A Domain-Specific Language for Modeling Performance Testing: Requirements Analysis and Design Decisions

A Domain-Specific Language for Modeling Performance Testing
Requirements Analysis and Design Decisions

Maicon Bernardino, Avelino F. Zorzo, Elder Rodrigues, Flávio M. de Oliveira
Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre, RS, Brazil
bernardino@acm.org, eldermr@gmail.com, {avelino.zorzo, flavio.oliveira}@pucrs.br

Rodrigo Saad
Dell Computers of Brazil Ltd.
Porto Alegre, RS, Brazil
rodrigo_saad@dell.com

Abstract—**Performance is a fundamental quality of software systems. The focus of performance testing is to reveal bottlenecks or lack of scalability of a system or an environment. However, usually the software development cycle does not include this effort on the early development phases, which leads to a weak elicitation process of performance requirements. One way to mitigate that is to include performance requirements in the system models. This can be achieved by using Model-Based Testing (MBT) since it enables to aggregate testing information in the system model since the early stages of the software development cycle. This also allows to automate the generation of test artifacts, such as test cases or test scripts, and improves communication among different teams. In this paper, we present a set of requirements for developing a Domain-Specific Language (DSL) for modeling performance testing of Web applications. In addition, we present our design decisions in creating a solution that meets the specific needs of a partner company. We believe that these decisions help in building a body of knowledge that can be reused in different settings that share similar requirements.**

Keywords—**performance testing; domain-specific language.**

I. Introduction and Motivation

Performance testing can be applied to improve quality of a Web-based service or application hosted on cloud computing or virtualization environments, since it supports the verification and validation of performance requirements [1]. Furthermore, it also supports evaluation of infrastructure's resource consumption while the application is under different workloads, *e.g.*, to accurately measure the resources required by an application that will respect the established Service Level Agreements (SLA). Despite the fact that performance testing is a well-known technique to validate performance requirements of an application or service, there is a lack of a modeling standard or/and language to support the specific needs of the performance testing domain.

Nevertheless, there are some notations, languages, and models that can be applied to represent a system behavior, *e.g.*, UML (Unified Modeling Language) [2], UCML (User Community Modeling Language) [3], CBMG (Customer Behavior Modeling Graph) [4], and WebML [5]. Some available modeling notations, *e.g.*, UML testing profiles, rely on the use of textual annotations on models, *i.e.*, stereotypes and tags, to support the modeling of performance aspects of an application. The use of notations, languages or models improve the performance testing activities, *e.g.*, reducing misinterpretation and providing a common document to stakeholders, system analysts and testers. Moreover, the use of a well-defined and concise notation, language or model, can support the use of Model-Based Testing (MBT) to generate inputs to the performance testing automation process, *e.g.*, test data, test scenarios and scripts can be automatically generated [6].

However, despite the benefits of using a UML profile to model specific needs of the performance testing domain, its use presents some limitations: (a) most of available UML design tools do not provide support to work with only those UML elements that are needed for a specialized language. Thus, the presence of unused and not required elements may result in an error-prone and complex activity; (b) UML diagrams are restricted to the semantics that is defined by Object Management Group (OMG) [2]. Therefore, in some cases the available UML elements and their semantics can restrict or even prevent the modeling of some performance characteristics of the Web domain.

It is important to highlight that UML is useful to analyze and design the architecture and the behavior of a system. Furthermore, it is a standard notation that does not imply in an implementation decision; besides, it is helpful for representing higher level concepts and the initial glossary domain. When compared to UML, Domain-Specific Languages (DSLs) are less general, and are based on an implementation strategy. That is, UML is used at an implementation independent level, whereas DSLs are used at an implementation dependent level. DSLs are restricted languages that can be used to directly model concepts in a specific problem domain. These languages can be textual, like most programming languages, or graphical. Furthermore, each DSL is a domain-specific code generator that maps domain-specific models into the required code.

In spite of the fact that performance testing is an active research field, researches investigating how to apply MBT approaches to automate the performance testing activities essentially started to be reported in the last decade and it is still on its early stages [6][7][8]. Furthermore, the lack of a standard to represent performance testing information is one of the major challenges to be explored from both, academic and industrial practitioners.

In this work, we discuss the requirements and design decision on the development of a modeling notation for performance testing. Thus, we propose a DSL that focus on meeting specific needs for modeling performance testing of Web applications. In this context, we also discuss the use of our DSL to support an MBT approach to generate performance testing artifacts to test these applications. Our contribution is twofold: (a) we identify a set of requirements, specific to our research context, that are not fully addressed by any known languages, model or notation. Thus, we elicit some practical scenarios that language providers and/or implementers may consider supporting; (b) we report our design decisions in supporting these requirements for an in-house solution. These decisions, in turn, may be reused or adapted to improve existing tools or devise new ones targeting similar needs.

This paper is organized as follows. Section II discusses background on performance testing, DSL and related work. Section III presents the context in which our work was developed. Section IV describes the domain analysis process. Section V enumerates the elicited requirements, which we address with specific design decisions, discussed in Section VI. Section VII briefly presents an example of use. Section VIII concludes the paper.

II. BACKGROUND

A. Performance Testing

Software Performance Engineering (SPE) [9] describes and provides support to improve the performance through two distinct approaches: an early-cycle predictive model-based, and a late-cycle measurement-based. Performance testing is an essential measurement-based activity in the software development process, since it helps to identify bottlenecks that impact performance and scalability in a system. It is used to understand the behavior of a system under a certain workload.

According to Meier *et al.* [1], performance testing can be applied in different domains of applications, such as desktop, Web services, and Web applications. The process of designing and executing performance testing to a specific domain is composed by a set of well-defined activities. Therefore, to support the performance testing process, as well as its activities, a set of tools has been developed, *e.g.*, HP LoadRunner or Microsoft Visual Studio. Some of these tools support the generation of performance test scenarios and scripts through Capture and Replay technique or just manually coding scripts. Another technique that can be applied is MBT, which is useful to automate the performance testing process. A few academic and industrial performance testing tools based on models can be found in the literature, *e.g.*, SWAT [6], MBPeT [10] or PLeTs [11]. Despite the existence of some MBT tools, few of them use the same model, *i.e.*, a modeling standard for performance testing has not yet been set. Furthermore, there are some theoretical notations that do not allow test automation, *e.g.*, UCML [3].

B. Domain-Specific Language

DSLs, also called application-oriented, special purpose or specialized languages, are languages that provide constructs and notations tailored for a particular domain. DSLs are specific domain and problem-oriented computer languages [12]. DSLs are created to solve specific problems of a particular domain, *e.g*, in our case performance testing. However, to create a DSL, a domain analysis phase is required, which leads to a solid body of knowledge about the domain. During this phase, the domain's rules, features, and properties must be identified and documented. Currently, there are several tools, called Languages Workbenches (LWs), to support the creation and maintaining of a DSL, such as Eclipse Modeling Framework (EMF) [13], MetaEdit+ [14], among others. These tools are not restricted to analysis and code generation, LWs allow a DSL developer to create DSL editors with similar power to modern IDEs [12]. Thereby, a DSL can be classified in accordance with its creation techniques/design, that are the following: internal, external, and based on LWs.

Therefore, the use of DSLs presents some advantages, such as [15]: (a) better expressiveness in domain rules, allowing to express the solution at a high level of abstraction. Consequently, domain experts can understand, validate, modify or develop their own solutions; (b) improves the communication and collaboration among software project stakeholders; (c) supports artifacts and knowledge reuse. Inasmuch as DSLs can retain the domain knowledge, the adoption of DSL allows the reuse of the retained domain knowledge by a mass of users, including those that are not experts in the domain; (d) a DSL can provide a better Return Of Investment (ROI) than a traditional model. Despite that, the use of DSLs can present some disadvantages, such as [15]: (a) high cost to design and maintain a DSL, especially if the project presents a moderate or a high complexity. (b) high cost for training DSL users, *i.e.*, steep learning curve; (c) difficulty to define an adequate scope for a DSL; (d) a company could become dependent of an in-house language that is not used anywhere else; (e) in case of executable DSLs there are issues related to the performance loss when compared to source code written by a developer.

C. Related Work

The DSL community currently lacks evidence on the driving factors on the development of a DSL for the performance testing domain, its corresponding requirements and design decisions. There are few works describing DSLs requirements [16][17]. Whilst Kolovos [16] presents the core requirements and discuss the open issues with respect to the DSL requirements. Athanasiadis [17] describes the key requirements for a DSL to the environmental software domain, and discusses its benefits. However, these works present only an incipient discussion about the design decisions and, are not focused on the performance testing domain.

Conversely, there are some works [18][19] reporting design decisions for creating DSLs. Kasai *et al.* [18] proposed design guidelines that covers the following categories: language purpose, realization, content; concrete syntax; and abstract syntax. Frank [19] suggests guidelines to support the design decisions on a DSL development, *aka* DSML. Moreover, the author presents a useful discussion concerning the design decisions to DSLs. Although these studies are relevant, the design decisions were not taken to meet real requirements (*i.e* industrial requirements), but only based on the DSL creators knowledge and academic experience.

Some studies propose similar DSLs for the testing domain [20][21]. Bui [20] presents a DSL, DSLBench, for benchmark generation, while Spafford [21] presents a DSL, Aspen, for performance modeling. Different from our DSL proposal, focused in the measurement-based approach, Aspen supports the predictive-based approach. However, these works do not present any feedback from industrial cases where these DSLs are used, or discuss where they succeed or fail when applying on an MBT approach. Gatling [22] proposed an internal DSL based on industrial needs and tied to a testing tool. Unlike our proposal that provides a graphical DSL to represent the performance notation, Gatling provides only a textual DSL based on the Scala language. Moreover, our DSL is not tied or dependent of any performance testing tool or load generator. Wert *et al.* [23] present a novel automated approach for performance problem detection, in which they combined systematic search based on a decision tree with goal-oriented experimentation. However, this work is not based on MBT.

III. CONTEXT

This study is being conducted in cooperation with the Technology Development Lab (TDL) of a global IT company. This cooperation aims to develop new strategies and approaches for software testing. The software process adopted by TDL is briefly described as follows. First the development teams implement the unity tests, which performs the preliminary tests in a dedicated development environment. After a stable version that meets its requirements is available, it is published in the test environment for the test team to perform functional and integration tests. If failures are found, the development team removes faults from the application and the process is restarted. If no failure is found, the software version becomes available for the performance testing team to perform load and stress testing in a new environment (called performance environment). It is important to mention that for some applications, sometimes the complexity or size of the production environment is so high that testing is executed only in a partial production environment and proportional estimations are used [10]. Finally, when the application meets the defined quality requirements, *e.g.*, response time, that version is deployed to the production.

Empirical evidences of our previous work [24], developed in collaboration with the TDL, indicate that a performance MBT approach is valuable to mitigate the effort to generate performance scripts. Our findings also indicate that performance testing teams, usually, choose notations and modeling tools by their own convenience. Thus, many models can be found across the company performance testing teams, which leads to a segmented knowledge about applications requirements. Hence, the use of different models can present other issues, such as inhibit communications among testing teams and increases the possibility of requirements misinterpretation. To attenuate these issues, it is necessary that a single notation is used across the entire performance testing division. Since there is not a standard modeling language to model the specific needs of the performance testing domain, we focused our effort on investigating and proposing a DSL to this domain.

IV. Domain Analysis

Before we start to describe our DSL, it is important to mention some of the steps that were taken prior to the definition of the requirements and design decisions. These steps were taken in collaboration with researchers from our group and test engineers from TDL.

The first step is related to the expertise that was acquired during the development of a Software Product Line (SPL) to generate MBT tools called PLeTs [11]. This SPL was split in two main parts: one to analyse models and generate abstract test cases from those models, and; another that would take the abstract test cases and derive actual test scripts to be executed by performance testing tools. Actually, our SPL is divided in four main features: parser, test case generation, script generation, and execution. Several models were studied during the first phase of the development of our SPL, *e.g.*, UCML, UML Profiles, Finite State Machines (FSM). Likewise, several performance testing environments and tools were studied, such as HP LoadRunner, Microsoft Visual Studio, among others.

The second step is related to the use of some of the above models and tools to actual applications, such as TPC-W and Moodle. Furthermore, we also used some of the products generated by our SPL to test those applications. Some of other real applications from our partner were also tested in the context of our collaboration. Those real applications were tested in very complex environments, which gave us a very thorough understanding of the needs a testing team has.

Besides that, we also based our DSL on well-known concepts from SWEBOK, IEEE Std. 610.12-1999, IEEE Std. 829-2008, and other literature, such as Meier *et al.* [1]. These references were chosen to mitigate the bias, provide a theoretical basis and ensure the coherency among concepts, features, and properties of the performance domain. The above steps provided us with a small Performance Testing Body Of Knowledge (PTBOK) that is used to define the performance testing requirements and design decisions for our DSL. Furthermore, prior to create any DSL to support modeling performance testing in the target DSL, the TDL first considered the use of off-the-shelf solutions, provided that specific requirements were met.

V. Language Requirements

This section enumerates the requirements we collected from our expertise and also from the software engineers from TDL. These requirements are related to features and concepts from performance testing domain. Moreover, we discuss some mechanisms for implementing the proposed DSL.

RQ1*) The DSL must allow to represent the performance testing features.* One of the main functions of the performance testing is to reveal bottlenecks of a system. Therefore, the applications should be measured and controlled in small parts that can be defined as transactions. This allows to measure the performance quality for each activity of a system. For instance, to define the response time SLA based on these transactions.

RQ2*) The technique for developing our DSL must be based on LW.* Since we do not want to develop new tools, *i.e.*, editor or compiler, as in an external DSL; neither we intend to embed our DSL in a GPL, we will base our DSL on a LW. This will allow us to focus on the analysis domain and development of the new DSL rather than spend effort on implementing new tools or having to choose a GPL language that might not be appropriate for the DSL that we want.

RQ3*) The DSL must support a graphical representation of the performance testing features.* This requirement does not concern the language itself, but the LW that will support its development. Thereunto, we desire that the LW supports a graphical-based editor for creating DSLs. Moreover, the LW should allow to implement the domain concepts, their translation rules, designing symbols and elements of the language, and also to generate different code for different tools.

RQ4*) The DSL must support a textual representation.* The proposed DSL should also include a custom language that is close to a natural language. This will facilitate its adoption by test engineers that are used to use textual representation. The language should have features and keywords that remember the performance testing domain.

RQ5*) The DSL must include features that illustrate performance counters.* In performance testing there are many performance counters, *e.g.*, response time or network throughput, that provide means to analyze both application quality level and host infrastructure.

RQ6*) The DSL must allow to model the behavior of different user profiles.* This requirement is a specific function of the performance domain, which should allow that the behavior of different user profiles, such as a buyer or a new clients, is modeled according to the System Under Test (SUT). In our context we will focus on Web applications.

***RQ7)** Traceability links between graphical and textual representations should require minimal human intervention/effort.* Traceability is an important feature in software solutions, mainly when involve model transformation, *e.g.*, translation from a graphical to a textual representation. The proposed DSL should automate the mapping process of graphical elements of the model to their respective textual counterparts.

***RQ8)** The DSL must be able to export models to formats of specific technologies.* This requirement should ensure that models written in our proposed DSL can be exported to the format of the input of specific testing tools, *e.g.*, HP LoadRunner, MS Visual Studio or Apache JMeter.

***RQ9)** The DSL must generate model information in a eXtensible Markup Language (XML) file.* This requirement aims to ensure that we can export our DSL to any other technology in the future. That is, we export all information from the system model into a XML file, so anyone that wants to use our solution can import the XML into their technology.

***RQ10)** The DSL must represent different performance test elements in test scripts.* The modeled diagram using the proposed DSL must represent multiples elements of test scripts, such as conditional or repetition control flows, among others.

***RQ11)** The DSL must allow the modeling of multiple performance test scenarios.* Performance testing is responsible to carry out testing of part of or the whole system under normal and/or stress workload. The DSL, therefore, should be able to generate multiples performance test scenarios, *i.e.*, under normal and stress workload conditions.

Currently, to the best of our knowledge, no existing language or model (commercial or not) meets all of the presented requirements. Therefore, given the automation needs of performance testing, we propose a DSL for modeling performance testing of Web applications.

VI. Design Decisions

In this section, we describe our design decisions for creating a DSL that supports the requirements discussed in Section V. For each design decision, we mention the associated requirements that are being dealt with.

***DD1)** To use a LW that supports graphical DSLs* (RQ2, RQ3). To attend these requirements we performed a literature review on existing LWs, including academic, commercial or open-source. The work of Erdweg *et al.* [25] presents the state-of-the-art in LWs and defines some criteria (the authors call them features) that help someone to decide which tool should be adopted. Given the requirements of our proposed DSL, we chose the MetaEdit+ from MetaCase [14], because it supports most of the features evaluated by work.

***DD2)** The features of the performance testing domain will be used in an incremental way* (RQ1, RQ5). Developing a DSL requires a series of phases, such as analysis, design, implementation, and use [26]. Usually researchers focus their attention to the implementation phase, but only a few of them focus on the analysis of the domain and design of the DSL. Nevertheless, there are some methodologies for domain analysis, which helps to unravel the knowledge about the problem domain analyzed. Among them we can highlight Domain Specific Software Architectures (DSSA), Feature-Oriented Domain Analysis (FODA), and Organization Domain Modeling (ODM). Some works present an approach based on the formalization of domain analysis through ontologies [27][28]. Thus, in order to determine the features that represent the performance testing domain, we adopted a strategy to identify and analyze the domain using an ontology [29]. This ontology provides the basis for determining the concepts, relationships, and constraints that represent the performance testing domain. Besides the ontology, we have used the PTBOK (Section IV).

***DD3)** To provide a graphical language capable of representing the behavior of user profiles for different performance test scenarios* (RQ6, RQ11). To attend these requirements we analysed different models and graphical representations that support performance testing. Among the approaches and techniques, the most relevant for our work were UML profiles. Besides that, it is also important to mention a theoretical language proposed by Scott Barber for modeling users behavior, called UCML. Based on these different approaches and techniques, the graphical language will have visual elements capable of representing the behavior of different user profiles. Besides the flow of activities that the user performs in the SUT, the graphical language will have visual elements to represent the performance test scenarios settings, including information about the performance testing domain, such as number of Virtual Users (VU), test duration, metrics to be evaluated (response time, memory available, processor time, among others). It is also possible to include the randomization and execution probabilities for each interaction that a VU executes during performance testing. Another very important feature is that the DSL can represent abstract data that will be instantiated in activity of the performance testing process, for example, during the generation of the performance test scripts.

***DD4)** To create a textual representation in a semi-natural language* (RQ4). Behavior-Driven Development (BDD) [30] is an agile software development process, in which acceptance testing, mainly functional testing, is essential to advance to next phase of a software project, since it facilitates the understanding among testing and development teams and stakeholders. Usually, tests are described in natural language in order to ensure this common understanding regarding the system requirements for all project members. Even though it is common to find languages that use natural language to describe functional testing, *e.g.*, Gherkin [30], to the best of our knowledge none of them includes performance testing features. Therefore, we intend to extend this language, to include the performance testing features described in Section IV. Gherkin is interpreted by a command line tool called Cucumber, which automates the acceptance testing execution.

***DD5)** To provide automated traceability between the graphical and textual representations* (RQ7, RQ10). Traceability is an important feature that should be mapped in the implementation of a DSL. Thus, it is required that the LW allows the creation of translation rules among models. In this case, the mapping among the graphical elements with their respective assets of the textual representation must be provided. It is important that this mapping is not an one-to-one mapping. Some graphical elements can be mapped to several instances of the textual elements. For example, a graphical decision point can be mapped to several textual scripts, one for each branch present in the graphical representation. In order to solve this mapping, algorithms such as the Chinese Postman Problem can be used.

***DD6)** To support the integration of the DSL with other technologies* (RQ8, RQ9). It should be able to export the models (test scenarios, abstract test cases, etc.) described in the DSL to other formats, such as XML or HP LoadRunner and MS Visual Studio input formats. The ability to export data in XML format will allow future users of the language to use it with other technologies or IDEs.

VII. DSL for Modeling Performance Testing

This section presents the DSL we developed to meet the requirements described in Section V and based on the design decision from Section VI. Our DSL is composed of three parts: monitoring, scenario, and scripting.

Monitoring: The performance monitoring part is responsible for determining all servers used in the performance testing environment. For each server (*i.e.*, application, databases, or even the load generator), information on the actual testing environment has to be included, *e.g.*, IP address or host name. It is worth mentioning that even the load generator has to be described in our DSL, since we can also monitor the performance of the load generator. Sometimes, the load generator has to be split in several servers if we really want to stress the application or database server. For each host, it is possible to indicate the performance counters that will be monitored. This monitoring part requires that at least two servers have to be described: one that hosts the application (SUT) and another to generate the workload and monitor the performance counters of the SUT.

Scenario: The performance scenario part allows to set user and workload profiles. Each user profile is associated to test scripts. If a user profile is associated with more than one test script, a probability is attributed between the user profile and each test script, *i.e.*, it describes the probability that that test script is executed. In addition to setting user profiles, in this part, it also is important to set one or more workload profiles. Each workload profile of is composed of several elements, defined as follows: (a) *virtual users*: number of VU who will make requests to the SUT; (b) *ramp up time*: time it takes for each set of ramp up users to access the SUT; (c) *ramp up users*: number of VU who will access the SUT during each ramp up time interval; (d) *Test duration*: refers to the total time of performance test execution for a given workload; (e) *ramp down users*: defines the number of VU who will left the SUT on each ramp down time; (f) *ramp down time*: defines the time it takes for a given ramp down user stop the testing.

Scripting: The performance script part represents each of the test scripts from the user profiles in the scenarios part. This part is responsible for determining the behavior of the interaction between VU and SUT. Each test script includes activities, such as transaction control or think time between activities. The same way as there is a probability for executing a test script, which is defined in the scenarios part, each test script can also contain branches that will have a user distribution associated to each path to be executed, *i.e.*, the number of users that will follow each path. During the description of each test script it is also possible to define a decision table associated to each activity. This decision table [12] represents the decisions that is taken by a user based on the response that an application provides. Actually, the decision table will be used to guarantee that the user distribution rate is achieved.

A. Example of Use: TPC-W

This section presents a small sample of the graphical representation described in previous sections. We instantiate the modeling performance testing process through the proposed DSL for the TPC-W e-commerce Web application. The goal is to give some understanding of the requirements and design decisions of a language to model user behavior, as well as the performance test scenarios. The graphical representation contains elements for virtual users, test duration, ramp up time, ramp up users, ramp down time, think time, among others.

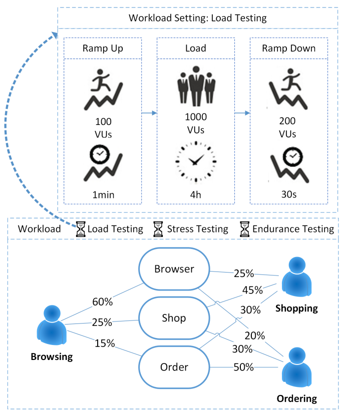
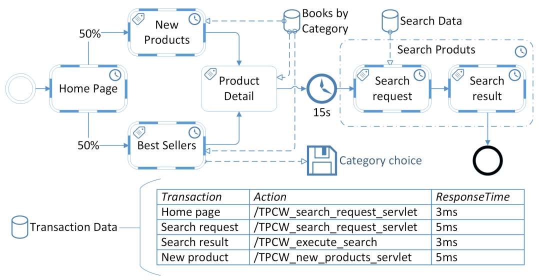


Figure 1. Graphical representation of a performance test scenario model.

Figure 1 gives an example of a performance test scenario model that represents the SUT functionalities divided into three scripts: `Browser`, `Shop` e `Order`. Each script has a percentage of the VU that will execute such script. For each script, a VU has a probability of buying a product from the site. This probability is bigger for script `Order` and smaller for script `Browser`. Due to space limitation this is not shown in this paper. The model also shows the interaction behavior of three different user profiles: `Browsing`, `Shopping` e `Ordering`. Basically, the user profiles differ from one another on the probability that they will order, shop or browse.

A snippet of the `Browser` script is presented in Figure 2. The model is composed of six activities, five of them with transaction control, shown by dashed border figures. The model also contains a `Think Time` of 15 seconds (Clock element) and three `Data Table`, *e.g.*, `Transaction Data`. In some cases, there is the necessity to store the results of processing of an activity into global variables or parameters, so that this data can be used in other activities, for example to decide a path in a decision point (see `Category choice`). The model also shows a transaction composed by a set of activities, see the `Search Products` transaction, which is composed of `Search request` and `Search result`.



Transaction	Action	ResponseTime
Home page	/TPCW_search_request_servlet	3ms
Search request	/TPCW_search_request_servlet	5ms
Search result	/TPCW_execute_search	3ms
New product	/TPCW_new_products_servlet	5ms

Figure 2. Performance test scripting model of the `Browser` script.

VIII. Lessons Learned and Final Remarks

Basically, the main lessons we have learned from the requirements and design decisions of our DSL are: (a) there are several techniques to achieve the best possible results from requirements elicitation process, such as interviews, ethnography, domain analysis, among others. We have used domain analysis to understand and capture our domain knowledge, and to identify of reusable concepts and components. Thus, we learned that eliciting requirements based on domain analysis, having as output the PTBOK, was effective to identify what we needed for our DSL; (b) 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 [27][28]; (c) one of the disadvantages of using DSLs is the high cost of training users who will use the DSL, *i.e.*, steep learning curve [15]. However, based on our previously experience using several load generator tools in an industrial setting, this disadvantage can be handled pragmatically, since the cost for a new staff to learn several load generators technologies is higher than compared to our DSL. Nonetheless, this drawback must be proved with empirical evidences; (d) Global Software Development refers to software development geographically or globally distributed, which aims to streamline the process of product development. In such scenario it is common that infrastructure and performance teams are located in different countries. For this reason, it is important to adopt a standard language for creating scripts and models for performance testing, hence we chose the English as default for the textual representation of our DSL, implicitly we avoid a cacophonous language [12]; (e) we adopt an incremental development methodology for creating our proposed DSL. This methodology allows us to improve the DSL on each interaction, which is composed by the following steps: analysis, development, and utilization [15].

This paper presented a set of requirements elicited in the context of an industrial partner. Through a pilot study and based on our PTBOK, we collected specific needs for performance modeling adoption to create a DSL for modeling performance testing, and argue that existing models and languages do not meet the specificity of the requirements at hand. We then presented our design decisions for creating a DSL. We claim that the reported requirements and design decisions as the two contributions of this work, since currently few studies bring such discussion. Our work adds to that in the sense that the elicited requirements evidence practical scenarios that other load generators may consider supporting; the design decisions, in turn, may be reused or adapted to improve existing DSLs, models or languages, or even new ones, targeting similar requirements.

IX. Acknowledgment

Study developed in the context of PDTI 001/2014, financed by Dell Computers with resources of Law 8.248/91.

References

[1] J. Meier, C. Farre, P. Bansode, S. Barber, and D. Rea, Performance Testing Guidance for Web Applications: Patterns & Practices. Microsoft Press, 2007.

[2] OMG, “Object Management Group,” 2014, URL: http://www.omg.org [retrieved: 08, 2014].

[3] S. Barber, “User Community Modeling Language (UCML) for performance test workloads,” Sep. 2003, URL: http://www.perftestplus.com/articles/ucml.pdf [retrieved: 08, 2014].

[4] D. Menascé, V. Almeida, R. Fonseca, and M. Mendes, “A Methodology for Workload Characterization of E-commerce Sites,” in 1^{st} ACM Conference on Electronic Commerce. ACM, 1999, pp. 119–128.

[5] N. Moreno, P. Fraternali, and A. Vallecillo, “WebML modelling in UML,” IET Software, vol. 1, no. 3, pp. 67–80, June 2007.

[6] M. Shams, D. Krishnamurthy, and B. Far, “A Model-based Approach for Testing the Performance of Web Applications,” in 3^{rd} International Workshop on Software Quality Assurance, 2006, pp. 54–61.

[7] 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, 2010.

[8] M. B. da Silveira *et al.*, “Generation of Scripts for Performance Testing Based on UML Models,” in 23^{rd} International Conference on Software Engineering and Knowledge Engineering, Jul. 2011, pp. 258–263.

[9] M. Woodside, G. Franks, and D. C. Petriu, “The Future of Software Performance Engineering,” in Future of Software Engineering, 2007, pp. 171–187.

[10] F. Abbors, T. Ahmad, D. Truscan, and I. Porres, “MBPeT - A Model-Based Performance Testing Tool,” in 4^{th} International Conference on Advances in System Testing and Validation Lifecycle, 2012, pp. 1–8.

[11] 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 22^{th} International Conference on Software Engineering and Knowledge Engineering, Jul. 2010, pp. 483–488.

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

[13] EMF, “Eclipse Modeling Framework,” 2014, URL: http://www.eclipse.org/modeling/emf/ [retrieved: 08, 2014].

[14] S. Kelly, K. Lyytinen, and M. Rossi, “MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment,” in 8^{th} International Conference on Advances Information System Engineering. Springer, 1996, pp. 1–21.

[15] D. Ghosh, “DSL for the Uninitiated,” Queue, vol. 9, no. 6, pp. 10–21, 2011.

[16] D. Kolovos, R. Paige, T. Kelly, and F. Polack, “Requirements for Domain-Specific Languages,” in 1^{st} Domain-Specific Program Development, Jul. 2006, pp. 1–4.

[17] I. N. Athanasiadis and F. Villa, “A Roadmap to Domain Specific Programming Languages for Environmental Modeling: Key Requirements and Concepts,” in ACM Workshop on Domain-Specific Modeling, 2013, pp. 27–32.

[18] G. Karsai *et al.*, “Design Guidelines for Domain Specific Languages,” in 9^{th} OOPSLA Workshop on Domain-Specific Modeling, 2009, pp. 7–13.

[19] U. Frank, “Domain-Specific Modeling Languages: Requirements Analysis and Design Guidelines,” in Domain Engineering, 2013, pp. 133–157.

[20] N. Bui, L. Zhu, I. Gorton, and Y. Liu, “Benchmark Generation Using Domain Specific Modeling,” in 18^{th} Australian Software Engineering Conference, Apr. 2007, pp. 169–180.

[21] K. L. Spafford and J. S. Vetter, “Aspen: A Domain Specific Language for Performance Modeling,” in International Conference on High Performance Computing, Networking, Storage and Analysis, 2012, pp. 1–11.

[22] Gatling, “Gatling Stress Tool,” 2014, URL: http://gatling-tool.org [retrieved: 08, 2014].

[23] A. Wert, J. Happe, and L. Happe, “Supporting Swift Reaction: Automatically uncovering performance problems by systematic experiments,” in 35th International Conference on Software Engineering, May 2013, pp. 552–561.

[24] E. M. Rodrigues *et al.*, “Evaluating Capture and Replay and Model-based Performance Testing Tools: An Empirical Comparison,” *“forthcoming”* in International Symposium on Empirical Software Engineering and Measurement, 1–8 2014.

[25] S. Erdweg *et al.*, “The State of the Art in Language Workbenches,” in Software Language Engineering, 2013, vol. 8225, pp. 197–217.

[26] M. Mernik, J. Heering, and A. M. Sloane, “When and How to Develop Domain-Specific Languages,” ACM Computing Surveys, vol. 37, no. 4, pp. 316–344, 2005.

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

[28] T. Walter, F. S. Parreiras, and S. Staab, “OntoDSL: An Ontology-Based Framework for Domain-Specific Languages,” in 12^{th} International Conference on Model Driven Engineering Languages and Systems, 2009, pp. 408–422.

[29] A. Freitas and R. Vieira, “An Ontology for Guiding Performance Testing,,” *“forthcoming”* in International Conferences on Web Intelligence and Intelligent Agent Technology, 2014, pp. 1–8.

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