2016 IEEE International Conference on Software Testing, Verification and Validation

# Canopus: A Domain-Specific Language for Modeling Performance Testing

Maicon Bernardino*†, Avelino F. Zorzo*, Elder M. Rodrigues*‡
*Postgraduate Program in Computer Science – Faculty of Informatics (FACIN)
Pontifical Catholic University of Rio Grande do Sul (PUCRS) – Porto Alegre, RS, Brazil
†FEEVALE University – Novo Hamburgo, RS, Brazil
‡Federal University of Health Sciences of Porto Alegre (UFCSPA) – Porto Alegre, RS, Brazil
bernardino@acm.org, avelino.zorzo@pucrs.br, elderr@ufcspa.edu.br

***Abstract*—Despite all the efforts to reduce the cost of the testing phase in software development, it is still one of the most expensive phases. In order to continue to minimize those costs, in this paper, we propose a Domain-Specific Language (DSL), built on top of MetaEdit+ language workbench, to model performance testing for web applications. Our DSL, called Canopus, was developed in the context of a collaboration[1] between our university and a Technology Development Laboratory (TDL) from an Information Technology (IT) company. We present, in this paper, the Canopus metamodels, its domain analysis, a process that integrates Canopus to Model-Based Performance Testing, and applied it to an industrial case study.**

***Index Terms*—performance testing; domain-specific language; domain-specific modeling; model-based testing.**

## I. Introduction and Motivation

It is well-known that the testing phase is one of the most time-consuming and laborious phases of a software development process [1]. Depending on the desired level of quality for the target application, and also its complexity, the testing phase can have a high cost. Normally defining, designing, writing and executing tests require a large amount of resources, *e.g.* skilled human resources and supporting tools. In order to mitigate these issues, it would be relevant to define and design the test activity using a well-defined model or language and to allow the representation of the domain at a high level of abstraction. Furthermore, it would be relevant to adopt some technique or strategy to automate the writing and execution of the tests from this test model or language. One of the most promising techniques to automate the testing process from the system models is Model-Based Testing (MBT) [2].

MBT provides support to automate several activities of a testing process, *e.g.* test cases and scripts generation. In addition, the adoption of an MBT approach provides other benefits, such as a better understanding on the application, its behavior and test environment, since it provides a graphical representation about the System Under Test (SUT). Although MBT is a well-defined and applied technique to automate some testing levels, it is not fully explored to test non-functional requirements of an application, *e.g.* performance testing. There are some works proposing models or languages to support the design of performance models. For instance, the SPT UML profile [3] relies on the use of textual annotations on models, *e.g.* stereotypes and tags to support the modeling of performance aspects of an application. Another example is the Gatling [4] Domain-Specific Language (DSL), which provides an environment to write textual representation of an internal DSL based on industrial needs and tied to a testing tool.

Although these models and languages are useful to support the design of performance models and also to support testing automation, there are a few limitations that restrict their integration in a testing process using an MBT approach. Despite the benefits of using an UML profile to model specific needs of the performance testing domain, its use can lead to some limitations. For instance, most of the available UML design tools do not provide support to work with a well-defined set of UML elements, which is needed to work with a restricted and specialized language. Thus, the presence of several unnecessary modeling elements may result in an error-prone and complex activity.

Most of these issues could be mitigated by the definition and implementation of a graphical and textual DSL to the performance testing domain. However, to the best of our knowledge, there is little investigation on applying DSL to the performance testing domain. For instance, the Gatling DSL provides only a textual representation, based on the Scala language [4], which is tied to a specific load generator technology - it is a script-oriented DSL. The absence of a graphical representation could be an obstacle to its adoption by those performance analysts that already use some type of graphical notation to represent the testing infrastructure or the SUT. Furthermore, as already stated, the use of a graphical notation provides a better understanding about the testing activities and SUT to the testing team as well as to developers, business analysts and non-technical stakeholders. Another limitation is that the Gatling DSL is bound to a specific workload solution.

Therefore, it would be relevant to develop a graphical modeling language for the performance testing domain to mitigate some of the limitations mentioned earlier. In this paper, we propose Canopus, a DSL that aims to provide a graphical and textual way to support the design of performance models, and that can be applied in a model-based performance

[1]Study developed by the Research Group of the PDTI 001/2016, financed by Dell Computers with resources of Law 8.248/91.

978-1-5090-1827-7/16 $31.00 © 2016 IEEE
DOI 10.1109/ICST.2016.13

testing approach. Hence, our DSL scope is to support the performance testing modeling activity, aggregating information about the problem domain to provide better knowledge sharing among testing teams and stakeholders, and centralizing the performance testing documentation. Moreover, our DSL will be used within an MBT context to generate performance test scripts and scenarios for third-party tools/load generators. It is important to mention that during the design and development of our DSL, we also considered some requirements from a Technology Development Lab (hereafter referred to as TDL) of an industrial partner, in the context of a collaboration project to investigate performance testing automation. These requirements were: a) the DSL had to allow for representing the performance testing features; b) the technique for developing our DSL had to be based on Language Workbenches; c) the DSL had to support a graphical representation of the performance testing features; d) the DSL had to support a textual representation; e) the DSL had to include features that illustrate performance counters (metrics, *e.g.* CPU, memory); f) the DSL had to allow the modeling of the behavior of different user profiles; g) traceability links between graphical and textual representations should require minimal human intervention; h) the DSL had to be able to export models to specific technologies, *e.g.* HP LoadRunner, MS Visual Studio; i) the DSL had to generate model information in an eXtensible Markup Language (XML) file; j) the DSL had to represent different performance test elements in test scripts; and, k) the DSL had to allow the modeling of multiple performance test scenarios. The rational and design decisions for our DSL are described in [5].

In this paper, we discuss the performance testing domain and describe how the metamodels were structured to compose our DSL. Besides, we also present how Canopus was designed to be applied with an MBT approach to automatically generate performance test scenarios and scripts. To demonstrate how our DSL can be used in practice, we applied it throughout an actual case study from the industry.

This paper is organized as follows. Section II introduces related background on performance testing and DSL, and discusses related work. Section III presents an analysis about the performance domain and describes the Canopus metamodels. Section IV describes a model-based performance testing process that integrates Canopus. Section V presents how we applied Canopus in an industrial case study. Section VI concludes the paper with some future directions and points out lessons learned.

## II. Background

This section introduces Model-Based Performance Testing and Domain-Specific Languages, and discusses the related work.

### A. Model-Based Performance Testing

Performance engineering is an essential activity to improve scalability and performance, revealing bottlenecks of the SUT [6], and can be applied in accordance with two distinct approaches: a predictive-based approach and a measurement-based approach. The predictive-based approach describes how the system operations use the computational resources, and how the limitation and concurrence of these resources affect such operations, usually, the design is supported by formal model, *e.g.* Finite State Machine (FSM). As for the measurement-based approach supports the performance testing activity, which can be classified as load testing, stress testing and soak (or stability) testing [7]. Each one of these differ from each other based on their workload and the time that is available to perform the test. They can also be applied to different application domains such as desktop, mobile, web service, and web application.

Based on that, a variety of testing tools have been developed to support and automate performance testing activities. The majority of these tools take advantage of one of two techniques: Capture and Replay (CR) or Model-Based Testing (MBT). In a CR-based approach, a performance engineer must run the tests manually one time on the web application to be tested, using a tool on a "capture" mode, and then run the load generator to perform "replay" mode. In an MBT [2] approach, a performance engineer designs the test models, annotates them with performance information and then uses a tool to automatically generate a set of test scripts and scenarios.

There are some notations, languages and models that can be applied to design performance testing, *e.g.* Unified Modeling Language (UML), User Community Modeling Language (UCML) [8] and Customer Behavior Modeling Graph (CBMG) [9]. Some of these notations are only applicable during the designing of the performance testing, to support its documentation. For instance, UCML is a graphical notation to model performance testing, but it does not provide any metamodels able to instantiate models to support testing automation. Nevertheless, other notations can be applied later to automate the generation and execution of performance test scripts and scenarios during the design and execution phases. Another way to design testing for a specific domain is using a DSL.

### B. Domain-Specific Languages

Domain-Specific Languages (DSL), also called application-oriented, special purpose or specialized languages, are languages that provide concepts and notations tailored for a particular domain [10]. Nevertheless, to develop a DSL, a Domain-Specific Modeling (DSM) is required to lead to a solid body of knowledge about the domain. DSM is a methodology to design and develop systems based on Model-Driven Development (MDD). One important activity from a DSM is the domain analysis phase, which deals with domain's rules, features, concepts and properties that must be identified and defined.

A strategy to support the creation and maintenance of a DSL is to adopt tools called Language Workbenches (LW) [11]. LW is an environment to develop metamodels. There are some available LW, such as Microsoft Visual Studio Visualization and Modeling SDK [12], Generic Modeling Environment

(GME) [13], Eclipse Modeling Framework (EMF) [14], and MetaEdit+ Workbench [15]. Besides implementing the analysis and code generation process, an LW also provides a better editing experience for DSL developers, being able to create DSL editors with power similar to modern IDEs [10]. Thereby, DSL are classified with regard to their creation techniques/design as: internal, external, or based on LW [16].

The use of DSL presents some opportunities, such as [17]: a) better expressiveness in domain rules, allowing the user to express the solution at a high level of abstraction. Consequently, domain experts can understand, validate, modify and/or develop their solutions; b) improving the communication and collaboration among testing and development teams, as well as non-technical stakeholders; c) supporting artifact and knowledge reuse. Inasmuch as DSL can retain the domain knowledge, the adoption of DSL allows the reuse of the preserved domain knowledge by a mass of users, including those inexperienced in a particular problem domain; d) enabling better Return On Investment (ROI) in the medium- or long-term than traditional modeling, despite the high investment to design and deploy a DSL.

### C. Related Work

There is a lack of novel models, languages and supporting tools to improve the performance testing process, mainly in the automation of scenarios and script generation. To the best of our knowledge, there is little work describing performance testing tools [18] [19]. For instance, [18] proposes the solution called WALTy (Web Application Load-based Testing), which is an integrated toolset with the objective of analyzing the performance of web applications through a scalable *what-if* analysis. This approach is based on a workload characterization being derived with information extracted from log files. The workload is modeled by using Customer Behavior Model Graphs (CBMG) [9]. This approach focuses in the final phase of the development process, since the system must be developed to collect the log application and then generate the CBMG. Conversely, our approach aims to support the performance requirements identification, and it is performed in the initial phase of the development process to facilitate the communication among project stakeholders.

Krishnamurthy [19] presents an approach that uses application models that capture dependencies among the requests for the web application domain. This approach uses Extended Finite State Machines (EFSM), in which additional elements are introduced to address the dependencies among requests and input test data. The approach is composed of a set of tools for model-based performance testing. One tool that integrates this set of tools is SWAT (Session-Based Web Application Tester), which in turn uses the `httperf` tool to submit the synthetic workload over the SUT. However, this approach presents some disadvantages, such as being restricted to generating workload for a specific workload tool and the absence of a model to graphically design the performance testing.

Although these works introduce relevant contribution to the domain, none of them proposed a specific language or modeling notation to design performance testing. There are a few studies investigating the development of DSL to support performance testing [4] [20] [21]. Bui *et al.* [20] propose DSLBench for benchmark generation, Spafford [21] presents the Aspen DSL for performance modeling in a predictive-based approach [6]. Meanwhile, Gatling DSL [4] is an internal DSL focused in supporting a measurement-based approach. Differently from them, our work provides a graphical and textual DSL to design performance testing in a measurement-based approach.

## III. Canopus: a Domain-Specific Language for Modeling Performance Testing

This section discusses the analysis of the performance testing domain and the metamodels that compose our DSL. As mentioned in Section I, the requirements and design decisions for Canopus are described in [5].

### A. Domain Analysis

The proposed DSL aims to allow a performance engineer to model the behavior of web applications[2] and their environment. Through the developed models it is possible to generate test scenarios and scripts that will be used to execute performance testing. It is important to mention that before we started to develop our DSL, some steps were taken in collaboration with researchers from our group and test engineers from the TDL to clearly define our problem domain.

The first step was related to the expertise that was acquired during the development of a Software Product Line (SPL) to derive MBT tools called PLeTs [22]. These SPL's artifacts can be split into two main parts: one that analyses models and generates abstract test cases from those models, and one that takes the abstract test cases and derive concrete test scripts to be executed by performance testing tools. Several models were studied during the first phase of the development of our SPL, *e.g.* UCML, UML profiles, CBMG and FSM. Likewise, several performance testing environments and tools were studied, such as HP LoadRunner, MS Visual Studio and Neo Load.

Our second step was to apply some of the models and tools identified in the previous step to test some open-source applications, such as TPC-W (Transaction Processing Performance-Web) and Moodle Learning Management System. Furthermore, we also applied some of the MBT tools generated from our SPL to test those applications and to support the test of real applications in the context of our collaboration with the TDL (see [23]). Those real applications were hosted in highly complex environments, which provided us with a deep knowledge on the needs of performance testing teams.

During our last step, we conducted a web-based survey that has been answered by the performance test experts. An example question[3] is as follows: "Do you concur that the following monitoring elements that compose our monitoring metamodel for performance testing are representative?". The

[2]Although we focus on web application, our DSL is not intended to be limited to this domain.

[3]The complete survey details can be found at http://tiny.cc/ICST-2016.

participant had to answer using a five points Likert scale [24], based on their perception of representativeness of the DSL elements to symbolize performance testing concepts.

The steps above provided us with an initial performance testing body of knowledge that was used as a base to define the performance testing requirements and design decisions for our DSL [5]. Thus, in order to determine the concepts, entities and functionalities that represent the performance testing domain, we adopted a strategy to identify and analyze the domain using an ontology [25]. Besides this ontology, we used the body of knowledge to provide the basis for determining the objects, relationships, and constraints to express the performance testing domain. This set of elements composes the metamodels of Canopus DSL, which will be described in the next section.

To define the Canopus DSL, we need a model to create our performance testing models, *i.e.* a metamodel, which can be used to create abstract concepts for a certain domain [26]. A metamodel is composed of metatypes, which are used to design a specific DSL. The development process of generating such metamodels is called metamodeling, which is a framework defined by meta-metamodels to generate metamodels. There are several metamodeling environments, *a.k.a.* Language Workbenches (LW) (see Section II-B). To support the creation of our DSL, we chose MetaEdit+, one of the first successful commercial tools. MetaEdit+ supports the creation and evolution of each of the Graph, Object, Port, Property, Relationship and Role (GOPPRR) [26] [27] metatypes.

A Graph metatype is a collection of objects, relationships and roles. These are bound together to show which objects a relationship element connects through which roles. A graph is also able to maintain information about which graphs its elements decompose into. A graph is one particular model, usually shown as a diagram. The Object metatype is the main element that can be placed in graphs. Examples of objects are the concepts of a domain that must be represented in a graph. It is worthwhile to highlight that all instances of a created object can be reused in other graphs. The Relationship metatype is an explicit connection among two or more objects. Relationships can be attached to an object via roles. The Role metatype specifies how an object participates in a relationship. A Port metatype is an optional specification of a specific part of an object to which a role can connect to. Thus, ports allow additional semantics or constraints on how objects can be connected. The Property metatype can be included in all other GOPPRR metatypes to specify what information can be attached to them. In Figure 1 some examples of the basic elements of these metatypes are labeled.

Canopus has 7 metamodels presented by 7 packages (see Figure 2). The main metamodels that compose our DSL are `Canopus Performance Monitoring`, `Canopus Performance Scenario`, and `Canopus Performance Scripting`, which together compose the `Canopus Performance Model`.

*1) Canopus Performance Monitoring Metamodel:* The Performance Monitoring (CPM) metamodel is intended to be used to represent the servers deployed in the performance testing environment, *i.e.* application, databases, or even the load generators. Moreover, for each one of these servers, information about the testing environment must be provided, *e.g.* server IP address or host name. It is worth mentioning that even the load generator must be described in our DSL, since in several cases it can be desirable to monitor the performance of the load generator.

The CPM metamodel requires that at least two servers have to be modeled: one that hosts the SUT and another that hosts the load generator and the monitoring tool. The metatypes supported by the CPM metamodel are: 3 objects (`SUT`, `Load Generator (LG)`, `Monitor`); 2 relationships (`Flow`, `Association`); and, 4 roles (`From`, `To`, `Source`, `Target`). Moreover, these metatypes are bound by 4 bindings. For instance, a `Flow` relationship connects, using the `From` and `To` roles, the `LG` to the `SUT` objects. Furthermore, two objects (`LG` and `SUT`) from the metamodel can be decomposed (*i.e*, into subgraphs) in a `Canopus Performance Metric` model.

*2) Canopus Performance Scenario Metamodel:* The Performance Scenario (CPSce) metamodel is used to represent the users workload profiles. In the CPSce metamodel, each user profile is associated to one or more scripts. If a user profile is associated with more than one test script, a probability must be attributed to every script belonging to this profile, *i.e.* it defines the number of users that will execute a test script. In addition to setting user profiles, in the CPSce metamodel it is also important to set one or more workload profiles. Each workload profile is decomposed into a subgraph, a `Canopus Performance Workload` (PW) metamodel, which in turn is composed by 6 objects, defined as follows:

- Virtual Users (VU): number of VU who will make

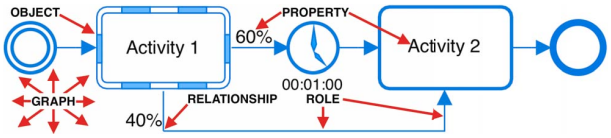


Figure 1: GOPPRR metatypes from MetaEdit+ Workbench

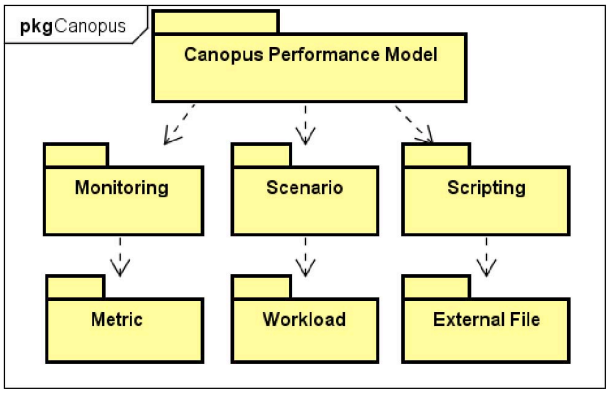


Figure 2: Canopus class diagram

requests to the SUT;
- Ramp Up Time (RUT): time each set of ramp up users takes to access the SUT;
- Ramp Up Users (RUU): number of VU who will access the SUT during each ramp up time interval;
- Test Duration (TD): refers to the total performance test execution time for a given workload. It is important to mention that the test duration will take, at least, the ramp up time multiplied by the number of intervals of ramp up time plus the ramp down time multiplied by the number of intervals of ramp down time, *i.e.* $TD \geq n \times RUT + m \times RDT$, where $n = \lceil VU/RUU \rceil - 1$ and $m = \lceil VU/RDU \rceil - 1$;
- Ramp Down Users (RDU): defines the number of VU who will leave the SUT on each ramp down time;
- Ramp Down Time (RDT): time a given set of ramp down users takes to leave the SUT.

The CPSce metamodel is composed by 3 objects (`User Profile`, `Script`, and `Workload`); 1 relationship (`Association`); 2 roles (`From` and `To`). Besides, there is a unique binding, which connects a `User Profile` to `Script` objects, respectively, through `From` and `To` roles. Moreover, each `Script` object attached to a scenario model must be decomposed into a subgraph, *i.e.* a `Canopus Performance Scripting` metamodel.

*3) Canopus Performance Scripting Metamodel:* The Performance Script (CPScr) metamodel represents each of the scripts from the user profiles in the scenarios part. The CPScr metamodel is responsible for determining the behavior of the interaction between VU and SUT. Each script includes activities, such as, transaction control or think time between activities. Similarly to the percentage for executing a script, which is defined in the CPSce metamodel, each script can also contain branches that will have a user distribution associated to each path to be executed, *i.e.* the number of VU that will execute each path. During the description of each script, it is also possible to define a set of parallel or concurrent activities. This feature is represented by a pair of fork and join objects. Our DSL also allows an activity to be decomposed into another CPScr metamodel. The CPScr metamodel also supports that a parameter generated in runtime can be saved to be used in other activities of the same script flow (the SaveParameters object handles this feature). The composition of the CPScr metamodel are: 6 objects, 4 relationships, 9 bindings and 2 metamodels.

## IV. A Model-Based Performance Testing Process

The aim of our Domain-Specific Modeling (DSM), using Canopus, is to improve a performance testing process to take advantage of MBT. Figure 3 shows our process for modeling performance testing using Canopus. The process incorporates a set of activities that have to be performed by two different parties: `Canopus` and `Third-Party`. The main activities that define our performance testing process are: `Model Performance Monitoring`, `Model Performance Scenario`, `Model Performance Scripting`, `Generate Textual Representation`, `Generate Third-Party Scripts`, `Generate XML`, `Execute Generated Scripts`, `Monitor Performance Counters`, and `Report Test Results`. The details related to the activities of our performance testing process are described next:

*1) `Model Performance Monitoring`:* The model performance monitoring is the first activity of our process, which is executed by the `Canopus` party. In this activity, the SUT, monitor servers and performance metrics that will be measured are defined. The milestone of this activity is the generation of a `Canopus Monitoring Model`. This model is composed of SUT, Load Generator (LG) and Monitor objects. A Monitor object is enabled to monitor the SUT and LG objects; this object is controlled by a `Canopus Performance Metric` that can be associated with one or more of these objects. A `Canopus Performance Metric` model represents a set of predefined metrics, *e.g.*, memory, processor, throughput, etc. Each one of them is associated with a metric counter, which in turn are linked to a criterion and a threshold.

*2) `Model Performance Scenario`:* The next activity of our process consists of modeling the performance test scenario. The `Canopus Performance Scenario Model` is the output of this activity. This model is composed of user profiles that represent VU that can be associated with one or more script objects. Each one of these scripts represents a functional requirement of the system from the user profile point of view. Furthermore, a script is a detailed VU profile behavior, which is decomposed into a `Canopus Performance Scripting Model`. Besides, each scenario allows to model several workloads in a same model. A `Canopus Performance Workload` is constituted of setup objects of test scenario, *e.g.* number of virtual users.

*3) `Model Performance Scripting`:* In this activity, each script object, modeled in the `Canopus Performance Scenario Model` from the previous activity, mimics (step-by-step) the dynamic interaction by VU with the SUT. This activity generates a `Canopus Performance Scripting Model` that is composed of several objects, such as activity, think time, save parameters and data table. It is important to notice that activity and data table objects can be decomposed into new sub-models. The former can be linked to a `Canopus Performance Scripting Model` that allows to encapsulate a set of activities to propose their reuse into other models. The latter is associated with a `Canopus External File` that fills a dynamic model with external test data provided by a performance engineer. After the three first activities from our process, the performance engineers has to decide whether they generate the textual representation from the designed models (Monitoring, Scenarios and Scripting), or input for a third-party tool, or even a generic XML file that might be integrated to any other tool that accepts XML as input.

*4) `Generate Textual Representation`:* This activity consists of generating a textual representation in a semi-natural language, a DSL based on the Gherkin [28] language that extends it to include performance testing information. Our

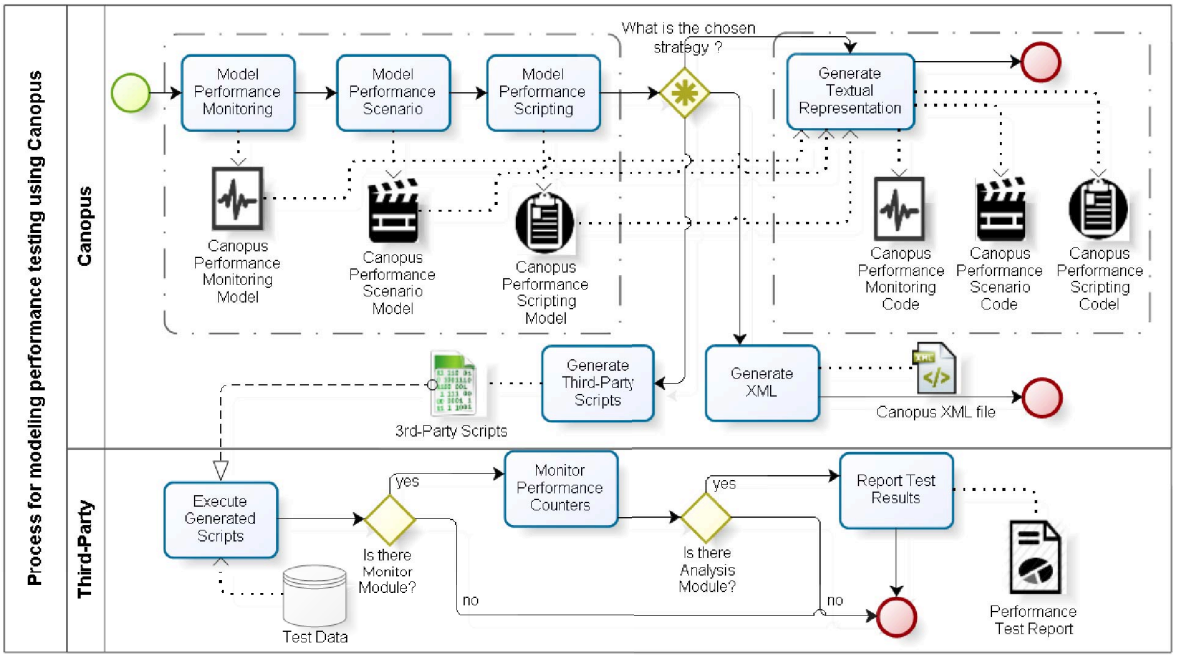


Figure 3: Model-based performance testing process using Canopus

design decision to deploy this feature in Canopus is to facilitate the documentation and understanding among development, testing teams and stakeholders.

*5) `Generate Third-Party Scripts`:* Canopus was designed to work also as a front-end to third-party performance tools. Therefore, even though we can generate a "generic" textual representation, our process can also generate input for any testing tool, hence, it can be integrated with different load generator tools such as HP LoadRunner or MS Visual Studio.

*6) `Generate XML`:* This activity is responsible to generate a Canopus XML file. We included this feature to support the integration of Canopus with other technologies. Hence, Canopus can export entire performance testing information from Canopus models to an XML file. The ability to export data in XML format might allow future Canopus users to use other technologies or IDEs. For instance, in a previous work, we developed a model-based performance testing tool, called PLeTsPerf tool [23]. This tool can parse our Canopus XML file and process the automatic generation of performance test scenarios and scripts to HP Load Runner.

The other three activities shown in Figure 3 are not part of the Canopus process and are just an example of one possible set of steps that can be executed by a performance engineer, depending on the third-party tool that is used. For example, the `Execute Generated Scripts` activity consists of executing the performance scenarios and scripts generated for a third-party tool. During the execution of this activity the load generator consumes the test data mentioned in the data table object in `Canopus Scripting Model`; `Monitor Performance Counters` activity is executed if the third-party tool has a monitoring module; and, `Report Test Results` activity is also only executed if the performance tool has an analysis module. Some of these activities can use information that exist in the Canopus models, *e.g.* `Canopus Monitoring Model`.

## V. An Industrial Case Study: Changepoint

In this section we present an industrial case study, using the Changepoint application[4], in which our DSL is applied to model performance testing information.

Changepoint is a commercial solution to support Portfolio and Investment Planning, Project Portfolio Management and Application Portfolio Management. This solution can be adopted as a "out of the box solution" or it can be customized based on the client needs. In the context of our case study, Changepoint is customized in accordance with the specific needs of a large IT company. Moreover, as Changepoint is a broad solution, in our case study we focus on how our DSL is applied to model the Project Portfolio Management module. That is, we show how to instantiate the modeling performance testing process. The goal of this case study is to evaluate and also demonstrate how our DSL can be used throughout an actual case study in an industrial setting.

To provide a better understanding of how Canopus was applied in the context of our case study, we show how to model performance testing using each one of the Canopus metamodels. Basically, we intend to explore the following Research Questions (RQ): **RQ1.** *How useful is it to design performance testing using a graphical DSL?* **RQ2.** *How intuitive is a DSL to model a performance testing domain?*

[4]www.changepoint.com

### A. Performance Monitoring

Figure 4 shows the performance testing environment where the Changepoint application is deployed to. This environment is composed of five hosts: *Database_Server* is a physical server hosting the Changepoint database; *WebApp* is a virtual machine hosting a web server; *APP_Server* is a virtual machine hosting the applications server; *Spy* is a physical server hosting a performance monitoring tool; and, the *Workload* is a physical server hosting a workload generator tool. The LoadRunner load generator is used to generate VU (synthetic workload) to exercise the *Web_Server* and *APP_Server* servers, and its monitoring module is hosted on the *Spy* server for monitoring the level of resources used by the *WebApp*, *APP_Server* and *Database_Server* servers. The monitoring module can be set up with a set of performance metrics, as well as the acceptable thresholds of computational resources, *e.g.* processor, memory, and disk usage. For instance, Figure 5 presents a single metric of the Canopus Performance Metric model. This figure depicted four objects; metric, counter, criterion and threshold. In this case, the metric is *memory* that is related to the *Available MBytes counter*. In turn, each counter can be related to two or more criteria and thresholds. A snippet of a textual representation of the Canopus Performance Monitoring model is depicted in Figure 6. It is important to highlight that the textual representation supports the definition of several metrics, including the set up infrastructure information (hostname, IP address) based on the monitoring model (Figure 4) such as monitors, load generators, and SUT servers. This model presents a memory metric composed of three criteria based on available memory, each one of them is associated to a number of VU that are accessing the SUT. For instance, when there are less than 350 users, a host must have at least 3000 MB of available memory (see Figure 6).

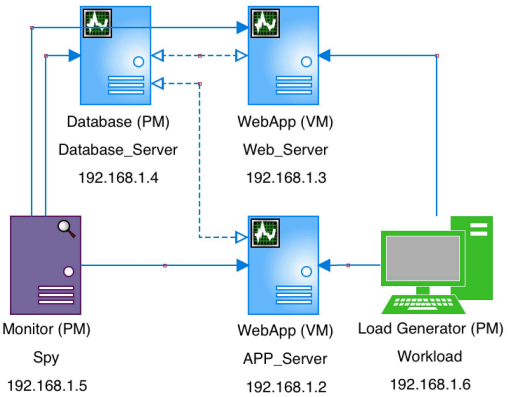


Figure 4: Graphical representation of the Changepoint `Canopus Performance Monitoring` model

### B. Performance Scenario

The Changepoint usage scenarios and workload distribution were defined based on the application requirements, describing the user profiles and their respective interactions. Figure 7 shows part of the graphical representation of the `Canopus Performance Scenario` model for the Changepoint application. This model contains 7 user profiles: `Business Administrator`, `Development Team`, `Post Sales`, `Pre Sales`, `Project Manager`, `Resource Manager` and `Solution Architect`. There are 63 script objects associated with these user profiles, which represent the behavior of each user profile when interacting with a system functionality. Moreover, each association relationship, between a script object and a user profile, has a percentage that defines the number of VU, from the total number of users defined in the workload model, *e.g.* 96.1%. The user profile total number of VU is defined by the percentage annotated in the user profile element in this model.

For instance, the `Development Team` user profile represents 77% of the workload of an entire Changepoint scenario. This workload is applied to execute the `Submit Expected Time` script based on the `Stability Testing` workload object. This object is decomposed in a `Canopus Performance Workload` model, depicted in Figure 8. This workload model defines that 1000 VU will be running over Changepoint during 4 hours (test duration time). Besides, this model also presents information about the users ramp up and ramp down. For instance, during the test execution 100 VU will be added during every 10-minute interval. In the same way, the ramp down defines the way in which the VU will

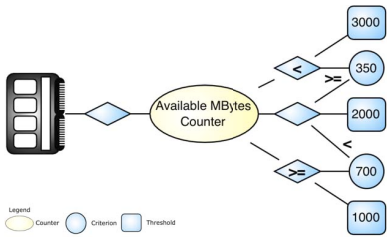


Figure 5: Graphical representation of the `Canopus Performance Metric` model

```
1    Feature Monitor the performance counters of the system
            and environment.
2 Monitoring Control the performance counters of the
            application.
3      Given that "APP_Server:192.168.1.2" WebApp monitored by
            "Spy:192.168.1.5" monitor
4        And workload generated through "Workload:192.168.1.6"
            load generator(s) for the WebApp on "AppServer"
5        And the "Changepoint Performance Scenario" test
            scenario
6       When the "Memory" is monitored
7       Then the "Available MBytes Counter" should be at least
            "2000" MB when the number of virtual user are
            between "350" and "700"
8        And at least "1000" MB when the number of virtual
            user is greater than or equal to "700"
9        And at least "3000" MB when the number of virtual
            user is less than "350"
```

Figure 6: Snippet of textual representation of the Changepoint `Canopus Performance Monitoring` model

leave the SUT. Thus, the `Stability Testing` workload defines that 1000 VU will be simultaneously executing during 2h10min, the users ramp up time (1h40min) and users ramp down time (10min), therefore, completing 4 hours of test duration time.

A textual representation of the graphical performance test scenario, depicted in Figure 7 and Figure 8, is presented in Figure 9. This textual representation, as mentioned before, is structured based on the Gherkin language, but with elements that represent the semantics for performance testing, *e.g.* virtual users, ramp up and ramp down, think time. It is worth to highlight that in this textual representation the percentage distribution among VU is already expressed in terms of values based on the workload. Due to space limitation, only a snippet of the `Canopus Performance Scenario` models are presented in this paper[5].

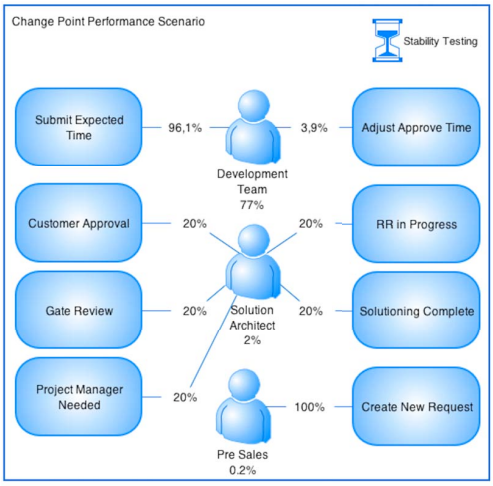


Figure 7: A partial graphical representation of the Changepoint `Canopus Performance Scenario` model

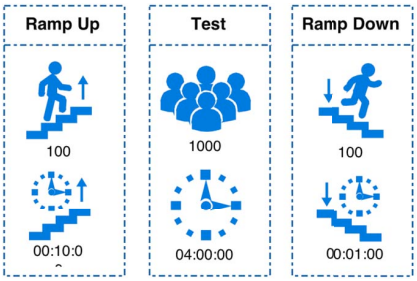


Figure 8: Graphical representation of the Changepoint `Canopus Performance Workload` model

### C. Performance Scripting

As presented in Section III-A3, it is possible to associate a subgraph model for each script element from

[5]The models designed during this case study and survey details can be found at http://tiny.cc/ICST-2016.

```
1  Feature Execute the test scenario for a workload
2  Scenario Evaluate the "Stability Testing" workload for
          "1000" users simultaneously
3     Given "100" users enter the system for each "00:10:00"
4       And "100" users leave the system for each "00:01:00"
5       And "1000" users register into the system
          simultaneously
6       And performance testing execution during "04:00:00"
7      When "1%" ("20") of the virtual users execute the
          "Solution Architect" user profile:
8       ...
9      When "77%" ("770") of the virtual users execute the
          "Development Team" user profile:
10     Then "3.9%" ("30") of them execute the "Adjust Approve
           Time" script
11      And "96.1%" ("740") of them execute the "Submit
           Expected Time" script
12      ...
```

Figure 9: Snippet of textual representation of the Changepoint `Canopus Performance Scenario` model

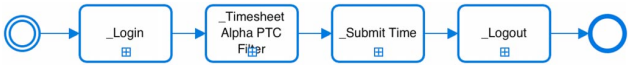


Figure 10: Graphical representation of the Changepoint `Canopus Performance Scripting` model for the `Submitted Expected Time` script

a `Canopus Performance Scenario` model. Therefore, each script element presented in the Changepoint scenario model is decomposed into a `Canopus Performance Script` model, which details each user interaction with the SUT functionalities. Figure 10 presents a snippet of the `Submit Expected Time` script. This model is composed of four activities, each one of them has another subgraph model associated, represented by "+" (plus) on the bottom script element. This decomposition allows to reuse part of the modeled performance scripts. For instance, in this case study a `_Login` script element is decomposed into a model that is included into several others scripts.

Figure 11 shows the `Canopus Performance Script` model of the `Timesheet Alpha PTC Filter` script from the main model presented in Figure 10. This model is designed with 15 activity elements, 2 data tables (`Server.dat` and `TaskId.dat`) elements, a think time element (Clock element), and a couple of join and fork elements, *i.e.*, the `Cal JQ`, `Time Sheet Status`, `Cal JQ Time Sheet Status` and `Form Time Sheet` are executed in parallel. In some activities, test data must be dynamically generated. This data can be parametrized using data table elements. For instance, the `TaskId.dat` element provides test data for four activities, *e.g.* `Time Sheet Open Task`.

A textual representation of the performance scripting model from Figure 10 is shown in Figure 12. An advantage of using a textual representation is to present all performance testing information annotated in the model, facilitating the view of all relevant information. Although the graphical representation provides a better way to represent the main domain concepts, there are some information that are better represented in the

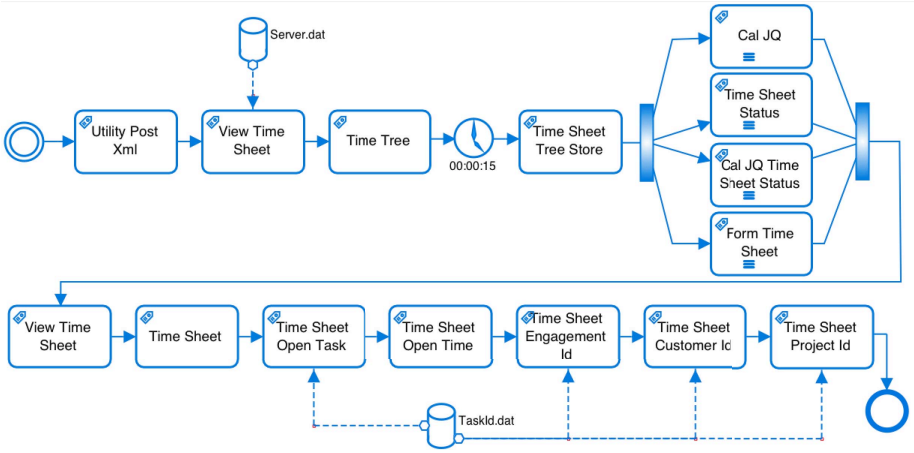


Figure 11: Graphical representation of the Changepoint CPScr model for the `_Timesheet Alpha PTC Filter` script

```
1 Feature Execute the performance test script for different
        user profiles
2  Script performance test script based on Test Case N. 1
        from "Timesheet Alpha PTC Filter"
3   Given the "Utility Post Xml" activity through "http://{
        Server}/Utility/UtilityPostXml.aspx?rid={{rId}}&sno
        ={{sno}}" action
4    When the system randomize the "Server" data within
       {Server.Server} that is dynamically generated and up-
        dated on each interaction based on a strategy random
5     And the system loaded the "rid" and "sno" data values
        previously stored
6    Then I will be taken to "http://{Server}/Core/TimeSheet/
        vwTimeSheet.asp?rid={{rId}}&sno={{sno}}" action in
        the "View Time Sheet" activity
7    ...
8    Then I will be taken to "http://{Server}/Core/Treeviews/
        TimeSheet.aspx?cid=tvEngTime&rid={{rId}}&reId=&sno
        ={{sno}}&id={EngagementId}" action in the "Time
         Sheet Engagement Id" activity
9     And the system randomize the "TaskId" data within {Tasks
        .TaskId}, which is dynamically generated and update
        on each interaction based on a strategy random
10    And the system randomize the "EngagementId" data within
        {Tasks.EngagementId}, which is dynamically generated
        and updated on each interaction based on a strategy
        same as "TaskId"
11    ...
12 Examples: TaskId.dat
13 |CUSTOMERID    | ENGAGEMENTID | PROJECTID     | TASKID       |
14 |C8DDF527-4A13| 9E988BD0-AD4D| FA13E2DC-FE3E| E9DD7653-13BB|
15 |C8DDF527-4A13| 555549F3-3473| E8453F8B-F8B7| C8AFD538-A7C7|
16 |C8DDF527-4A13| 23B6FFD9-5E9C| 074C8F57-B7C9| 39097DBF-6DF3|
```

Figure 12: Snippet of textual representation of the Changepoint `Canopus Performance Scripting` model

elements from the textual representation. Furthermore, some activities (see Figure 12) have a dynamic parameter that refers to a data table, such as the `Time Sheet Engagement Id` activity (line 8) that refers to the {`EngagementId`} parameter, which in turn refers to a data table presented at the end of the script (line 10). Note that the test data associated to the {`EngagementId`} parameter will be updated on each interaction of a virtual user based on the {`TaskId`} parameter (line 9).

### D. Case Study Analysis

We investigated and answered each one of the research questions stated previously in Section V, based on the results of our case study and interviews conducted with a performance testing team. Moreover, a web-based survey was answered by fifteen performance test experts. The purpose of this survey was to evaluate the graphical elements and their representativeness to symbolize performance elements that compose each Canopus metamodel (Scripting, Scenario, Monitoring). The subjects answered a survey composed of: (i) statements to find whether the element is representative for a specific metamodel, based on a five points Likert scale [24]: Disagree Completely (DC), Disagree Somewhat (DS), Neither Agree Nor Disagree (NAND), Agree Somewhat (AS) and Agree Completely (AC); (ii) open questions to extract their opinions on each metamodel. The answers were summarized in the frequency diagram shown in Figure 13. The numbers in this figure are based on the evaluation of 37 elements: 13 elements for the CPScr metamodel, 10 for CPSce metamodel and 14 for CPM metamodel. The frequency diagram presents the results grouped by each set of evaluated elements for each metamodel. As can be seen in the figure, 81.54% (60.51% AC + 21.03% AS) of the answers understand that the Monitoring elements are representative for the CPM metamodel. For the CPSce metamodel, 90.67% (73.33% AC + 17.33% AS) agree that the elements for that metamodel are representative. Finally, 85.71% (67.62% AC + 18.1% AS) agree that the elements represent the features they intend for the CPScr metamodel. These results are used as part of the evaluation of our DSL.

Each of the research questions mentioned in Section V, are answered next.

**RQ1.** *How useful is it to design a performance testing using a graphical DSL?* The graphical representation of a notation, language or model is useful to better explain issues to non-technical stakeholders. This was also confirmed by the performance team that reported that using our approach, it is possible to start the test modeling in early phases of the development

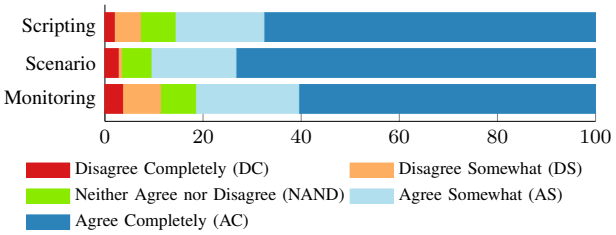


Figure 13: Frequency diagram of the graphical elements that compose Canopus, grouped by metamodel

process. Furthermore, it would be possible to engage other teams during all process, mainly the Business System Analyst (BSA). The BSA is responsible to intermediate the business layer between the developer team and the product owner. Another interesting result pointed out by the subjects is that the textual representation is also very appropriate, since it allows to replace the performance testing specification. However, as expected, there is a steep curve on the understanding of the DSL notation and an initial overhead when starting using an MBT-based approach instead of a CR-based approach.

**RQ2.** *How intuitive is a DSL to model a performance testing domain?* The case study execution indicates that the use of our DSL is quite intuitive, since the performance testing domain is represented throughout graphs, objects, relationships and properties. Visually, for instance, the scripting model can show the different flows that have been solved in several test cases on the fly, and also the decomposition features that can map objects into other graphs. This feature is also related to the reuse of partial models among models, characteristic of a DSL that allows to improve the productivity and to reduce the spent time on performance testing modeling activity.

## VI. Final Remarks and Lessons Learned

This paper presented Canopus, a DSL that was developed in the context of a collaboration between our university and a TDL from an IT company. Throughout an industrial case study, we discussed the domain analysis and presented the Canopus metamodels, as well as a process to integrate Canopus to model-based performance testing in the context of a real environment. Finally, we also showed how Canopus can be used throughout a case study to model performance testing.

Analyzing the threats to validity of our proposal, we understand that having applied Canopus to a single industrial case study addressing only one application, could be a threat to the conclusions of the viability of our DSL. We are aware that we must further investigate the suitability of Canopus to other real industrial scenarios. Furthermore, the selection of a representative case study, as well as the complexity and the size of the models designed during the study may not be representative to generalize the results. Nevertheless, to mitigate this threat we interviewed several performance engineers, as well as selected a set of distinct software projects from our TDL partner to choose the most representative case study to evaluate Canopus. In summary, the main lessons we have learned from the development process of Canopus are:

*LL1) Domain analysis:* domain analysis based on ontologies is a good alternative to transform the concepts and relationships from the ontology into entities and functionalities of the DSL. There are several methods and techniques for describing this approach, for instance [29] [30].

*LL2) DSL over a General Purpose Language (GPL):* a DSL provides a better understanding of a domain than an adapted GPL, *e.g.* UML profile. In previous studies [23] [31], we empirically investigated the advantages and disadvantages on using an MBT approach or a CR approach. The results provided evidence towards proposing our DSL, since the UML approach that was applied did not completely support the entire domain concepts and rules. We are aware that we must conduct further investigation to discuss the advantages on using a DSL instead of UML or other GPL on the performance testing domain.

*LL3) Learning curve:* one disadvantage of using a DSL is the high cost of training users who will use the DSL, *i.e.* steep learning curve [17]. However, this disadvantage can be handled pragmatically, since the cost for a new staff to learn several load generators technologies could be higher than compared to our DSL.

*LL4) Performance testing engagement:* the experience with our industrial partner point out that it is common to the performance team to engage only on the final steps of software development. Our DSL brings the performance team to engage in early stages of the software development process.

*LL5) Incremental development methodology for creating a DSL:* we adopted an incremental development methodology for creating Canopus. This methodology allowed us to improve the DSL on each interaction, which is composed of the following steps: analysis, development, and utilization [17].

*LL6) More reuse using models than scripts:* it is easier to reuse a Canopus model to create/compose other models than when reusing scripts in a CR-based approach. Moreover, the use of a CR-based approach could limit the reuse of previously generated artifacts, inducing the tester to rewrite new scripts from scratch.

In [32] we present an empirical experiment that shows that Canopus is better suited for modeling performance testing than UML models. Currently, we are working close to our partner, which is trying Canopus to other actual projects. This will give us some good insights on which elements should be improved, altered or even included in Canopus.

## Acknowledgments

We thank several performance engineers from DELL and researchers from the Research Center on Software Engineering (CePES) at PUCRS that helped in development of our DSL.

## REFERENCES


[1] Y. Yang, M. He, M. Li, Q. Wang, and B. Boehm, "Phase Distribution of Software Development Effort," in *2nd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. New York, USA: ACM, 2008, pp. 61–69.

[2] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, USA: Morgan Kaufmann, 2006.

[3] OMG, "UML Profile for Schedulability, Performance, and Time Specification - OMG Adopted Specification Version 1.1," 2005.

[4] Gatling, "Gatling Stress Tool," Available in: http://gatling.io/.

[5] M. Bernardino, A. F. Zorzo, E. Rodrigues, F. M. de Oliveira, and R. Saad, "A Domain-Specific Language for Modeling Performance Testing: Requirements Analysis and Design Decisions," in *9th International Conference on Software Engineering Advances*, Nice, France, 2014.

[6] M. Woodside, G. Franks, and D. C. Petriu, "The Future of Software Performance Engineering," in *The Future of Software Engineering*. Washington, USA: IEEE, 2007, pp. 171–187.

[7] I. Molyneaux, *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*, 1st ed. O'Reilly, 2009.

[8] S. Barber, "User Community Modeling Language (UCML) for performance test workloads," Available in: http://www.perftestplus.com/articles/ucml.pdf, sep. 2003.

[9] D. A. Menascé, V. A. F. Almeida, R. Fonseca, and M. A. Mendes, "A Methodology for Workload Characterization of E-commerce Sites," in *1st ACM Conference on Electronic Commerce*. New York, USA: ACM, 1999, pp. 119–128.

[10] M. Fowler, *Domain Specific Languages*, 1st ed. Addison-Wesley, 2010.

[11] ——, "A Pedagogical Framework for Domain-Specific Languages," *Software, IEEE*, vol. 26, no. 4, pp. 13–14, July–Aug. 2009.

[12] Microsoft, "DSL Tools Web Site," Available in: http://msdn.microsoft.com/en-us/library/bb126259.aspx, 2015.

[13] GME, "Generic Modeling Environment," Available in: http://www.isis.vanderbilt.edu/projects/gme, 2015.

[14] EMF, "Eclipse Modeling Framework," Available in: http://www.eclipse.org/modeling/emf/, 2015.

[15] MetaCase, "MetaEdit+," Available in: http://www.metacase.com/mep/.

[16] D. Ghosh, "DSL for the Uninitiated," *Queue*, vol. 9, no. 6, pp. 10:10–10:21, Jun. 2011.

[17] A. van Deursen, P. Klint, and J. Visser, "Domain-Specific Languages: an Annotated Bibliography," *SIGPLAN Notices*, vol. 35, pp. 26–36, Jun. 2000.

[18] G. Ruffo, R. Schifanella, M. Sereno, and R. Politi, "WALTy: A User Behavior Tailored Tool for Evaluating Web Application Performance," in *3rd IEEE International Symposium Network Computing and Applications*. Washington, USA: IEEE, 2004, pp. 77–86.

[19] D. Krishnamurthy, M. Shams, and B. H. Far, "A Model-Based Performance Testing Toolset for Web Applications," *Engineering Letters*, vol. 18, no. 2, pp. 92–106, May 2010.

[20] N. Bui, L. Zhu, I. Gorton, and Y. Liu, "Benchmark Generation Using Domain Specific Modeling," in *18th Australian Software Engineering Conference*. Melbourne, Australia: IEEE, Apr. 2007, pp. 169–180.

[21] Spafford, Kyle L. and Vetter, Jeffrey S., "Aspen: A Domain Specific Language for Performance Modeling," in *International Conference on High Performance Computing, Networking, Storage and Analysis*. Los Alamitos, USA: IEEE, 2012, pp. 84:1–84:11.

[22] E. M. Rodrigues, L. D. Viccari, A. F. Zorzo, and I. M. Gimenes, "PLeTs Tool - Test Automation using Software Product Lines and Model Based Testing," in *22th International Conference on Software Engineering and Knowledge Engineering*. Redwood City, USA: Knowledge Systems Institute Graduate School, Jul. 2010, pp. 483–488.

[23] E. M. Rodrigues, M. Bernardino, L. T. Costa, A. F. Zorzo, and F. M. Oliveira, "PLeTsPerf - A Model-Based Performance Testing Tool," in *8th IEEE International Conference on Software Testing, Verification and Validation*. Graz, Austria: IEEE, April 2015, pp. 1–8.

[24] R. Likert, "A technique for the measurement of attitudes," *Archives of Psychology*, vol. 140, no. 55, 1932.

[25] A. Freitas and R. Vieira, "An Ontology for Guiding Performance Testing," in *IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technologies*, Aug. 2014, pp. 400–407.

[26] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, 2007.

[27] S. Kelly, K. Lyytinen, and M. Rossi, "MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment," in *Proceedings of the 8th International Conference on Advances Information System Engineering*. London, UK: Springer-Verlag, 1996, pp. 1–21.

[28] M. Wynne and A. Hellesoy, *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2012.

[29] R. Tairas, M. Mernik, and J. Gray, "Models in Software Engineering," M. R. Chaudron, Ed. Germany: Springer, 2009, ch. Using Ontologies in the Domain Analysis of Domain-Specific Languages, pp. 332–342.

[30] T. Walter, F. Silva Parreiras, and S. Staab, "OntoDSL: An Ontology-Based Framework for Domain-Specific Languages," in *12th International Conference on Model Driven Engineering Languages and Systems*. Germany: Springer, 2009, pp. 408–422.

[31] E. M. Rodrigues, R. S. Saad, F. M. Oliveira, L. T. Costa, M. Bernardino, and A. F. Zorzo, "Evaluating Capture and Replay and Model-based Performance Testing Tools: An Empirical Comparison," in *8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. New York, USA: ACM, 2014, pp. 9:1–9:8.

[32] M. Bernardino, E. Rodrigues, and A. Zorzo, "Performance Testing Modeling: an empirical evaluation of DSL and UML-based approaches," in *Proceeding of the 31st ACM Symposium on Applied Computing*. ACM, April 2016, pp. 1–6.