Traffic-aware Stress Testing of Distributed Systems Based on UML Models

Vahid Garousi‡, Lionel C. Briand§ and Yvan Labiche‡

‡ Carleton University
Department of Systems and Computer Engineering
Software Quality Engineering Laboratory
1125 Colonel By Drive
Ottawa, ON K1S5B6, Canada
{vahid, briand, labiche}@see.carleton.ca

§ Simula Research Laboratory
Department of Software Engineering
Martin Linges v 17, Fornebu
P.O.Box 134,
1325 Lysaker, Norway
briand@simula.no

ABSTRACT
A stress test methodology aimed at increasing chances of discovering faults related to network traffic in distributed systems is presented. 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. Different variants of our stress testing technique already exist (they stress different aspects of a distributed system) and we focus here on one variant that is designed to identify and to stress test the system at the instant when data traffic on a network is maximal. Using a real-world distributed system specification, we design and implement a prototype distributed system and describe, for that particular system, how the stress test cases are derived and executed using our methodology. 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.

Categories and Subject Descriptors

General Terms
Algorithms, Reliability, Verification.

Keywords
Stress testing, model-based testing, distributed systems, UML, network traffic.

1. INTRODUCTION
Distributed Systems (DS) are becoming more important to our everyday life. Examples include command and control systems, aircraft aviation systems, robotics, and nuclear power plants [29]. The development and testing of a DS is difficult and takes more time than the development and testing of a distributed system without real-time constraints or a non-distributed system, which runs on a single computer [31]. Sources of failures in the United States Public Switched Telephone Network (PSTN) are investigated in [17]. 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 motivation for our work can be stated as follows: because DS are by nature concurrent and are often real-time, there is a need for methodologies and tools for testing and debugging DS under stress conditions such as heavy user loads and intense network traffic. These systems should be tested under stress before being deployed in the field in order to assess their robustness to distribution-specific problems. In this work, our focus is on network traffic, one of the fundamental factors affecting the behavior of DS.

Distributed nodes of a DS 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 DS might behave well with normal network traffic loads (e.g., in terms of amount of data, number of requests), but the communication might turn out to be poor and unreliable if many network messages or high loads of data are concurrently transmitted over a particular network or towards a particular node.

Since 1997, UML has become the de facto standard for modeling object-oriented software for nearly 70 percent of IT industry [26]. The new version of UML, version 2.0 [23] offers an improved modeling language compared to UML 1.x versions. Some of the high level improvements are: enhanced architecture modeling, extensibility, support for component-based development, modeling of relationships and model management [26]. As we expect UML to be increasingly used for DS, it is therefore important to develop automatable UML model-driven, stress test
techniques and this is the main motivation for the work reported here.

Assuming that the UML design model of a DS is in the form of Sequence Diagrams (SD) annotated with timing information, and the systems’ network topology is given in a specific modeling format, we propose a technique to derive test requirement to stress the DS with respect to network traffic in a way that will likely reveal robustness problems. Note that, for a DS 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. 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.

Though the stress test technique reported in this article targets distributed systems, there are important aspects of real-time systems that we do not address here. As we further discuss in Section 6.6, such systems often have to react to external or internal events that exhibit arrival patterns (e.g., periods). Such patterns impose constraints on the time instant when interactions between distributed objects can take place. These issues are addressed in [15] where a more sophisticated optimization technique based on Genetic Algorithms is proposed. The current paper, however, focuses on the modeling issues to enable such optimization and a simpler optimization approach when no arrival pattern constraints are present.

Contributions of this work can be summarized as: (1) an approach for UML model-based network traffic usage analysis in DS, and (2) a stress test strategy aiming at increasing chances of discovering faults related to network traffic.

The remainder of this article is structured as follows. Related works are discussed in Section 2. An overview of the stress test methodology is given in Section 3. The assumed input system models for the methodology 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. The results of applying the methodology to a case study system is described in Section 7, which shows the applicability and assesses the effectiveness of the methodology in revealing faults related to network traffic. Finally, Section 8 concludes the article and discusses some of the future research directions.

2. RELATED WORKS

To the best of our knowledge, no existing work addresses the automated derivation of test requirements from UML models for performance stress testing of DS from the perspective of maximizing the chance of exhibiting network traffic faults. There have not been many works on systematic generation of stress and load test suites for software systems, with the notable exception of [2, 4, 32, 33].

Authors in [2] propose a class of load test case generation algorithms for telecommunication systems which can be modeled by Markov chains. The black-box techniques proposed are based on system operational profiles. The Markov chain that represents a system’s behavior is first built. The operational profile of the software is then used to calculate the probabilities of the transitions in the Markov chain. The steady-state probability solution of the Markov chain is then used to guide the generation process of the test cases according to a number of criteria, in order to target specific types of faults. For instance, using probabilities in the Markov chain, it is possible to ensure that a transition in the chain is involved many times in a test case so as to target the degradation of the number of calls that can be accepted by the system. From a practical standpoint, targeting only systems whose behavior can be modeled by Markov chains can be considered a limitation of this work.

Yang proposed a technique [32] to identify potentially load sensitive code regions to generate load test cases. The technique targets memory-related faults (e.g., incorrect memory allocation/de-allocation, incorrect dynamic memory usage) through load testing. The approach is to first identify statements in the module under test that are load sensitive, i.e., they involve the use of malloc() and free() statements (in C) and pointers referencing allocated memory. Then, data flow analysis is used to find all Definition-Use (DU)-pairs that trigger the load sensitive statements. Test cases are then built to execute paths for the DU-pairs.

Briand et al. [4] propose a methodology for the derivation of test cases that aims at maximizing the chances of deadline misses within a system. They show that task deadlines may be missed even though the associated tasks have been identified as schedulable through appropriate schedulability analysis. The authors note that although it is argued that schedulability analysis simulates the worst-case scenario of task executions, this is not always the case because of the assumptions made by schedulability theory. The authors develop a methodology that helps identify performance scenarios that can lead to performance failures in a system.

Zhang et al. [33] describe a procedure, similar to ours, for automating stress test case generation in multimedia systems. The authors consider a multimedia system consisting of a group of servers and clients connected through a network as a SUT. Stringent timing constraints as well as synchronization constraints are present during the transmission of information from servers to clients and vice versa. The authors identify test cases that can lead to the saturation of one kind of resource, namely CPU usage of a node in the distributed multimedia system. The authors first model the flow and concurrency control of multimedia systems using Petri-nets coupled with timing constraints. A specific flavor of temporal logic [1] was used to model temporal constraints. The following are some of the limitations of their technique: (1) The technique 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

---

1 A network interaction interleaving is a possible sequence of network interactions among a subset of objects on a subset of nodes.
system may only exhibit failures if stress testing is prolonged for a period of time; (4) If this technique is applied in a UML-based development, it requires additional knowledge (Petri Nets and a specific flavor of temporal logic-see [33]) which can be an impediment to its use.

3. OVERVIEW OF THE METHODOLOGY
An overview of our model-based stress test methodology is presented using an activity diagram in Figure 1. Note that only the steps in gray background are addressed by the current paper.

![Figure 1. Overview of our model-based stress test methodology.](image)

A UML model of a SUT, following specific but realistic requirements, is used in input. A test model (TM) is then built to facilitate subsequent automation steps. The TM and a set of stress test parameters (objectives) set by the user are then used by an optimization algorithm to derive stress test requirements. Test requirements can finally be used to specify test cases to stress test a SUT.

The 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 business logic of a SUT), and schedule them such that their maximum traffic messages run at the same time. The detailed steps of Figure 1 are described in the next sections:

- Input models (Section 4)
- Building the test model (Section 5)
- Derivation of stress test requirements (optimization algorithm) – (Section 6)

4. INPUT SYSTEM MODEL
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 [16]). Others 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). The next subsections describe the latter two diagrams.

4.1 Network Deployment
The structure of the distributed architecture of a SUT as we need it to be described is shown in Figure 2 as a metamodel. Such network information is paramount as one of our objectives is to stress, not only nodes in a network, but also (sub-)networks. An example of a distributed architecture is depicted in Figure 3-(a) which shows networks in a hierarchical structure (each network can have many subnets and only one supernet), nodes belonging
to networks, and objects distributed on nodes, e.g., node1 hosts three objects (o1,1, o1,2, and o1,3).

Each node is connected to other nodes through a network path which is defined as a sequence of networks. For example, node1 is connected to node3 through the network path <network1, SystemNetwork, network2> in Figure 3-(a).

![Figure 2. Metamodel for distributed architectures.](image)

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

We want to describe such a distributed architecture using UML 2.0 so as to be able to use it as an input for our testing methodology in the context of a UML-based development. When compared to UML 1.x [26], UML 2.0 [23] shows significant improvements with respect to modeling application architecture, nodes, and communication paths. However, modeling a hierarchical set of networks and their inter-connectivity is not directly addressed in the UML 2.0 specification [23].

We thus decided to model the system network as a package structure where the entire system network is the root (high level) package and other networks and nodes are the sub-packages in a hierarchical manner. We refer to such a diagram as a Network Deployment Diagram (NDD). As an example, the architecture in Figure 3-(a) is modeled by the NDD in Figure 3-(b). Note that packages represent networks of the system and nested relationships among the packages denote nested networks.

4.2 Modified Interaction Overview
The naming of Modified Interaction Overview Diagram (MIOD) comes from the UML 2.0’s Interaction Overview Diagram (IOD) [23]. 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. Some of the existing representations to model such constraints among SDs are [6, 8, 21]. However, as we discuss in [15], 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, e.g., the Login SD of an ATM system should be executed before the Withdrawal SD. As an example, the MIOD of a simplistic ATM system is depicted in Figure 4. This example MIOD has two partitions to model which actor (e.g., user) can trigger (or interact with) SDs.

![Figure 4. An example Modified Interaction Overview Diagram (MIOD).](image)

5. BUILDING THE TEST MODEL

The Test Model (TM) consists of four entities: (1) a control flow model, (2) a network interconnectivity tree, (3) network traffic usage patterns, and (4) inter-SD constraints. These entities are needed in our TM to facilitate the automated derivation of test requirements. How they are built from a given input UML model is described next.

5.1 Control Flow Model

In UML 2.0 [23], 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 [14], asynchronous messages and parallel combined fragments entail concurrency inside SDs.

In a SD of a distributed system, some messages are local (sent from an object to another on the same node), while others are distributed (sent from an object on one node to an object on another node) thus entailing network traffic. For example, referring to Figure 5, message A is local (within node n1) and message C is distributed (from n1 to n2). Furthermore, different CFPs can have different sequences of messages and each message can have different signatures and a different set of parameters. Therefore, the network traffic usage of each CFP can be different from other CFPs. Thus, comprehensive model-based stress testing should take into account the different CFPs of a SD.

In the current work, we use the Model-based Control Flow Analysis (MBCFA) technique presented in [14], which was formalized using meta-modeling and consistency-rules in the Object Constraint Language (OCL) [22]. We also introduced Concurrent Control Flow Graphs (CCFG) as a means to analyze the concurrent control flow of SDs. A CCFG is generated for each SD. In cases where a SD calls (refers to) another SD, there would be control flow edges connecting their corresponding CCFGs to form an Inter-SD CCFG. An inter-SD CCFG is similar to the concept of inter-procedural CFG [19]. As an example, the CCFG in Figure 6 corresponds to the SD in Figure 5. In a CCFG, nodes correspond to messages of the corresponding SD (e.g., in Figure 6, node labels are message names in Figure 5), and edges between nodes denote the flow of control among messages (e.g., message B follows message A). Join and fork nodes are used to represent concurrent control flow entailed by asynchronous messages and parallel combined fragments in a SD. For example, the fork node after node C (Figure 6) corresponds to the concurrent control flow entailed by the asynchronous message C.

The timing information of messages has been modeled using the UML Profile for Schedulability, Performance, and Time (UML-SPT) [25] and will be discussed later in Section 5.3.

![Figure 5. A SD with asynchronous messages and timing information.](image)

**Figure 5. A SD with asynchronous messages and timing information.**

![Figure 6. CCFG of the SD in Figure 5.](image)

**Figure 6. CCFG of the SD in Figure 5.**

The concept of Concurrent Control Flow Path (CCFP), formally presented in [14], is similar to the conventional CFGs, except that they account for concurrent control flows as they are derived from CCFGs. A CCFP is a path from the start node of a CCFG to its final node. In our notation for CCFPs, parentheses denote concurrent control flow, and can be nested hierarchically. For example, some of the CCFPs in the CCFG of Figure 6 are represented in Figure 7.

\[ \rho_1 = \text{ABC} \left( \frac{DE}{FG} \right) \]
\[ \rho_2 = \text{ABC} \left( \frac{HI}{FG} \right) \]
\[ \rho_3 = \text{ABC} \left( \frac{HH}{FG} \right) \]

**Figure 7. Some CCFPs of the CCFG in Figure 6.**

The symbol \( \rho \) (Figure 7) will be used in the rest of this article to refer to CCFPs. Four CCFPs in Figure 7 are due to the decision node corresponding to a loop in the CCFG. Following a strategy similar to what is done for code coverage, a loop can either be bypassed (if possible), taken only once, a representative or average number of times, and a maximum number of times. These situations correspond to CCFPs \( \rho_1, \rho_2, \rho_3 \) and \( \rho_4 \), respectively.
5.2 Network Interconnectivity Tree

A Network Interconnectivity Tree (NIT) is a data structure built from a NDD (Section 4). 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. The root of the tree is always the entire system network while system networks and nodes are its children. In a NIT, networks and nodes are shown as rectangles and circles, respectively. For example, the NIT of the NDD in Figure 3-(b) is shown in Figure 8-(a). Then, assuming a tester’s goal is to stress test the SUT only with respect to Network2, the test strategy will only consider nodes under Network2 in the NIT.

Figure 8. (a): NIT built from the NDD in Figure 3-(b). (b): Derivation of network path between two nodes using getNetworkPath().

To identify the network path between any two given nodes, we define the network path function getNetworkPath(ni, nj) where ni and nj are the sender and the receiver nodes of a message, respectively. (An algorithm for this function can be found in [15].) For example, the derivation of the network path between n1 (the sender) and n2 (the receiver) is depicted in Figure 8-(b) and is formally represented as:

getNetworkPath(n1, n2)=<Network2, SystemNetwork, Network1>

5.3 Network Traffic Usage Pattern

A network traffic usage pattern is the pattern describing how a CCFP entails traffic on a network. As discussed in Section 5.1, the network traffic usage of each CCFP can be different from other CCFPs. We present in this section a resource usage analysis (RUA) technique to measure the traffic usage for each CCFP. The measurement heuristics are described first.

5.3.1 Heuristics

In order to analyze the usage pattern of each CCFP, we need to analyze the traffic usage entailed by its messages. Only distributed messages in SDs are of interest here (not local messages) since they are the only ones entailing network traffic.

A Distributed CCFP (DCCFP) is a CCFP where only distributed messages are modeled. For example, considering r1 in Figure 7, since all messages except A and B are distributed, the transformation to obtain a DCCFP is:

\[ r_1 = ABC \left( \begin{array}{c} DE \\ FG \end{array} \right) \Rightarrow DCCFP(r_1) = C \left( \begin{array}{c} DE \\ FG \end{array} \right) \]

where DCCFP() denotes a function returning the corresponding DCCFP of a CCFP.

In order to measure the traffic entailed by each distributed message, we measure 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\(^2\). We define the data size of an object to be the summation of sizes (in bytes) of the attributes in its class. We formalize next the heuristics discussed in this section.

Admittedly, other measures (perhaps more accurate) of network traffic can be considered. We however consider our measurement as a reasonable surrogate for network traffic.

5.3.2 Formalizing Messages

In order to precisely define how we perform traffic analysis of SDs, we formally define SD messages. Similar to the tabular representation of messages, proposed by UML 2.0 [23], each message annotated with timing information (using the UML-SPT profile [25]) can be represented as a tuple:

\[ \text{message}=(\text{sender}, \text{receiver}, \text{methodOrSignalName}, \text{parameterList}, \text{returnList}, \text{startTime}, \text{endTime}, \text{msgType}) \]

where

- \text{sender} denotes the sender of the message and is itself a tuple of the form sender=(object, class, node), where:
  - \text{object} is the object (instance) name of the sender.
  - \text{class} is the class name of the sender.
  - \text{node} is where the sender object is deployed.
- \text{receiver} denotes the receiver of the message and is itself a tuple of the same form as \text{sender}.
- \text{methodOrSignalName} is the name of the method or the signal class name on the message.
- \text{parameterList} is the list of parameters for call messages.
- \text{returnList} is a sequence of the form \langle p_1, C_1, \text{in/out} \rangle, \ldots, \langle p_m, C_m, \text{in/out} \rangle, where \( p_i \) is the \( i \)-th parameter of class type \( C_i \) and \text{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, \text{in} \rangle, \langle o_2, C_2, \text{in} \rangle. If the method call has no parameter, this set is empty.
- \text{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 \text{var}_1=\text{val}_1, C_1 \rangle, \ldots, \langle \text{var}_n=\text{val}_n, C_n \rangle, where \text{val}_i is the return value for variable \text{var}_i with type \( C_i \).
- \text{startTime} is the start time of the message (modeled by UML-SPT profile’s RTstart tagged value).
- \text{endTime} is the end time of the message (modeled by UML-SPT profile’s RTend tagged value).
- \text{msgType} is a field to distinguish between signal, call and reply messages. Although the messageSort attribute\(^3\) 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

\(^2\) 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.

\(^3\) The messageSort attribute of a message specifies the type of communication reflected by the message [23], and can be any of these values: synchCall (synchronous call), synchSignal (synchronous signal), asyncCall, or asyncSignal.
explanations on this and an approach to distinguish between call and reply messages can be found in [14].

5.3.3 Network Traffic Usage Function

We present a Network Traffic Usage Function (NTU) in Equation 1, which is used to measure the amount of traffic entailed by a distributed message. A dash (-) symbol indicates that a field can take any arbitrary value (a “don’t care” field). 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 sum of data sizes of all the attributes of the signal class referred to by the message. For a call message (function CallDT 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, specified by the user.

\[
\text{NTU} : \text{Message} \rightarrow \text{Real}
\]

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

\[
\text{SignalDT(msg)} = \text{dataSize}(\text{msg.methodOrSignalName})
\]

\[
\text{CallDT(msg)} = \sum_{i \in \text{classDiagram}} \text{dataSize}(\text{C})
\]

\[
\text{ReplyDT(msg)} = \sum_{i \in \text{classDiagram}} \text{dataSize}(\text{C})
\]

Equation 1. Network Traffic Usage (NTU) function.

As an example, suppose call message msg has parameterList= \(<(\text{A},\text{B})>\), where classes A and B are defined in the class diagram of Figure 9.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute1 : long[100]</td>
<td>attribute1 : long[100]</td>
</tr>
<tr>
<td>attribute2 : long[500]</td>
<td>attribute2 : char[100]</td>
</tr>
</tbody>
</table>

Figure 9. Two classes with data fields.

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

\[
\text{NTU}(\text{msg}) = \text{CallDT}(\text{msg}) = \text{dataSize}(\text{A}) + \text{dataSize}(\text{B}) = (8 \times (100 + 500)) + (8 \times 100 + 1 \times 400) = 6 KB \text{ (kilobytes)}
\]

where the target programming language is assumed to be Java. long and char are among the primitive data types\(^4\) in Java. Data size of a long and a char variable in Java are four bytes and one byte, respectively.

5.3.4 Usage Pattern of Distributed Concurrent Control Flow Paths

Using NTU, we present a Network Traffic Usage Pattern (NTUP) function for DCCFPs in Equation 2.

NTUP is a function from the set of DCCFPs, networks, and time domain to real values (usage pattern values). The usage pattern of a DCCFP \(p\) on a network \(net\) at a particular time instant \(t\) is the sum of NTU values of the subset of DCCFPs’ messages whose start/end time interval includes \(t\) and that go through \(net\) (using getNetworkPath() in Section 5.2). \(Dur()\) denotes the time duration of a message and since a message can span over several time

\[
\text{NTUP}(p, net, t) = \sum_{\text{msg} \in \text{returnList}} \text{NTU(msg, t)} \text{ Dur(msg)} \text{ msg.start} \leq t \leq \text{msg.end} \text{ net \in \text{getNetworkPath}(msg.sender.node, msg.receiver.node)}
\]


As an example, the NTUP of DCCFP\((p)\), discussed in Section 5.3.1, is shown in Figure 10-(a). The network traffic usage values, shown in Figure 10-(b), are example values used when computing this NTUP. Figure 10-(a) shows that the maximum traffic entailed by this DCCFP is 80 KB at time instance equals to 13 ms.

5.4 Inter-Sequence Diagram Constraints

As discussed in Section 4.2, 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., SDs A and B in Figure 11-(a) are independent). An ISDS is the largest (maximal) set of SDs, in which any two SDs are independent, thus enabling all the SDs in the set to run concurrently. A MIOD can lead to several ISDSs and, as discussed in Section 6, the ISDS with maximum traffic (among all the ISDSs for a given MIOD) will be chosen to generate stress test requirements.

To derive the set of ISDSs of a MIOD, we first build a graph: nodes are SDs and there is an edge between two nodes if and only if the two corresponding SDs are independent. Finding the ISDSs is then a graph-based problem. More specifically, every strongly-connected component in this graph is an ISDS. (More details can be found in [15].) Standard algorithms can then be used to find those strongly-connected components.

For example, the MIOD of a power distribution controller system (used as a case study in Section 7) is shown in Figure 11-(a).

\(^4\) Data types that are not defined in terms of other types.
messages are all sent concurrently. This concurrent schedule of ISDS are considered in order to ensure we comply with inter-SD constraints. The heuristic can be informally visualized by the example in Figure 12.

For the ISDG in Figure 11-(b), four ISDSs are identified:

- ISDS₁ = {A, B, D, E}
- ISDS₂ = {A, B, F}
- ISDS₃ = {C, D, E}
- ISDS₄ = {C, F}

6. DERIVATION OF STRESS TEST REQUIREMENTS

This section describes how stress test requirements are derived from our test model.

6.1 Heuristics

Given a specific network to stress test, we identify a message (or a set of messages) in a DCCFP of a SD which imposes maximum traffic on the network. Let us refer to such messages as maximum stress messages. Then, using the start times of the maximum stress messages selected in each DCCFP, the selected set of DCCFPs can be scheduled in such a way that the maximum stressing messages are all sent concurrently. This concurrent schedule of DCCFPs will cause a maximum possible traffic on the selected network, which in turn will increase the probability of exhibiting distributed traffic- related faults in the system under stress test. 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. The heuristic can be informally visualized by the example in Figure 12.

Figure 11. (a): The MIOD of a power system. (b): The corresponding Independent-SDs Graph (ISDG). (c): One of the strongly-connected components of the ISDG (shown with dashed edges).

Figure 12. An example illustrating the heuristic.

Given the usage patterns of the DCCFPs of SDs in an ISDS, the heuristic is to first find the DCCFP of each SD, which has a message with the maximum traffic. In Figure 12, we assume we have three SDs and the corresponding DCCFPs are denoted DCCFP₀⁻max, i uniquely identifying SDs. The DCCFPs are then scheduled such that their maximum stressing messages are sent at the same time (right-hand side diagram in Figure 12). Note that the NIT is used to determine if a distributed message between two nodes goes through the selected network net.

6.2 Formulation as an Optimization Problem

According to the heuristic defined above, we formulate the stress test requirements derivation process as the optimization problem [13] in Figure 13.

Objective Function: Maximize the traffic on a specified network

Variables:
- A subset of DCCFPs and the number of instances of each DCCFP
- Schedule to run the selected DCCFP instances

Constraints:
- Inter-SD sequential and conditional constraints

Figure 13. Formulation as an optimization problem.

Note that multiple concurrent invocations of a SD might be allowed in a system, e.g., a SD which is triggered by five sensors concurrently. Therefore multiple DCCFP instances of such a SD can be executed to maximize stress during testing. Our technique derives the number of multiple invocations of a SD from the information specified in a system context diagram [16], 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 (SD₁, …, SDₙ), a test requirement will be a schedule of a selected set of DCCFPs in the form of:

(ρ₀⁻max, φ₀⁻max), …, (ρᵢ⁻max, φᵢ⁻max, …), where for the i-th entry of the sequence, ρᵢ⁻max is a DCCFP in the DCCFP set of SDᵢ, DCCFP(SDᵢ), that entails the maximum traffic over the selected network. φᵢ⁻max is the start time of ρᵢ⁻max, i.e., the time to trigger ρᵢ⁻max. Intuitively, if none of the DCCFPs of SDᵢ has any message going through the given network, it means that that SDᵢ does not have any traffic on the network and hence it will not be included in the test requirement. In such a case, the i-th entry is null.

6.4 Algorithm

We now present the algorithm which solves the optimization problem in Figure 13 and generates test requirements to stress test data traffic in a network net at any time instant, referred to as StressNetInstDT(net). The main steps of the algorithm are presented using the activity diagram in Figure 15. The algorithm is best explained by applying to an example.

Suppose the test model of a SUT with four ISDSs and six SDs as described in Figure 11 is built as specified in Section 5. Further assume that each SD X has two CCFPs (and DCCFPs), labeled ρₓ₋₁, ρₓ₋₂. Step 1 of the algorithm in Figure 15, which is to find DCCFPs for SDs with maximum network traffic, identifies DCCFPs ρₓ₋₁, ρₓ₋₂, ρₓ₋₁, ρₓ₋₁, ρₓ₋₁, and ρₓ₋₂ for SDs A to F, respectively. Note that in order to find those DCCFPs, Step 1.1 first finds the maximum traffic messages and determines the start times of such messages, which are used in Step 3. Step 1.2 selects...
the DCCFP with maximum traffic value, which is used for stress testing purposes. In our example, the maximum traffic values of DCCFPs (labeled $\max DT(p_{x,i})$ for each SD $X$ and DCCFP $j$) are specified below in KB:

$$
\max DT(p_{x,1}) = 10 \quad \max DT(p_{x,2}) = 15 \quad \max DT(p_{x,3}) = 12
$$

$$
\max DT(p_{x,4}) = 35 \quad \max DT(p_{x,5}) = 80 \quad \max DT(p_{x,6}) = 45
$$

Using ISDS as an example, Step 2 then calculates the total traffic of each ISDS as follows:

$$
\max DT(\text{ISDS}) = \sum_{i=1}^{\text{isds}} \max DT(p_{i,\text{on}})
$$

and similarly:

$$
\max DT(\text{ISDS}) = 70 \quad \max DT(\text{ISDS}) = 127 \quad \max DT(\text{ISDS}) = 57
$$

The ISDS with maximum DT value is then chosen: $\text{ISDS}_x$. The last step (Step 3) is to schedule the DCCFPs of $\text{ISDS}_x$ ($p_{B,1}, p_{B,2}, p_{B,3}, p_{E,1}$) so that their maximum stress messages happen at the same time. Assume that start times of maximum traffic messages for $p_{B,1}, p_{B,2}, p_{B,3}$, and $p_{E,1}$ are 5 ms, 10 ms, 15 ms, and 20 ms, respectively (Figure 14, left). Then the four DCCFPs are scheduled at times 15 ms, 10 ms, 5 ms, and 0 ms, respectively (Figure 14, right). Thus, since $A$ and $F$ are not part of the selected ISDS, the output stress test requirements will be:

$$
\langle p_{A,1}, 15\text{ms}\rangle, \langle p_{B,1}, 10\text{ms}\rangle, \langle p_{D,1}, 5\text{ms}\rangle, \langle p_{E,1}, 0\text{ms}\rangle, \langle p_{F,1}, 5\text{ms}\rangle, \langle p_{F,2}, 10\text{ms}\rangle,
$$

Figure 14. Stress test scheduling.

6.6 Discussions

Note that, due to space constraints, we only discussed one of our stress test heuristics in this paper, which is stress testing data traffic of a network in a single time instant. Alternatively, a network could be stress tested in a period of time. These tests might reveal different types of faults in a DS, as discussed in [15].

Our methodology has a class of 32 different test strategies to test different nodes and network in a DS, such as instant stress testing towards a node, or period stress testing from a node. The stress test parameters in Figure 1 are used to specify the type of testing strategy desired by a tester and other specific parameters of the strategy, such as the network to be tested for the strategy discussed in this article.

Furthermore, we presented the simplified version of our methodology in this paper where only sequential and conditional constraints were considered between SDs. DS usually have another type of constraints, referred to as arrival-pattern constraints [25], which relate to the timing of SDs. The time instant when a SD can start running might be constrained. Each single SD might only be allowed to execute in some particular time instants. For example, consider a SD which is triggered by a timer on a periodic basis, say every 5 seconds. In such a case, it is not possible to trigger the SD (or any of its DCCFPs) in time instant 4 seconds. As we discuss in [15], taking into account the arrival-pattern class of constraints requires more advanced optimization techniques, since they bring a new set of non-linear constraints into the optimization problem [15]. We present a more sophisticated optimization technique in [15], to derive stress test requirements while complying with arrival-pattern constraints of a DS (in addition to sequential and conditional ones).

7. CASE STUDY

Our stress test methodology can be used to stress test distributed systems, with perhaps an emphasis on safety-critical and data-intensive systems. Distributed Control Systems (DCS) [18] and Supervisory Control and Data Acquisition (SCADA) Systems [10] are two kinds of such systems.

We surveyed numerous existing systems (e.g. [7, 9, 11, 12]) 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. [5, 27, 30]) 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 [28]: (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 (Section 4). The UML model was defined and the system was implemented using Borland Delphi [3], which is a well-known Integrated Development Environment (IDE) for Rapid Application Development (RAD).

The objective of our case study was to compare the durations for SDs D and E in Figure 11-(a) by running Operational Profile Tests (OPT) and Stress Tests (ST). The two SDs are parts of the power application software model. Recall there was a time constraint on these SDs’ executions and our goal is to assess whether stress testing can help detect violations of this constraint. We considered OPTs as a useful baseline of comparison as test cases are derived from the operational profile of SCAPS. This is a common testing practice to assess a system based on its expected usage in the field [20]. To derive operational profile test cases, we assumed an operational profile for SCAPS, which takes into account its 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 [28].

Recall that we modeled a HRT constraint in the MIOD of SCAPS in Figure 11-(a). This specifies the maximum acceptable value for the durations of SDs D and E: it should be less than 1,300 ms (milliseconds). Figure 16 shows the observed values of this duration by running 500 Operational Profile Tests (OPT) and Stress Tests (ST). The X-axis is the test type and the Y-axis is duration in milliseconds. The quantile regions and the histograms of the two distributions are also depicted, and reported in Table 2.

Due to indeterminism in distributed environments, it is expected the duration of distributed messages can be different across different executions. Figure 16 shows that, despite such a variation, all OPTs executions satisfy the HRT constraint above, since all their durations are less than 1,300 ms (bold horizontal line). However, showing sharply different results, the constraint is violated in %96.4 (482/500) of stress test cases. These results suggest that our ST strategy is much more effective at stressing the system to exhibit violations of our HRT constraint than standard, operation profile-based testing.

When the experiments show that there can be scenarios in which one or more of the RT constraints can be violated in a DS, such as in our example above, performance engineering techniques should be applied to redesign the system and/or increase resources (for example, network capacities) or change the RT constraint values to more realistic levels. We present a Stress-Test based Performance Engineering (STPE) approach in [15] which can assist testers and system analysts in fixing the distribution-related faults. Following STPE, the designer uses stress test results to evaluate the performance throughput of a SUT, analyze missed Real-Time constraints, and provide guidelines to enhance performance and robustness of the system in terms of Real-Time constraints.

<table>
<thead>
<tr>
<th>Level</th>
<th>Min.</th>
<th>10%</th>
<th>25%</th>
<th>Median</th>
<th>75%</th>
<th>90%</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPT</td>
<td>953</td>
<td>1029</td>
<td>1059</td>
<td>1094</td>
<td>1125</td>
<td>1156</td>
<td>1241</td>
</tr>
<tr>
<td>ST</td>
<td>1276</td>
<td>1305</td>
<td>1317</td>
<td>1329</td>
<td>1344</td>
<td>1358</td>
<td>1382</td>
</tr>
</tbody>
</table>

8. CONCLUSIONS

A stress test methodology aimed at increasing chances of discovering faults related to network traffic in distributed systems was presented. The technique uses the UML 2.0 model of a system, augmented with timing information. In particular we were very careful to design an adequate and realistic input test model precisely, which is one of the main contributions of this article. It mainly entails (1) a Network Deployment Diagram (following the UML package notation) that describes the distributed architecture in terms of system nodes and networks and (2) a Modified Interaction Overview Diagram (following the UML 2.0 interaction overview diagram notation) that describes execution constraints between sequence diagrams. Our stress testing technique relies on a careful identification of the control flow in UML 2.0 Sequence Diagrams and the network traffic they entail. These data are used to generate stress test requirements composed of specific control flow paths (in Sequence Diagrams) along with time values indicating when those paths have to be triggered so as to stress the network the to the most extent possible.

Using the specification of a real-world distributed system, we designed and implemented a prototype system and described how the stress test cases were derived and executed. We furthermore reported the results of applying our stress test methodology on the prototype system and discussed its effectiveness in detecting violation of a hard real-time constraint when compared to test cases based on an operational profile. Our first results are promising as they suggest that our stress test cases can help to increase the probability of exhibiting network traffic-related faults in distributed systems.

Some of our future works include: (1) experimenting with the other 31 stress testing techniques that we mentioned as being defined, (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 (3) specifying, visualizing, and analyzing stress test requirements and stress test process using the UML 2.0 Testing Profile [24], (4) risk
assessments and fault analysis of distributed-type faults, and (5) tool support. We also plan to stress test more complex distributed systems and investigate the effectiveness of the methodology.

ACKNOWLEDGMENTS
This work was in part supported by Siemens Corporate Research, Princeton, NJ and a Canada research chair.

REFERENCES