Programming and Evolvable Machines, Springer*, 2006.

[8] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns, and Java*, 2nd Edition ed: Prentice Hall, 2003.

[9] J. Brunton, G. Digby, and A. Doherty, "Design and Operational Philosophy for a Metro Power Network SCADA System," Proceeding of International Conference on Power System Control and Management, pp. 176-180, 1996.

[10] R. J. A. Buhr, "Use Case Maps as Architectural Entities for Complex Systems," *IEEE Transactions on Software Engineering*, vol. 24, no. 12, 1998.

[11] BWI Co., "ElipseSCADA," in *http://www.bwi.com/proot/2775*, 2004.

[12] E.-K. Chan and H. Ebenhoh, "The Implementation and Evolution of a SCADA System for a Large Distribution Network," *IEEE Transactions on Power Systems*, vol. 7, no. 1, pp. 320-326, 1992.

[13] P. Chardaire, A. Kapsalis, J. W. Mann, V. J. Rayward-Smith, and G. D. Smith, "Applications of Genetic Algorithms in Telecommunications," Proceeding of Applications of Neural Networks to Telecommunications, pp. 290-299, 1995.

[14] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. GilChrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development - The Fusion Method*: Prentice Hall, 1994.

[15] Z. Constantinescu, P. Petrovic, A. Pedersen, D. Federici, and J. Campos, "QADPZ (Quite Advanced Distributed Parallel Zystem)," in *http://qadpz.sourceforge.net*, 2003.

[16] A. Daneels and W. Salter, "What is SCADA?," Proceeding of International Conference on Accelerator and Large Experimental Physics Control Systems, pp. 39-343, 1999.

[17] K. De Jong, "Learning with Genetic Algorithms: An Overview," *Machine Learning*, vol. 3, no. 3, pp. 121-138, 1988.

[18] 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.

[19] Y. Ebata, H. Hayashi, Y. Hasegawa, S. Komatsu, and K. Suzuki, "Development of the Intranet-based SCADA for Power System," Proceeding of IEEE Power Engineering Society Winter Meeting, pp. 1656-1661, 2000.

[20] European Information Society Technologies, "Component Based Open Source Architecture for Distributed Telecom Applications," in *http://coach.objectweb.org*, 2003.

[21] V. Garousi, L. Briand, and Y. Labiche, "Control Flow Analysis of UML 2.0 Sequence Diagrams," Proceeding of European Conference on Model Driven Architecture-Foundations and Applications, LNCS 3748, pp. 160-174, 2005.

[22] V. Garousi, L. Briand, and Y. Labiche, "Traffic-aware Stress Testing of Distributed Real-Time Systems based on UML Models using Genetic Algorithms," Technical Report SCE-06-09, Carleton University, http://www.sce.carleton.ca/squall/pubs/tech_report/TR_SCE-06-09.pdf, 2006.

[23] V. Garousi, L. Briand, and Y. Labiche, "Traffic-aware Stress Testing of Distributed Systems based on UML Models," Proceeding (to appear) of International Conference on Software Engineering, 2006.

[24] H. Gomaa, *Designing Concurrent, Distributed, and Real-Time Applications with UML*: Addison-Wesley, 2000.

[25] J. J. Grefenstette and H. G. Cobb, "Genetic Algorithms for Tracking Changing Environments," Proceeding of International Conference on Genetic Algorithms, pp. 523-530, 1993.

[26] R. L. Haupt and S. E. Haupt, *Practical Genetic Algorithms*: Wiley-Interscience, 1998.

[27] R. Horst and P.M. Pardalos (eds.), *Handbook of Global Optimization*: Kluwer, Dordrecht, 1995.

[28] B. F. Jones, H.-H. Sthamer, and D. E. Eyres, "Automatic Structural Testing using Genetic Algorithms," *Software Engineering Journal*, vol. 11, no. 5, pp. 299–306, 1996.

[29] 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," Proceedings of International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies, pp. 629-633, 2000.

[30] R. Kuhn, "Sources of Failure in the Public Switched Telephone Network," *IEEE Computer*, vol. 30, no. 4, pp. 31-36, 1997.

[31] J. Lahtinen, P. M. Silander, and H. Tirri, "Empirical Comparison of Stochastic Algorithms," Proceedings of Nordic Workshop on Genetic Algorithms and their Applications, pp. 45-60, 1996.

[32] S. J. Louis and G. J. E. Rawlins, "Predicting Convergence Time for Genetic Algorithms," Technical Report 370, Computer Science Department, Indiana University 1993.

[33] S. Mackay, E. Wright, and J. Park, *Practical Data Communications for Instrumentation and Control*: Newnes, June, 2003.

[34] S. W. Mahfoud and D. E. Goldberg, "Parallel Recombinative Simulated Annealing: A Genetic Algorithm," *Journal on Parallel Computing*, vol. 21, no. 1, pp. 1-28, 1995.

[35] S. Y. Mahfouz, "Design Optimization of Structural Steel Work," Ph.D. Thesis, Department of Civil and Environmental Engineering, University of Bradford, 1999.

[36] 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," Proceedings of International Conference and Exhibition on Electricity Distribution, pp. 172-172, 2001.

[37] 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.

[38] H. Mühlenbein, "Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization," Proceedings of International Conference on Genetic algorithms, pp. 416-421, 1989.

[39] J. D. Musa, "The Operational Profile in Software Reliability Engineering: An Overview," Proc. Int. Symp. on Software Reliability Engineering, 1992.

[40] C. Nebut, F. Fleurey, Y. L. Traon, and J.-M. Jézéquel, "Requirements by Contracts allow Automated System Testing," Proceedings of International Symposium on Software Reliability Engineering, pp. 85-96, 2003.

[41] Object Management Group (OMG), "OCL 2.0 Specification," 2005.

[42] Object Management Group (OMG), "UML 2.0 Superstructure Specification," 2005.

[43] Object Management Group (OMG), "UML Profile for Schedulability, Performance, and Time (v1.0)," 2003.

[44] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data Generation using Genetic Algorithms," *Journal of Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 263–282, 1999.

[45] M. A. Pawlowsky, "Crossover Operators," Practical Handbook of Genetic Algorithms Applications, L. Chambers Ed., pp. 101-114, 1995.

[46] T. Pender, *UML Bible*: Wiley, Sept. 2003.

[47] 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 International Conference on Genetic algorithms, pp. 51-60, 1989.

[48] T. Seki, T. Tsuchiya, T. Tanaka, H. Watanabe, and T. Seki, "Network Integrated Supervisory Control for Power Systems based on Distributed Objects," Proceedings of International Symposium on Applied Computing, pp. 620-626, 2000.

[49] J. E. Smith and T. C. Fogarty, "Adaptively Parameterized Evolutionary Systems: Self Adaptive Recombination and Mutation in a Genetic Algorithm," *Proceeding of International Conference on Parallel Problem Solving From Nature*, pp. 441-450, 1996.

[50] 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," Proceedings of IEEE Power Engineering Society Winter Meeting, pp. 695-700, 2002.

[51] 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.

[52] D. Thierens, "On the Scalability of Simple Genetic Algorithms," Report 1999-48. Information and Computing Sciences, Utrecht University, The Netherlands, 1999.

[53] 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.

[54] N. Tracey, J. Clark, and K. Mander, "Automated Program Flaw finding using Simulated Annealing," *ACM SIGSOFT Software Engineering Notes*, vol. 23, no. 2, pp. 73-81, 1998.

[55] N. Tracey, J. Clark, and K. Mander, "The way Forward for Unifying Dynamic Test-case Ceneration: The Optimization-based Approach," Proc. of Int. Workshop on Dependable Computing and its Applications, pp. 169-180, 1998.

[56] 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.

[57] M. Wall, "GAlib: A C++ Library of Genetic Algorithm Components," Documentation version 2.4, Massachusetts Institute of Technology 1996.

[58] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary Test Environment for Automatic Structural Testing," *Journal of Information and Software Technology, Special issue on Software Engineering using Metaheuristic Innovative Algorithms*, vol. 43, no. 14, pp. 841-854, 2001.

[59] E. Weyuker and F. I. Vokolos, "Experience with Performance Testing of Software Systems: Issues, an Approach and Case Study," *IEEE Transactions on Software Engineering*, vol. 26, no. 12, pp. 1147-1156, 2000.

[60] Wikipedia, "Definition of Controllability," in *http://en.wikipedia.org/wiki/Controllability*, 2005.

[61] Wikipedia, "Definition of Observability," in *http://en.wikipedia.org/wiki/Observability*, Last accessed: Feb. 2006.

[62] C. S. D. Yang, "Identifying Potentially Load Sensitive Code Regions for Stress Testing," Proceedings of the Mid-Atlantic Student Workshop on Programming Languages and Systems, 1996.

[63] J. Zhang and S. C. Cheung, "Automated Test Case Generation for the Stress Testing of Multimedia Systems," *Journal on Software Practice and Experience*, vol. 32, no. 15, pp. 1411-1435, 2002.

Appendix A- Proof of the Formula to Calculate the Unbounded Range Starting Point (URSP) of a Bounded Arrival Pattern

The following is the proof of formula (Equation 6) to calculate the Unbounded Range Starting Point (URSP) of a bounded arrival pattern (AP), given the minimum and maximum inter-arrival times (*minIAT* and *maxIAT*) of the AP The formula is used in Section 6.5.4 for determining a suitable maximum search time for our GA.

Recall from Section 5.5.4 the concept of Accepted Time Intervals (ATI) for a bounded arrival pattern. For example, the gray eclipses in the timing diagram in Figure 11 depict the ATIs of the arrival pattern (*'bounded', (4, ms), (5, ms))*, i.e. m*inIAT=4ms, maxIAT=5ms*. To devise a formula to find the calculate the URSP of a bounded AP, we formalize bounded APs time properties as demonstrated in Figure 83.

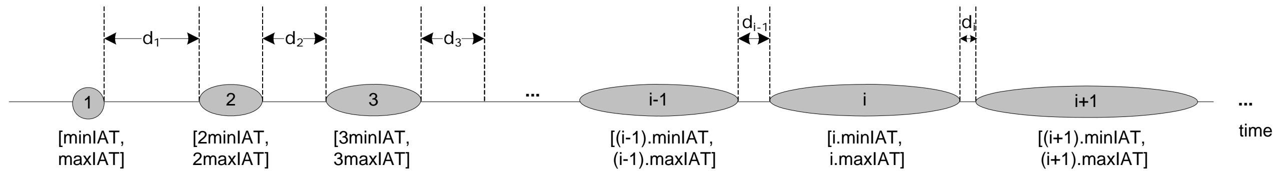


Figure 83-Formalization of bounded APs time properties.

ATIs are indexed and are referred as ATI_i. Recall from Section 5.5.4 (analysis of arrival patterns) that each ATI's start and end times are multiples of the AP's m*inIAT* and *maxIAT*, respectively. For example, the consecutive ATIs of the arrival pattern *('bounded', (4, ms), (5, ms))* are: *[4 ms, 5 ms], [8 ms, 10 ms], [12 ms, 15 ms], [16 ms, 20 ms]*, and so on. In the parametric form, consecutive ATIs are shown in Figure 83 as: *[*m*inIAT, maxIAT]*, *[*2m*inIAT, 2maxIAT]*, *[3*m*inIAT, 3maxIAT]*, *[k.*m*inIAT, k.maxIAT]*, and so on. d_i denotes the closest distance between two neighboring ATIs ATI_i and ATI_{i+1}. It is obvious that:

$$d_i = (i+1).minIAT - i.maxIAT$$

The URSP of a bounded AP appears when two consecutive ATIs overlap[1], i.e., the start time of the next ATI is smaller than or equal to the end time of the current ATI. This, in turn, entails than $d_k \leq 0$ in such a case. k here denotes the index of the ATI whose start time is the URSP of the bounded AP. Such a situation is visualized in Figure 84.

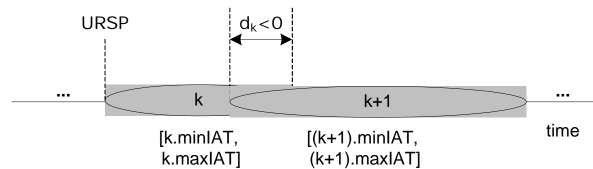


Figure 84-Two overlapping ATIs.

Therefore, in order to find the value of URSP, we should find the *k*-th ATI's start time such that $d_k \leq 0$. To prove that for every bounded AP, there exists a URSP, we should prove that there exists a $d_k \leq 0$. This can be proved if we show that the d_i values of a bounded AP are *descending*, i.e., $d_i > d_{i+1}$. Since supposing that d_i will start from a positive value and it is descending, it will at some point be equal to zero or less than zero.

Theorem. The d_i values of a bounded AP are *descending*, i.e., $d_i > d_{i+1}$.
Proof. We follow a proof-by-contradiction approach. Assume to the contrary that $d_i \leq d_{i+1}$. Then:

[1] Two ATIs are said to be *overlapped* if they have at least one common ATP.

$$d_i \leq d_{i+1}$$
$$(i+1)minIAT - i.maxIAT \leq (i+1+1)minIAT - (i+1).maxIAT$$
$$maxIAT \leq minIAT$$

We can see that, after takeing the proof-by-contradiction approach, we have got a conclusion which contradics the assumption of m*inIAT*<*maxIAT* for bounded APs (Section 5.5.4). This means the proof of the main theorem, i.e., the d_i values of a bounded AP are *descending*, i.e., $d_i > d_{i+1}$.

Therefore, in order to find the value of URSP, we should find *k*-th ATI's start time such that $d_k \leq 0$ (the smallest *k*). If we find the index, *k*, of the ATI, URSP can be found easily by *URSP*=*k.minIAT*. The value of *k* can be found as the following:

$$d_k \leq 0$$
$$\Rightarrow (k+1)minIAT - k.maxIAT \leq 0$$
$$\Rightarrow k(maxIAT - minIAT) \geq minIAT$$
$$\Rightarrow k \geq \frac{minIAT}{maxIAT - minIAT}$$
$$\Rightarrow k = \left\lceil \frac{minIAT}{maxIAT - minIAT} \right\rceil \quad \text{(since } k \text{ is an integer.)}$$

Therefore, the URSP of a bounded AP can be calculated by:

$$URSP = \left\lceil \frac{minIAT}{maxIAT - minIAT} \right\rceil .minIAT$$

For example, the URSP of the bounded AP (*'bounded', (4, ms), (5, ms))* is 16 ms which can be verified visually in Figure 11.

$$URSP = \left\lceil \frac{4}{5-4} \right\rceil .4 = 16ms$$