

Choosing a Test Modeling Language: a Survey

Alan Hartman¹, Mika Katara², and Sergey Olvovsky¹

¹ IBM Haifa Labs, Mt. Carmel Campus
Haifa 31905, ISRAEL

{hartman,olvovsky}@il.ibm.com

² Tampere University of Technology, Institute of Software Systems
P.O.Box 553, FI-33101 Tampere, FINLAND
mika.katara@tut.fi

Abstract. Deployment of model-based testing involves many difficulties that have slowed down its industrial adoption. The leap from traditional scripted testing to model-based testing seems as hard as moving from manual to automatic test execution. Two key factors in the deployment are the language used to define the test models, and the language used for defining the test objectives. Based on our experience, we survey the different types of languages and sketch solutions based on different approaches, considering the testing organization, the system under test, etc. The types of languages we cover include among others domain-specific, test-specific as well as generic design languages. We note that there are no best practices, but provide general guidelines for various cases.

1 Introduction

A common view in the software development community is that there is a need to raise the level of abstraction from a code-centric view to a model-driven one. In Model-Driven Development (MDD), behavioral and structural models are treated as first class entities that are used to generate code automatically. However, there is an ongoing debate whether to use generic modeling languages (GMLs) or domain-specific (DSMLs) ones. GMLs, such as UML, are advocated by the OMG's Model-Driven Architecture (MDA) [1] initiative and others. DSMLs are gaining popularity through industrial success stories that report huge improvements in productivity [2], and the growing availability of tooling support for DSMLs.

Unfortunately, this debate has not yet reached the area of testing. Based on our experience from the model-based testing of industrial systems developed using code-centric practices, we think that the same debate needs to be extended to cover testing. Moreover, another equally important and related question is whether to model tests using a design language or a test-specific language. In this respect, the driving force of the MDD community seems to be the ability to generate applications, while test generation has been somewhat neglected. This is in contrast with the new developments in the more traditional software development process areas. In these, Test-Driven Development and other approaches deeply rooted in testing are seen to increase productivity.

From the viewpoint of black-box testing, the reasons for choosing between the different types of test modeling languages may differ significantly from those relevant to application and document generation. We believe the main differences to be:

1. Developers generally need to introduce details of the target platform and other implementation specifics – while testers may often disregard these aspects during modeling.
2. Test models may focus more on ”user experience” and system boundary – as opposed to the need to model precise internal system behavior by developers.
3. Developers usually have a better command of high level generic design languages, e.g. UML or SDL – whereas testers are better versed in the language of user experience and requirements documentation.

In model-based testing, tests are generated automatically from models that describe the behavior of the system under test (SUT) from a perspective of testing. The usual goal is to make the SUT fail the test in order to find defects.

Even if we build the most sophisticated model-based test automation system, there are still great difficulties in deploying the methodology and the tools (see, for instance [3, 4]). In fact, based on our experience, the leap from traditional scripted testing to model-based testing seems as hard as moving from manual to automatic test execution.

Two key factors in the deployment of model-based testing are the language used to define the test models, and the language used for defining the test objectives. The test models describe the behavior of the SUT, whereas the test objectives describe the behavior expected from the test generation software. Whether visual or textual, these languages are used for the interaction between the test automation system and its user. In many cases, we can translate existing test artifacts to test models ([5]), but even in this case, the aforementioned languages play an important intermediate role. There are also other languages used for defining test execution directives, or test set ups, but we consider these somewhat less important from the deployment point of view.

In this survey we discuss types of test modeling languages to be used in order to deploy model-based testing practices. The best answer is context-sensitive, but we try to develop some guidelines for choosing between different approaches. Since there is no clear division between the different language types, but rather a wide spectrum, we concentrate on design vs. test-specific, as well as generic vs. domain-specific languages.

The remainder of this paper is structured as follows. We briefly present the theory and practice of model-based testing in Section 2. In Section 3, we discuss the differences between using a design language vs. a testing language for test modeling. The discussion is continued in Section 4 concerning DSMLs and GMLs. Section 5 contains further considerations on the subject from other points of view. Finally, Section 6 presents some guidelines for decision making, and Section 7 draws some conclusions.

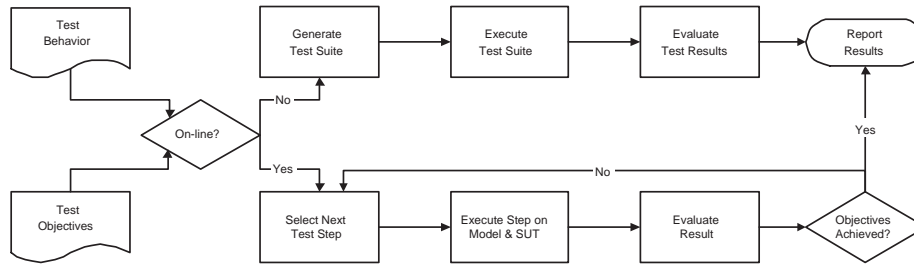


Fig. 1. MBT process: on-line vs. off-line testing.

2 Model-based testing

In this section we briefly review the theory and practice of model-based software testing, and its role in model-driven development.

2.1 Model-based vs. model-driven testing

In principal, any form of software testing can be seen as model based. The tester always forms a mental model of the system under test before engaging in activities such as test case design. The term Model Based Testing (MBT) is applicable when these mental models are documented and subsequently used to generate tests, or to evaluate their results. The term Model Driven Testing (MDT) has also been used frequently. Our interpretation of MDT is that it refers to a particular style of model-based testing, inspired by the OMG’s Model Driven Architecture (MDA) initiative. The underlying principles behind MDA (and hence MDT) are the separation between platform specific and platform independent aspects of the software, and the use of automated transformations to pass between different levels of abstraction. In model-driven testing, this means that the testing model is independent of the testing platform, and that a transformation is used to pass from platform independent test cases to platform specific ones.

2.2 On-line vs. off-line testing

Model-based testing involves the following key activities: modeling the behavior of the SUT, describing the test objectives, generating the test cases (using input from the behavioral model and test objectives), running the test cases on the SUT, and evaluating the test results to decide whether the testing is complete. These activities are depicted in the upper part of Figure 1. Alternatively, especially when testing reactive systems, we may choose to execute the test steps once they are generated as depicted in the lower part of Figure 1. In this so-called on-line testing approach, test cases and suites are implicit and testing is seen as a game [6] between the MBT tool and the SUT. To differentiate between these two alternatives, the former scheme is referred to as off-line testing.

The choice between these two approaches also affects the tool architecture. In the on-line case, the test generation software is connected to the SUT using an adapter that constantly translates inputs and outputs between the test automation system and the SUT. In the off-line case, the test cases (or suites) are generated first and, often after a translation or transformation, are executed at the SUT. An example of a tool implementing both approaches is Spec Explorer, which is currently used by several Microsoft product groups on a daily basis [7].

2.3 Behavioral modeling

This paper discusses the choice of appropriate languages to enable the aforementioned activities, and therefore focuses on languages that are used to describe SUT behavior and test objectives. Behavioral models can take many forms: diagrams (e.g., UML state diagrams), grammars, tables (e.g., decision tables), control flow graphs, and others. They have two main functions: to describe the set of stimuli that can be applied to the SUT in any given situation and to describe the possible responses to those stimuli. Models which do not describe the responses can still be useful for test generation. However, if an oracle is not supplied, the success or failure of a test case must be determined by other means.

Behavioral models for testing may be at different levels of abstraction. In the most extreme case, test engineers will re-implement the entire SUT independently in order to have an accurate test oracle. This is usually prohibitively expensive. On the other hand, when the SUT is developed using a purely model-driven approach, and the testing model simply reuses the implementation model, the only part of the system being tested is the model transformation and not the system itself. Model-based testing usually falls between these two extremes, with a behavioral model at a higher level of abstraction than the implementation, but with significant portions of the testing model developed independently of the design models.

When there is a significant difference between the abstraction level of the behavioral model and the SUT, there is a need to describe the transformation between abstract test cases and concrete test scripts at the implementation level. This paper does not deal with transformation languages, but we note that this is often a significant issue in the success or failure of model-based testing.

2.4 Test objectives

Test objectives are often formulated in natural language; however, where test generation is automated, the test objectives are required inputs for test generation algorithms. The test objectives may be couched in terms of the model (model coverage), the implementation (code coverage), user experience (usage profiling, use cases), or combinations thereof. Some of the early attempts at model-based testing failed due to the complexity of defining test objectives for a given automation system.

The objectives may be described in a variety of ways. These reflect both the variability of the objectives themselves and the varying sophistication level of the

users of model-based testing systems. In order to test compliance with a complex requirement, the test engineer may be required to generate a long sequence of related inputs with complicated constraints. Test objectives of this kind are usually referred to as test purposes, and can be formally specified, for instance, as sequence diagrams or state machines (TGV [8] and AGEDIS [9]). Less specific test objectives may be expressed as coverage requirements on the model as in Testmaster [10] and GOTCHA [11], on the input combinations as in Modelware [12], or on other aspects of the test suite or on-line test execution trace. Some tools take a very simplistic approach to the description of test objectives and offer the engineer a choice of testing “levels”, with higher levels generating more test cases and greater coverage while not involving the engineer in the details of specifying objectives.

3 Design vs. test-specific languages

The section covers the use of a design language for test modeling, and goes on to discuss the benefits of using a test-specific language.

3.1 Using a design language

When using a design language, such as UML or SDL, for test modeling – there are certain distinct advantages. Many tool vendors offer modeling tools for these languages. In the case of UML, for instance, its visual notation has become an industry-standard for software design.

Another advantage of the popularity of the language is the availability of qualified people for designing the models. Some best practices have already developed around testing with UML, and some of these are codified into the UML Testing Profile [13]. Moreover, consultation services are readily available.

A key step in any model-based testing methodology is the validation (e.g., formal inspection) of the test model by the developers and other application stakeholders. This is a vital step in resolving specification ambiguities and synchronizing the understanding of the requirements. Thus, it is important for developers to understand test models. A UML-based modeling language will employ a vocabulary and approach that is native and easy to understand for software developers and will obviate their need to study unfamiliar DSMLs.

The disadvantage of these languages and the supporting tools lies in the fact that they are usually much more complex to learn and use than the alternatives. Practitioners often complain about the perceived need to perform system design twice – once for development and once for testing.

An example of adaptation and use of a design language can be found in the AGEDIS project reports [9]. While the experiment reports found many positives, all noted the complexity of the solution and the counter-intuitive modeling.

3.2 Using a test-specific language

Using a test-specific language can ease the deployment of the model-based approach, because testers do not need to learn UML or any other language whose main purpose is not the support of testing activities.

TTCN-3 [14] is an example of a test-specific language that has been defined solely for the purposes of test specification. Although the language concepts are mostly inherited from the scripted protocol testing domain, the language may be suitable for expressing high level test models. Distinct advantages also include the ability to specify tests both in textual and visual notations as well as the close relationship with the UML Testing Profile. However, also in this case, one should be careful to avoid the aforementioned problem of performing the system design twice. There is a choice of tools that execute the specification by interpreting the TTCN-3 code or by compiling it into some programming language.

In addition, different tool vendors have defined languages to be used with their tools. Such languages are optimized for the tool at hand, and can be supported by advanced editors and other utilities that ease the learning curve. These languages are often proprietary, which increases risks of their adaptation. However, in many cases they can still be used for model-based testing in conjunction with a higher-level specification language and a source-to-source compiler.

4 Domain-specific vs. generic language

Next, we compare domain-specific and generic approaches to test modeling.

4.1 Domain-specific languages

Due to the context-sensitive nature of testing, domain-specific approaches offer many attractive advantages. The basic idea of using a domain-specific solution is to introduce a language solely for the purpose of test modeling in the particular domain at hand. This way, it is possible to tailor the modeling language for the needs of the testers in this domain.

In most cases, there are no commercial tools available for modeling and generating tests in a particular domain, and custom made tools are needed. Firstly, there needs to be a test design tool for modeling the tests using the domain-specific language. Secondly, a translator (a compiler or an interpreter) is needed from the language to the underlying test execution environment. Development of such tools does not need to start from scratch; there are tools available for developing domain-specific modeling environments, such as MetaEdit+ [15] or XMF Mosaic [16]. In principle, such tools may mitigate many of the risks and difficulties associated with DSMLs. However, we have not found reports on the use of such tools for testing.

Alternatively, a domain-specific layer of abstraction can be built on top of an MBT tool that uses a generic language. This means developing a translator from the domain-specific language to the generic one. As in any development of a

source-to-source translator, the difficulty of this task depends on the differences between the input and output languages.

As an example of the domain-specific approach, consider a DSML for testing mobile phones through a graphical user interface (GUI) [17]. The purpose of the DSML is to model the behavior of the phone user at a high level of abstraction. The language consists of so-called action words [18], such as “send an SMS”, “answer a call”, and “add a new contact”. The action words are used as transition labels in test models given as LTSs (Labeled Transition Systems, i.e., simple finite state machines). Because the underlying test automation system operates at a lower level of abstraction, each action word is implemented by a sequence of so-called keywords, which correspond to key strokes on the phone’s keyboard. A keyword sequence implementing an action word is given in a separate LTS. The component LTSs are composed automatically to obtain a final test model, which in turn is used for generating the actual tests on-line through a commercially available generic test automation system. Test objectives are stated using LTSs, in conjunction with a textual coverage language in the top tier of the 3-tier test model architecture [19].

Another, quite different, DSML-based approach is HOTTest [20] used for testing database applications. There, tests are specified using a textual language, based on the Haskell functional programming language. The textual test model is first translated to an extended finite state machine from which the actual tests are generated. It’s argued that the strong type system of the modeling language makes it easier to introduce domain knowledge and capture domain-specific requirements than using more conventional model-based testing approaches.

Genesys-Pro [21] is an example that uses DSMLs for testing areas other than software. IBM uses this proprietary approach for verifying processor designs. The tests are specified in test templates defined by an XML schema. The Genesys-Pro test generator uses a constraint solver to create test cases after translating the test template into a set of equational constraints. It has been reported that it takes an experienced engineer from two to six months to learn to utilize the full capabilities of the language. However, novice users can exploit the tool with minimal learning time to create basic scenarios. Compared to their previous approach, fewer defects escape into silicon despite the increased complexity of the design.

4.2 Generic modeling languages

One of the advantages of using a generic test modeling language is the possibility of model-based testing practitioners to move between different domains while still enjoying similar modeling experiences. The look and feel and the basic (e.g. UML-based) terminology will remain the same across different domains.

To support domain-specific modeling, UML provides a standard extension mechanism called ”profiles”. Profiles are collections of stereotypes, tagged values and constraints usually defined as Object Constraint Language (OCL) expressions that can be defined to support modeling concepts found in different domains. Depending on the profile, the result can be fundamentally different

from the plain UML. There are several predefined profiles freely available from OMG, including the aforementioned Testing Profile.

Although a generic approach should liberate us from any specific tool, in practice, there are problems with exchanging models between tools. Moreover, not all tools handle profiles (and possible conflicts between different profiles) in standard ways, so we may be shackled to a specific tool capable of processing the needed profiles.

Another problem with using profiles is that while they support incorporating the concepts of the problem domain into the test models, they lack the ability to hide unnecessary details. That is, when stereotyping a UML class to encapsulate test data, we should be able to forget everything we know about classes, methods, attributes, visibility etc. and concentrate on defining the test data. However, even though we can define constraints to hide the superfluous details, the user interface of the tool is not usually customizable to the same extent. The menus remain cluttered with options having nothing to do with the task of defining test data. This makes the work of the tester unnecessarily complex. Depending on the background of the testing personnel, these kinds of usability problems can hamper the deployment of a generic approach significantly.

An additional problem manifests itself when the domain is hard to define through a UML profile. There are cases where defining stereotypes and tags is simply not sufficient to describe the target environment. Extending UML through other means is not frequently supported by UML tools and can hardly be supported by any infrastructure developed for previous UML profile-based solutions. This problem may be either very hard or impossible to solve. Moreover, some previously unseen interoperability problems could surface later.

Yet another issue is the SUT adapter/translator needed in generic solutions. It is a software component that handles the transfer and translation of messages between the test execution system and the SUT. Because the generic tools are designed to work with any kind of SUT, they need to be adapted when deployed in a new context. Depending of the type of the SUT, this component can be very simple or more complex. In any case, it needs to be developed before the test execution can be started and maintained as any other piece of software. There do exist tool-specific adapters for certain domains, and there is work on generic adapters for TTCN-3 [22]. However, the effort needed to develop this component should be considered before choosing a generic tool.

On the application and document generation side of MDD, there is active research to bridge the gap between DSMLs and UML profiles that will most probably enhance model-based testing. For example, one interesting approach [23] uses metamodels for defining translations between the two types of models.

5 Further considerations

In this section, we cover additional aspects concerning the choice of a test modeling language.

5.1 Visual vs. textual languages

The question of whether to use visual or textual languages for test modeling is somewhat orthogonal to the previous discussion, and is a matter of personal background and taste of the testers. On the one hand, most testers would probably prefer a visual language for understanding a model. Model inspections and reviews – especially the ones where both testers and designers are present – are much easier with visual models. On the other hand, textual languages can be very productive in test creation [20]. Another example of a textual language used in industrial settings is Gotcha Definition Language, supported by the Gotcha tool [11]. When developing large models with visual languages, one has to utilize the abstraction and encapsulation mechanisms effectively in order to avoid cluttering the models. Modular development may be easier with a textual language, at least for testers with some programming experience.

It is usually much harder to provide validation and simulation support for visual models. Hence, debugging and refactoring may often be easier in the case of textual languages. A language like TTCN-3, offering both types of notations with a mapping between the two, is viewed as the best solution in this respect.

There are also vast libraries of free and open source software, that facilitate testing with Java, Python, or Perl. However, since these languages have not been designed for modeling as such, the level of abstraction may be too low. If, for instance, it takes several lines of code to model a test event, there is practically no difference with traditional test scripting from the modeler's point of view. A better approach might be to use an existing programming language that has been extended to support model-based testing such as in [24].

5.2 Commercial vs. in-house vs. open-source tools

In principle, the domain-specific approach does not rely on commercially available tools as much as the generic approach. However, custom-made tool solutions are often considered more risky than generic ones. This is especially true when you need to decide on starting development of a new tool for a domain-specific language before having any experience in the approach. On the other hand, a well-known vendor of a commercial tool can become a reliable partner providing support for many years – but prove to be too costly in the long term.

Custom made tools demand a heavy upfront investment, which is only justified when they are used in numerous consecutive projects, for example in a product family development. This could also tie the organization to a specific partner used for tool development. An alternative is to develop an in-house tool. However, many testing organizations lack such competencies or prefer concentrating on their core business. Moreover, if the domain changes, the organization has to be prepared to maintain the language and the associated tools. Depending on the organization, this might be infeasible without outside help.

Naturally, the quality of the tools is always an issue; without industry-strength tools, it is easy to fail in deployment, even with the best possible language. There is a much lower level of competition in domain-specific test

tool market than in the one for generic tools. Consequently, the organization has to make a careful choice regarding the tool maker. We see that in many cases, the best solution is an open-source tool, supported by a critical mass of organizations and individuals with similar needs.

5.3 Proprietary vs. standard language

If an organization that uses a standard language, such as UML, SDL, or TTCN-3, wishes to move to another tool, (e.g., for licensing or new functionality reason) it should be possible to preserve the investment in existing models due to the standard nature of language. Thus, a standards-based approach for test modeling provides freedom in the choice of tools.

Another example of benefits gained by using standard languages is the TTCN-3 SIP Test Suite [25] for testing the SIP protocol [26]. SIP, the Session Initiation Protocol, is a signaling protocol for Internet services, such as conferencing, telephony, presence, events notification and instant messaging. By utilizing a standard test suite for conformance testing of a protocol implementation, many interoperability and other problems can be resolved in the early phases of developing SIP protocol implementations. Without using a standard testing language, it would be very difficult to benefit from the standard test suite.

However, since one size rarely fits all, a custom made proprietary solution might be more favorable because of the anticipated improvements in productivity. If the testing organization owns the proprietary language, which is often the case with DSMLs, it can govern the language and the associated tools without outside interference. On the other hand, if the language is owned by some other organization, possibly a partner, a joint understanding about the development of the language and the tools is needed.

6 Pitfalls and solution considerations

SUTs are not equal with regards to testing difficulty. In the simplest case, there are test interfaces designed to make testing easier. In a more difficult case, test execution has to be done through a GUI, using text recognition, to convert bitmap characters to text strings. There is also a vast difference in testing control vs. data intensive systems, concurrent vs. single threaded ones, or deterministic vs. non-deterministic ones. Obviously, it is impossible to develop best practices covering all the different SUT types.

The decision to select a language should be based on business objectives. We believe that the major consideration is to weigh the trade-offs between the solution's long-term risks and development/adoption costs against the ease-of-use, provided by that solution. In the following section, we cover some pitfalls and solutions for the less obvious long-term risk factors.

6.1 Testing solution language and tooling choice pitfalls

The pitfalls we identified concerning language and tooling choice:

Domain slippage. Over time, the domain that requires testing may change, standards on which it is based may evolve and new technologies may appear. The changes may not be drastic enough to demand a totally new solution. Even worse, they may be gradual, requiring constant updates to the modeling language. DSMLs present a higher risk, since they are specific to the addressed problems.

Underestimating effort needed in language definition. As domain-specific languages gain popularity for application generation, there is a danger that such a solution may be adapted for testing leading to low solution quality. Designing a language is a difficult task that needs expert knowledge and should not be underestimated.

Language ownership. If the adopting organization does not own the language definition, it may find over time that the language changed in ways that are not compatible with the organization's goals. While it is not a problem in itself, since the organization already owns the required tools, it may create significant difficulties. For example, support for existing tools may be discontinued, hiring may become a problem, acquiring new licenses may be impossible, etc. Language definition ownership and the use of GMLs can reduce the risk. Proprietary standards owned by tool vendors are high risk.

Tool lock-in. Dependence on a specific tool may become a problem. Support costs may skyrocket, new licenses may become extremely expensive and tool vendors may become less responsive to the organization's needs – e.g. support of new features in future releases. The problem remains more severe for DSMLs, even though proprietary storage formats and the absence of export capabilities for some GMLs may cause similar problems.

Development process change. A change in the development process may make the chosen testing tools extremely inappropriate. For example, adopting automatic workflow systems to handle bug tracking, requirements management and version control – all bundled together to achieve traceability – may require tools that can work within the process management system. Small vendors (or internal solutions) often present higher risk.

Unsustainable vendors. Many systems need to be maintained and supported for decades. A small vendor (or even a large one) may become unavailable for support or tool/language updates over the years. The choice of non-standard DSMLs, and small vendors (whose customer base won't be large enough to provide migration paths by competitors) increase the risk.

Solution complexity. Many MBT solutions are technologically complex when compared to traditional testing approaches. The organization might not be able to find professionals with the required background, nor educate a large enough number of its own testers. The problem might well remain hidden until the market for software developers heats up. In such a case, top testers often move on to development positions and the organization may find itself unable to substitute them. GMLs and design languages are more risky due to their higher complexity.

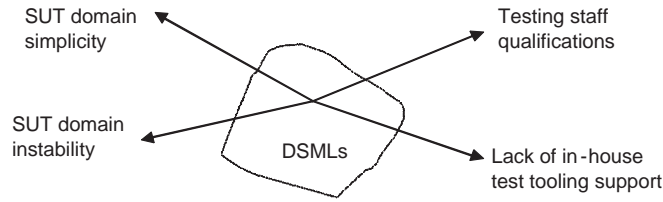


Fig. 2. DSML vs. generic solution

6.2 Solution guidelines for different organization and project types

Figure 2 summarizes our view on the strengths of the DSML approach, which is perceived to challenge the more generic approaches. DSMLs are good in capturing complex domains, and require only domain skills from their users. However, in-house tool support is probably needed and problems can occur in case of domain changes. If the domain is simple and the required skills exist in the testing staff, a more generic solution may be more favorable. Such an approach is seen more robust in the case of domain changes. In addition, tools and the associated services can be bought from third parties.

Below, we provide some general guidelines that take into consideration different factors. The guidelines are listed in the order of importance but should not be considered to be mutually exclusive:

- An organization lacking the infrastructure to develop and support tools in-house and is possibly unable to educate testers: *Reliable large vendor solution, GML and probably test-specific language*. We believe that the existence (or lack thereof) of an in-house team to develop/support testing solutions is possibly the single most important factor in choosing a solution. Good support becomes paramount, domain slippage can cause high costs when handled by external tool vendors (DSML risk), language evolution may render an external DSML solution unusable, and external education and support for testers mean that an easy to use language (test specific) is preferable.
- An organization that is either developing point-in-time solutions for many domains (e.g., services or outsourcing), or has a wide range of products in different domains: *GML*. The first case means that the high risk of adopting new solutions (DSML) is repeated for each new project, and the cost is not amortized over long term/multiple projects. GML, on the other hand, means a stable testing methodology and the reuse of tester’s expertise. The second case means that GML solutions allow easy reassignment of testers between testing teams, without requiring re-education.
- Big, long term projects in a stable domain (e.g. Air Traffic Management Systems, systems for government services, defense or financial organizations): *An in-house solution*, with a DSML, test-specific and possibly even internally owned language. A second (but worse) alternative is a widely accepted

public standard language with either an open source solution or a set of implementations by different vendors. Preventing vendor lock-in is extremely important in this case, both due to possibly escalating costs, and the risk of loss of tool/language support over the project lifecycle. This is best addressed, by an in-house solution, but can be prevented by adopting a widely accepted language. However, domain slippage and language evolution are better addressed by an in-house solution. An additional justification for an in-house solution is that projects of this type are usually run by a large organization that has the skills and resources needed to run a language definition/maintenance project. An investment in a DSML development could be amortized over the long life cycle of the project. The investment in a test-specific language, as well as dedicating a testing team to exclusively support the project for years, limits the risk and improves the quality of these (often safety critical) systems. Thus, specialization of the personnel is not an issue. Ownership of the language is preferable and its cost (in hiring language definition experts) acceptable because the language should support the project needs for a long term. An organization may benefit from publicizing the language and making it a de-facto industry standard in the area.

- An organization using developers for component/functional/system testing: *Design language*. Lower education costs for testers, who are already familiar with the language and only need to learn the new tools, easier buy in by these "testers", and less opposition to perform testing tasks (which are often disliked by developers).
- An organization lacking advanced testers: *Test-specific language*. An organization like this will struggle to adopt MBT in any case. It should be carefully considered whether a more conventional solution would serve the business goals better or not. To succeed the organization must adopt easy-to-use tools and languages that can be mastered by its personnel.
- An organization that uses domain experts to create testing scenarios: *DSML*. Reduces the education costs and increases the chances that domain experts will be ready to create testing scenarios. These experts are often either scientists or expert engineers, and are usually not keen on participating in testing-related activities. The choice will also decrease the time they spend on the activities, and reduces the high cost of these specialists.
- An organization that relies on internal experts to adopt/provide and internally support best practice solutions (e.g., advanced start-up companies): *Open source tools*, standardized or internally created languages (likely DSMLs). Open source gives you the possibility to make necessary changes to the tools when needed. Experts in testing tools (as in other areas of computer sciences) often prefer the flexibility of open source solutions and the ability to both provide their solution to a wide community and cooperate with other experts in their day-to-day work. Such experts are hard to find and expensive to hire. They are usually motivated not just by high compensation (they can find employment anywhere), but mainly by their work definition. Thus, open source and open solutions are usually preferable, since this keeps these people satisfied and makes them more efficient.

7 Conclusions

In this paper we discussed the choice between different types of languages for test modeling. We also provided some guidelines to help in the decision between the alternatives. We believe this decision may have far-reaching consequences in the deployment of model-based practices to a testing organization.

As noted by the authors of [27], there are no best practices in software testing. When selecting between different types of languages, the right choice depends on various aspects. If, for instance, two different organizations need to use common languages and tools for testing, the choice of a test modeling language may be governed by numerous reasons that are less than obvious.

Public case studies on deployments using various approaches are needed. However, since testing organizations, and especially tool vendors, are usually not willing to share their experiences in unsuccessful projects, the published evaluations are presumably biased towards success stories. References to successful deployments are important, but because of the context-sensitive nature of the problem, they should not be over-emphasized in decision making. Thus, we think that the debate between the different approaches needs to be continued.

Acknowledgments

The work of the first and third author was partially supported by the MODELWARE project. MODELWARE is a project co-funded by the European Commission under the “Information Society Technologies” Sixth Framework Programme (2002-2006). Information included in this document reflects only the authors’ views. The European Community is not liable for any use that may be made of the information contained herein. The work of the second author was partially funded by Nokia Foundation.

References

1. OMG: Model Driven Architecture. Available at <http://www.omg.org/mda/> (2006)
2. Domain-Specific Modeling Forum: DSM case studies and examples. Available at <http://www.dsmforum.org/cases.html> (2006)
3. Robinson, H.: Obstacles and opportunities for model-based testing in an industrial software environment. In: Proceedings of the 1st European Conference on Model-Driven Software Engineering, Nuremberg, Germany (2003) 118–127
4. Baker, P., Loh, S., Weil, F.: Model-driven engineering in a large industrial context – Motorola case study. In: Proceedings of MoDELS 2005. Volume 3713 of Lecture Notes in Computer Science. Springer (2005) 476–491
5. Hartman, A., Kirshin, A., Olvovsky, S.: Model driven testing – as an infrastructure for custom made solutions. In: Proceedings of the 4th Workshop on System Testing and Validation (STV’06), Potsdam, Germany (2006)
6. Nachmanson, L., Veanes, M., Schulte, W., Tillmann, N., Grieskamp, W.: Optimal strategies for testing nondeterministic systems. In: ISSTA ’04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, Boston, MA, USA, ACM (2004) 55–64

7. Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Testing concurrent object-oriented systems with Spec Explorer. In: Proceedings of Formal Methods 2005. Number 3582 in Lecture Notes in Computer Science. Springer (2005) 542–547
8. Jard, C., Jéron, T.: TGV: theory, principles and algorithms – a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. STTT **7** (2005) 297–315
9. Hartman, A.: AGEDIS project final report. Available at [http://www.agedis.de/documents/d423_3/FinalPublicReport\(D1.6\).PDF](http://www.agedis.de/documents/d423_3/FinalPublicReport(D1.6).PDF) (2004)
10. Apfelbaum, L., Doyle, J.: Model based testing. Software Quality Week (1997)
11. Farchi, E., Hartman, A., Pinter, S.: Using a model-based test generator to test for standard conformance. IBM Systems Journal **41** (2002) 89–110
12. Modelware: Modelware project homepage. Available at <http://www.modelware-ist.org> (2006)
13. OMG: UML Testing Profile. Available at http://www.omg.org/technology/documents/formal/test_profile.htm (2006)
14. : TTCN-3 homepage. Available at <http://www.ttcn-3.org> (2006)
15. MetaCase: MetaEdit+ homepage. Available at <http://www.metacase.com> (2006)
16. Xactium: XFMosaic homepage. Available at <http://www.xactium.com> (2006)
17. Kervinen, A., Maunumaa, M., Pääkkönen, T., Katara, M.: Model-based testing through a GUI. In: Proc. 5th International Workshop on Formal Approaches to Testing of Software (FATES 2005). (2005) To appear in LNCS 3997.
18. Buwalda, H.: Action figures. STQE Magazine, March/April 2003 (2003) 42–47
19. Kervinen, A., Maunumaa, M., Katara, M.: Controlling testing using three-tier model architecture. In: Proceedings of the Second Workshop on Model Based Testing (MBT 2006), Vienna, Austria (2006) To appear.
20. Sinha, A., Smidts, C.: HOTTest: A model based test design technique for enhanced testing of domain specific applications. ACM Transactions on Software Engineering and Methodology (to appear)
21. Behm, M., Ludden, J., Lichtenstein, Y., Rimon, M., Vinov, M.: Industrial experience with test generation languages for processor verification. In: Proceedings of the 41st Annual conference on Design Automation (DAC-04), San Diego, CA, USA, ACM (2004) 36–40
22. Hyrkkänen, A.: General purpose SUT adapter for TTCN-3. Master’s thesis, Tampere University of Technology, Department of Information Technology (2005)
23. Abouzahra, A., Bézivin, J., Didonet Del Fabro, M., Jouault, F.: A practical approach to bridging domain specific languages with UML profiles. In: Proceedings of the Best Practices for Model Driven Software Development at OOPSLA’05, San Diego, California, USA (2005)
24. UniTesK: UniTesK tools homepage. Available at <http://www.unitesk.com> (2006)
25. ETSI: Conformance test specification for SIP – part 3: Abstract test suite (TTCN-3 code). Available at http://portal.etsi.org/docbox/EC_Files/EC_Files/ts_10202703v030101p0.zip (2003)
26. IETF: IETF RFC 3261 – SIP: Session Initiation Protocol. Available at <http://www.ietf.org/rfc/rfc3261.txt> (2002)
27. Kaner, C., Bach, J., Pettichord, B.: Lessons Learned in Software Testing. Wiley (2001)