Information and Software Technology 52 (2010) 625–640

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Experience and challenges with UML-driven performance engineering of a Distributed Real-Time System

Vahid Garousi

Software Quality Engineering Laboratory (SoftQual), Department of Electrical and Computer Engineering, Schulich School of Engineering, University of Calgary, 2500 University Drive NW, Calgary, Alberta, Canada T2N 1N4

ARTICLE INFO

Article history:
Received 6 July 2009
Received in revised form 14 January 2010
Accepted 18 January 2010
Available online 1 February 2010

Keywords:
Software Performance Engineering
Performance tuning
Stress testing
Experiment
Distributed Real-Time Systems
UML

ABSTRACT

Context: Performance-related failures of Distributed and Real-Time Software Systems (DRTS's) can be very costly, e.g., explosion of a nuclear reactor. We reported in a previous work a stress testing methodology to detect performance-related Real-Time (RT) faults in DRTS's based on the design UML model of a System Under Test (SUT). The stress methodology aimed at increasing the chances of RT failures (violations in RT constraints).
Objective: After stress testing a SUT and finding RT faults, an important immediate question is how to fix (debug) those RT faults and prevent the same RT violations in the future and after deployment. If appropriate solutions to this challenge cannot be found, stress testing and its findings (detection of RT faults) will be of no or little use to the quality assurance goals of the development team.
Method: To move towards systematically solving performance-related problems causing RT faults, we develop a customized version of the standard Software Performance Engineering process and conduct an experiment on a DRTS. The process is iteratively applied to a SUT, while results from stress testing reveal that there are still scenarios in which RT constraints are violated.
Results: Application of the performance engineering paradigm in this context on a real DRTS enables systematic analysis of performance-related defects and their fixations.
Conclusion: The contributions of this work are an initial approach to software performance engineering based on stress testing, and an analysis, based on experimentation, of the open issues that need to be addressed in order to improve the approach.

1. Introduction

Distributed Real-Time Systems (DRTS's)[1] are becoming more important to our everyday life. Examples include command and control systems, aircraft aviation systems, robotics, and nuclear power plant systems [1]. The development and testing of such a system is difficult and takes more time than the development and testing of a distributed system without Real-Time (RT) constraints or a centralized system running on a single computer [2].

Performance-related defects in software systems can be very costly; e.g., the controller software of a nuclear reactor should send appropriate control signals every two seconds or catastrophic results might occur. Such defects can result in catastrophic and life-threatening outcomes (e.g., explosion of a nuclear reactor), damaged customer relations, cost and time overruns due to performance tuning activities, and missed market windows.

Sources of performance failures in the United States Public Switched Telephone Network (PSTN), as a large system, are investigated in [3]. 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. Hence, it is evident that although overloads do not happen frequently, the failure resulting from them can be quite expensive.

DRTS's offer unique challenges for performance engineering due to the complexity of interactions between different distributed nodes, components and their concurrent nature. Therefore, utilizing SPE techniques to systematically locate, fix and prevent RT faults in those systems is critically required. We (software engineers) cannot afford to deploy safety-critical DRTS's which have the potential to cause catastrophic consequences due to performance-related defects.

Furthermore, according to the literature on power distribution systems implemented using distributed software systems (e.g., [4,5]), if RT constraints are not met in these systems, e.g., power is not restored *on time* after a network failure in a part of a power distribution grid, costly power blackouts are inevitable, e.g., the Northeast North American blackout on August 2003 whose out-

E-mail address: vgarousi@ucalgary.ca

[1] For reading convenience, a list of acronyms is provided in the appendix.

0950-5849/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2010.01.003

age-related financial losses alone were estimated at $6 billion USD [6].

We presented in [7] a stress test methodology aimed at increasing the chances of discovering RT faults related to network traffic. Two extended versions of the base methodology were also presented in [8,9]. The technique in [7] uses the UML 2.0 [10] model of a distributed System Under Test (SUT), augmented with timing information, and is based on an analysis of the control flow in UML sequence diagrams. It yields stress test requirements that are constructed from specific control flow paths along with time values indicating when to trigger them.

The technique has been evaluated empirically in a number of studies [11,12] on real and hypothetical DRTS's and the results have confirmed the robustness and effectiveness of the stress test technique in revealing RT faults. In one experimentation [7], we used the specifications of a real-world power grid controller system to design and implement a prototype system. We then derived, for that particular system, the stress test suites and executed them using our test methodology. The stress test results indicated that the methodology is significantly more effective at detecting RT faults when compared to test cases based on an operational profile [13], in which the system was tested under its expected workload.

After stress testing a SUT and finding RT faults, an important immediate question is how to fix (debug) those RT faults and prevent the same RT violations in the future and after deployment. If appropriate solutions to this challenge are not provided, the stress testing and its findings (detection of RT faults) will be of no or little use to the organizations' quality assurance goals. In our case study [7], by a root cause analysis of faults, all RT faults were found to have been caused by performance-related defects and thus the bottom-line to eliminate RT faults is to correct the performance issues which were the causes of those RT faults.

To experiment with systematically solving performance-related problems which can lead to RT faults, we present in this paper a performance engineering process for DRTS's, referred to as *Stress-Test Performance Engineering (STPE)*, based on stress testing and an experience report using that process when applied to a prototype DRTS. Although we are able to solve the performance-related failures in our experiment (Section 5), the experiment raises several important questions and practical challenges in UML-driven performance engineering of DRTS's as open issues (Section 6).

To design the STPE process, we customize and adapt a well-know paradigm called *Software Performance Engineering (SPE)* [14]. SPE is a paradigm for general software systems, so we have adapted it for the context of DRTS's, stress testing and UML-driven development processes.

While the SPE literature is extensive (e.g., the list of references in [15]), no existing work has studied performance engineering of DRTS's based on stress test results. Furthermore, no existing work has provided performance tuning guidelines to help developers cope with RT faults in UML-driven developments. Resolving this need is one of the contributions of the current work. The other contribution of our experience report is to highlight the practical questions and challenges in performance engineering of these types of systems. For example, we find out that cost-effective performance engineering, in this context, would require designing an intelligent and systematic decision-support mechanism to support performance engineering decisions. We identify the challenges in building such a mechanism, and aim at offering solutions to some of those challenges.

The rest of this article is structured as follows. Related works are discussed in Section 2. To better understand the proposed performance engineering process and the experiment, some background information is presented in Section 3. Section 4 presents the customized process. Section 5 reports the results of an experiment in which the process is applied to a prototype DRTS. Section 6 evaluates our methodology by comparing it to a few related methodologies and discusses the main research questions (open issues) raised by this study to be addressed by future works. The appendix provides a list of acronyms used in this article.

2. Related work

There have been many works in the Software Performance Engineering (SPE) literature (e.g., the list of references in [15]), focusing on different type of performance testing (e.g., stress and load testing), different types of systems under test, and different types of performance-related faults.

For space constraints, we limit our focus in this article to the most relevant work which uses SPE practices in the context of stress testing (e.g., [2,16]), and distributed real-time systems (e.g., [17,18]). For a more comprehensive survey of related work, readers are referred to [19].

Note that (system) performance engineering is more than just Software Performance Engineering. Performance engineering deals with tuning all computing (IT) resources (including both hardware and software) to meet performance non-functional requirements for a system [20]. For example in the current work, we test, measure and tune both hardware and software components to conduct the performance engineering tasks.

The SPE literature can be classified into two broad categories [21]: (1) model-based analysis and (2) monitoring (measurement)-based approaches. The existing works in the first category (e.g., [2]) build analytical performance models, e.g., Layered Queuing Networks (LQN) [22], to analyze and tune performance. The works in the second category apply performance testing techniques, e.g., stress and load testing, to detect faults. Each category has its own advantages and disadvantages, e.g., model-based techniques can be used even before the system is developed, but the executable system should be ready for applying testing-based techniques. On the other hand, techniques of the 2nd category usually provide more realistic and robust results compared o the 1st category as the real system is performance tested instead of a representative model. Further comparisons on the above two groups can be found in many SPE literature, e.g. [14].

The work in [2] reports upon the issues and the experiences with performance testing of industrial software systems. The authors reveal some interesting empirical insights with respect to the locality of performance-related problems, e.g., they found that 93% of the performance-related project-affecting issues in an industrial software project were concentrated in the 30% of source code sections that were deemed to be most problematic. This observation confirmed the authors' experience that projects that are in "good shape" are very likely to recognize that addressing performance issues is a necessary part of the architectural phase of development.

During the industrial performance testing project in [2], the authors were able to uncover and correct several software faults that substantially compromised the performance behavior of the SUT. They were also convinced that if the system had been deployed without having identified and corrected these problems, the system would have provided performance that would have been viewed by their customers as being entirely unacceptable. As a summary, the experience paper in [2] reveals useful and practical issues into the performance testing of industrial systems. However, it does not present an iterative SPE process which can provide performance tuning guidelines to help developers prevent RT faults. Furthermore, similar to [16,2] does not discuss the notions of RT constraints and violations in understanding and predicting performance, but rather discusses general types of performance faults (e.g., unacceptable throughput).

The work in [16] presents a methodology for understanding and predicting the performance of component-based distributed systems both during development and after they have been deployed. The presented methodology includes three parts: (1) monitoring, (2) modeling and (3) performance prediction. Performance predictions are based on UML models created dynamically by monitoring and analyzing a live or under-development system. The system is monitored using non-intrusive methods and runtime data is collected. In addition, static data is obtained by analyzing the deployment configuration of the target application. UML models enhanced with performance indicators are created based on both static and dynamic data, and are used by the methodology to pinpoint performance bottlenecks. Although the methodology in [16] can be used for understanding and predicting the performance of distributed systems under stress conditions, it does not deal with RT constraints and their violations in understanding and predicting performance.

There are a variety of approaches to applying SPE practices to distributed real-time systems (e.g., [17,18]). The work in [17] presents a toolset for performance engineering of client–server systems. The presented toolset offers the possibility of early performance evaluation of designs for several types of layered-service systems, including client–server, distributed, and transaction processing systems. The toolset was reported to be able to identify the features that affect the performance of layered-service systems, e.g., the placement of modules in tasks, and the number of clients. These features were modeled and formulated in an analytical performance model notation, referred to as "stochastic rendezvous networks". To get the best performance configuration, this model was *solved* by performance solvers through analytical solvers and simulations. The two solution approaches are complementary: analytic techniques were used to explore alternatives, with simulation techniques used to confirm the results in practice.

The paper in [23] provides interesting management-level insights into the SPE of distributed systems by identifying two types of performance management practices used in organizations: reactive and proactive performance management. The authors believe that reactive techniques are inappropriate for today's distributed systems, since if critical problems are detected late in development, it may be difficult and expensive to correct them before deployment, or even impossible to correct them without a major re-design.

In summary, we have reviewed above some of the relevant SPE practices which are applied in industry to date. However, no experience has been reported with regard to the study and integration of all the three aspects of performance engineering (measurement, evaluation, and tuning) for systems based on stress test results with a focus on RT faults. Our stress test methodology reported in [7] belongs to testing-based SPE category since an actual SUT is performance tested by the test cases generated using the methodology.

The novelty of the current work with respect to three of our relevant earlier publications [7–9] is as follows. All those three earlier contributions were only stress test techniques. After using any of those three techniques to detect RT faults in a SUT, none of them addressed the important question of how to fix (debug) those RT faults. In other words, those works did not cover the performance engineering aspect of dealing with RT faults, and that is the main contribution of the current work. Any of our three previous techniques [7–9] can be used in conjunction with the current STPE process to complete the *loop* in performance engineering of DRTS's: detecting RT fault and taking measures to resolve (fix) them. The relationships among our three earlier contributions [7–9] are as follows.

The *Time-Shifting Stress Test Methodology (TSSTM)* [7] was our first and base approach with two simplifying assumptions: (1) it assumed that the timing information of messages in the SUT is predictable or precise (as specified in its UML models) and (2) TSSTM is only applicable to DRTS's without timing constraints (e.g., arrival patterns) for RT tasks. The techniques reported in [8 and 9] which were extended versions of TSSTM were developed to address the above two simplifying assumptions of TSSTM, respectively.

In order to devise deterministic test requirements (from time point of view) that yield a stress test scenario of network traffic in a SUT, our TSSTM methodology [7] required that the timing information of messages in SDs is available and as precise as possible. The determinism of a test requirement in this context corresponds to the certainty by which executing test cases corresponding to that test requirement will maximize traffic on a given network. However in reality, the timing information of messages is not always available and/or precise (certain) before execution. Furthermore, such timing information can change across different executions. In other words, TSSTM might generate imprecise and not necessarily maximum stressing test cases in the presence of such time uncertainty and, thus, it might not be very effective in revealing RT faults. To address the above limitation of TSSTM, we presented in [8] a modified testing methodology, referred to as *Wait-Notify Stress Test Methodology (WNSTM)*, which can be used to stress test systems in which the timing information of messages is unpredictable or imprecise.

To take into account different types of event arrival patterns (e.g., periodic arrival) common in DRTS's while generating stress test requirements, we reported a technique referred to as *Genetic Algorithm-based Stress Test Methodology (GASTM)* in [9]. Such patterns impose constraints on the time instant when interactions between distributed objects can take place. We made use of specifically-tailored Genetic Algorithms to automatically generate test requirements which comply with such timing constraints and lead to high traffic-aware stress in a SUT. While, if the base approach (TSSTM) is applied to those SUTs, it will generate strenuous but possibly *invalid (illegal)* test requirements that will possibly violate the timing constraints of RT tasks.

3. Background

Since our SPE approach is based on RT constraints and their violations, a brief introduction to RT constraints in DRTS's is presented in Section 3.1. An overview of our stress test methodology [7], as the background to this work, is provided in Section 3.2.

3.1. An overview of Real-Time constraints

Real-Time (RT) constraints are timing constraints on operations in RT systems. There are usually two types of RT constraints: *hard* and *soft* [1]. A hard RT constraint on an operation enforces that the operation *must* complete within the specified time frame or the operation is, by definition, incorrect, unacceptable, and usually has no value.

On the other hand, in the case of a soft RT constraint for an operation, the value of the operation declines steadily after the deadline expires. Tasks completed after their respective soft RT deadlines are less important than those whose deadlines have not yet expired [1].

The Unified Modeling Language (UML) [10] is increasingly used in the development of DRTS's (e.g., [24]). Therefore, we assume that RT constraints are specified in UML models of a SUT and are provided to our stress test methodology. To model RT constraints in UML models, the UML profile for Schedulability, Performance, and Time (UML-SPT) [25] proposes comprehensive modeling constructs to model timing information. Although UML-SPT briefly mentions soft and hard RT constraints (Section 2.2.3 of [25]), it

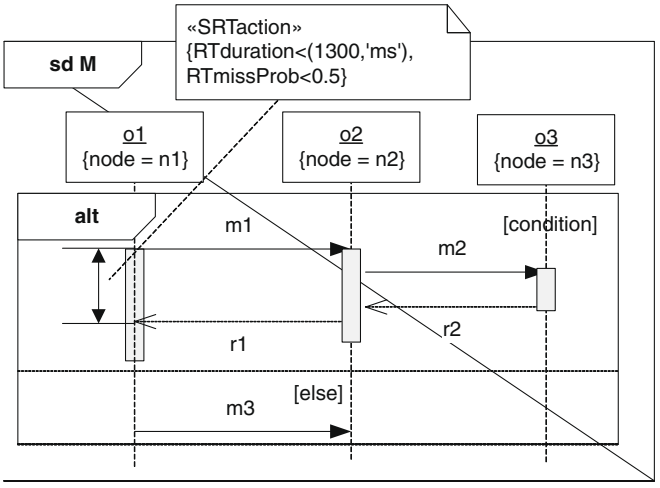


Fig. 1. An example usage of the «SRTaction» stereotype in a UML sequence diagram.

does not propose any specific stereotypes to distinguish between them.

Note that the UML-SPT was the standard profile when this research was conducted. As of this writing (January 2010), the Object Management Group (OMG) has adopted a beta specification is working on the finalization stage of a new improved profile, called the *UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE)* [26], which is expected to replace SPT in the near future. From an objective standpoint, the framework reported in this paper can easily be modified to be applicable with model developed using MARTE as well since for the purpose of analysis in this paper, only the format of the tagged-values in SPT and MARTE are different. For example, to specify the deadline of 1300 ms for a RT message, the SPT [25] and MARTE [26] notations are as follows, respectively: `RT duration < (1300 ms)`, and `relDl = (1300` ms). relDl stands for relative deadline.

In order to distinguish hard and soft RT constraints, as the UML-SPT profile [25] does not provide any stereotypes, we proposed in [27] two extensions to the *RTaction* stereotype of the UML-SPT referred to as *HRTaction* (hard RT action) and *SRTaction* (soft RT action). As of this writing (January 2010), the adopted beta specification of MARTE [26] also still has no defined stereotype to distinguish soft vs. hard RT constraints.

Furthermore, in order to model the statistical threshold probability up to which SRT constraints are allowed to be violated, we defined a tagged-value referred to as *RTmissProb* for SRT constraints [26]. The extreme values for *RTmissProb* values denote "that no violation is allowed" (*RTmissProb* = 0) and "that all violation instances are allowed" (*RTmissProb* = 1). In the former case, a SRT constraint turns into a HRT constraint since all violations are interpreted as a RT fault. In the latter case, a SRT constraint is in fact not a constraint since all of its violation instances are allowed and not considered a RT fault.

Similarly, we presented a tagged-value referred to as *RTcriticality* for HRT constraints, [0, . . ., 1], which indicates the degree to which the consequences of missing a hard RT deadline are unacceptable. The extreme values for *RTcriticality* values, i.e., 0 and 1, denote "virtually no serious consequences" and "catastrophic results for violating" a HRT constraint, respectively. An example usage of the *SRTaction* stereotype in a Sequence Diagram (SD) is demonstrated in Fig. 1. The SRT constraint denotes that the execution duration of the message sequence $\langle m1, m2, r2, r1\rangle$ should not exceed 1300 ms (ms). As the *RTmissProb* value specifies, this constraint may be violated across different executions up to a probability of 0.5.

RTmissProb and *RTcriticality* values are used in the performance evaluation stage of our performance engineering methodology (Section 4.2) to assess whether a SRT or a HRT constraint, respectively, is violated when the SUT is executed for a large number of times. Those values are usually provided by domain experts and in consultation with the clients of the system [1].

3.2. An overview of our stress test methodology

An overview of the TSSTM stress test methodology [7] is presented using the UML activity diagram in Fig. 2. Only the steps with a gray background are addressed in [7]. A UML model of a SUT, following specific but realistic requirements, is used as input. Note that context diagrams in the input do not belong to the standard UML 2.0, but they have been used in a many OO methodologies, e.g., the Concurrent Object Modeling and Architectural Design Method (COMET) framework [28].

A test model is built to facilitate subsequent automation steps. As shown in Fig. 2, such a model includes, among other entities, the control flow information of the set of all SDs in the SUT in the form of control flow graphs. The test model and a set of stress test parameters (e.g., the node or network to be under stress test) 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 the TSSTM methodology [7] was to choose the maximum number of sequence diagrams (to create maximum possible traffic) which can realistically be executed concurrently (according to the business logic of a SUT), and schedule them such that their maximum traffic messages run concurrently.

TSSTM [7] was our base stress test methodology, which requires the timing information of the messages in the system to be deterministic. Also it was not applicable to RT systems in which messages could have arbitrary arrival patterns.

To address the above two limitations of TSSTM [7], we have developed two additional stress test methodologies called:

- Wait-Notify Stress Test Methodology (WNSTM) [8] to address the first above shortcoming.
- Genetic-Algorithm-based Stress Test Methodology (GASTM) [9] to address the second above shortcoming.

The reader can refer to the above three papers for details on TSSTM [7], WNSTM [8] and GASTM [9].

4. Stress test performance engineering

We use the ideas from the SPE paradigm [14] to devise the STPE process specific to DRTS's and stress testing. Similar to SPE [14], our STPE process is an iterative process in which the SUT is stress-tested until no performance-related defects (i.e., RT violations) are discovered. In each iteration (cycle), STPE uses stress test results to evaluate the performance of a SUT, analyze missed RT constraints, and provide performance engineers with guidelines for enhancing performance.

Based on the above introduction, both SPE and our STPE process are illustrated using the UML activity diagrams in Fig. 3, where nodes with soft and hard edges correspond to activities (steps in the process) and data, respectively [10]. For easier referencing, activities of the STPE are numbered. Activities of the SPE are also numbered in a way that the correspondences of STPE activities to those of SPE are identified, e.g., the "Generate Stress Test Requirements" activity of the STPE corresponds to the "Construct Performance Models" activity of the SPE.

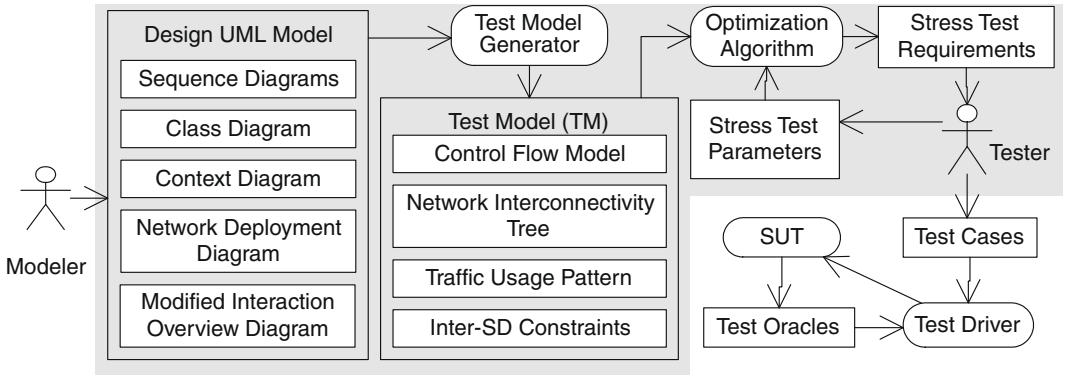


Fig. 2. Overview of the TSSTM methodology [7].

The activity nodes of SPE are quite self explanatory. For further details, the reader is referred to [14]. We explain next our STPE process and its relationship to SPE. Note the two main building blocks of SPE and STPE (i.e., performance measurement and tuning) are similar. The differences in some of the activities is due to the fact that SPE was proposed for general software systems and was intended to support both model-based analysis and monitoring (testing)-based performance approaches, while our STPE is a more specific monitoring-based approach based on a specific testing technique and is targeted for SPE of DRTS's, detecting and fixing their RT violations. The similarity of our approach to SPE can be followed by relating the activities between the two similar processes.

In STPE, the design-phase UML model of a SUT is provided as an input, which is used to derive stress test requirements using either our TSSTM [7] or WNSTM methodology [8], depending on whether the timing information of messages in the SUT is predictable or not. The *cycle* variable and the transition guards using the *cycle* variable in Fig. 3 relate to the iterative nature of STPE. Recall from above that a SUT is stress tested with STPE until no performance-related defects are encountered. The *cycle* variable keeps track of the STPE cycle (iteration) number.

We discuss next the activities of the STPE process in detail. The activities are grouped into three high-level steps (shown as complex activities in Fig. 3):

- Performance measurement.
- Performance evaluation.
- Performance tuning.

4.1. Performance measurement

Step 1. *Generate Stress Test Requirements*: The first step of performance measurement in STPE is to use either the TSSTM [7], WNSTM [8] or GASTM [9] to generate stress test requirements which maximize the probability of revealing RT faults. The decision to use the right stress test methodology is based on the properties of the SUT, i.e., if its timing information are precise and whether its tasks have arrival patterns (refer to Section 3.2 for details). For brevity reasons, and without loss of generality, we assume in the remaining of this section that the TSSTM [7] is used. Stress test requirements generated by TSSTM are made of specific control flow paths along with timing values indicating when to trigger them.

Step 2. *Generate Stress Test Cases*: TSSTM only generates stress test requirements, while testers build stress test cases corresponding to those test requirements. The steps 1 and 2 correspond to the "Construct Performance Models" step of SPE (Fig. 3). Stress test requirements are made of specific control flow paths along with timing values indicating when to trigger them. To actually be able to trigger a specific control flow path, appropriate set of input values or conditions should be generated, e.g., in a boiler control system, the boiler temperature should be 120 °C and the all the control sensors should send their data concurrently. There is a large body of knowledge to generate test input data once test requirements are known (e.g., [29]).

Step 3. *Run Stress Test Cases*: The next step of performance measurement in STPE is to run the stress test cases derived in the previous step to assess the performance of the SUT. By executing the stress test cases, we evaluate the performance of the SUT under the most stressed conditions and determine whether the SUT is prone to RT faults under such conditions. Since the execution times in DRTS's can be indeterministic [1], due to, e.g., unexpected network delay, jitter, etc., the engineers should execute each test case for a large number of times (e.g., 500, or 1000) and then study the test results statistically.

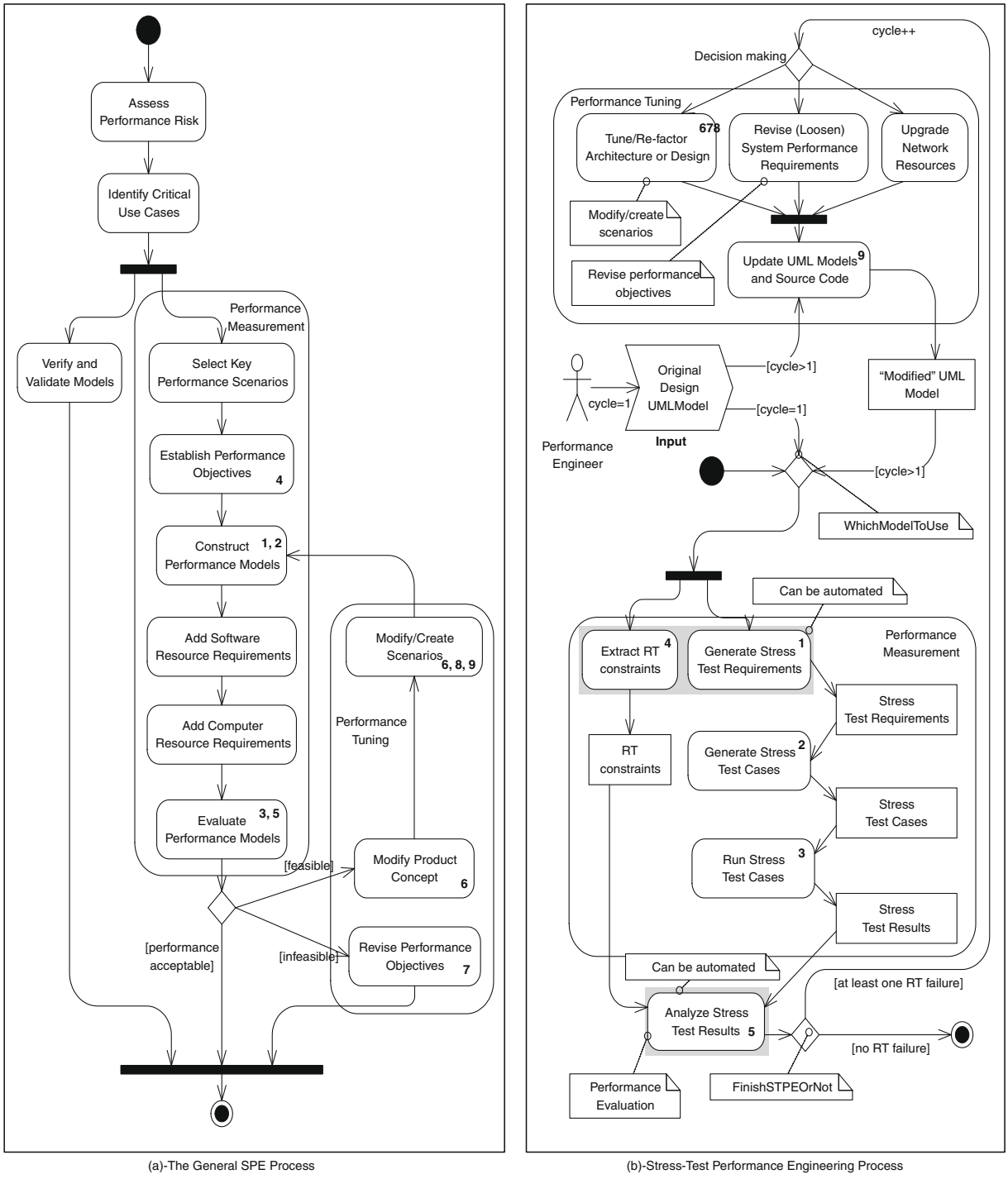
Step 4. *Extract RT constraints*: Concurrently to generating and running stress test cases, the RT constraints of the SUT are extracted from its UML model to be used in performance evaluation (Step 5), which serve as the performance objectives (corresponding to Step 4 of the SPE process in Fig. 3).

4.2. Performance evaluation

Step 5. *Analyze Stress Test Results*: At this stage, we have both stress test results (measured durations of messages bound by RT constraints) from Step 3, and the RT constraints (performance objectives) from Step 4. It is now time to compare the actual test case results to the performance objectives. In other words, the runtime duration of the message(s) bounded by each RT constraint should be compared to the deadline value of the corresponding RT constraint.

Due to the nature of HRT and SRT constraints, STPE evaluates those two constraints separately. For each HRT constraint, if the corresponding measured duration of the HRT constraint is greater than the specified duration, a HRT fault is said to have occurred (i.e., the stress test case has revealed a failure).

On the other hand, for each SRT constraint, the ratio of the number of missed instances of the constraint over total number of runs is calculated. For example, suppose a SRT constraint is violated 30 times in 100 executions of a test case. In this case, the above ratio will be 30%. Only if the ratio is greater than the maximum miss



(a)-The General SPE Process (b)-Stress-Test Performance Engineering Process

Fig. 3. The relationship of our Stress-Test Performance Engineering Process to the general SPE Process [14].

probability (*missProbability*) of the SRT constraint (e.g., 15%), a SRT fault is said to have occurred.

Root-cause analysis: In case of any RT failure, the root-cause(s) for poor performance (causing that failure) are identified by this step as follows which will be used as guidelines for corrections in the next performance tuning steps. A RT failure means that the particular stress test requirement generated by the stress test methodology is entailing a traffic load on the network or the node under test which it cannot handle on time. For example, assume that the execution of a test requirement ends up a RT failure. Since the network the or node under stress, the control flow paths and also the specific messages inside each of those paths are known (from the test requirement), we can use that information as guidelines for corrections in performance tuning steps (next).

4.3. Performance tuning

The performance tuning stage of STPE is undertaken if and only if the performance evaluation in Step 5 reports that at least one RT failure (HRT or SRT) has occurred. It is evident that, after finding performance-related defects using our approach, a performance engineer may have several alternatives to tune (fix) them.

The performance tuning stage of STPE has been designed to be an *open* activity, i.e., any *suitable* performance tuning activity may be applied at this stage. By suitable, we mean those activities which can eliminate the chances of RT failures. As a starting point, we are proposing three such activities in this work: Steps 6–8.

Step 6. *Tune/Re-factor Architecture or Implementation*: According to our stress testing experiments in [7], some RT faults occur due to overloads in network traffic when distributed nodes are communicating with each other. Thus, if a system's architecture is refactored such that some of the messages exchanged across distributed nodes are replaced by local messages (inside a node), the probability of RT faults associated with overload in network traffic may decrease. Other architecture tuning practices such as load balancing [30] and message prioritization [31] may also be utilized to mitigate such performance-related defects.

Step 7. *Revise (weaken) System Performance Requirements*: This activity is an adaptation of the "Revise performance objectives" stage of the SPE. As mentioned in the SPE, if no feasible cost-effective alternative is found to satisfy a set of performance objectives, we need to modify the performance goals to reflect this reality. It may not seem reasonable to change the performance objectives if we can't meet them (*if we can't hit the target, we redefine the target*); however revising the performance objectives is not wrong if it is done at the outset of the project and its business objectives, as mentioned in the SPE and also followed in software industry. For example, by discussing such a performance limitation with all stakeholders of a industrial robot controller project, it might be acceptable (e.g., safe) in the context of the project to increase a HRT constraint from 2 to 3 s, while making sure that the new constraint will be met in all worst-case scenarios.

Step 8. *Upgrade Network Resources*: Another way to reduce the chances of RT faults due to network traffic is to increase network resources and infrastructure. According to our stress test experiments in [7], network resources (e.g., the bandwidth of a network) are often performance bottlenecks leading to RT faults. Thus, if we can pinpoint the bottleneck network resource(s), we can either upgrade or replace the resource(s) with more capable ones, e.g., increasing the bandwidth of a network from 10 to 100 megabits per second (Mbps).

4.4. Decision-support for cost-effective performance tuning

Although each of the above alternatives can be useful in solving the performance-related defects, performance engineers are usually under technical, economic, and strategic constraints in choosing an alternative to conduct the tuning step (Fig. 4). This is similar in nature to the constraints engineers face when choosing to use Commercial off-the-shelf (COTS) products in software development [32]. The best solutions lie in the middle intersection of all three types of constraints.

An example technical constraint in our context is that although we desire to upgrade the network resources (e.g., to more than 100 Mbps), some of the specific-purpose hardware devices that we currently have will no longer be operational with the ultra-speed networks. As another example, if we want to refactor the system architecture in particular way, that would not be allowed by the middleware that we currently have in use (e.g., CORBA).

Economic constraints are usually wide-spread in software projects, e.g., if we decide to proceed with network resource upgrades to satisfy the violated RT constraints, we would go beyond the project budget. Strategic constraints in this context generally refer to our contracts with our clients and information from the DRTS domain experts, especially in terms of the predefined RT constraint values.

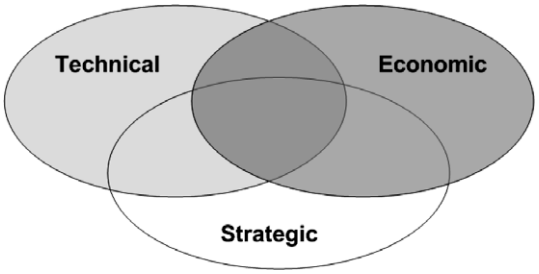


Fig. 4. Three types of constraints in choosing an alternative to conduct performance tuning.

Choosing the best performance tuning alternative would require paying careful attention to all the above three types of constraints in the context of a given system. The decision-making challenge is not trivial and ad hoc decisions cannot help much in general, e.g., one might choose to increase the network bandwidth (or other infrastructures) by a large arbitrary amount (which means extra costs), while fixing the detected RT failure could have been done by a minor change (refactoring) in the system code. Thus, we need to have a decision-support mechanism in place to help performance engineers choose the most cost-effective activity (shown by a decision node before performance tuning in Fig. 3).

4.4.1. A decision-making heuristic

Based on the software-engineering decision-support body of knowledge [33], we provide a simple rank-based heuristic to support performance engineers in choosing the most cost effective tuning alternative. The current notion of cost-effective performance tuning is similar in goals to two related subjects, namely *value-based software engineering* [34] and *value-based management of software testing* [35]. We thus similarly refer to our heuristic as *Value-Based Performance Engineering (VBPE)*. Note that this is a first step in devising such a heuristic and further more sophistical versions are sought to be developed using the advances in software-engineering decision-support body of knowledge [33].

For brevity, let us refer to the three tuning activities (steps) 6–8 as *REF*, *RPR*, and *UNR*, respectively. To compare cost-effectiveness of each alternative, let *cost(alternative)* denote the estimated cost of each alternative. We essentially need to incorporate the three dimensions of constraints (Fig. 4) into one quantitative value, which in general is a challenging task. In the state-of-the-art and also – practice [33], the effort is usually spent to (try to) convert the technical and strategic dimensions into the economic dimension as well. For example, in some cases, the stakeholder(s) might agree to the *RPR* option but with a reduction in project's payable revenue since the quality of the system (as a whole) might be perceived as been decreased. In some other cases, the *RPR* option might not be feasible (acceptable) at all.

Also, a major technical limitation in performance tuning might be solved by spending a large amount of cost (e.g., changing the middleware of a DRTS from CORBA to another platform).

To calculate the rankings of the three alternatives: *REF*, *RPR*, and *UNR*, we need to apply various techniques to estimate the dollar expenditure values for each. Fortunately, the guidelines provided by Step 5 (analyze stress test results) can partially help performance engineers in this cost-estimation analysis in the following ways:

- Identifying the network(s) which need upgrade.
- Measuring the amount of upgrade needed in network resources.
- Identifying part(s) of the system architecture or implementation for refactoring.
- Suggestions for types of refactoring.
- Suggestions for negotiating revisions to RT constraints with clients by identifying the RT constraints which have violations and the severity of those violations (how much beyond each RT deadline the executions are currently).

For example, as discussed in Section 4.2, a RT failure would mean that the generated stress test requirement is entailing a traffic load on the network under test which it cannot transmit on time. According to TSSTM [7], the stress traffic amount (in KBs for example) is known (as reported by the test methodology output), and we know that the current bandwidth of the network under test is not enough for that traffic load. Therefore, the amount of bandwidth to be increased can be calculated and can thus be used to estimate costs of *UNR*.

To estimate costs of *REF*, we need to analyze the code using the sections of the code which have caused the RT failure (available in the stress test results), assess different code complexity metrics (e.g., cohesion, coupling) of code modules, use change-impact-analysis techniques [36], and finally estimate (e.g., in man hours) the effort needed to refactor the code or design in the proper way to fix the fault.

To estimate costs of *RPR*, the stress test results can again be used to assess how *bad* the RT failure is. For example, if the stress load has caused a hard RT constraint of 2 s to be observed as 5 s, this is too severe and the chances that the business logic of the system and/or the stakeholder(s) would allow relaxing this constraint deadline to be set to 5 s are very rare. This would mean a high value for *cost(RPR)* to not let *RPR* be selected. On the contrary, if the hard RT constraint of 2 s was observed as 2.1 s, it is possible (and perhaps a good idea) to try the *RPR* option. If the business logic and the stakeholder(s) allow this slight change (2 vs. 2.1 s), the cost of applying *REF* or *UNR* will be saved.

As another example, consider the following usage scenario of the heuristic. If we want to apply UNR to upgrade our network infrastructure, it would cost $50 K. On the other hand, due to enormous complexity of a SUT and also since we do not have enough design documentation, we predict that taking the *REF* path might impose a higher cost with respect to refactoring (refactoring time is money). Also, taking the *RPR* path is not free as performance engineers will most probably need to *negotiate* the performance requirements' *relaxation* with the primary stakeholders (e.g., clients) of the SUT.

4.4.2. Further decisions need to be made

In choosing which alternative to employ, further questions would also be raised. For example, if we choose to proceed with *REF*, should the system architecture be refactored (e.g., network clustering), or the source code (e.g., the way the distrusted communication message is programmed)? Furthermore, if there are several candidate source code locations for refactoring, which one should be conducted? Answering to such questions would need careful application of concepts such as change-impact-analysis [36], and performance patterns [20]. Obviously, the above decision-making process is by no means trivial.

Also, if we want to conduct *RPR* on a SRT constraint, should its deadline value be changed or its *RTmissProb* value? Answering to this question would need consultation with different stakeholders, especially the domain experts. Similarly, if we want to proceed with the *UNR* alternative, and there are several network locations to be upgraded, which location should be upgraded?

All the above decision-related questions lead to the formation of a decision *tree* in this context, a typical example of which is shown in Fig. 5. A more systematic heuristic would need to traverse each path (from the root to a leaf node) in this tree and estimate its cost given the information from the Step 5 of the STPE (stress test results), and also detailed source-code/network analysis.

We will discuss in Section 5 how we have analyzed some of the choices we faced in our experiment using the above heuristic and also the decision tree in Fig. 5.

4.4.3. Single vs. multiple fixes in each iteration

Adding to the complexity is the fact that, in general, more than one performance tuning activity can be applied in each STPE iteration with the hope of eliminating the chances for RT failures. However this can be complex and probably not very cost effective in some cases. Imagine applying several major fixes while the failure could have been fixed by only one of those changes.

To simplify the performance tuning stage and also the analysis of improvements, the current version of the STPE applies only one performance tuning activity at a time (in each iteration).

4.4.4. Iteration backtracking

The STPE process is flexible in the sense that, once an alternative tuning activity is applied and the resulting performance behavior is not satisfactory as expected, the performance engineer may *backtrack* in the changes, e.g., undo the last tuning activity, and perform another one. Of course, the time and cost associated with such *try and errors* should be carefully considered in the tuning process.

Also backtracking in this context sometimes makes sense, and sometimes it does not. For example, backtracking of source code refactor is viable, e.g., a developer can just undo the code changes. However, backtracking of a major network upgrade which has incurred several major purchases might not be possible (if the network hardware cannot be *returned* to the vendor).

4.4.5. Greedy vs. long-term-looking/holistic approach

The final word of caution in conducting the tuning stage is that the above heuristics in its current version is following a greedy ap-

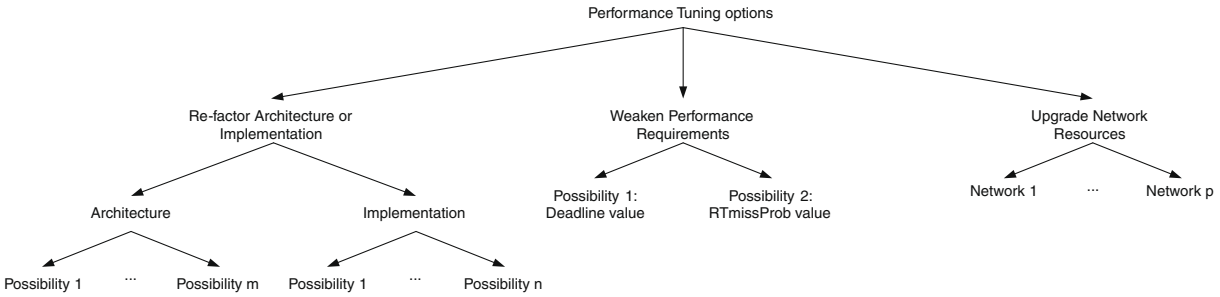


Fig. 5. A decision tree for Value-Based Performance Engineering (VBPE).

proach (as defined in optimizations literature). To better explain this notion, suppose that there are several RT violations in a given SUT. The rule-of-thumb in the SPE state-of the-art and -practice [20] has been to follow the divide-an-conquer approach, in which each RT violation (or a few which relate to the same system module) is (are) handled separately.

In such a case, prioritization of which RT violation to fix becomes the next natural question. To solve this prioritization question, engineers usually use the priority of HRT to SRT constraints, and then the severity of constraints.

Focusing on each RT violation, the current version of the STPE and the tuning heuristic are used to find the most cost-effective option and then to apply it. Since the system/problem parameters are used to solve one RT violation (or few similar ones) at a time, thus the solution strategy is greedy. Solutions from this strategy are usually not expected to be the most optimal ones overall when all the RT violations are considered in a holistic approach based on the notion of global optimization.

For example, consider a SUT with two HRT violations: *HRT1* and *HRT2*. Using our guidelines, the engineer decides to choose the HRT violation with the highest severity (*HRT1*) and apply the 1st iteration of the STPE. By conducting a comprehensive cost analysis, she decides to apply the refactoring solution for the violation of *HRT1*, since this is less-expensive than applying *UNR* or *RPR* for "this constraint", in isolation. After retesting, *HRT1* is not violated anymore. She then moves forward with *HRT2* and chooses UNR since it proves to be the least expensive option. By applying UNR, she successfully fixes the violation of *HRT2*. In a post-process analysis, she finds out that if she had applied UNR in the first place for *HRT1* (even if it was more expensive for *HRT1* than REF), the overall cost would have been less. We will see a real example of the above pitfall in our experiment (Section 5.3).

Thus, users of the STPE approach should be aware that since the current version is based on a greedy approach, it can lead to *short-sighted* (locally optimum) decisions. Global optimization techniques to develop long-term-looking/holistic approaches in STPE are needed in future works.

4.5. Synchronize model and code

Step 9. *Update UML Models and Source Code*: Based on the performance tuning activities, the final step is performed to synchronize the UML models and the source code with the changes performed in the other performance tuning stages to make sure that the next STPE cycle is performed with the most up-to-date versions of the SUT's design models and implementation. This is since the stress testing techniques (Section 4.1) used in the STPE are using the UML models of a SUT as input artifacts.

The software industry has lately observed effective tools in this area, e.g., IBM Rational Software Architect, Eclipse, Enterprise Architect, SDE for Visual Studio, and Altova UModel. For example, in our experiments (iteration 1 in Section 5.3), we used the IBM Rational Software Architect (RSA) which was able to successfully achieve 100% automated synchronization (i.e., there was no need for manual changes). If automated modeling tools are not used, for smaller-scale changes (e.g., changing the parameter of a function call), manual synchronization of UML models with the system source code should not be a major effort.

5. Experiment

We applied our STPE process to a case study SUT to demonstrate its feasibility. Our case study system is described in Section 5.1. Section 5.2 reports the network and hardware configurations of our case study. We report in Section 5.3 the results of applying STPE to our case study system.

5.1. System under analysis

Our case study SUT is a prototype *SCADA*-based power distribution system: SCADA for *Supervisory Control And Data Acquisition* [37]. The system is referred to as *SCAPS* (a *SCAda-based Power System*) [38], and is a system to control the power distribution grid across Canada consisting of several provinces. Each province has several cities and regions. There is one central server in each province which gathers the SCADA data from Tele-Control units (TCs) from all over the province and sends them to the national server.

The national server performs the following RT data-intensive safety-critical functions as part of the power application software: (1) Overload monitoring and control, (2) Detection of separated (disconnected) power grids, and (3) Power restoration after grid failure. We designed SCAPS to be used in Canada. To simplify the design and implementation of a prototype version of SCAPS, we considered only two Canadian provinces in the system: Ontario (ON) and Quebec (QC). The complete UML design model of SCAPS and more details on its business logic are presented in [38].

To familiarize the reader with SCAPS' functionality, a subset of its UML models is shown in Figs. 6–8. The SD in Fig. 6a corresponds to the *Overload Monitoring (OM)* control of the province of Ontario (ON). The *queryONData(dataType)* SD (Fig. 7) queries the load and grid connectivity data from TCs in the province of Ontario.

OC (Overload Control) SD (Fig. 8) checks if there is an overload situation in any of the two provinces, and if yes, a new power distribution load policy is generated by an object of type *ASA* (Automatic System Agent) and is sent to the respective provincial controller using *setNewLoadPolicy()*.

We do not show the SCAPS class diagram in this article due to its large size, but it is presented in [38]. Parameter *dataType* in Fig. 7, used in call messages *queryONData*, is an instance of class *LoadStatus*. An instance of the *LoadStatus* class stores information about the load levels of different parts of a power grid.

As we discussed in detail in [38], the RT constraints in these SDs are based on realistic estimates of the duration times for critical messages used in real SCADA-based power systems (e.g., [4,5]).

Based on the system business logic and a survey in the SCADA-based power systems (e.g., [4,5]), we specified six RT constraints in SCAPS using our extended UML stereotypes *SRTaction* and *HRTaction* (Section 3). One of the six RT constraints ($HRTC_4$, shown in Fig. 8) is the most critical constraint in SCAPS since it had the highest *RTcriticality* value. The RT information of $HRTC_4$ and $SRTC_1$ are explained next since they are discussed in our experiments (Section 5.3). The RT properties for other four RT constraints in SCAPS are discussed in [38].

$HRTC_4$ in SD *OC* (Fig. 8): It is critical that all the load data stored in the primary national server (*SEV_CA1*) should be backed up on the backup national server (*SEV_CA2*) in less than 1500 ms. Thus, a *RTcriticality* value of 0.95 was assigned to $HRTC_4$, denoting a critical HRT constraint. According to the SCADA-based power systems literature (e.g., [4,5]), backing up critical data in such systems is very important. This is done so that, in case of a failure in the primary national server, the backup server can continue to control the power system without service interruption.

$SRTC_1$ in SD *OC* (Fig. 8): As soon as an overload situation is observed in Ontario, the load balancing policy (i.e., preparing a load balancing regime and sending the regime to the provincial server) by the ASA should be executed in less than 1000 ms. A *RTduration* value of 0.9 (a high constraint missing threshold) was assigned to this constraint, meaning not a critical SRT constraint.

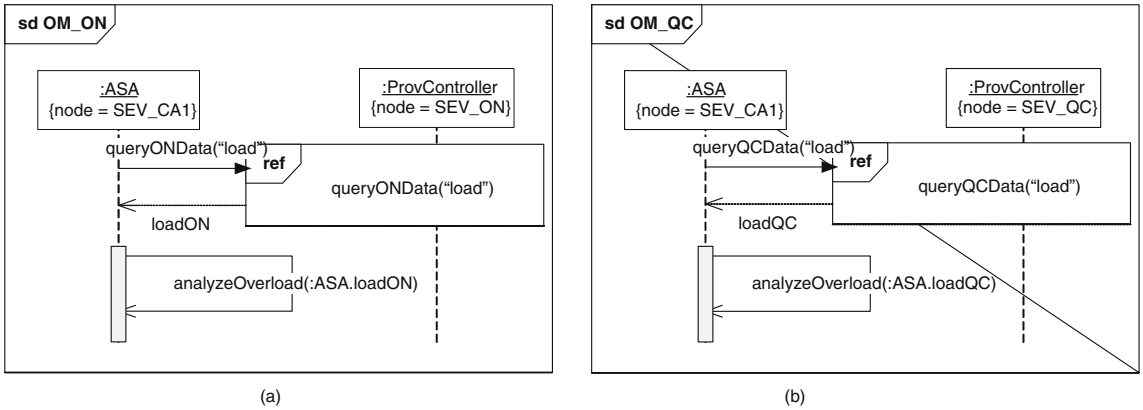


Fig. 6. SDs *OM_ON* and *OM_QC* (overload monitoring).

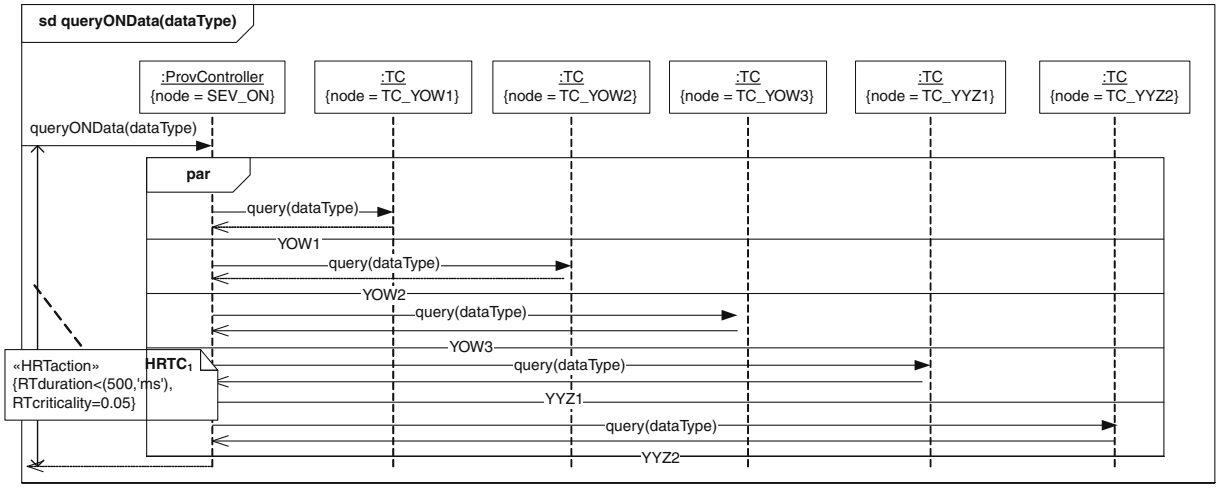


Fig. 7. SD *queryONData(dataType)*.

5.2. Network and hardware configurations

To manage the deployment complexity of this SUT, six PCs were used to play the roles of the 17 nodes in the SUT: *SEV_CA1* (one PC), *SEV_CA2* (one PC), *SEV_ON* (one PC), *SEV_QC* (one PC), *TC_YOWx* and *TC_YYZx* (one PC) and *TC_YMXx* and *TC_YQBx* (one PC). *TC_YOWx* denotes all the three TCs in the city of Ottawa: *TC_YOW1*, *TC_YOW2* and *TC_YOW3*. The decision of deploying all TCs from each province on one PC was made to simplify the system's deployment, controllability as well as its observability (less nodes to control and monitor at runtime).

The detailed network and hardware configurations of the machines used in our case study are available in [19]. The network cards of *SEV_CA1* and *SEV_CA2* were wireless (IEEE 802.11 g) with a speed of 19 Mbps. All others network cards were wired (Ethernet) and had a speed of 100 Mbps.

5.3. Applying STPE

The results of applying the STPE process to our case study SUT are discussed in this section. We continued applying STPE iterations until all chances of RT violations were eliminated. As discussed next, we found out that after three STPE iterations, we were able to eliminate all RT violations. For each of the three iterations, the three phases of STPE (performance measurement, evaluation and tuning) are presented next.

5.3.1. First Iteration

5.3.1.1. Performance measurement. Similar to other safety-critical control systems, SCAPS was designed such that its behavior is close to deterministic with respect to time (i.e., it is a *closed* systems). Since the system did not have a stochastic behavior, overlapping stress test requirements generated by TSSTM [7] was not hard to enforce repeatedly.

To measure and evaluate performance using STPE, we ran the test case corresponding to the generated stress test requirement for a large number of times (i.e., 500). Violations in one constraint ($HRTC_4$) were observed in the first iteration. To show an example of a SRT constraint and that it was not violated, we also discuss the durations of messages bounded by $SRTC_1$ in our experiment.

The execution time distributions of durations of $HRTC_4$ and $SRTC_1$ across all 500 runs for the three iterations of STPE are depicted in Fig. 9. The inter-quartile range boxes are shown. There are a few outliers in each data set. The *x*-axis is the iteration number and the *y*-axis is the measured time duration of $HRTC_4$ and $SRTC_1$.

5.3.1.2. Performance evaluation. Recall from Section 4.2 that, to evaluate performance, we need to compare stress test results (measured durations of RT constraints) with RT constraint deadlines specified in the UML models. According to Fig. 8, the RT deadline of $HRTC_4$ and $SRTC_1$ are 1500 and 800 ms, respectively (illustrated as horizontal bold lines in Fig. 9).

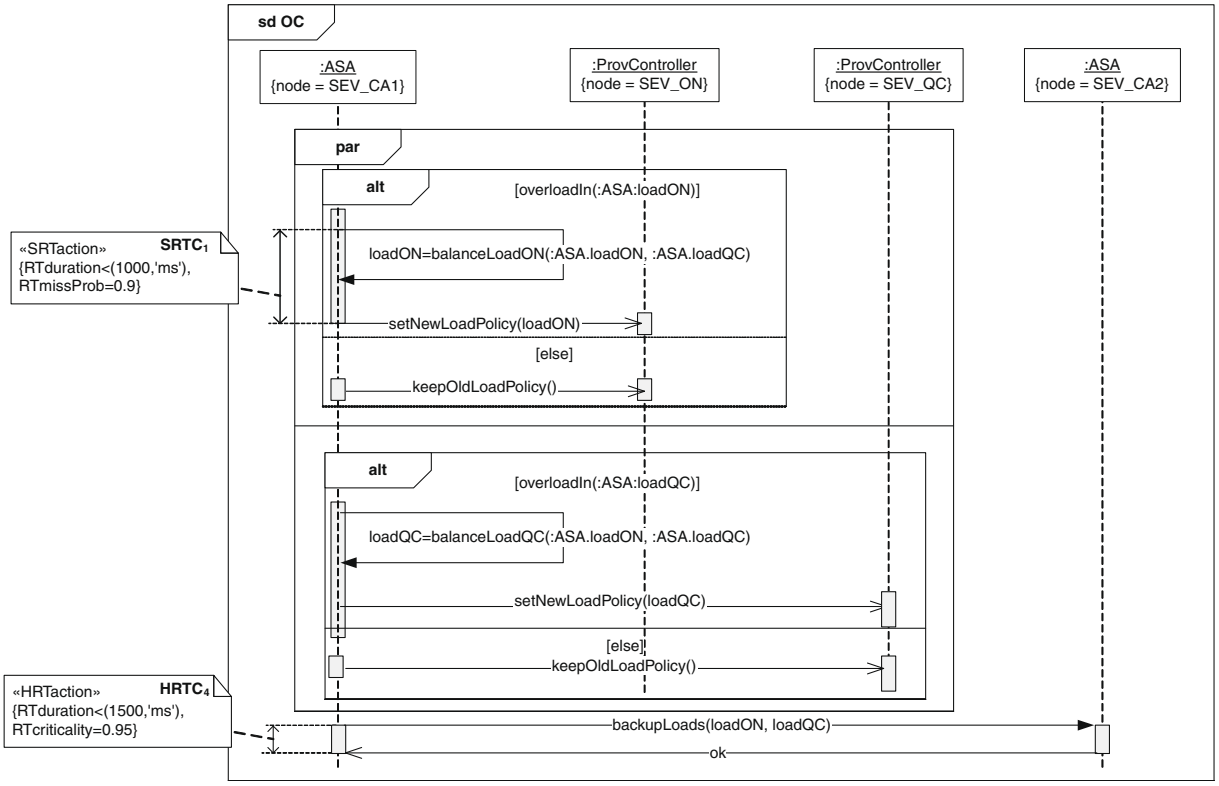


Fig. 8. SD *OC* (overload control).

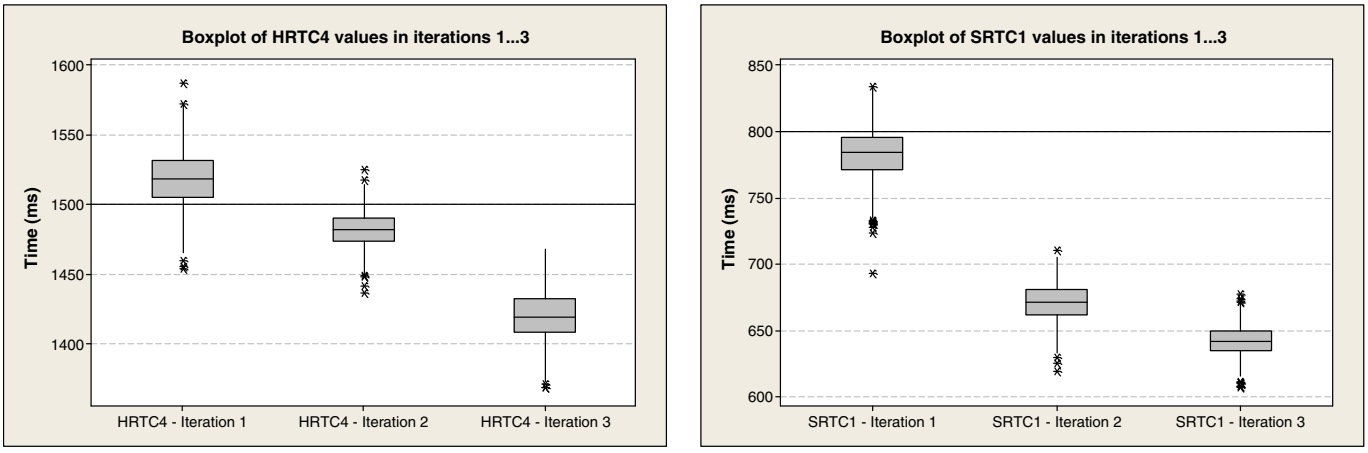


Fig. 9. Execution time distributions of messages bounded by $HRTC_4$ and $SRTC_1$ across three iterations of STPE.

Recall from Section 4.2 that if any of the measured HRT constraint durations are greater than its specified deadline, a HRT violation has occurred. For iteration 1, we can clearly see in Fig. 9 that most of the 500 measured $HRTC_4$ durations are greater than its deadline (1500 ms). More precisely, by analyzing the distribution data, we found out that 84.2% (421/500) of the measured durations violated $HRTC_4$. Thus, the outcome of the performance evaluation in iteration 1 is that a HRT violation has occurred. The STPE process thus suggests performing performance tuning.

For each SRT constraint, the ratio of the number of missed instances of the constraint over total number of runs is calculated. According to the data shown in Fig. 9, $SRTC_1$ is violated in 14.8% (74/500) of executions. Since this ratio is not greater than the maximum miss probability (*missProbability*) of the constraint (i.e., 90%), no SRT violation has occurred.

5.3.1.3. Performance tuning. Violations in one constraint ($HRTC_4$) were observed in the first iteration. Thus, we need to follow the performance tuning decision-support (VBPE) to choose the most cost-effective tuning alternative. We estimate the cost associated with each of the three possible tuning alternatives: REF, RPR, and UNR. *cost(REF)* does not seem to be very high, since the design of SCAPS is well documented and it is not very complex to be refactored. Also, the test results of the first iteration help us in finding the root cause of poor performance, i.e., the nation-wide network cannot transmit on time the large amount of traffic entailed by message *backupLoads* constrained by $HRTC_4$ in Fig. 8. On the other hand, *cost(RPR)* and *cost(UNR)* seem to be quite high. We assume that the deadline values are enforced by the business logic of the system (e.g., violating the set deadline can result in power blackouts) and thus are not negotiable, i.e., option *RPR* is not feasible

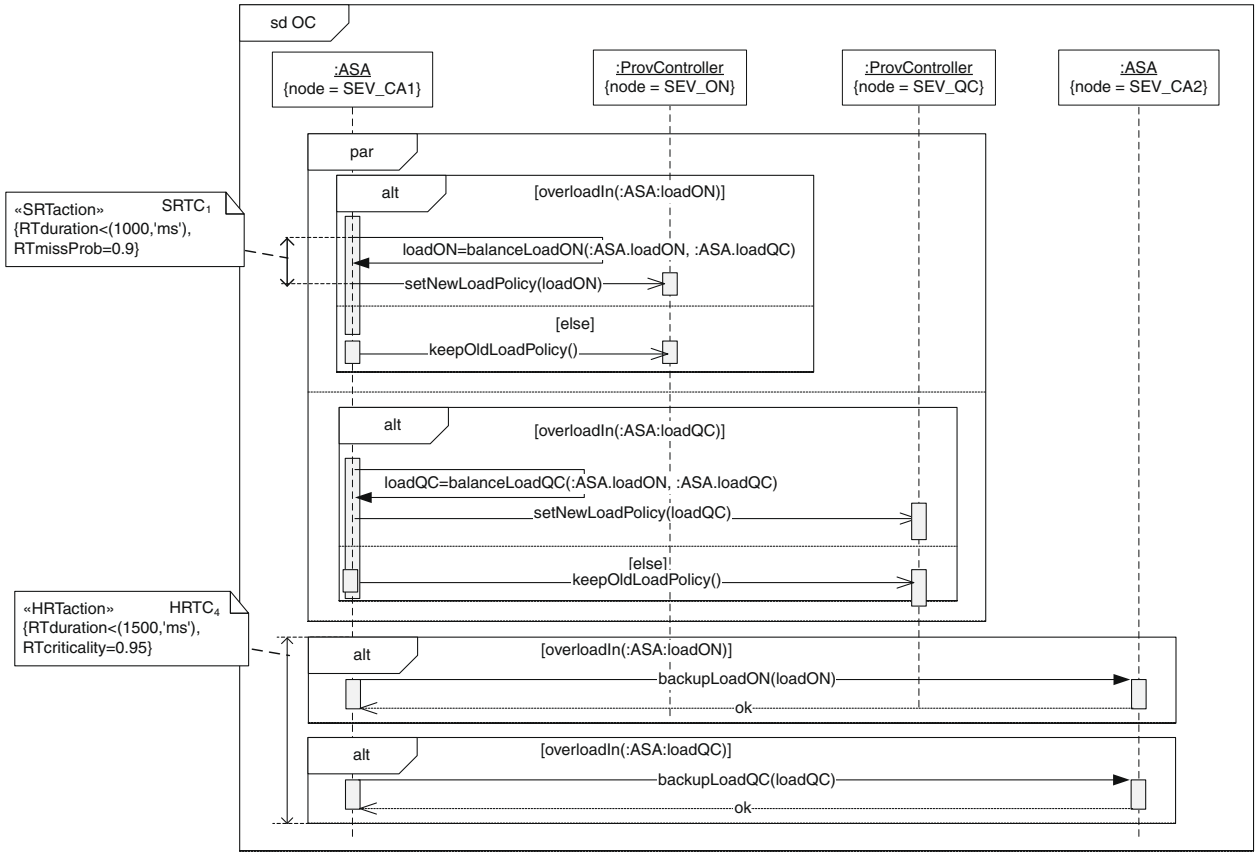


Fig. 10. Modified SD *OC* (overload control) after refactoring.

at all. *UNR* is also quite expensive as more modern network hardware has to be installed.

We thus decide to follow the *architecture/design refactoring* option. By using the guidelines provided by STPE (test results), we carefully inspect the design to find candidate design entities for refactoring. The message *backupLoads* constrained by $HRTC_4$ in Fig. 8 is the root cause of failure, so we analyzed to see if this message can be handled differently, and the answer was yes. Since this message has two parameters: *loadON* and *loadQC*, we can send them separately in two different messages instead of one, and can thus reduce the chances of stressing the network in on huge message traffic. Also, according to SD OC (Fig. 8), the message *backupLoads* was triggered even if none of the load parameters were updated in the SD (i.e., unnecessary duplicate backup of the old data). We thus refactored the SD by including UML alternative (if) constructs to only send the load data to the backup server only if needed. This would hopefully reduce the chances of stress traffic. The modified SD *OC* is shown in Fig. 10.

To synchronize model with the code after code refactoring, the IBM Rational Software Architect (RSA) was used which was able to successfully do a 100% synchronization (i.e., there was no need for manual changes).

5.3.2. Second Iteration (after refactoring design)

5.3.2.1. Performance measurement. Similar to the first STPE iteration, we again used our stress test methodology to generate the stress test requirement which maximizes the chances of revealing RT faults.

We ran the test case corresponding to the stress test requirement generated by the test methodology for 500 times. The execution time distribution of messages bound of $HRTC_4$ and $SRTC_1$ across all 500 runs for the second iteration is depicted in Fig. 9.

5.3.2.2. Performance evaluation. To evaluate performance in the second iteration, we again compared the measured durations of $HRTC_4$ with its RT deadline (1500 ms). We can clearly see in Fig. 9 that still some of the measured $HRTC_4$ durations are greater than its deadline (1500 ms). More precisely, by analyzing the distribution data, we found out that 12.4% of the measured durations were greater than $HRTC_4$'s deadline. Thus, the outcome of the performance evaluation in iteration 2 is that HRT violations still occur after the performance tuning step of iteration 1, but the frequency of RT violations has decreased. We found out that our refactoring activity in iteration 1 was helpful, but was not *enough* to eliminate the RT failure. The STPE process thus again suggests performance tuning.

5.3.2.3. Performance tuning. Violations in $HRTC_4$ are still observed. Thus, we need to follow the performance tuning decision-support (VBPE) to choose the most cost-effective tuning alternative. We estimate the cost associated with each of the three possible tuning alternatives (REF, RPR, and UNR). To consider *REF,* we looked for candidate source code/architecture entities for refactoring, but we were not able to find any entity for refactoring such that it relates to the root cause of the failure, i.e., messages *backupLoadON* and *backupLoadQC*.

The test results of the second iteration suggest that the network connecting nodes *SEV_CA1* and *SEV_CA2* cannot transmit on time the large amount of traffic entailed by the above two messages. In upgrading resources for this particular network in SCAPS, we can have several alternatives e.g., replacing the wireless network card (speed: 19 Mbps) of *SEV_CA1* or *SEV_CA2* with wired network cards (speed: 100 Mbps). *cost(UNR)* would equal to the cost of such upgrades.

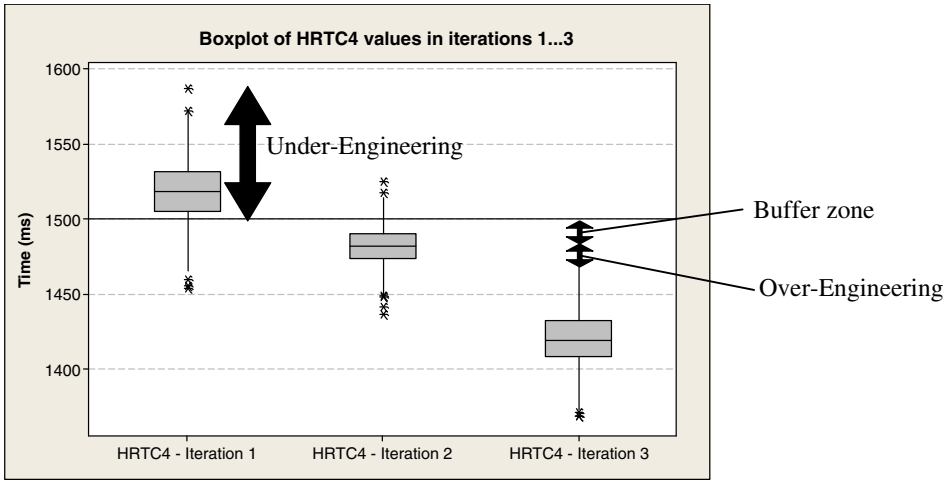


Fig. 11. Using stress test results to analyze instances of under- and over-engineering.

Similar to iteration 1, we assume that the deadline values are still hardly enforced by the business logic of the system and thus are not negotiable. Thus option *RPR* is not feasible at all.

Therefore, the VBPE heuristic suggests that *UNR* is the most cost-effective tuning alternative in this iteration. We chose to apply an upgrade on *SEV_CA1* as there was no preference to choose *SEV_CA1* over *SEV_CA2* or vice versa. We acknowledge that our criteria to select a network component to upgrade at this stage is not systematic, and further decision-support mechanisms on this selection problem is necessary in future works, which will possibly need root-cause analysis as to determine which network component needs upgrading first.

There was no update to UML models or the source code needed in this iteration as only the hardware setting was changed.

5.3.3. Third Iteration (after upgrading network resources)

5.3.3.1. Performance measurement. We again executed the test case generated by our test methodology based on the latest UML models of the SUT. The execution time distributions of durations of $HRTC_4$ and $SRTC_1$ across all 500 runs for the third iteration are depicted in Fig. 9.

5.3.3.2. Performance evaluation. Similar to iterations 1 and 2, $SRTC_1$ is not violated, although interestingly its duration has decreased. This can be explained since the system has a wider overall bandwidth to transmit data and thus the latency of message and data transmission is reduced. Unlike iterations 1 and 2, $HRTC_4$ is now not violated anymore. This means a success for STPE in that it helped us fix the RT failures in $HRTC_4$. The outcome of the performance evaluation in iteration 3 is that no violation of $HRTC_4$ has occurred, and thus, no more performance tuning is needed.

5.3.4. Outcome

Note that we performed similar performance measurement and evaluation steps on the other four RT constraints in SCAPS (i.e., $HRTC_1$, $HRTC_2$, $HRTC_3$, and $SRTC_2$) which due to space constraints are not reported in this article. Based on the performance tuning in iterations 1 and 2, the STPE process demonstrated that the other four RT constraints are also not violated in the sense that none of them were violated. We were thus able to make the SUT reliable after three STPE iterations.

5.4. Under-, over-, and right-engineering

It is clear that the approach presented so far in this paper is aimed at catching instances of *under*-engineering [39], that is, when the software or network resources are unable to deliver the required performance requirements. However, the general approach could also be applied to detect and correct instances of *over*-engineering as well.

Let us illustrate the approach using the stress test results for RT constraint $HRTC_4$ (discussed above). Fig. 11 visualizes the idea of under- and over-engineering analysis on this RT constraint. As discussed in Section 5.3, for meeting this RT constraint in the first iteration of STPE, the SUT was under-engineered, i.e., it did not have the capability to satisfy $HRTC_4$ in all test executions. To solve the problem, two iterations of STPE were conducted and, in the 3rd iteration, the system was able to meet the $HRTC_4$ constraint with a relatively small *buffer* (safety) zone. The worst-case execution time for the message bounded by this constraint in the 3rd iteration was 1470 ms which is only 30 ms below the 1500 ms constraint deadline. In building real-world safety-critical systems, engineers always allocate a buffer zone above the worst-case execution time to ensure that violations will never occur [1].

There is no systematic formula to calculate the amount of buffer zone in systems performance engineering in the literature. In real world, the decision stays at the discretion of the development team in consultations with domain experts on what ratio of the constraint deadline should be allocated for the buffer zone. Once a buffer zone interval is determined under the constraint deadline, to build the system in a cost effective manner, the system resources can be slightly revised so that the distance between the worst-case execution times for all messages are as close as possible to their buffer zone lines.

For the case of our example in Fig. 11, let us assume that the buffer zone size has been decided to be 15 ms. In this case, the worst-case execution time (1465 ms) for the message bounded by $HRTC_4$ is quite close to the buffer zone already and thus, qualitatively speaking, the extent of over-engineering is not major in this case.

In the presence of multiple RT constraints, the analysis of under- and over-engineering is unfortunately not as straightforward as it might seem. Referring to Fig. 9 and Section 5.3, recall that $SRTC_1$ was not violated even in iteration 1, with the lowest amount of system resources. Thus, adding more resources was only needed to satisfy $HRTC_4$, while those extra resources can be considered over-engineering with respect to $SRTC_1$, bringing its execution times more and more under the deadline value.

While the overall goal of performance engineering is to make sure all RT constraints are met, however, to conduct an effective VBPE, the issues of under- and over-engineering (necessity and sufficiency) should be looked into carefully by shifting system's resources from the modules or subsystems which have been over-engineered to the under-engineered ones.

For example, since $SRTC_1$ was satisfied in iteration 2 by a relatively large buffer zone, the corresponding over-engineering could be addressed by moving the high-speed network resources to where they were needed (i.e., the network infrastructure which affect the duration of $HRTC_4$). Thus, it seems that the systematic and system-wide VBPE would require another decision-support module to enable developers achieve *right*-engineering by moving the resources from over-engineered areas of the system to the under-engineered ones. This would need to analyze all the various parameters and factors in the given problem. Complex coupling and inter-connections among system modules, and their impact on the runtime durations of RT constraints can make this a challenging optimization problem, which we leave to future work.

This discussion also seems to relate to the recent cloud and utility computing paradigms [40]. In these paradigms, resources consumers *"consume resources as a service and pay only for resources that they use*" [40]. Resources are shifted from consumers with less demand to those with higher demands dynamically and in an agile fashion. However, most cloud and utility computing services so far have been employed for enterprise IT systems (such as conventional web services and application), and using such paradigms for safety-critical and DRTS's have not been discussed in the literature so far. Due to the nature of safety-critical systems, those systems have often their own dedicated resources which are not shared by other system [1].

Further, note that since the entire approach is towards right-engineering, gradual less-expensive fixes should be tried first, and if they are not enough to solve a given RT violation problem, we move to more expensive fixes. This was the case in iteration 2 as the fix (refactoring) in iteration 1 was not enough to solve the problem with $HRTC_4$ violation. Thus, the time spent on refactoring the code (in iteration 1) was not actually wasted. It provided to us the insight that refactoring by itself is not enough to solve the problem, and further fixation (bringing additional cost) is needed. These types of gradual steps, actual experimentations and analyzes are usually needed to justify the need for larger costs in real projects.

6. Evaluation of the methodology and open issues

6.1. Evaluation

Since the STPE methodology is a customization of SPE paradigm [14], quantitative evaluation of STPE would require comparing its efficiency, effectiveness, the results, and the guidelines it produces to other SPE-based methodologies. However, to our knowledge, there exists no SPE-based methodology which is specific to DRTS's and is using the information from stress testing.

Thus, we compare STPE to general measurement-based SPE approaches [21]. Recall from Section 2 that the SPE literature is classified into two broad categories: (1) model-based analysis (e.g., using Layered Queuing Networks) and (2) monitoring (measurement)-based approaches. As we discussed earlier, the current technique falls in the second category.

According to a recent major SPE road-map paper [21], the state of industrial performance measurement and testing techniques is captured in a series of 14 articles by Scott Barber [41], and previously in a report by the Software Engineering Institute [42]. By critically comparing STPE with this series of articles and techniques [41], we can summarize the advantages and novelty of STPE (compared to approaches currently employed in industry) as follows.

6.1.1. Performance measurement

Although stress test generation is not a contribution of the current paper, but three of our previous works [7–9], our stress testing approach has been recognized as one of the two most notable testing technique in the context of SPE by three pioneer SPE researchers, as reported in [21]. Most industrial SPE approaches rely on the so-called "baseline/benchmark tests" (see part 2 of [41]) which are less systematic, less predictable and less rigorous than our approach. Most of the papers on the topic suggest using load tests based on expected operational profile of the system in the field [13]. However, for DRTS's for example, performance some serious failures as shown by our experiments can occur only under worst-case test scenarios (Section 4.1), not addressed by other researchers/practitioners.

6.1.2. Performance evaluation

The series of articles in [41] focuses the application of SPE on enterprise applications (e.g., web-based systems). Thus, considerations of hard and soft RT constraints as performance requirements are not discussed in [41]. For performance evaluation, SPE practitioners rather use the less-formal notion of "user expectations", by asking questions such as: "*Are user expectations being met at various user loads?*", and "*Do all components perform as expected under load?*" [41]. As an industrial software performance engineer, Scott Barber [41] mentions that, as per his experience, most users consider response time under 3 s as "no delay or fast", 3–5 s as typical performance, 5–8 s as slow, 8–15 s as frustrating, and more than 15 s response time as unacceptable. It is obvious that the above informal performance evaluation approach cannot be used to build safety-critical DRTS's. The novelty of STPE is that, instead of rule-of-thumb "user expectations", it proposes a formal systematic approach to model performance requirements in the UML models of the SUT (Section 3.1) supporting both hard and soft RT constraints. Our technique can even be used as a more rigorous approach (than what is currently used) for enterprise applications (e.g., web-based systems) since soft RT constraints are being taken into account in more rigorous development of such systems.

6.1.3. Performance tuning

For tuning purposes, SPE practitioners aim at answering similar questions to those we raised in this paper, e.g., "*What components need to be or can be tuned?*", "*Is the network adequate?*" [41,42]. Barber reports that practitioners aim at assessing "server hardware adequacy", analysis of bottlenecks, and network "cluster management". However, the report [41] does not seem to provide a systematic decision-support approach for cost-effective performance tuning, in which the cost of each tuning alternative is compared to the overall benefit to be gained from performance enhancement. Our STEP approach at least proposes the idea of such a decision-support approach, raises the main challenges in this regards (Section 4.4) and provides partial solutions to such challenges.

From the perspective of right-engineering and efficiency of performance-related design/deployment decisions, we can compare STPE to a related requirements engineering approach [39], and related Agile development (lean software engineering) approaches [43,44].

The work in [39] is by three engineers from Praxis Critical Systems, a UK-based firm which develops safety- critical systems for railway, aerospace, defense, and nuclear domains. The authors use strong (what they call "rich") traceability features in requirements engineering of such systems. Their experience shows that one advantage of rich traceability is that the explicit justification makes it easier to detect over- or under-engineering. Any over-engineering

manifests itself, e.g., quantitative specifications that are more stringent than required, such as unnecessarily quick performance. This seems to align with our context, in which the STPE suggests to weaken system performance requirements (RT constraints) as seemed appropriate by the stakeholders. In other words, if a stringent RT constraint of 1 s is not necessary for a task, and 3 s would also be acceptable, the design can be *relaxed* accordingly. The authors [39] further mention that any under-engineering means that it is not possible to construct a valid justification (in the implementation level) for at least one statement of the requirements. The paper reports successful results from applying the approach to the development of a safety-critical protection system [39]. This is also inline with our approach in that if the worst-case execution time of a RT task is below its deadline value, under-engineering has occurred.

Right-engineering of software performance requirements has also been the focus of many engineers in the Agile community lately [43,44]. The popular lean software development approach [44] advocates tuning the software performance to the most realistic level by avoiding spending more effort and resource on the part of the system which is unnecessarily.

In a recent talk by Stockdale in an Agile conference, the speakers talked about an Agile approach toward performance tuning (the talk video is available online at [43]).The speaker compares and differentiates between the performance tuning needs of various applications domains (safety-critical, banking, and enterprise). He called performance tuning to be the art and science of "optimizing use of finite resources". He presents various Agile-based performance patterns in various categories: code-based patterns (e.g., using code profilers), architecture-based patterns (e.g., using software-level caches), design-based (e.g., putting expensive requests in queue). At the end, the speaker concludes that although "hardware doubles every 18 months", the software that we develop need to be performance tuned to best utilize the hardware resources. Interestingly, one of the three guidelines we have looked into (refactoring architecture or design) closely matches with the above Agile approach toward performance tuning [43].

6.2. Open issues

Overall, the current STPE approach and the experience report provided some partial solutions to the sophistical challenge of applying SPE to DRTS's. The paper also brought forward the following list of open questions to the research community in the context of decision-support mechanisms in performance tuning of DRTS's, which would not have been possible without actual experimentations:

1. The current version of the STPE process is currently providing a rank-based decision-making heuristic to choose the best performance tuning guideline to apply. The measurements are conducted on a mixed qualitative/quantitative manner. More development of the heuristic based on the software-engineering decision-support body of knowledge [33] is necessary.
2. To simplify the performance tuning stage and also the analysis of improvements, the current version of the STPE applies only one performance tuning activity at a time (in each iteration). Extension of the approach to conduct multiple performance tuning fixes in each iteration are needed. Recall that existence of at least one RT violation in a SUT necessitates the need for an STPE iteration.
3. Recall from Section 4.4 that the rule-of-thumb in the SPE state-of the-art and -practice [20] has been to follow the divide-an-conquer approach, in which each RT violation (or a few which are under the same system module) is (are) handled separately. Focusing on each RT violation, the current version of the STPE and the tuning heuristic are used to find the most cost-effective option and then to apply it. Since the system/problem parameters are used to solve one RT violation (or few similar ones) at a time, thus the solution strategy is greedy. Solutions from this strategy are usually not expected to be the most optimal ones overall when all the RT violations are considered in a holistic approach using global optimization techniques. Thus, global optimization techniques to develop long-term-looking/holistic approaches in STPE are needed in future works. We saw a real example of the above pitfall in our experiment.

Acknowledgements

This work was supported by the Discovery Grant No. 341511-07 from the Natural Sciences and Engineering Research Council of Canada (NSERC) and also by the Alberta Ingenuity New Faculty Award No. 200600673. The author would like to thank James Miller for his helpful comments and suggestions on the early drafts of this article.

Appendix A. List of abbreviations

ASA	Automatic System Agent
CFP	Control Flow Path
DRTS	Distributed and REAL-RIME SYSTEM
GASTM	Genetic Algorithm-Based Stress Test Methodology
HRT	Hard Real-Time
OM	Overload Monitoring
REF	Tune/Re-Factor Architecture or Design
RPR	Revise (Weaken) System Performance Requirements
RT	Real-Time
SCADA	Supervisory Control and Data Acquisition System
SCAPS	A SCAda-Based Power System
SD	Sequence Diagram
SPE	Software Performance Engineering
SRT	Soft Real-Time
STPE	Stress-test Performance Engineering
SUT	System Under Test
TC	Tele-Control Unit
TSSTM	Time-Shifting Stress Test Methodology
UML-SPT	UML Profile for Schedulability, Performance, and Time
UNR	Upgrade Network Resources
VBPE	Value-Based Performance Engineering
WNSTM	Wait-Notify Stress Test Methodology

References

[1] J.J.P. Tsai, Y. Bi, S.J.H. Yang, R.A.W. Smith, Distributed Real-Time Systems: Monitoring, Visualization, Debugging, and Analysis, John Wiley and Sons, 1996.
[2] E. Weyuker, F.I. Vokolos, Experience with performance testing of software systems: issues, an approach and case study, IEEE Transactions on Software Engineering 26 (12) (2000) 1147–1156.
[3] R. Kuhn, Sources of failure in the public switched telephone network, IEEE Computer 30 (4) (1997) 31–36.
[4] E.-K. Chan, H. Ebenhoh, The implementation and evolution of a SCADA system for a large distribution network, IEEE Transactions on Power Systems 7 (1) (1992) 320–326.
[5] Y. Ebata, H. Hayashi, Y. Hasegawa, S. Komatsu, K. Suzuki, Development of the intranet-based SCADA for power system, in: Proceeding of IEEE Power Engineering Society Winter Meeting, 2000, pp. 1656–1661.
[6] M. Hirsh and D. Klaidman, Blackout 2003: What Went Wrong, Newsweek, 25 August, 2003.

[7] V. Garousi, L. Briand, Y. Labiche, Traffic-aware stress testing of distributed systems based on UML models, in: Proceedings of International Conference on Software Engineering, 2006. pp. 391–400.
[8] V. Garousi, Traffic-aware stress testing of distributed real-time systems based on UML models in the presence of time uncertainty, in: Proceedings of IEEE International Conference on Software Testing, Verification and Validation, 2008, pp. 92–101.
[9] V. Garousi, L. Briand, Y. Labiche, Traffic-aware stress testing of distributed real-time systems based on UML models using genetic algorithms, Elsevier Journal of Systems and Software 81 (2) (2008) 161–185 (Special Issue on Model-Based Software Testing).
[10] Object Management Group (OMG), UML 2.1.1 Superstructure specification, 2007.
[11] V. Garousi, Empirical analysis of a genetic algorithm-based stress test technique for distributed real-time systems, in: Proceedings of the Genetic and Evolutionary Computation Conference, Search-Based Software Engineering (SBSE) track, 2008, pp. 1743–1750.
[12] V. Garousi, A genetic algorithm-based stress test requirements generator tool and its empirical evaluation, IEEE Transactions on Software Engineering, Special Issue on Search-Based Optimization, in press, doi:10.1109/TSE.2010.5.
[13] M.S. Gittens, The Extended Operational Profile Model for Usage-Based Software Testing, Doctoral thesis, University of Western Ontario, 2004.
[14] C.U. Smith, Performance Engineering of Software Systems, Addison-Wesley, 1990.
[15] C.U. Smith, L.G. Williams, Software performance engineering, in: Encyclopedia of Software Engineering, second ed., John Wiley and Sons, 2002.
[16] A. Mos, J. Murphy, Performance management in component-oriented systems using a model driven architecture: the SPL trade approach, in: Proceedings of International Enterprise Distributed Object Computing Conference, 2002, pp. 227–237.
[17] G. Franks, A. Hubbard, S. Majumdar, J.E. Neilson, D.C. Petriu, J.A. Rolia, C.M. Woodside, A toolset for performance engineering and software design of client-server systems, Journal of Performance Evaluation vol. 24 (1995) 117–136.
[18] G. Franks, S. Majumdar, J. Neilsony, D. Petriu, J. Rolia, M. Woodside, Performance analysis of distributed server systems, in: Proceedings of International Conference on Software Quality, 1996, pp. 15–26.
[19] V. Garousi, Iterative stress-test performance engineering of distributed real-time systems, Technical report, University of Calgary, SERG-2007-03, 2007, <http://www.enel.ucalgary.ca/~vgarousi/downloads/papers/tr/SERG-2007-03.pdf>.
[20] C.U. Smith, L.G. Williams, Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software, Addison-Wesley Professional, 2001.
[21] M. Woodside, G. Franks, D.C. Petriu, The future of software performance engineering, in: International Conference on Software Engineering, Future of Software Engineering, 2007, pp. 171–187.
[22] J.A. Rolia, K.C. Sevcik, The method of layers, IEEE Transactions on Software Engineering 21 (8) (1995) 689–700.
[23] C.U. Smith, L.G. Williams, Will your new distributed system require performance adjustments?, 2007, <http://www.perfeng.com/distrmgt.htm>. (accessed: May).
[24] B. Douglass, Doing Hard Time, Developing Real-Time Systems with UML Objects, Frameworks, and Patterns, Addison-Wesley, 1999.
[25] Object Management Group (OMG), UML Profile for Schedulability, Performance, and Time (v1. 1), 2005.
[26] Object Management Group (OMG), UML profile for modeling and analysis of real-time and embedded systems (MARTE), version 2.0 beta, 2008, <http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf>. (accessed January 2010).
[27] V. Garousi, Measuring the cost effectiveness of stress test orders for distributed real-time systems based on fault criticalities, Technical report, University of Calgary, SERG-2007-01, 2007, <http://www.enel.ucalgary.ca/~vgarousi/downloads/papers/tr/SERG-2007-01.pdf>.
[28] H. Gomaa, Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison-Wesley, 2000.
[29] B. Korel, Automated software test data generation, IEEE Transactions on Software Engineering 16 (8) (1990) 870–879.
[30] M.H. Willebeek-LeMair, A.P. Reeves, Strategies for dynamic load balancing on highly parallel computers, EEE Transactions on Parallel and Distributed Systems 4 (6) (1993) 979–993.
[31] P. Maheshwari, T.N. Kien, A. Erradi, QoS-based message-oriented middleware for web services, in: Proceedings of Workshop on Web Information Systems, 2004, pp. 241–251.
[32] M. Motsko, T. Oberndorf, E.-J. Pairo, J. Smith, Rules of Thumb for the Use of COTS Products, Technical report, Carnegie Mellon University, CMU/SEI-2002-TR-032, 2002.
[33] G. Ruhe, Software engineering decision support: methodology and applications, in: Tonfoni, Jain (Ed.), Innovations in Decision Support Systems, International Series on Advanced Intelligence, vol. 3, 2003, pp. 143–174.
[34] B.S.A. Aurum, B.W. Boehm, H. Erdogmus, P. Grünbacher, Value-Based Software Engineering, Springer Verlag, 2005.
[35] R. Ramler, P. Grünbacher, S. Biffl, Value-based management of software testing, in: Value-Based Software Engineering, Springer Verlag, 2005, pp. 225–244.
[36] R.S. Arnold, Software Change Impact Analysis, IEEE Computer Society Press, 1996.
[37] A. Daneels, W. Salter, What is SCADA?, in: Proceeding of International Conference on Accelerator and Large Experimental Physics Control Systems, 1999, pp. 39–343.
[38] V. Garousi, L. Briand, Y. Labiche, Traffic-Aware Stress Testing of Distributed Systems Based on UML Models, Technical report SCE-05-13, Carleton University, 2005, <http://www.sce.carleton.ca/squall/pubs/tech_report/TR_SCE-05-13.pdf>.
[39] J. Hammond, R. Rawlings, A. Hall, Will it work?, in: Proceedings of IEEE International Symposium on Requirements Engineering, 2001, pp. 102–109.
[40] B. Hayes, Cloud computing, Communications of the ACM 51 (7) (2008) 9–11.
[41] S. Barber, Beyond Performance Testing, parts 1–14, IBM developer works, rational technical library, 2004, <www-128.ibm.com/developerworks/rational/library/4169.html>. (accessed: Jan. 2010).
[42] M.H. Klein, State of the Practice Report: Problems in the Practice of Performance Engineering, Technical report, Software Engineering Institute, CMU/SEI-95-TR-020, 1996.
[43] M. Stockdale, Performance Tuning: an Agile Approach, in: Agile Vancouver Conference Much Ado About Agile IV, talk video, 2009, <http://www.vimeo.com/8091960>.
[44] M. Poppendieck, T. Poppendieck, Lean Software Development: An Agile Toolkit, Addison-Wesley Professional, 2003.