Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 81 (2008) 161–185
The Journal of Systems and Software
www.elsevier.com/locate/jss

Traffic-aware stress testing of distributed real-time systems based on UML models using genetic algorithms

Vahid Garousi [a,*,1], Lionel C. Briand [b], Yvan Labiche [c]

[a] University of Calgary, Department of Electrical and Computer Engineering, Software Engineering Research Group, 2500 University Drive N.W. Calgary, AB T2N 1N4, Canada
[b] Simula Research Laboratory, Department of Software Engineering, Martin Linges v 17, Fornebu, P.O. Box 134, 1325 Lysaker, Norway
[c] Carleton University, Department of Systems and Computer Engineering, Software Quality Engineering Laboratory, 1125 Colonel By Drive, Ottawa, ON, Canada K1S5B6

Available online 7 June 2007

Abstract

This paper 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 plant systems (Tsai et al., 1996). However, the development and testing of such systems is difficult and takes more time than for systems without real-time constraints or distribution (Weyuker and Vokolos, 2000). Furthermore, based on an analysis of sources of failures in the United States Public Switched Telephone Network (PSTN) (Kuhn, 1997), 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

* Corresponding author.
E-mail addresses: vgarousi@ucalgary.ca (V. Garousi), briand@sce.carleton.ca, briand@simula.no (L.C. Briand), labiche@sce.carleton.ca (Y. Labiche).
[1] Vahid Garousi performed the work reported in this paper while being a PhD student at the Software Quality Engineering Lab. (SQUALL) at Carleton University, Ottawa, Canada.

0164-1212/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2007.05.037

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 (Binder, 1999). Since 1997, UML has become the de facto standard for modeling object-oriented software for nearly 70% of IT industry (Pender, 2003). The new version of UML, version 2.0 (Object Management Group (OMG), 2005) offers an improved modeling language compared to UML 1.x versions. Some of the high level improvements are: enhanced architecture modeling and extensibility mechanisms, support for component-based development, and model management (Pender, 2003). 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.[2] 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.

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

The current work is an extended version of the work in Garousi et al. (2006b), 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 Garousi et al., 2006b) 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 *GA-based test Requirement tool for real-time distribUted Systems* (GARUS) 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 (Garousi et al., 2006b; Avritzer and Weyuker, 1995; Briand et al., 2006; Yang, 1996; Zhang and Cheung, 2002) on systematic generation of stress and load test suites for software systems.

Avritzer and Weyuker (1995) 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. (2006) 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 (1996) proposes a technique 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 and Cheung (2002) 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 (Allen, 1983) 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 Garousi et al. (2006b). An important aspect of real-time systems taken into account in the current work, which was left out in Garousi et al. (2006b), 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., Wegener et al., 2001; Tracey et al., 1998a,b; Pargas et al., 1999; Jones et al., 1996) 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[3] 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 Wegener et al. (2001), Tracey et al. (1998a), Pargas et al. (1999) and Jones et al. (1996) aim at generating adequate test data for branch coverage and other white-box testing criteria for a program under test (Bowen et al., 2002). 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 (Bowen et al., 2002). ET techniques are also used for black-box testing. For instance, Tracey et al. (1998b) 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 it, 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 ET 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

[3] 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 (Bowen et al., 2002).

(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 existing 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 Fig. 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 Fig. 1 using a color coding scheme (refer to the legend) between the contributions of the work in Garousi et al. (2006b) and the current article. The stress testing technique in Garousi et al. (2006b) 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 by the tester) are not addressed in this paper 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 Fig. 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 Fig. 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 (Garousi et al., 2006a). 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

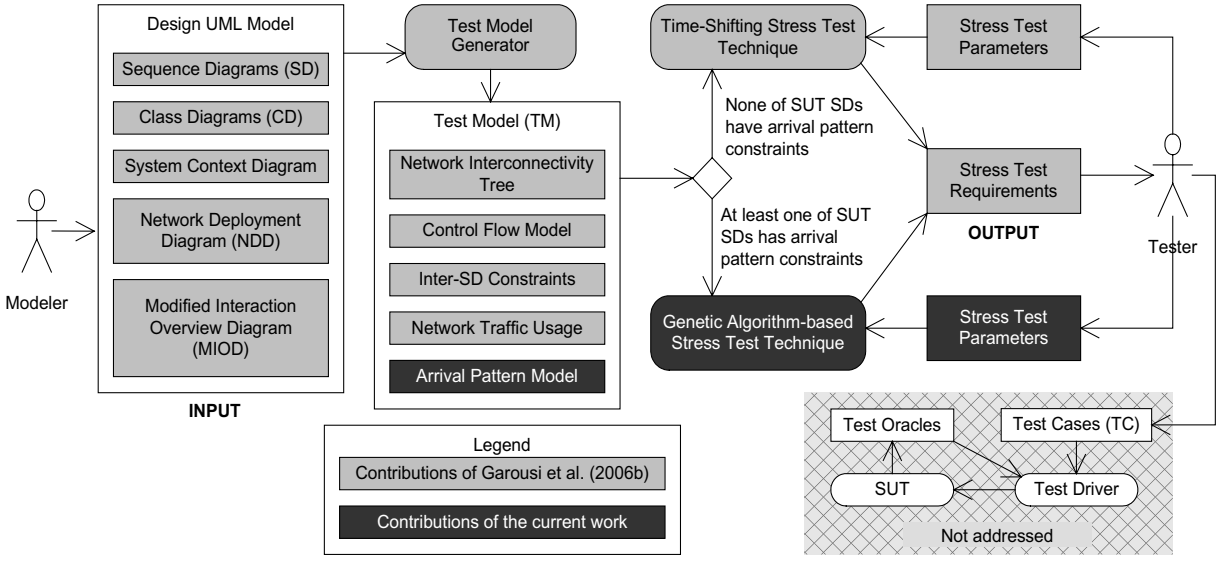


Fig. 1. An overview of our model-based stress test methodology.

time interval, *Int*); and (4) a stress test *type* (*DT* for maximizing amount of data traffic, or *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).

4. Input system models

The assumed input system models for the stress test methodologies in this article and Garousi et al. (2006b) 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) (Object Management Group, 2003) in SDs. Thus, we present in Section 4.1 only an overview of the assumed input system models. Interested reader can refer to Garousi et al. (2006a,b) 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; Gomaa, 2000). 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 Fig. 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 Fig. 3a 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 $o_{1,3}$).

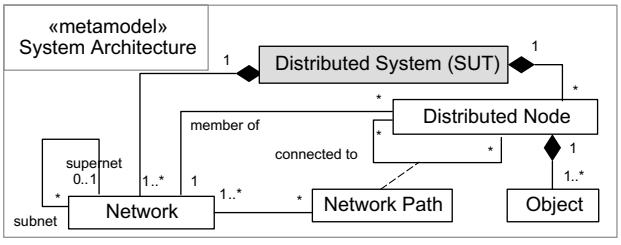


Fig. 2. Metamodel for distributed architectures.

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 $\langle Network_1, SystemNetwork, Network_2\rangle$ in Fig. 3a. In the current work, we consider that there is only one path between two nodes, rather than several paths. Though this is a simplifying assumption, it is still realistic in numerous cases as many proprietary SUT networks (e.g., a distributed controller system of a factory) do not have a complex topology. Considering multiple paths would increase the complexity of our network traffic usage model (Section 5.4) since it would require a detailed analysis of the routing policy used in the network of the SUT.

Modeling a hierarchical set of networks and their interconnectivity is not directly addressed in the UML 2.0 specification (Object Management Group (OMG), 2005). 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 Fig. 3a is modeled by the network deployment diagram (NDD) in Fig. 3b.

4.1.2. Modified interaction overview

The name modified interaction overview diagram (MIOD) comes from the UML 2.0's interaction overview diagram (IOD) (Object Management Group (OMG), 2005). 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., Coleman et al., 1994; Buhr, 1998; Nebut et al., 2003). However, as we discuss in Garousi et al. (2006b), 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 in Fig. 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.

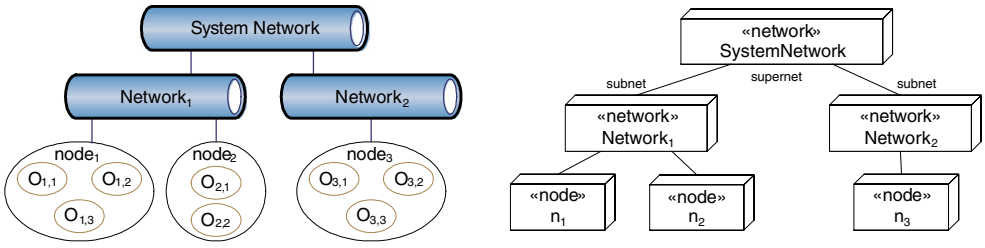


Fig. 3. (a): An example distributed architecture. (b): An example Network Deployment Diagram (NDD).

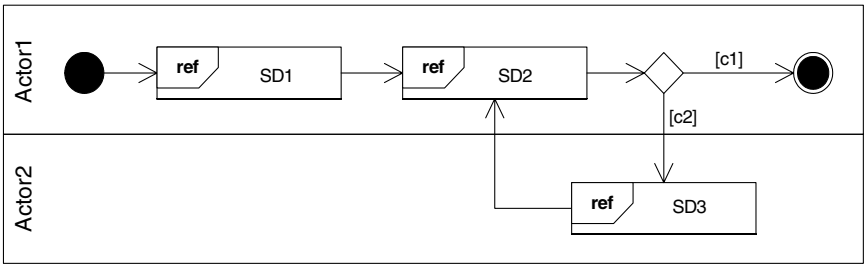


Fig. 4. An example MIOD.

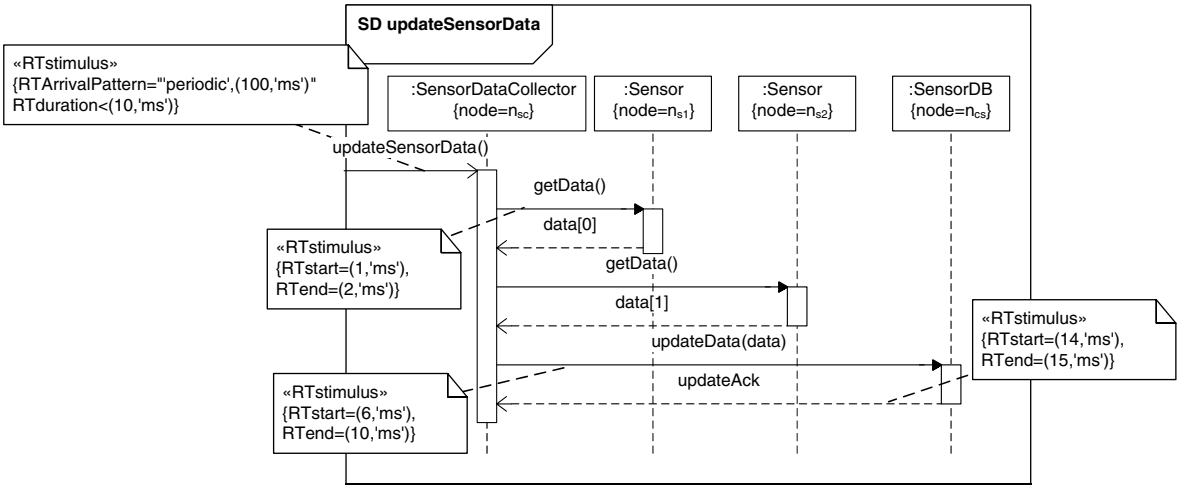


Fig. 5. Example of time modeling using the UML-SPT profile.

4.2. SDs with arrival pattern information

We expect that arrival pattern[4] information is provided in the specifications, e.g., if a SD is going to be triggered by a sensor on a periodic basis, the period value of this periodic arrival pattern should be provided in the system specifications. Such information is supposed to be obtained in the design stage. Methodologies such as the one by Douglass (1999) and the concurrent object modeling and architectural design method (COMET) framework (Gomaa, 2000) discuss ways to obtain arrival pattern information in the design stage. It is therefore reasonable to expect such information as part of the design model of a SUT.

To model arrival pattern information, modelers can use the *RTArrivalPattern* tagged-value which is a modeling construct in the *TimeModel* package of the UML profile for schedulability, performance, and time (UML-SPT) (Object Management Group, 2003). Our technique assumes that the arrival pattern information is given using the *RTArrivalPattern* tagged-value in the UML model of a SUT. As an example, the UML 2.0 SD in Fig. 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

[4] 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.

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. In the UML-SPT, the time origin ($t = 0$) of constraints in a SD is assumed to be the execution start time of the SD.

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 Fig. 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 Fig. 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 UML models composing the TM are discussed in Garousi et al. (2006b) 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 Arrival Patterns Model (Section 5.5).

5.1. Control flow model

In UML 2.0 (Object Management Group (OMG), 2005), 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 Garousi et al. (2005), 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 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 Garousi et al. (2006b), we used the model-based control flow analysis (MBCFA) technique presented in Garousi et al. (2005), which was formalized using meta-modeling and consistency-rules in the object constraint language (OCL) (Object Management Group, 2005). 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., Fig. 7a 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 Garousi et al. (2006a).) 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

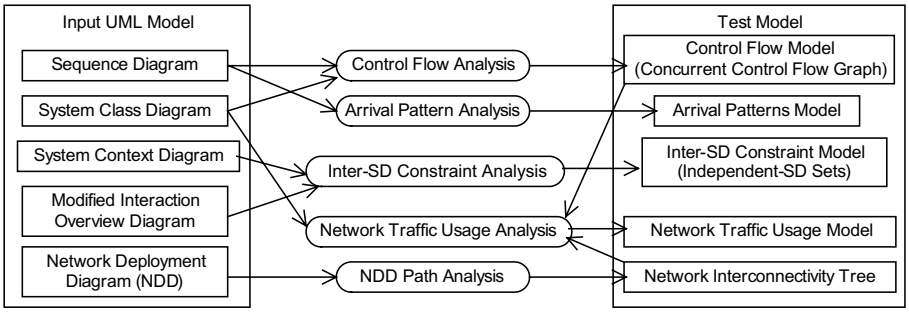


Fig. 6. An overview of how test models are built from input UML models.

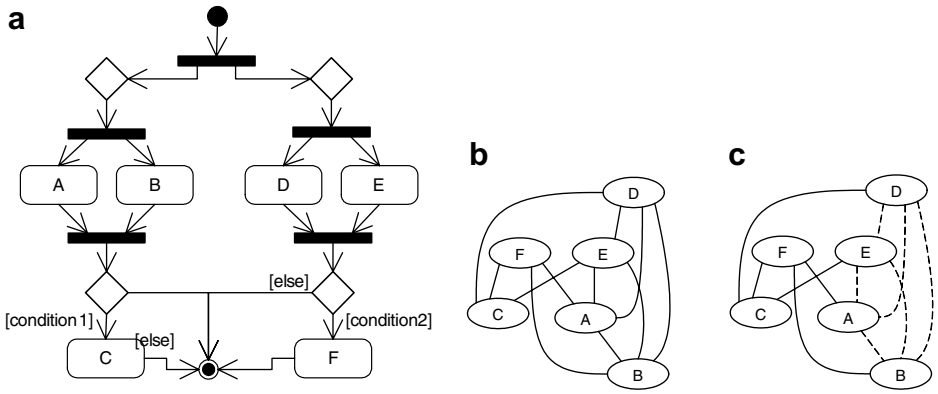


Fig. 7. (a): The MIOD of our case study system. (b) and (c): Deriving independent SD sets of the MIOD.

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.

To derive the set of ISDSs of a MIOD, we use a graph-based approach in which we first build a graph, e.g., Fig. 7b, 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[5] in this graph is an ISDS. Standard graph algorithms can then be used to find those maximal-complete subgraphs. For the MIOD in Fig. 7, four ISDSs are identified (e.g., $ISDS_1$ is illustrated in Fig. 7c):

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

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 path between any two given nodes. For example, 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 Garousi et al. (2006a).) For example, the derivation of the network path between $node_1$ (the sender) and $node_3$ (the receiver) in Fig. 3a 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 estimate traffic usage for messages and 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.[6] 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 (Object Management Group (OMG), 2005), each message annotated with timing information (using the

[5] A maximal complete subgraph (clique) of a graph is a subset of vertices, each pair of which are connected by an edge, that cannot be enlarged by adding any additional vertex from the graph (Moon and Moser, 1965).

[6] 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.

UML-SPT profile Object Management Group, 2003) 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.
- *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 $\langle(p_1, C_1, in/out), \ldots, (p_n, C_n, in/out)\rangle$, where p_i is the ith 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 $\langle(o_1, C_1, in), (o_2, C_2, in)\rangle$. 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 $\langle(var_1 = val_1, C_1), \ldots, (var_n = val_n, C_n)\rangle$, 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 built-in way to separate call and reply messages. Further explanations on this and an approach to distinguish between call and reply messages can be found in (Garousi et al., 2006a).

To formalize our network traffic usage model, we define a *network traffic usage* (NTU) function (Eq. 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.

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

Eq. (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 Fig. 8a. Using these class specifications, we can estimate the size of the message to be 5.8 KB, as illustrated in Fig. 8b, assuming the target programming language is Java (the size of a *char* and a *long* variable are two and eight bytes, respectively).

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 (see Eq. 2).

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 how an APM will help our stress test requirement derivation process (Section 5.5.1) to derive *valid* test requirements, i.e., test requirements which comply with SDs' APs. Types of arrival patterns we consider in this

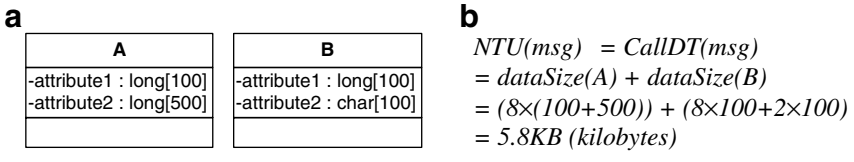


Fig. 8. (a): Two classes with data fields. (b): An example of computation of NTU.

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

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

Eq. (2). Network traffic usage pattern (NTUP) function.

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 therefore 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 Fig. 9. Let us first consider a simple search heuristic (used in our earlier work Garousi et al., 2006b), to be used when there is no arrival pattern: Fig. 9a. 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 Fig. 9a, 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, Fig. 9b. 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 cannot 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 Fig. 9b.

5.5.2. Types of arrival patterns

We also assume that SD APs are modeled using the UML-SPT profile's *RTarrivalPattern* tagged-value (Object Management Group, 2003), such as in Fig. 5. We provide next an overview on the five types of APs in the UML-SPT profile:

- Bounded: An AP where the inter-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 Fig. 10 determines if a DCCFP arrival time satisfies an AP. The function will be used in our stress test derivation technique (Section 6) to select legal schedules for a SD's DTCCFPs. The AP can be any of the fol-

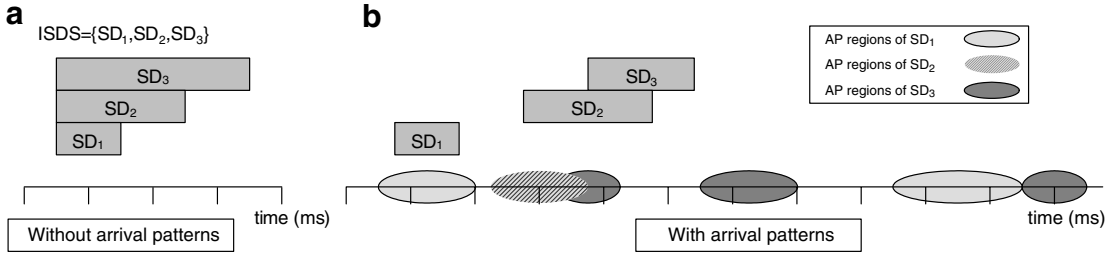


Fig. 9. Impact of arrival patterns on the derivation of test requirements.

```
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 }
```

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

lowing: {'*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 inter-arrival time* (*MinIAT*, *MaxIAT*). We assume that *MinIAT* and *MaxIAT* of a bounded AP cannot 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=3 ms*, is indeed a periodic arrival pattern with *period = 3 ms*. Consider a bounded AP with $MinIAT = 4\ ms$ and $MaxIAT = 5\ ms$. The gray eclipses in Fig. 11 depict the *Accepted Time Interval* (ATI) of the AP, i.e., the time intervals where an AP is satisfied.

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 Fig. 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 [4–5 ms]. If an event arrives in time = 4 ms, according to the fact that $MinIAT = 4\ ms$ and $MaxIAT = 5\ ms$, the next event can arrive in interval [8–9 ms]. Similarly, if an event arrives in time = 5 ms, according to the fact that $MinIAT = 4\ ms$ and $MaxIAT = 5\ ms$, the next event can arrive in interval [9–10 ms]. In a similar fashion, a value between 4 and 5 ms will cause the next arrival time to be in the range [8–10 ms]. Therefore, the second ATI is [8–10 ms]. The next ATIs are [12–15 ms], [16–20 ms], [20–25 ms], [24–30 ms] and so on. Since a bursty 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 and explanations for the rest of the pseudo-code in Fig. 10 can be found in Garousi et al. (2006a).

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 Fig. 12a formalizes the fundamental concepts.

Each SD has an ATS. An ATS is made of several accepted time point (ATP), for irregular and periodic (with no deviation) arrival patterns, *or* several accepted time interval (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 Fig. 12a. Each ATI has a start time and an end time of type *RTtimeValue* (from the UML-SPT), denoting the start and end times of an interval. ATP is of type *RTtimeValue* too. The end time of an ATI can be null, which denotes an ATI which has no upper bound (this is further justified below).

Three ATS examples are illustrated in Fig. 12b, which comply with the metamodel in Fig. 12a. $ATS_{bounded}$ 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*)

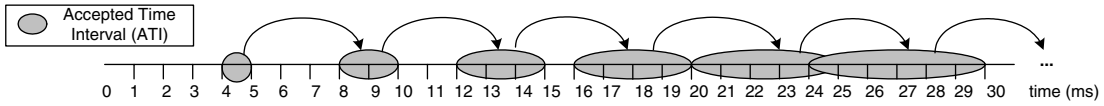
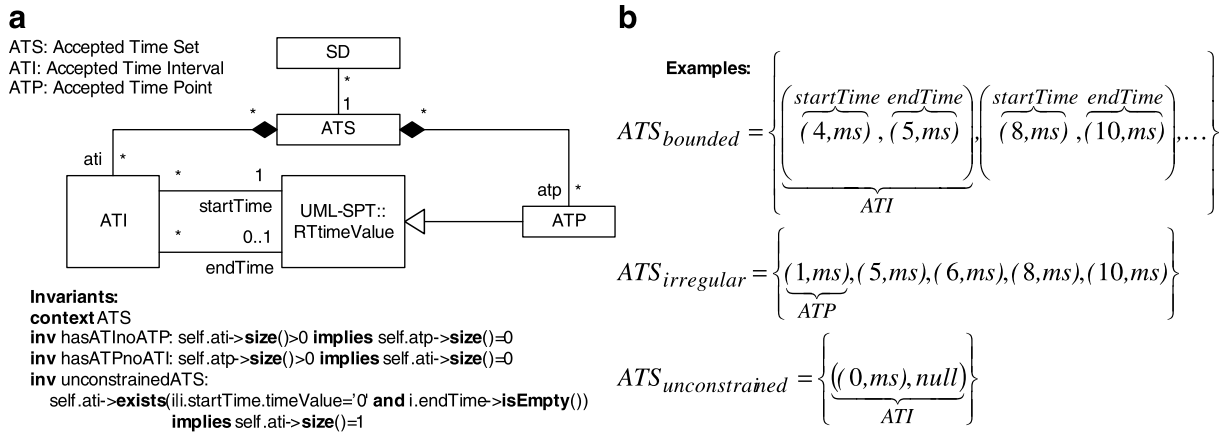


Fig. 11. Accepted Time Intervals (ATI) of a *bounded* arrival pattern ('*bounded*', (4, *ms*), (5, *ms*)), i.e. $MinIAT = 4\ ms$, $MaxIAT = 5\ ms$.



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

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

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

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

in Fig. 12a. 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]3–5 ms[. 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.

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 6.2 formulates the stress test generation problem as an optimization problem. The output stress test requirements format is presented in Section 6.3. 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

When SDs have AP constraints, DCCFPs executions cannot 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).

Fig. 13a shows the NTUPs for three DCCFPs of a given ISDS (the ISDS contains three SDs and each SD has one DCCFP): messages that impose maximum traffic on the network (marked with vertical lines) execute at different time instants. Fig. 13b shows the periods of time (ATSs) during which it is legal to trigger the three DCCFPs. Each of the three DCCFPs has a specific AP (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, when considering AP constraints (Fig. 13b), it is not always possible to achieve the maximum possible stress that one would obtain without considering AP constraints. Considering that ATSs of different SDs in a SUT can be in general very

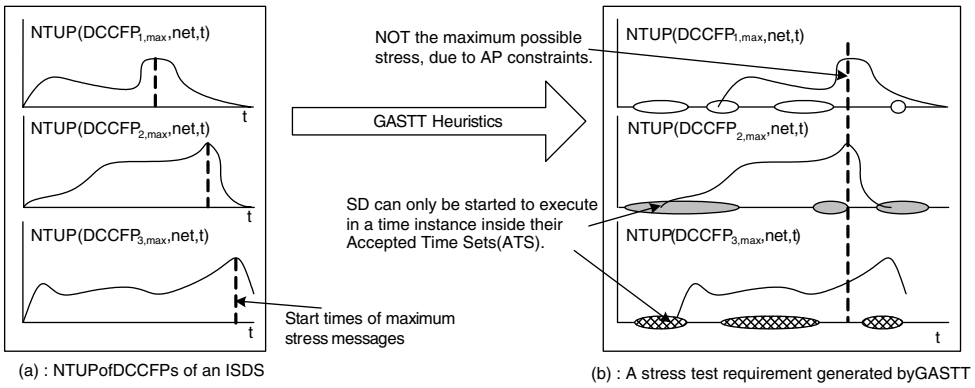


Fig. 13. Heuristic for GASTT.

different from each other, we can also see in Fig. 13 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 therefore not expected to be as simple as the one in Garousi et al. (2006b).

In our work, we assume that a DCCFP execution ends before the next acceptable time instant (ATI or ATP) in the ATS as illustrated in Fig. 13b: the displayed execution of the second DCCFP (starting in the first ATI) ends before the next immediate time instant in the ATS (i.e., the start of the second ATI). We consider this a realistic assumption since in DRTSs designers usually enforce, by specifying appropriate arrival patterns, that this is the case in order to ensure schedulability and safety.

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 Fig. 14.

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 (Gomaa, 2000), 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 ($SD_1, \ldots, SD_n$), a test requirement will be a schedule of a selected set of DCCFPs in the form of: $\langle(\rho_{1max}, \alpha\rho_{1max}), \ldots, (\rho_{nmax}, \alpha\rho_{nmax})\rangle$, where for the ith 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 ith 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" (Horst, 1995), 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 (Thierens et al., 1999) 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 Lahtinen et al. (1996)

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

Fig. 14. Formulating the problem of generating stress test requirements as an optimization problem.

indicate that SA outperforms GAs, while others, such as Chardaire et al. (1995) 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 (Mahfoud and Goldberg, 1995; Mahfouz et al., 1999). 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 (Mahfoud and Goldberg, 1995). 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 (Haupt and Haupt, 1998). Recall we need to optimize the selection of SDs' DCCFPs and their schedule, i.e., their start times. Therefore, the length of a chromosome is the number of SDs in the SUT. Additionally, we need to encode both DCCFP identifiers and their arrival times in a chromosome. A gene can be depicted as a pair $(\rho_{i,\text{selected}}, \alpha\rho_{i,selected})$, 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 Fig. 15a. 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.

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

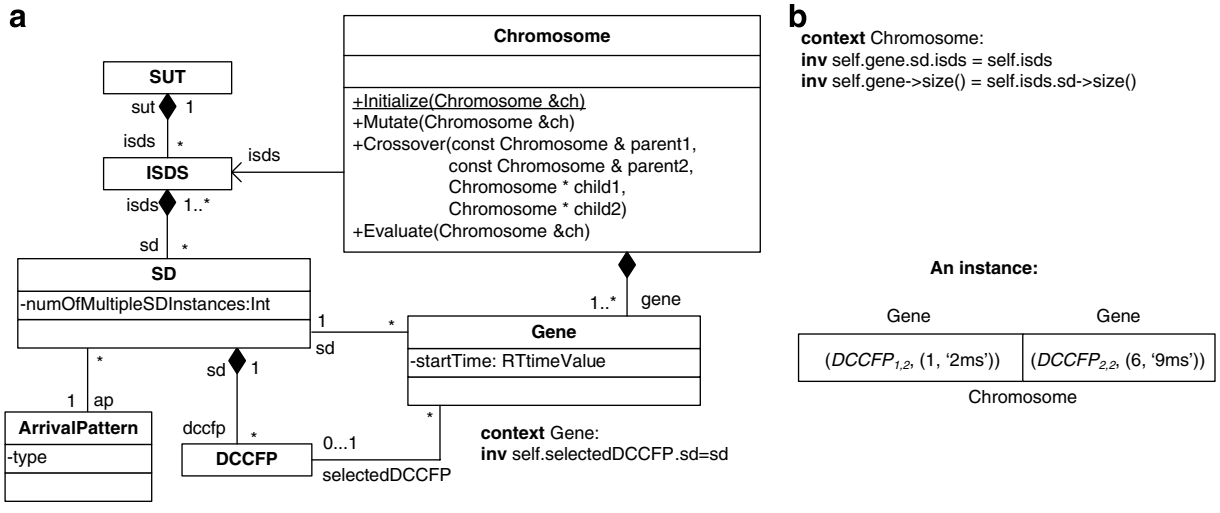


Fig. 15. (a): Metamodel of chromosomes and genes in our GA algorithm. (b): Part of an instance of the metamodel.

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 invariants for the *Chromosome* and *Gene* classes (Fig. 15). For example, the SD of a Gene instance should be the SD containing the selected DCCFP of that Gene instance. 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 Fig. 15b, which complies with the metamodel in Fig. 15a. 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 (Haupt and Haupt, 1998) 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.

6.5.3. 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* (Fig. 15a) as presented in Fig. 16.

6.5.4. Constraint #2: arrival pattern constraints

Given a chromosome, the OCL post-condition in Fig. 17 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.

6.5.5. Initial population

Determining the population size of a GA is challenging (Atallah, 1999). 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 Haupt and Haupt (1998) list a variety of works that suggests adequate population sizes. The authors reveal that the work of De Jong (1988) suggests a population size ranging from 50 to 100 chromosomes. Grefenstette and Cobb (1993) recommend a range between 30 and 80, while Schaffer et al. (1989) 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 Fig. 18. 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

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

Fig. 16. Constraint #1 of the GA (an OCL expression).

```
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
```

Fig. 17. Constraint #2 of the GA (an OCL function).

```
Function CreateAChromosome(): Chromosome
c: Chromosome
1   ISDS=a random ISDS
2   For all SD_i ∈ ISDS
3       c.gene_i.selectedDCCFP = a random DCCFP from SD_i
4   For all SD_i ∉ ISDS
5       c.gene_i=null
6   Intersection=ATS(SD_1) ∩ ATS(SD_2) ∩… ∩ATS(SD_i), where SD_j,j=1…i ∈ ISDS
7   If Intersection≠{ }
8       Choose a random time instance t_schedule in Intersection
9           For all c.gene_i ≠null
10          c.gene_i.startTime= t_schedule
11  Else
12      For all c. gene_i ≠null
13          c.gene_i.startTime= A random time instance t_i in ATS(SD_i)
14  Return c
```

Fig. 18. Pseudo-code to generate chromosomes of the GA's initial population.

GA's initial population,*CreateAChromosome*() is invoked 80 times.

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, in an attempt to produce an initial population that already puts the system under stress. (The impact of this heuristics remains to be studied though.) This is done by first calculating the intersection of ATSs for SDs in the selected ISDS (line 6), using an intersection operator described in Garousi et al. (2006a). 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 and 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 and 13).

When selecting a random time instance for a gene, we need a range from which to select time values. When calculating an intersection of ATSs, we also need to select 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 Fig. 18, we ensure the initial population of chromosomes complies with both constraints of Section 6.5.2. Note that the above algorithm does not necessarily ensure that all the ISDSs are represented in the initial population (this depends on the number of ISDSs and the size of the 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.

6.5.6. 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 further discussed in Garousi et al. (2006a). We also present in Garousi et al. (2006a) a set of heuristics-based constraints that a suitable MST should satisfy depending on the kind of AP (e.g., periodic, irregular). For example, one of those constraints is that a suitable MST should be greater than all maximum times in ATSs of all irregular APs. This constraint will allow our GA to effectively search in the time domain, considering all possible times from all irregular APs. All those constraints are considered together when searching for a suitable MST by the GA.

6.5.7. 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 (Atallah, 1999). 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 6.2 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 Eq. 3.

The first line of Eq. 3 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 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 Fig. 15b is calculated. The calculation process is shown in Fig. 19. The chromosome contains two genes, which correspond to $DCCFP_{1,2}$ and $DCCFP_{2,2}$. The search range is [2 ms, 20 ms]: 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 Fig. 19. The output value of ISTOF is 110 KB.

6.5.8. Operators

Operators enable GAs to explore a solution space (Haupt and Haupt, 1998) 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

$$ISTOF\ \ Chromosome \rightarrow\ real$$

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

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

Eq. (3). Instant stress test objective function (ISTOF).

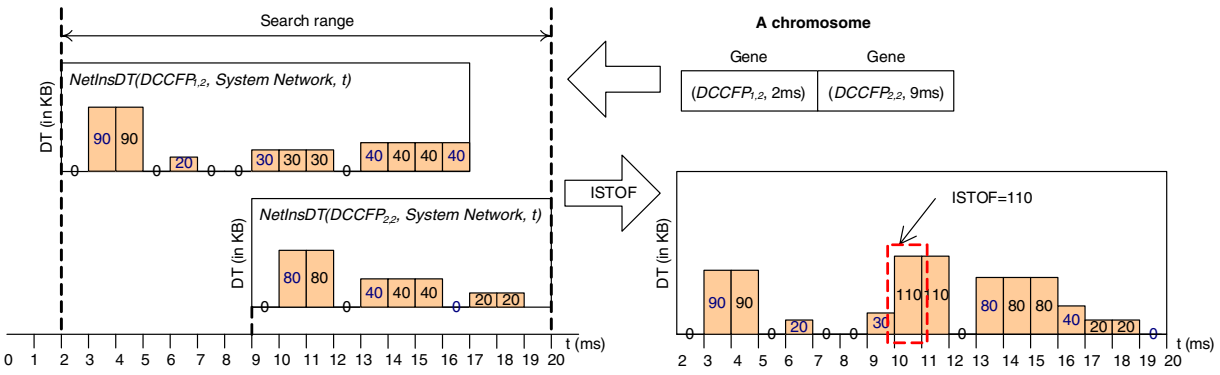


Fig. 19. Computing the instant stress test objective function (ISTOF) value of a chromosome.

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.

6.5.9. Crossover operator

Crossover operators aim at passing on desirable traits or genes from generation to generation (Haupt and Haupt, 1998). 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 off-springs. 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 (Haupt and Haupt, 1998). 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 Pawlowsky, 1995) 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 the chromosome's 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 Fig. 20 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 Fig. 20b and c, respectively. In *nPointCrossover*, the fragments of Parent 1 and Parent 2 are alternately interchanged (Fig. 20b): 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 Fig. 20c. Bold genes indicate the fragments interchanged by *nPointProbCrossover*. Three of the four generated children (all except Child 2 in Fig. 20c) 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 Fig. 20c violates constraint #1, the two temporary children (Child 1 and Child 2 in Fig. 20c) are abandoned, and this particular execution of *nPointProbCrossover* is repeated.

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.

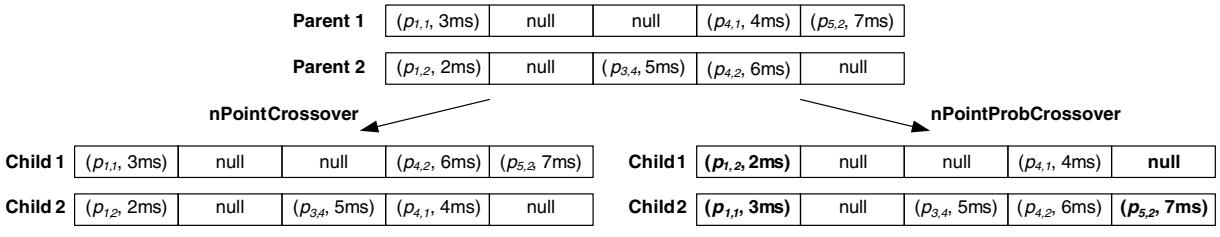


Fig. 20. Two example uses of the crossover operators.

Crossover rates are critical. A crossover rate is the percentage of chromosomes in a population being selected for a crossover operation. If the crossover 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 (Haupt and Haupt, 1998). De Jong (1988) concluded that a desirable crossover rate should be about 60%. Grefenstette and Cobb (1993) 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%.

6.5.10. 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 set (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 (Haupt and Haupt, 1998). Back (1992) enumerates some of the more common mutation rates used. The author states that De Jong (1988) suggests a mutation rate of 0.001, Grefenstette and Cobb (1993) suggests a rate of 0.01, while Schaffer et al. (1989) 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 (1989) suggests a mutation rate defined by $1/length$. Smith and Fogarty (1996) 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.: $1.75/\lambda\sqrt{length}$. Each of the three mutation operators defined above are applied with a probability corresponding to a third of the mutation rate.

7. Empirical analysis

We implemented a prototype tool to support the application of the GASTT methodology (GARUS). This section presents a carefully designed empirical study, using this tool, to validate the design choices of our GA. We only provide below a short functional overview (Fig. 21) but technical details about the tool can be found in Garousi et al. (2006a).

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* (these are classes in the tool's class diagram Garousi et al., 2006a). Furthermore, *ga*'s parameters (e.g. mutation rate) are set according to our discussions in Section 6.5. GARUS then evolves *ga* using our mutation and crossover operators (Section 6.5.6). When the evolution of

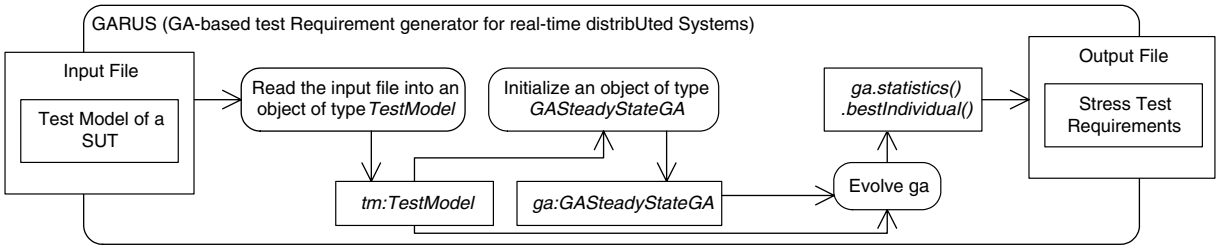


Fig. 21. Overview activity diagram of GARUS.

*ga*finishes, i.e., after a predefined number of generations, the best individual is saved in an output file.

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. Using the above information, test requirements generated by GARUS can be validated according to a number of criteria. Due to space constraint, we describe here two of the main criteria (four others can be found in Garousi et al. (2006a)):

1. *Repeatability of GA results across multiple runs*: It is important to assess how stable and reliable the results of the GA will be. To do so, the GA is executed a large number of times and we assess the variability of the average and best chromosome's fitness values.
2. *Convergence efficiency across generations towards a maximum*: In order to assess the design of the selected mutation and cross-over operators, as well as the chosen chromosome representation, it is useful to look at the speed of convergence towards a maximum fitness plateau (Louis and Rawlins, 1993). 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.

Using the above criteria, we have analyzed the stress test requirements generated by running GARUS on an *experimental* test model, which was specifically designed for that purpose. The test model has a relatively small size (five SDs and two to five CFPs for each SD) and APs of every possible type supported by the UML-SPT profile (Object Management Group, 2003) (periodic, bounded, bursty, irregular, and unbounded as discussed in Section 5.5.2) are used. Note that though the experimental test model used here is small, the scalability of our tool with respect to variations in model size was assessed through another experiment reported in Garousi et al. (2006a). Three of the main observations from our experiments on scalability analysis were: (1) as the size of the test model gets larger, the variation in maximum ISTOF values (objective function) across executions remain constant; (2) the GA can reach a maximum plateau even when the size of a specific component (SD, ISDS, DCCFP, etc) of a given model is very large (up to 100 ISDSs in a SUT, 200 SDs, 30 SDs in an ISDS, and 50 DCCFPs in a SD); and (3) test model size does not have an impact on the 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. We report next our empirical analysis results regarding repeatability and convergence efficiency.

7.1. Repeatability of GA results across multiple runs

Since GAs are heuristics, their performance and outputs can vary across multiple runs. We investigate the *repeatability*of GA results by analyzing the variation in maximum ISTOF values and maximum stress time values.

Fig. 22a depicts the distributions of maximum ISTOF and stress time values for 1000 runs of the experimental test model (explained above; Garousi et al., 2006b). 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 1.

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 Fig. 22a that with a few runs a chromosome with a fitness value close to the observed maximum would likely be identified. Since each run lasts a few seconds, relying on multiple runs to generate a stress test requirement should perhaps take a few minutes for very large examples and should not lead to practical problems.

Corresponding portions of max stress time values for the most frequent maximum ISTOF value (72 units of traffic) have been highlighted in black in Fig. 22b. 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 testing strategy to further explore

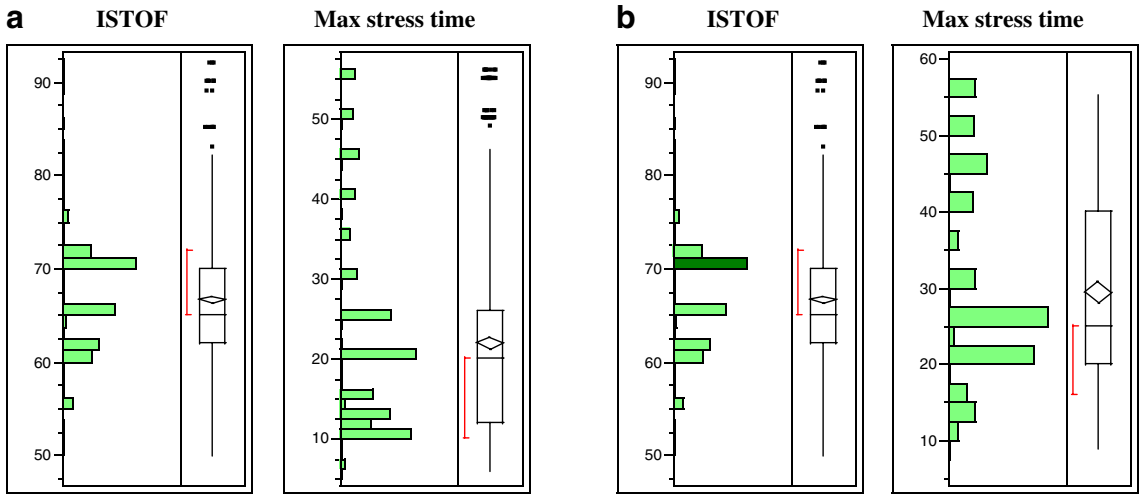


Fig. 22. (a): Histogram of maximum ISTOF and stress time values for 1000 runs. (b): Corresponding max stress time values for one of frequent large ISTOF value (72 units of traffic).

Table 1
Descriptive statistics of the maximum ISTOF values over 1000 runs

Min	Max	Average	Median	Standard deviation
50	92	66.672	65	6.4

Values are in units of data traffic (e.g. KB).

would be to cover all (or a subset of) such test requirements with maximum ISTOF values in different time instances. Indeed, although their ISTOF values are the same, a SUT's reaction to different test requirements might vary, since different DCCFPs (and hence set of messages) in different time instances may be triggered. This might in turn lead to uncovering different RT faults in the SUT.

7.2. Convergence efficiency across generations towards a maximum

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 an experimental test model is shown in Fig. 23, where the *x*-axis is the generation number and the *y*-axis is the probability of achieving such plateau in such 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 than 100 generations will be required (even for our large experimental models). This number is in line with the experiments reported in the GA literature (Haupt and Haupt, 1998) but is however likely to be dependent on the number and complexity of SDs as well as their ATSs.

We further observe that, from the initial to the final populations, the maximum fitness values typically increase by about 80%, which can be considered a large improvement.

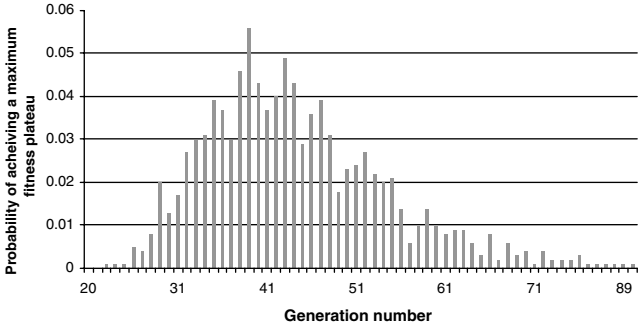


Fig. 23. Histogram of the generation numbers when a stable maximum fitness plateau is reached in 1000 runs of an experimental test model by GARUS.

So, though we cannot guarantee that a GA does find the global maximum, we clearly generate test requirements that will significantly stress the system.

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 8.2.

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) (Mackay et al., 2003) and *supervisory control and data acquisition* (SCADA) *Systems* (Daneels and Salter, 1999) are two kinds of such systems.

We surveyed numerous existing systems (e.g. Constantinescu et al., 2003; Ebata et al., 2000; Information Society Technologies, 2003; BWI Co, 2004) 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. Stojkovic and Vujosevic, 2002) 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* (Toshida et al., 1998): (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 Garousi et al. (2006a).

We used our GARUS tool (Section 7) with the SCAPS UML model (Garousi et al., 2006a) as input to derive stress test requirements maximizing instant data traffic on the SCAPS nation-wide network. 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 Fig. 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.

8.2. Stress test results

In this section we compare the durations for SDs *D* and *E* presented in Fig. 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 (Musa, 1992). 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 (Toshida et al., 1998).

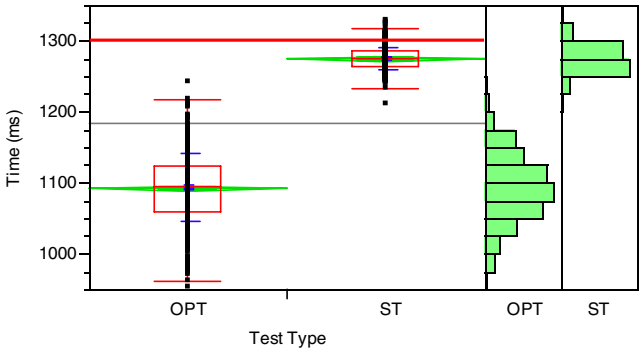


Fig. 24. Duration of a hard RT constraint by running Operational Profile Tests (OPT) and Stress Tests (ST).

Table 2
Quantiles of the distribution in Fig. 24

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

Values are in milliseconds.

Recall that we also modeled a HRT constraint in the MIOD of SCAPS in Fig. 7. It specifies the maximum acceptable value for the durations of SDs *D* and *E*: they should be less than 1300 ms (milliseconds). Fig. 24 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 2.

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 Fig. 24. The 1300 ms deadline (HRT constraint) is illustrated by a horizontal bold line in Fig. 24 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 paper 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 Garousi et al. (2006b), 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; and (3) Generalizing the assumption we made in Section 4.1.1 in which we considered only one network path between two nodes rather than several paths to simply our network traffic usage model (Section 5.4). Such a generalization will require analysis of data load in different parts of a network using the information provided by the routing policy used in the backbone network of a SUT.

Acknowledgement

This work was in part supported by Siemens Corporate Research, Princeton, NJ, and a Canada Research Chair (CRC) grant. Vahid Garousi was further supported by a start-up grant from the Department of Electrical and Computer Engineering and the Schulich School of Engineering of the University of Calgary.

Appendix. Glossary of acronyms

AP	arrival pattern
APM	arrival pattern model
ATI	accepted time intervals
ATP	accepted time points
ATS	accepted time set
CCFG	concurrent control flow graphs
CCFP	concurrent control flow path
CFP	control flow path
DCCFP	distributed concurrent control flow path
DRTS	distributed real-time systems
DT	data traffic
GA	genetic algorithms
GARUS	GA-based test requirement tool for real-time distributed systems
GASTT	genetic algorithm-based stress test technique
HRT	hard real-time
IOD	interaction overview diagram
ISDS	independent SD set
ISTOF	instant stress test objective function
MaxIAT	maximal Inter-Arrival Time
MinIAT	minimal inter-arrival time
MIOD	modified interaction overview diagram
MST	maximum search time
MT	message traffic
NDD	network deployment diagram
NIT	network interconnectivity tree
NTU	network traffic usage
NTUP	network traffic usage pattern
OPT	operational profile test
RUA	resource usage analysis
SD	sequence diagram
ST	stress test
SUT	system under test
TM	test model
TSSTT	time-shifting stress test technique
UML-SPT	UML profile for schedulability, performance, and time

References

Allen, J.F., 1983. Maintaining knowledge about temporal intervals. Communications of the ACM 26, 832–843.

Atallah, M.J., 1999. Handbook of Algorithms and Theory of Computation. CRC (Chemical Rubber Company) Press.

Avritzer, A., Weyuker, E.J., 1995. The automatic generation of load test suites and the assessment of the resulting software. IEEE Transactions on Software Engineering 21, 705–716.

Back, T., 1992. Towards a practice of autonomous systems. Proceeding of European Conference on Artificial Life, 263–271.

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

Bowen, J.P., Bogdanov, K., Clark, J., Harman, M., Hierons, R., Krause, P., 2002. FORTEST: formal methods and testing. Proceedings of the International Computer Software and Applications Conference, 91–101.

Briand, L.C., Labiche, Y., Shousha, M., 2006. Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. Journal of Genetic Programming and Evolvable Machines 7, 145–170.

Buhr, R.J.A., 1998. Use case maps as architectural entities for complex systems. IEEE Transactions on Software Engineering 24.

BWI Co., 2004. ElipseSCADA. <http://www.bwi.com/proot/2775>.

Chardaire, P., Kapsalis, A., Mann, J.W., Rayward-Smith, V.J., Smith, G.D., 1995. Applications of genetic algorithms in telecommunications. Proceeding of Applications of Neural Networks to Telecommunications, 290–299.

Coleman, D., Arnold, P., Bodoff, S., Dollin, C., GilChrist, H., Hayes, F., Jeremaes, P., 1994. Object-Oriented Development – The Fusion Method. Prentice Hall.

Constantinescu, Z., Petrovic, P., Pedersen, A., Federici, D., Campos, J., 2003. QADPZ (Quite Advanced Distributed Parallel Zystem). <http://qadpz.sourceforge.net>.

Daneels, A., Salter, W., 1999. What is SCADA? Proceeding of International Conference on Accelerator and Large Experimental Physics Control Systems, 39–343.

De Jong, K., 1988. Learning with genetic algorithms: an overview. Machine Learning 3, 121–138.

Douglass, B., 1999. Doing Hard Time, Developing Real-Time Systems with UML Objects, Frameworks, and Patterns. Addison Wesley.

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

Garousi, V., Briand, L., Labiche, Y., 2005. Control flow analysis of UML 2.0 sequence diagrams. In: Proceeding of European Conference on Model Driven Architecture-Foundations and Applications, LNCS, vol. 3748, pp. 160–174.

Garousi, V., Briand, L., Labiche, Y., 2006a. Traffic-aware stress testing of distributed real-time systems based on uml models using genetic algorithms, Technical Report SCE-06-09, Carleton University. <http://squall.sce.carleton.ca/pubs/tech_report/TR_SCE-06-09.pdf>.

Garousi, V., Briand, L., Labiche, Y., 2006b. Traffic-aware Stress testing of distributed systems based on UML models. In: Proceeding of International Conference on Software Engineering, pp. 391–400.

Gomaa, H., 2000. Designing Concurrent, Distributed, and Real-Time Applications with UML. Addison-Wesley.

Grefenstette, J.J., Cobb, H.G., 1993. Genetic algorithms for tracking changing environments. Proceeding of International Conference on Genetic Algorithms, 523–530.

Haupt, R.L., Haupt, S.E., 1998. Practical Genetic Algorithms. Wiley-Interscience.

Horst, R., Pardalos, P.M. (Eds.), 1995. Handbook of Global Optimization. Kluwer, Dordrecht.

European Information Society Technologies, 2003. Component based open source architecture for distributed telecom applications. <http://coach.objectweb.org>.

Jones, B.F., Sthamer, H.H., Eyres, D.E., 1996. Automatic structural testing using genetic algorithms. Software Engineering Journal 11, 299–306.

Kuhn, R., 1997. Sources of failure in the public switched telephone network. IEEE Computer 30, 31–36.

Lahtinen, J., Silander, P.M., Tirri, H., 1996. Empirical Comparison of Stochastic Algorithms. Proceedings of Nordic Workshop on Genetic Algorithms and their Applications, 45–60.

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

Mühlenbein, H., 1989. Parallel genetic algorithms, population genetics and combinatorial optimization. Proceedings of International Conference on Genetic Algorithms, 416–421.

Mackay, S., Wright, E., Park, J., 2003. Practical Data Communications for Instrumentation and Control. Newnes.

Mahfoud, S.W., Goldberg, D.E., 1995. Parallel recombinative simulated annealing: a genetic algorithm. Journal on Parallel Computing 21, 1–28.

Mahfouz, S.Y., 1999. Design optimization of structural steel work, Ph.D. Thesis, Department of Civil and Environmental Engineering, University of Bradford.

Moon, J.W., Moser, L., 1965. On Cliques in Graphs. Israel Journal of Mathematics 3, 23–28.

Musa, J.D., 1992. The operational profile in software reliability engineering: an overview. In: Proceedings of the International Symposium on Software Reliability Engineering.

Nebut, C., Fleurey, F., Traon, Y.L., Jézéquel, J.-M., 2003. Requirements by contracts allow automated system testing. Proceedings of International Symposium on Software Reliability Engineering 85–96.

Object Management Group (OMG), 2005. UML 2.0 Superstructure Specification.

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

Object Management Group (OMG), 2005. OCL 2.0 Specification.

Pargas, R.P., Harrold, M.J., Peck, R.R., 1999. Test-data generation using genetic algorithms. Journal of Software Testing, Verification and Reliability 9, 263–282.

Pawlowsky, M.A., 1995. In: L. Chambers (Ed.), Crossover Operators, Practical Handbook of Genetic Algorithms Applications, pp. 101–114.

Pender, T., 2003. UML Bible. Wiley.

Schaffer, J.D., Caruana, R.A., Eshelman, L.J., Das, R., 1989. A study of control parameters affecting online performance of genetic algorithms for function optimization. Proceedings of International Conference on Genetic algorithms, 51–60.

Smith, J.E., Fogarty, T.C., 1996. Adaptively parameterized evolutionary systems: self adaptive recombination and mutation in a genetic algorithm. Proceeding of International Conference on Parallel Problem Solving From Nature, 441–450.

Stojkovic, B., Vujosevic, I., 2002. 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, 695–700.

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

Toshida, N., Uesugi, M., Nakata, Y., Nomoto, M., Uchida, T., 1998. Open distributed EMS/SCADA system. Hitachi Review 47, 208–213.

Tracey, N., Clark, J., Mander, K., 1998a. The way forward for unifying dynamic test-case ceneration: the optimization-based approach. Proceedings of the International Workshop on Dependable Computing and its Applications, 169–180.

Tracey, N., Clark, J., Mander, K., 1998b. Automated program flaw finding using simulated annealing. ACM SIGSOFT Software Engineering Notes 23, 73–81.

Tsai, J.J.P., Bi, Y., Yang, S.J.H., Smith, R.A.W., 1996. Distributed Real-Time Systems: Monitoring, Visualization, Debugging, and Analysis. John Wiley & Sons.

Wegener, J., Baresel, A., Sthamer, H., 2001. Evolutionary test environment for automatic structural testing, journal of information and software technology. Special issue on Software Engineering using Metaheuristic Innovative Algorithms 43, 841–854.

Weyuker, E., Vokolos, F.I., 2000. Experience with performance testing of software systems: issues, an approach and case study. IEEE Transactions on Software Engineering 26, 1147–1156.

Yang, C.S.D., 1996. Identifying potentially load sensitive code regions for stress testing. In: Proceedings of the Mid-Atlantic Student Workshop on Programming Languages and Systems.

Zhang, J., Cheung, S.C., 2002. Automated test case generation for the stress testing of multimedia systems. Journal on Software Practice and Experience 32, 1411–1435.