A Concept for Language-Oriented Security Testing

Conference Paper · June 2013

DOI: 10.1109/SERE-C.2013.16

CITATION
1

READS
179

4 authors, including:

Michael Felderer
University of Innsbruck
227 PUBLICATIONS **1,588** CITATIONS
SEE PROFILE

Matthias Farwick
Txture GmbH
33 PUBLICATIONS **352** CITATIONS
SEE PROFILE

Ruth Breu
University of Innsbruck
185 PUBLICATIONS **2,145** CITATIONS
SEE PROFILE

2013 Seventh International Conference on Software Security and Reliability Companion

A Concept for Language–oriented Security Testing

Philipp Zech, Michael Felderer, Matthias Farwick, and Ruth Breu
Institute of Computer Science
University of Innsbruck
Austria
Email: {philipp.zech, michael.felderer, matthias.farwick, ruth.breu}@uibk.ac.at

***Abstract*—Today's ongoing trend towards intense usage of web service based applications in daily business and everybody's daily life poses new challenges for security testing. Additionally, such applications mostly not execute in their own runtime environment but instead are deployed in some data center, run alongside multiple other applications, and serve different purposes for sundry user domains with diverging security requirements. As a consequence, security testing also has to adapt to be able to meet the necessary requirements for each application in its domain and its specific security requirements. In addition, security testing needs to be feasible for both service providers and consumers. In our paper we identify drawbacks of existing security testing approaches and provide directions for meeting emerging challenges in future security testing approaches. We also introduce and describe the idea of language–oriented security testing, a novel testing approach building upon domain–specific languages and domain knowledge to meet future requirements in security testing.**

***Index Terms*—Security Testing, Domain–specific Language, Language–oriented Programming, Service–centric Systems**

I. Introduction

Since the beginning of the millennium IT faced tremendous changes in its usage. A first indicator of those changes was the early adoption of web services back in the early 1990s [1]. However, it lasted more than another decade until the mid 2000s, when the first Service Oriented Architectures (SOA) emerged and, more generally, large Service–centric Systems (ScS). The goal was to, on the one side, slenderize the desktop and, on the other side, to project complex business processes onto IT to enhance business itself [2]. This trend continued up to today, with its latest derivative, the evolution of cloud computing [3]. Yet, the application of cloud computing also emerged out of another opportunity of such ScS, the possibility of outsourcing complete IT landscapes (both, hardware as well as software) thereby saving money due to only paying for the effective usage of hard- and software.

Taking a more distant look at such service–centric applications reveals that their only actual commonality is their shared runtime environment, providing support for a vast amount of different services or applications. However, due to the ongoing evolution of services computing, application specific requirements, especially the ones regarding security, changed dramatically. Whereas before outsourcing, when an application basically only needed to be secured against outsider attacks, in this new services computing paradigm, this requirement has drastically changed, as a malicious user now has the possibility to invade a system by already starting from its internals, due to the shared runtime environment [4]. Current security testing approaches fail to address such requirements, e.g., testing against insider attacks (see Section III).

Besides their shared environment, that poses new requirements for security testing, another serious issue for security testing arises when considering the domains of an ScS and their requirements. Here we distinguish between two domains, viz. the business domain, referring to the type of the system (e.g., e–Health or Enterprise Resource Planning), and the technological domain, referring to technological aspects of the ScS (e.g., what technology is used for service provisioning). Also important for testing is the specific intended functionality of the service. Is the service a building block for a composed service and hence, needs some kind of integration testing or is it just a simple management service, whose proper functioning can be assured by unit testing? Keeping these in mind requires to do much testing following different approaches to meet all requirements. And yet, this kind of testing does not even regard security related aspects at all. Adding the additional effort, necessary to do security testing in a meaningful manner, results in a long–lasting software development process that consumes high expenses. This testing dilemma becomes even worse if taking the insufficient tool support for security testing into account (see Section III).

What also makes security testing in such a service–centric environment challenging is, that testing needs to be feasible for both the service providers and the service consumers. The former wants to test services before deploying and providing them, whereas the latter in addition must have the possibility to test services before consuming or integrating them in their own applications and business processes. What has to be kept in mind at this point is a potential lack in technical knowledge by at least the service consumer. Current security testing approaches fail to abstract the necessary level of technical knowledge for that *untrained developers* can successfully test software systems on their own [5].

In our paper we identify drawbacks of current security testing approaches for ScS and, based on those drawbacks identify emerging requirements for future security testing approaches. We also introduce and motivate the idea of language–oriented security testing, a novel approach that attempts to focus especially on these requirements by applying domain specific abstractions. Additionally, we provide an in-depth discussion of our novel approach. Hence, the main contributions of our paper are:

978-0-7695-5030-5/13 $26.00 © 2013 IEEE
DOI 10.1109/SERE-C.2013.16

- An overview on the most important existing security testing approaches.
- Elicitation of requirements for making future security testing approaches useful and prolific.
- A novel approach for security testing ScS using language oriented programming as its base. It reduces the complexity of security testing and allows to focus on the technical and business related domain aspects of the system under test (SUT).

The remainder of our paper is structured as follows. Section II gives a review of the most important existing security testing approaches, followed by Section III, where we identify drawbacks of current approaches. Section IV presents emerging requirements that future security testing approaches must meet. Later, in Section V we motivate and sketch our idea of a novel testing approach called language–oriented testing that addresses the emerging requirements. Finally, we conclude in Section VI with a discussion of future work.

II. Status Quo in Security Testing

In this section we present a coarse classification of security testing and give a brief overview of the most important and relevant security testing techniques together with existing methods and tools in the area of security testing of ScS.

Security testing is characterized by software testing of security requirements like confidentiality, integrity, authentication, authorization, availability, and non–repudiation [6]. Security requirements can be classified into positive requirements and negative requirements [7]. Positive requirements define what a system should do, e.g., user accounts are disabled after three unsuccessful login attempts or network traffic must be encrypted. Negative requirements stress what a system should not do, e.g., unauthorized users should not be able to access data. Thus, positive requirements define the expected functionality of a system, and negative requirements define possible vulnerabilities. On the system level, positive requirements can be modeled as use cases and negative requirements as misuse cases [7]. Based on the classification into positive and negative security requirements, two types of security testing can be distinguished, functional security testing and security vulnerability testing [8]. Model-based testing is a variant of testing that relies on models to generate tests, to execute tests or to evaluate their results. Model-based testing offers big potential for automation, change management and abstraction in the testing process and is also suitable for security testing. In [9] we classify model-based security testing approaches according to the two dimensions, namely the degree of automatic test generation and consideration of risks into the test models by providing examples for each category. Based on the different perspectives used in securing a system, Schieferdecker et al. [6] distinguish several types of input models for test generation, e.g., (1) architectural and functional models, (2) threat, fault and risk models, and (3) weakness and vulnerability models.

A. Security Testing Techniques

As mentioned before, for testing positive security requirements established functional testing techniques can be applied. Michael and Radosevich [10] provide a detailed listing of functional testing techniques like equivalence testing or decision tables. Testing positive security requirements can be performed by the traditional test organization [11].

As the number of vulnerabilities is not enumerable, also the systematic testing of vulnerabilities is hard to do and typically requires more specific expertise than for functional requirements testing. Today's security vulnerability testing lacks approaches for systematic design, execution and evaluation of tests. In practice, security vulnerability testing uses the simulation of attacks as performed by hackers which is called penetration testing and attempts to compromise the security of a system [12], [13] by playing the role of a hacker trying to attack the system and exploit its vulnerabilities. Besides penetration testing, another well-known approach to vulnerability testing is fuzzing [14], [15]. Fuzzers have initially been developed for testing protocol implementations on possible security flaws due to improper processing of malicious input. Recently, the idea of fuzzing has been combined with the concept of model-based testing which allows for systematic and automated testing of software applications [16]. Most fuzzers are very specific and optimized for a certain type of input.

A promising way to manage the infinite number of vulnerabilities with limited test resources is the consideration of risks for testing purposes [17]–[20]. Risk-based testing is a type of software testing that considers risks of the software product as the guiding factor to solve decision problems in the design, selection and prioritization of test cases [17]. Based on a threat model or based on misuse cases, vulnerabilities can be identified and prioritized relying on a risk analysis as basis for further test activities. Model-based testing techniques provide additional support to lower the required level of expertise needed for security testing [9] due to several reasons. First, a security test model improves the test designer's understanding of the software's security aspects which results in more efficient test cases. By using models, the level of abstraction is raised which enables more people to design tests. Additionally, the model can be employed to automatically generate test cases. Finally, security models are often created in conjunction with a risk analysis. This risk information can on the one hand be used to derive test cases, and on the other hand for prioritizing test execution. So far, model-based security testing has mainly been applied for testing access control policies [21]–[25], identification of vulnerabilities [26]–[28] and security test generation from threat models [29].

B. Testing Service–centric Systems

Salva et al. [30] present an approach tailored on testing the security of ScS. However, by limiting their approach onto Symbolic Transition Systems (STS), representing a service's internal states, their technique cannot be used for other than testing security of stateful web services. An auspicious

approach is presented by Vieira, Anuntes et al. [31]–[33], using vulnerability scanning as a springboard for security testing of web services. However, as their methodology now only supports the detection and disposal of SQL and XPath vulnerabilities, from our point of view, it fails to test a service's security in a sound manner.

Other relevant approaches are presented by Weider et al. [34], Felderer et al. [35] and by Jurjens [26]. In the first quoted paper, an approach to sound security testing of web services, based on fault models, is presented by Weider et al. Nevertheless, their work, when writing our paper, neither considers test identification, test generation nor test automation at all. The work presented by Felderer et al. attempts to test the security of a ScS based on predefined security requirements, contained in the model of the system under test (SUT). Yet, this approach only incorporates manually defined positive requirements and does not incorporate misuse cases derived from risk analysis, thereby also failing to abstract security concepts. The idea laid before by Jurjens employs fault injection on the model level to properly test security requirements of software. However, the presented work has not that much in common with classical model–driven testing techniques (e.g., test code generation from models), it more or less focuses on checking the soundness and completeness of the model of the SUT under various mutations, caused by fault injection. Cloud computing allows the deployment and use of heterogeneous infrastructures, platforms, and software services.

Testing ScS in clouds is therefore manifold as shown by Riungu et al. [36] who highlight several aspects of testing in the cloud: Testing can be provided as a service to cloud consumers which is typically called Testing-as-a-Service (TaaS), TestSupport-as-a-Service (TSaaS) or SoftwareTesting-as-a-Service (STaaS). The cloud provider can test the cloud deployment itself when the system is developed or maintained. Also the customer can use cloud services on the infrastructure, platform or service level, and test their process integration. Riungu et al. highlight security testing in the cloud as an important research issue as it is challenging but also of high importance for users. However, according to a recent study by Chana et al. [37], at the moment not much research is performed in this area. One of the three mentioned approaches, by Banzai et al. [38], focuses on fault injection testing, the other two by Gu et al. [39] and Zech [40] focus on privacy aware testing.

The approaches presented by King et al. [41], Ciortea et al. [42], and Yu et al. [43] follow the idea of providing test as a service to cloud consumers for enhanced application testing. Nevertheless, as their approach attempts testing from inside the cloud environment, it is not applicable for security testing, also focusing on the examination of risks posed from outside the cloud. An approach to cloud infrastructure testing is proposed by Rings et al. [44], focusing on cloud interoperability testing. Yet, this approach does not take security aspects into consideration.

III. Issues With Existing Security Testing Approaches

In the following, we identify, based on Section II, the shortcomings of existing security testing approaches. The discussed shortcomings help in identify emerging requirements (see Section IV).

A. Testing Perspective

Talking about the testing perspective, one can choose between two sides, viz. *positive* and *negative* testing [7]. What comes close to the idea of negative testing are approaches that focus on vulnerability testing, namely penetration testing and fuzzing. In the case of fuzzing, the main drawback is that fuzzers are not general purpose. Yet, this does not mean that fuzzers are domain–specific, it eagerly means, that fuzzers are only tailored to exactly one specific protocol or component, but fail to cover a full system by addressing all its different components. If penetration testing, the problem is, that on the one side, doing it requires a high expertise in IT security [12], and on the other side, often fails to evolve with new hacking techniques [45].

Besides penetration testing and fuzzing, most other existing security testing approaches mainly follow the idea of positive testing [26], [35], to show the validity of the system. Thereby they fail to cover any negative requirement, which are by default existent and implemented in any software system and which should be hunted for.

An auspicious idea for performing the switch to a negative perspective is incorporating the aspect of risks in the testing process. Doing so allows to ignore any positive requirement but instead focusing on the negative ones by, e.g., using a risk analysis as a base for test design [46]. However, to do so in a valuable manner, requires a high degree of formalization by models, which, unfortunately is not yet supported by most security testing approaches (see Section III-B).

B. Testing Approaches and Techniques

Taking a closer look at existing approaches (see Section II) reveals that existing security testing approaches basically miss to provide a concise, straight–forward technique. For example, when considering penetration testing, it merely is nothing more than just executing scripts (e.g., test cases), which contain *coded* behavior of an attacker [45]. Yet, just executing those scripts against a SUT again and again does not suffice, as attackers evolve by finding new exploits, clearly not covered by outdated test cases (see Section III-A). When using a fuzzer, it also does not know anything about any specification of the system, probably it has restricted knowledge on the tested component by its interface, yet, after that it *dumbly* generates or alters data and feeds it as an input to the SUT. Hence, like penetration testing, fuzzing also lacks a concise approach on how to test a system in a sound manner. Yet, one could, e.g., when penetration testing is applied enhance the test cases by considering new vulnerabilities, or if using a fuzzer, extend its data generation or manipulation capabilities, however, in the end, it does not change the situation. Doing security testing in

such a manner probably reveals errors, yet, it does not reveal reasons for identified errors [45].

Another problem, yet not specifically coming along with any approach, but merely a common problem in testing of ScS is the lack of having access to an implementation. For example, a service consumer only has access to an interface instead of an implementation and hence, no behavioral description of the SUT. Thus, testing is restricted to be solely done in a black–box manner. However, to do valuable security testing, a behavioral description is beneficial, as it, on the one side, allows to develop (either manually or automated) more effective test cases and, on the other side, enhances identifying and locating vulnerabilities.

A promising approach is the application of model–based testing techniques for security testing, however, so far the hidden potential by abstraction and automation of test case generation as well as execution has not been unlocked completely (see Section II).

C. Testing Complexity

Generally, security testing is considered as a tough discipline [45]. This is due to that for valuable security testing, one needs a deep knowledge and understanding of how programs work and, even more important, how to exploit the used language or implemented concepts (e.g., buffer overflows, syntax–based exploits or circumvention of access control mechanisms). In penetration testing, for example, the necessary test cases need to be scripted by a security or vulnerability expert to be valuable. Yet, this is not the only intricacy of security testing, another one arises from the specific business domain, where the software is intended to be used. In this latter case, it is the duty of a testing approach to provide the tester with the necessary formal constructs (i.e.: linguistic abstractions[1]) to be able to let domain–specific peculiarities be taken into account in the overall testing process. Yet, it should be kept in mind, that such peculiarities should not be considered as a requirement, but more or less an additional fact on the SUT[2].

Again, applying model–based testing techniques is a favorable choice as doing so provides a convenient level of abstraction, necessary for formulating domain–specific peculiarities and lower the needed level of expertise and complexity of security testing (see Section II).

D. Focus of Testing

The focus of testing should not be confused with its perspective. The former states what kind of vulnerabilities a testing approach is capable to identify, whereas the latter states whether an approach is based on positive or negative requirements (see Section III-A). If penetration testing, the focus is quite vast, as a penetration test case can cover nearly any negative requirement, somehow testable with a scripted test case. Basically, the focus of a fuzzer is restricted to the application it has been developed for, yet it seldom can be used for anything else due to its tailored implementation.

If taking a closer look at security testing approaches for ScS, the variety is restricted. As presented in Section II-B, existing approaches mainly focus on detecting SQL and XPath injections [31]–[33], validating positive requirements [35], can only be used for model–checking the SUT [26]–[28] or solely allows testing access control policies [21]–[25].

However, the situation becomes even worse if searching for cloud–specific security testing approaches. Despite some small effort, existing approaches fail to focus on security testing a cloud itself but instead rather focus on how to use cloud deployments as testing services.

IV. Emerging requirements to Security Testing

Based on the issues discussed in the previous Section, we now discuss emerging requirements to be met by future security testing approaches.

A. R1: A Change in the Testing Perspective

As already stated in Section III-A, most of the existing security testing approaches, except penetration testing and fuzzing, focus on showing that a system behaves proper according to some functional security requirement. However, as criticized earlier, doing so does not suffice in showing that a system is secure (the same also applies to model–checking based techniques, as they also ignore the case, where a system is transformed in an invalid state due to manual intervention by a malicious agent). The simple reason for that only checking for positive requirements does not suffice is due to that a malicious agent assumes a system being erroneous and hence, ready to be exploited. For example, consider the positive requirement that a potential user may only log in to a system using valid credentials. Yet, at first sight this requirement seems legitimate, however, it fails to cover the cases where a malicious agent circumvents the whole login mechanism. A similar example can be found regarding SQL injections, e.g., a requirement stating, that any input must be sanitized before being processed. Again, the requirement is good, but it also fails to state how and where input needs to be sanitized, whether on client or on server side. If input sanitation is only done on client side, the requirement is useless, as again, a malicious user can circumvent the input validation by directly communicating to a server, thereby effectively avoid any input to be sanitized.

For future security testing approaches, to be valuable, they need to focus on exactly such *negative* requirements, as they state, what hackers do and how they are successful in breaking into systems. Yet, we do not propagate to completely ignore positive requirements, however, for security testing to be sound

[1]Linguistic abstractions are a concept from Language–oriented programming which propagate the declaration of new language concepts for abstracting a domain rather then reusing existing general–purpose concepts for abstraction.

[2]E.g., in cloud computing, one should have the chance to not only define at which layer a service operates, but also what kind of tasks it performs, e.g., data manipulation or resource provisioning; on the other side, the testing approach should be capable of dealing with such additional information and consider it during test case generation or test execution.

and complete, the consideration of negative requirements is vital.

B. R2: Abstraction, Automation and Evolution

A common characteristic of ScS and cloud deployments is their relatively high change frequency, e.g., new services may be added, existing ones may be altered or deleted, likely on an everyday basis. For a security testing approach to be valuable it needs to track such changes and react on them by considering them in future test runs. Existing approaches like penetration testing and fuzzing generally do not take a formal description of the SUT into account, resulting in poor test cases, not capable of identifying potential new vulnerabilities due to such changes in the system [45]. What also results from the lack of such a formal description is insufficient support for automation. For sure, one can automate the execution of test cases by using some scripted test scenario, yet, this is no solution. What is needed are testing approaches, which heavily rely on a formal description of the system, later used for the automated generation and execution of test cases. In an ideal scenario, the formal description of the system can further be used for automatically generating test data.

In recent years, model–based testing has grown to a mature testing technique, however, up to now, its potential has not been unlocked for security testing. Yet, it would solve the problem of abstracting a system description, automated test case generation and execution. Moreover, it enables coping with the evolution of the SUT and its related test cases offhand [47], and thereby addresses the problem of outdated test cases (see Section III-B). Additionally, it builds on a formal description of the SUT which allows to react on changes in the system for future testing. Also, resulting from its powerful mechanisms of abstraction, it would reduce the need for security or test experts for security testing a system. Applying model based techniques would also overcome the problem of only being able to do testing in a black–box manner (see Section III-B). Using models allows to integrate system behavior into the formal description of a SUT, thereby allowing for more effective security testing. Hence, for future testing approaches to be applicable in a convenient manner, they should build upon model–based testing techniques.

A need for abstraction not only arises if talking about the description of the SUT, such a need for abstraction is also necessary to deal with vulnerabilities in a meaningful manner. Currently, penetration testers and security experts rely on descriptions as contained, e.g., in the Common Weaknesses and Vulnerabilities (CWE) database [48]. However, descriptions contained in the databases like CWE are kept in an informal, textual manner. Besides leading to discrepancies in how such vulnerabilities are described, it can also lead to problems if trying to identify vulnerabilities due to a lack of understanding. Using formalization techniques to describe attacks in an abstract, machine and human readable way, would result in powerful security testing approaches, capable of detecting vulnerabilities already at a very early stage inside a formal description of the SUT, i.e., its model.

C. R3: Domain Orientation and Separation

Due to their flexibility and dynamics, ScS and cloud environments are used for a wide variety of purposes, e.g., e–Health, Business–2–Business (B2B), Enterprise Resource Planning (ERP), and the like. In the case of security testing of such a system, it is vital to incorporate the different domain peculiarities (e.g., has the system to deal with high–sensitive private data, is it responsible for performing financial calculations or predictions, and the like) during test design and execution. Unfortunately, existing security testing approaches completely fail to incorporate such domain peculiarities. Yet, such peculiarities are part of a detailed description of the concrete usage scenario of the SUT, and hence, should be a main driving force during test case design.

Yet, just intensifying the focus towards the domain does not suffice. What is also needed are modeling language concepts, allowing to transparently map domain requirements and characteristics into the affected models (e.g., a model of the SUT or a test requirements model). Hence, we again motivate the separation of the business and technical domain as already stated in Section I. Although doing so results in a larger set of modeling concepts and hence, a bulkier library of modeling languages, it is preferable due to two main reasons. First, a zoo of different domain–specific languages allows to enhance the overall testing process (e.g., detailed system and test model, domain tailored requirements model, and the like) resulting in more meaningful testing. And second, although at a first sight, a bulky set of languages may seem like an overhead, yet, by keeping in mind, that one only uses the necessary languages this overhead immediately vanishes (i.e., one can completely ignore the unused languages and their concepts).

Again, model–based testing techniques would provide the necessary mechanisms of formalization to let such domain peculiarities advect into the whole testing process through linguistic abstractions. Future security testing approaches need to meet this requirement for being valuable by stating something on the state of security of the SUT.

V. A Novel Approach to Security Testing

We now present our novel method overcoming the issues discussed in Section III and at the same time addressing the requirements, elicited in the previous Section.

A. Domain–specific Languages

As outlined before, security testing requires much domain knowledge as well as security expertise from the tester. The specifics of low-level protocols need to be known, but also knowledge about the particular domain of the SUT is required. Hence, we distinguish between two different domain types in our work. The technical domain, i.e., the technologies which the SUT exposes, as well as the business domain, such as e–Health or ERP.

Since the knowledge on the technical and business domain of an application is important for its security testing, applying abstractions that make it possible facilitating the testing of

these specifics can be of great advantage regarding development time and costs. In addition, the abstractions could allow non–expert security testers to do this crucial task.

As mentioned in Section II, model–based testing has been applied to functional security testing [9]. Additionally, domain–specific languages (DSLs) have been developed to ease the development of test cases at the code level [49]. However, the advantages of DSLs for the technical domain have not yet been exploited for security testing.

Fowler [50] defines a DSL as being, in essence, a bespoke computer programming language, which exposes fluency, has limited expressiveness and, most importantly, has a domain focus. The last two aspects are the most important for the use in conjunction with security testing. Limited expressiveness constrains the amount of mistakes that can be made when creating security tests. A domain focus allows to raise the level of abstraction necessary for developing test cases and thus facilitates the development process.

Opposed to general purpose modeling languages such as UML, the usage of (textual) DSLs is much more concise and usually less work. The reason is that for the creation of profile–based DSLs in UML, the large UML meta–model needs to be restricted, which leads to many constraints that in turn are hard to maintain. DSLs on the other hand are built for a specific purpose and only contain the needed model elements. However, typical DSL creation frameworks, such as xText [51], share the problem that language composition is a difficult process because of the parser–based nature of their implementation. Language composition is important to be able to combine specialized languages, e.g., combining a language for web–application security testing and a language for the corresponding business domain, that both rely on the foundation of a generic security testing language.

A very promising trend for security testing with DSLs is the advent of practically usable language workbenches such as Jetbrains Meta Programming System (MPS) [52]. Such workbenches allow for domain-specific language creation as well as composition and extension of existing languages. In the case of security testing, this concept allows for the mentioned example of language composition for an organization-specific security testing use case.

B. Building upon Modularity and Composition

Although the concepts of modularity and composition as described by Voelter [53] are no novelty, their usage in domain–specific language extension is. Besides allowing to extend existing languages with domain concepts (e.g., embedding SQL into Java), the techniques as described by Voelter [53] also allow to create modular languages, intended for domain modeling from scratch. Combined with language workbenches [53], language–oriented programming (LOP) [54] hence provides powerful mechanisms to address two (*R2* and *R3*) of the three requirements from Section IV by leveraging abstraction and enabling convenient domain customization by modularity and composition.

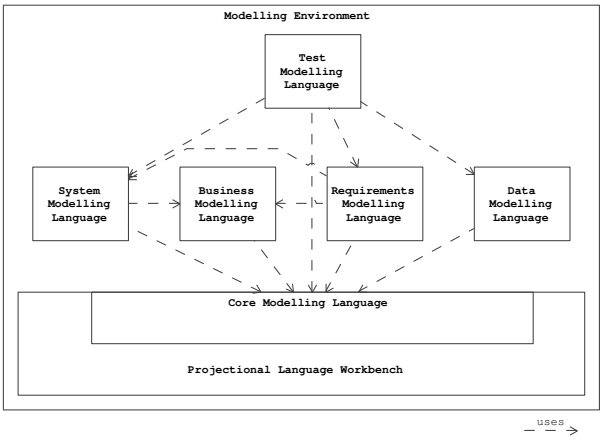

Fig. 1. Applying Language–oriented Programming on Security Testing

Figure 1 shows an exemplary architectural overview of a modeling environment for security testing following the concepts of LOP. As mentioned by Voelter [53], at the bottom of each language workbench a core language needs to be provided, that declares the base concepts for language extension. In the sample architecture from Figure 1, this core language is the CML (Core Modeling Language), provided by a language workbench. This CML in turn is then used to define the domain–specific language modules, that are necessary for the concrete use case of the modeling environment (as mentioned earlier, in this example the intended use case is security testing, hence the choice of the specific language examples). The remaining set of languages (SML, BML, DML, RML and TML) are an immediate result of the concept of domain orientation and separation (*R3*).

In the following, the various languages and their purpose are discussed in more detail:

- **Core Modeling Language** (CML) As already stated, the CML represents the core language of the language workbench, allowing to define language extensions, as well as creating languages from scratch. Apart from the remaining languages, the CML can be seen as a general purpose language more than a domain–specific one, as it must provide concepts for realizing languages for any purpose.
- **System Modeling Language** (SML) As its name already indicates, the SML is used for describing the SUT. Here, the kind of SUT can range from an embedded system to any kind of service centric or object–oriented system. Language modularity allows to provide DSLs for any system type. Besides, as languages in this case are defined from scratch, such a SML allows to directly map system specific requirements of e.g., technical nature.
- **Requirements Modeling Language** (RML) The RML supports the tester by enabling high–level and domain–specific definition of test requirements. Thereby the kind of test requirement a tester is allowed to define, e.g.,

a functional security requirement or a negative one, is restricted only by the RML itself. In security testing, such a language for defining requirements is crucial, as it directly affects the quality of the testing process by the meaningfulness of outcomes (e.g., a tailored requirements language allows to specify requirements, that can be mapped transparently to either the business or the technical domain, and hence, allows for more meaningful testing). The RML reuses the SML and BML (see below) to achieve modeling of domain–specific requirements.

- **Business Modeling Language** (BML) The BML helps integrating requirements and characteristics of the distinct business domain into the whole testing process. It is a modular language without any root concept (i.e., a concept of the BML can only be used as a child element in another language) and is used by the SML and the RML. This is possible due to language composition, allowing to seamlessly combine concepts of different modular languages. For example, one could, as mentioned earlier, define an SML for object oriented systems and just reuse a BML for e–Health (assuming it exists) without (re)defining the business specific language concepts.
- **Data Modeling Language** (DML) The purpose of the DML is to provide a tester with an easy and convenient way of defining test data. Thereby the DML may either allow to define the structure of possible test data (e.g., a protocol specification) further on used by a test data generator. On the other side, the DML could also provide concepts for e.g., web service messages, and, in combination with a projectional editor, offer a convenient way for defining test data in the form of custom web service messages by successfully hiding any low level details of the underlying message structure.
- **Test Modeling Language** (TML) The TML provides the tester with the necessary concepts for developing a test model (either manually or automatically by using some model transformation). Besides providing basic concepts necessary for a testing language, e.g., for defining assertions, the TML integrates the SML, BML and DML. Doing so is necessary to equip the TML with concepts for defining test data (DML), the test objective (RML) and any SUT related test aspects (SML). Hence, the TML is a composed language, consisting of the SML, DML and RML, with a few custom concepts for test modeling. Such kind of language development again mirrors the concept of language customization, allowing to easily develop new languages from existing ones by reusing or inheriting their concepts.

Especially the SML, the BML and the RML explain this concept of domain orientation and separation (*R3*), as their purpose is exactly to enhance the testing process by allowing transparent mapping of domain concepts (e.g., technical concepts in the SML, business concepts in the BML) as well as keeping the respective language small–sized, but yet domain–specific.

Not mentioned up to now and also not part of Figure 1 are any runtime and execution related concepts. In essence, the idea is to provide a generator for each of the mentioned languages (whereas the generator for the CML is provided by the language workbench itself), allowing to translate the language into any target language (i.e., another DSL or an executable language like Java) [53]. Depending on the underlying language workbench, transformations according to the taxonomy of Mens [55] are possible. As of this and due to that the concrete usage of generators depends on the specific use case of the language, we postpone the discussion on generation of target languages and any other runtime and execution related concepts to the next section.

Examples of language definitions and the like, will be given in the subsequent section. The purpose of this section was to give an idea of how LOP works, the discussed example shall not act as a pattern nor as a guideline of how to implement a security testing method, it is a plain example used to clarify the concepts of LOP. In the following we discuss a security testing method, building upon LOP and addressing the issues discussed in Section IV.

C. A Concept for Language–oriented Security Testing

As stated before, this section is dedicated to presenting our language–oriented security testing method. Figure 2 shows a sketch of our method, which we developed according to the requirements outlined in Section IV.

The upper left corner of Figure 2 shows an example of a SUT model developed using an instance of an SML as described earlier. As discussed, the purpose of the SML is allowing a tester or system expert to develop a domain–specific system model of a SUT. In our case, the model describes an ScS, aggregating a service called `SomeService` and defining another service (`SampleService`) by itself, which in turn offers some operations (`sampleOp` and `anotherOp`). The system model also contains necessary technical aspects, e.g., the endpoint or the underlying service engine (concepts of the SML; concepts for modeling service, operations, etc. are also part of the SML) as well as business aspects, e.g., the business domain where the service operates, in this case it's data related (a concept of the BML).

Next, the system model is subjected to a risk analysis. This stems from the change of perspective towards negative testing (*R2*). Performing a risk analysis allows to ignore any positive requirements, thereby identifying existing flaws in the system model, which later are considered as negative requirements. For that such a risk analysis works fully automated, there is a need for storing existing knowledge in a machine processable manner. For that we use logic programming [56], providing the necessary formalisms using facts to store such information in a deductive database system [57]. Further, using rules such knowledge can be matched against an input model for identify potential flaws (for a detailed discussion see our related work [46]). At first sight, this may contradict our idea of domain orientation and separation. However, in this concrete case the logic programming language (e.g., Datalog) can be

seen as metalanguage, providing facilities to define domain–specific languages for knowledge storage and retrieval. Hence, we fully comply to our idea and the facts and rules we developed using notions of logic programming can be seen as concepts of the RML, used to define test requirements[3]. Besides, integrating such knowledge in the testing process in a formalized manner tremendously eases the necessary level of expertise for security testing.

The upper right corner shows another model, also part of our method, in this case the test model based on the TML. For retrieving this model, the results of the risk analysis are parsed and transformed into the test model by a text–2–model transformation. The test model is a two–column tabular model, defining both, risks, i.e.: the objective of a test case (left column), and corresponding test cases (right column). In the previous section we stated that the TML uses the SML, which is not quite clear from Figure 1, however, the TML needs to know about the SML and its concepts for being able to link elements of the system model into the test model (e.g., associate a test case with an operation from the SUT).

So far, we have only talked about the language and modeling aspects of our method (and also for LOP), which is why we now attribute a few lines to the generators and runtime related concepts. As already stated, each of the languages comes along with its own generator, allowing to transform a language into any target language (modeling or execution). Taking a closer look at Figure 2 reveals that we do not use a dedicated generator for each language. In fact, the SML has an associated generator and the RML for the risk analysis. The generators of the SML are on the one side, used to generate executable tests (*System Model Test Suite Generator*) into Scala code, whereas on the other side, the *Problem Generator* translates the system model into a format, that is processable by the risk analysis (a transformation into facts). The last generator depicted in Figure 2 is the *Model Translator* allowing to transform the set of facts (which could be considered as a declarative risk model), retrieved by the risk analysis, into the test model. The TML in our case has no generator attached, as we directly execute the model, resulting in improved traceability and easy model feedback (by the *TestEngine* component, providing both, the test data *Fuzzer*, using the DML and the *Model Interpreter*). One could also attach a generator to the TML and generate some executable code in form of a test suite, to be executed against the SUT. After executing the test model, the *TestEngine* generates a test log (see lower-right corner of Figure 2).

Finally, by being completely model–based, our method also addresses the problems of abstraction, automation and evolution (*R1*). Besides, due to only using DSLs our method remains light-weight and extensible.

In what our approach mainly differs from existing work is that we choose to write own DSLs which meet necessary requirements *prior* to do testing. In this, we enable transparent mapping of domain concepts into the testing process by linguistic abstractions. Using classic approaches, such abstractions are only possible by reusing existing general–purpose concepts which do not support transparent mapping of domain concepts and thus, are affected with information loss.

[3]Our method only considers negative requirements, which are intrinsically contained in our RML, as it contains attack descriptions, which can be seen as a negative requirement, as they describe certain functionality and behavior which should not be allowed by the system.

VI. Conclusion and Future Work

In our paper we have investigated existing approaches to security testing, especially for ScS, and identified drawbacks, which make security testing a difficult task today:

- *Testing Perspective:* Just validating positive test requirements does not suffice anymore. For near–holistic security testing, negative requirements, resulting from a risk analysis, are vital, as they focus on what is possible in a software system, but yet, should be prohibited by all means.
- *Testing Approaches and Techniques:* Most of the existing approaches fail to provide high–level testing techniques (e.g., model–based testing techniques) and basically rely on scripting. For future test approaches to be usable, they need to provide mechanisms to easily adapt the testing process due to e.g., changing system or business requirements, changes in the system itself, and the like.
- *Testing Complexity:* At the time of this writing, security testing is a task that requires a high degree of security related knowledge. Hence, future security testing approaches must enable less trained security testers to do meaningful tests.
- *Focus of Testing:* Current security testing approaches mostly focus on specific vulnerabilities. This is not what is desirable. Future security testing approaches must focus on a wide range of attacks. This is motivated by the circumstance, that software systems are not subject to one kind of attack from one type of adversaries, but different kinds of attacks from various adversaries with varying intentions (e.g., a bank is only subjected to robbery, whereas an e–banking system can be subjected to many different attacks, e.g., Denial-of-Service, wire fraud, tampering with data, and the like).

Based on the identified limitations of Section III, we have elicited requirements for future security testing approaches, which should be followed, such that future software systems remain secure against many kinds of threats:

- *R1: Change in Testing Perspective* As already stated, simple positive testing does not suffice in a security context. Testers must focus on negative requirements, to assure a securely running system. After all, it is exactly these negative requirements a hacker is hunting for.
- *R2: Abstraction, Automation and Evolution* To be able to react on changes (e.g., technical or business related requirements, architectural changes), future approaches must build upon a formal method, providing the necessary mechanisms to identify such changes and adapt the testing process accordingly. This can easily be achieved by

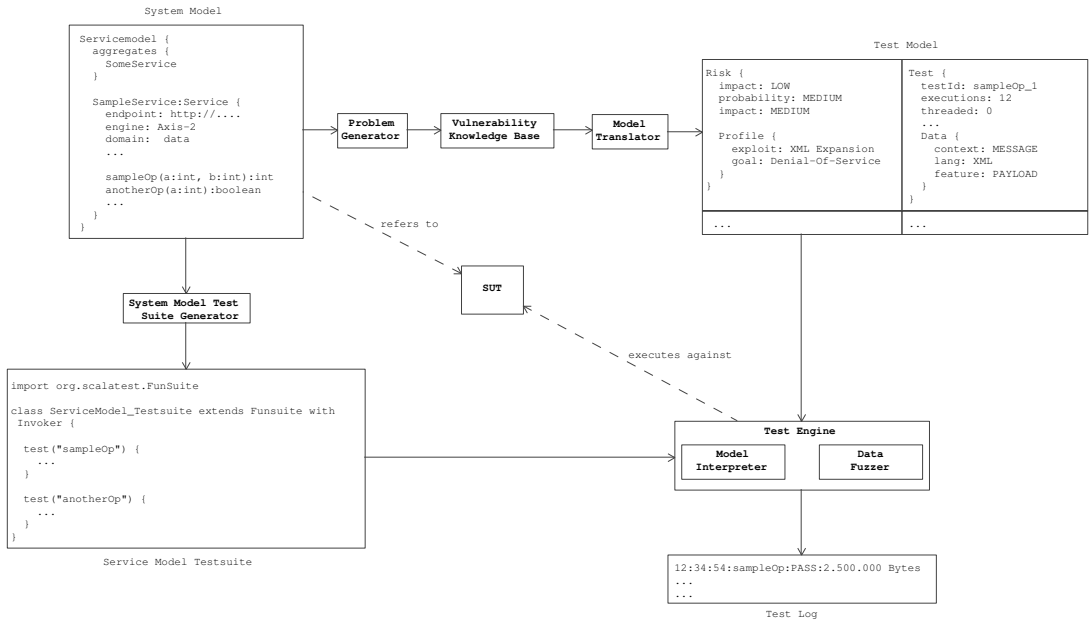

Fig. 2. Sketch of our Language–oriented Security Testing Method

building upon model–based techniques. Besides, building upon these techniques lowers the required level of expertise for security testing.

- *R3: Domain Orientation and Separation* By far the most important issue of our paper is to motivate an intensified domain focus in testing. This intensification comprises both, more domain orientation, but also, at the same time, more domain separation. This intensified domain focus allows to do testing in a highly domain–oriented manner (domain orientation) but at the same time, keeping the testing as diverse as necessary by addressing different technical and business aspects (domain separation).

For meeting these requirements (*R1, R2 and R3*), DSLs offer just the right features. First, DSLs allow to express any ever so complex business and technological requirement due to their transparent alignment to the respective problem domain. Second, DSLs can easily be reused and combined by inheritance, extension and composition, hence increasing the expressivity and usability regarding the used problem domain. Finally, using the concepts of LOP allowed us to implement a prototypical reference framework building upon Jetbrains MPS. Regarding future work, it will mainly be focused on our prototype, where the most relevant topics are:

- Showing the usability and effectiveness of our method and framework via case studies.
- Improving the system and test modeling languages by both, enhancing existing and adding new, necessary language features.
- Improving the set of attached language transformation generators and enhancing them with more flexibility by generable target languages (both, modeling and executable languages).

ACKNOWLEDGMENTS

This research was partially funded by the research projects MATE (FWF P17380), and QE LaB—Living Models for Open Systems (FFG 822740).

REFERENCES

[1] S. Kendall, J. Waldo, A. Wollrath, and G. Wyant, "A note on distributed computing," 1994.
[2] N. Josuttis, *SOA in practice*, 1st ed. O'Reilly, 2007.
[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.
[4] R. Wojtczuk and J. Rutkowska, "Following the white rabbit: Software attacks against intel vt-d technology," *ITL: http://www. invisiblethingslab. com/resources/2011/Software% 20Attacks% 20on% 20Intel% 20VT-d. pdf*, 2011.
[5] H. Thompson, "Why security testing is hard," *Security & Privacy, IEEE*, vol. 1, no. 4, pp. 83–86, 2003.
[6] I. Schieferdecker, J. Grossmann, and M. Schneider, "Model-Based Security Testing," *arXiv preprint arXiv:1202.6118*, 2012.
[7] I. Alexander, "Misuse cases: Use cases with hostile intent," *Software, IEEE*, vol. 20, no. 1, pp. 58–66, 2003.
[8] G. Tian-yang, S. Yin-sheng, and F. You-yuan, "Research on Software Security Testing," *World Academy of Science, Engineering and Technology Issure*, vol. 69, pp. 647–651, 2010.
[9] M. Felderer, B. Agreiter, P. Zech, and R. Breu, "A classification for model-based security testing," in *VALID 2011, The Third International Conference on Advances in System Testing and Validation Lifecycle*, 2011, pp. 109–114.
[10] C. Michael and W. Radosevich, "Risk-based and functional security testing," *Build Security In*, 2005.
[11] B. Potter and G. McGraw, "Software security testing," *Security & Privacy, IEEE*, vol. 2, no. 5, pp. 81–85, 2004.
[12] M. Bishop, "About penetration testing," *Security & Privacy, IEEE*, vol. 5, no. 6, pp. 84–87, 2007.
[13] B. Arkin, S. Stender, and G. McGraw, "Software penetration testing," *Security & Privacy, IEEE*, vol. 3, no. 1, pp. 84–87, 2005.

[14] B. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[15] A. Takanen, J. DeMott, and C. Miller, *Fuzzing for software security testing and quality assurance*. Artech House on Demand, 2008.

[16] Y. Yang, H. Zhang, M. Pan, J. Yang, F. He, and Z. Li, "A Model-Based Fuzz Framework to the Security Testing of TCG Software Stack Implementations," in *Multimedia Information Networking and Security, 2009. MINES'09. International Conference on*, vol. 1. IEEE, 2009, pp. 149–152.

[17] S. Amland, "Risk-based testing:: Risk analysis fundamentals and metrics for software testing including a financial application case study," *Journal of Systems and Software*, vol. 53, no. 3, pp. 287–295, 2000.

[18] C. Wysopal, L. Nelson, E. Dustin, and D. Dai Zovi, *The Art of Software Security Testing: Identifying Software Security Flaws*. Addison-Wesley Professional, 2006.

[19] D. Firesmith, "Security use cases," *Journal of object technology*, vol. 2, no. 3, 2003.

[20] P. Gerrard and N. Thompson, *Risk-based e-business testing*. Artech House Publishers, 2002.

[21] D. Basin, J. Doser, and T. Lodderstedt, "Model driven security: From UML models to access control infrastructures," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 15, no. 1, pp. 39–91, 2006.

[22] J. Julliand, P. Masson, and R. Tissot, "Generating security tests in addition to functional tests," in *Proceedings of the 3rd international workshop on Automation of software test*. ACM, 2008, pp. 41–44.

[23] P. Masson, M. Potet, J. Julliand, R. Tissot, G. Debois, B. Legeard, B. Chetali, F. Bouquet, E. Jaffuel, L. Van Aertrick *et al.*, "An access control model based testing approach for smart card applications: Results of the POSE project," *Journal of Information Assurance and Security*, vol. 5, no. 1, pp. 335–351, 2010.

[24] A. Pretschner, T. Mouelhi, and Y. Le Traon, "Model-based tests for access control policies," in *Software Testing, Verification, and Validation, 2008 1st International Conference on*. IEEE, 2008, pp. 338–347.

[25] Y. Traon, T. Mouelhi, and B. Baudry, "Testing security policies: going beyond functional testing," in *Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on*. IEEE, 2007, pp. 93–102.

[26] J. Jürjens, "Model-based security testing using UMLsec: A case study," *Electronic Notes in Theoretical Computer Science*, vol. 220, no. 1, pp. 93–104, 2008.

[27] J. Jürjens and G. Wimmel, "Specification-based testing of firewalls," in *Perspectives of System Informatics*. Springer, 2001, pp. 308–316.

[28] G. Wimmel and J. Jürjens, "Specification-based test generation for security-critical systems using mutations," *Formal Methods and Software Engineering*, pp. 471–482, 2002.

[29] D. Xu, M. Tu, M. Sanford, L. Thomas, D. Woodraska, and W. Xu, "Automated security test generation with formal threat models," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 4, pp. 525–539, 2012.

[30] S. Salva, P. Laurencot, and I. Rabhi, "An Approach Dedicated for Web Service Security Testing," *Software Engineering Advances, International Conference on*, pp. 494–500, 2010.

[31] M. Vieira, N. Antunes, and H. Madeira, "Using web security scanners to detect vulnerabilities in web services," in *DSN*. IEEE, 2009, pp. 566–571.

[32] N. Antunes, N. Laranjeiro, M. Vieira, and H. Madeira, "Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services," in *IEEE SCC*. IEEE, 2009, pp. 260–267.

[33] N. Antunes and M. Vieira, "Detecting SQL Injection Vulnerabilities in Web Services," *Dependable Computing, Latin-American Symposium on*, pp. 17–24, 2009.

[34] W. D. Yu, P. Supthaweesuk, and D. Aravind, "Trustworthy web services based on testing," *Service-Oriented System Engineering, IEEE International Workshop on*, pp. 167–177, 2005.

[35] M. Felderer, B. Agreiter, and R. Breu, "Security Testing by Telling TestStories," in *Modellierung 2010*. Gesellschaft fuer Informatik, 2010, pp. 195–202.

[36] L. Riungu, O. Taipale, and K. Smolander, "Research issues for software testing in the cloud," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE, 2010, pp. 557–564.

[37] I. Chana, A. Rana *et al.*, "Empirical evaluation of cloud-based testing techniques: a systematic review," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 3, pp. 1–9, 2012.

[38] T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, T. Hanawa, and M. Sato, "D-cloud: Design of a software testing environment for reliable distributed systems using cloud computing technology," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE Computer Society, 2010, pp. 631–636.

[39] L. Gu and S. Cheung, "Constructing and testing privacy-aware services in a cloud computing environment: challenges and opportunities," in *Proceedings of the First Asia-Pacific Symposium on Internetware*. ACM, 2009, p. 2.

[40] P. Zech, "Risk-based security testing in cloud computing environments," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 411–414.

[41] T. King and A. Ganti, "Migrating autonomic self-testing to the cloud," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*. IEEE, 2010, pp. 438–443.

[42] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: A software testing service," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 5–10, 2010.

[43] L. Yu, W. Tsai, X. Chen, L. Liu, Y. Zhao, L. Tang, and W. Zhao, "Testing as a Service over Cloud," in *Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on*. IEEE, 2010, pp. 181–188.

[44] T. Rings, J. Grabowski, and S. Schulz, "On the standardization of a testing framework for application deployment on grid and cloud infrastructures," in *Advances in System Testing and Validation Lifecycle (VALID), 2010 Second International Conference on*. IEEE, 2010, pp. 99–107.

[45] H. Thompson, "Why security testing is hard," *Security Privacy, IEEE*, vol. 1, no. 4, pp. 83 – 86, july-aug. 2003.

[46] P. Zech, M. Felderer, and R. Breu, "Cloud risk analysis by textual models," in *Proceedings of the 1st International Workshop on Model-Driven Engineering for High Performance and CLoud computing*, ser. MDHPCL '12. New York, NY, USA: ACM, 2012, pp. 5:1–5:6.

[47] P. Zech, M. Felderer, P. Kalb, and R. Breu, "A generic platform for model-based regression testing," *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pp. 112–126, 2012.

[48] The MITRE Corporation. (2012, Dec.) Common Weaknesses and Vulnerabilities (CWE) Database. [Online]. Available: http://cwe.mitre.org

[49] M. Odersky, L. Spoon, and B. Venners, *Programming In Scala*. Artima, 2011.

[50] M. Fowler, *Domain-Specific Languages*. Addison-Wesley, 2010.

[51] Http://www.eclipse.org/Xtext/ (retrieved 22/12/12).

[52] M. Völter and E. Visser, "Language extension and composition with language workbenches," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, ser. SPLASH '10. New York, NY, USA: ACM, 2010, pp. 301–304. [Online]. Available: http://doi.acm.org/10.1145/1869542.1869623

[53] M. Voelter and K. Solomatov, "Language modularization and composition with projectional language workbenches illustrated with MPS," *Software Language Engineering, SLE*, 2010.

[54] M. Ward, "Language-oriented programming," *Software - Concepts and Tools*, vol. 15, no. 4, pp. 147–161, 1994.

[55] T. Mens and P. V. Gorp, "A Taxonomy of Model Transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, no. 0, pp. 125 – 142, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066106001435

[56] J. Lloyd, "Foundations of logic programming," 1987.

[57] J. Lloyd and R. Topor, "A basis for deductive database systems," *The Journal of Logic Programming*, vol. 2, no. 2, pp. 93–109, 1985.