Design Principles for Integration of Model-Driven Quality Assurance Tools

Othon Crelier, Roberto S. Silva Filho, William M. Hasling, Christof J. Budnik
Siemens Corporate Research
Software Engineering Department
755 College Road East
Princeton, NJ, USA 08540
{Othon.Crelier.ext, Roberto.Silva-Filho, Bill.Hasling, Christof.Budnik}@siemens.com

Abstract— The engineering of software systems is supported by tools in different phases of the software development. The integration of these tools is crucial to assure the traceability of existing models and artifacts, and to support the automation of critical software development phases such as software testing and validation. In particular, the integration of novel software quality assurance tools into existing environments must be performed in a way that minimizes its impact on existing software process, while the benefits of the tool are leveraged. This guarantees the adoption of new methodologies with minimal interference in existing production workflow. In this paper we discuss our experience in integrating a model-driven software testing tool developed within SIEMENS with a widely-adopted model-driven design tool. In particular, we establish a set of design principles from the lessons learned in this integration. We conclude showing a design that prioritizes data integration over control and presentation that achieves a high degree of tool integration while minimizing the integration development effort.

Keywords: Tool Integration, Software Interoperability, Quality Assurance Tools, UML, TDE/UML, Model-based Testing.

I. INTRODUCTION

Different tools are used in supporting the phases of a software development process. Examples include requirements, modeling, design, development, testing, and configuration management tools. The integration of these tools into traceable and accountable processes is central to the development of high quality software: one that can be verified with respect to the requirements it was designed to support. In order to achieve higher levels of traceability and assurance between different development concerns, and to facilitate the incorporation of new features to existing tools and processes these tools need to be integrated [1]. This integration may involve different aspects such as data sharing, control (synchronizing the context among the tools) or presentation (following common user interface conventions) [2].

Current technology advancements in desktop presentation, operating systems and networks provide an ease of integration not seen two decades ago when research in this subject started [1]. In spite of these advances, software tool integration is still challenging. In particular, problems such as lack of data standardization, poor tool extensibility and lack of guidance during integration are still insufficient.

This paper describes our experiences integrating TDE/UML [5], a SIEMENS model-based testing tool, with Sparx Systems Enterprise Architect (EA) (http://www.sparxsystems.com/), an off-the-shelf UML modeling tool. We report on the lessons learned during the integration between TDE/UML and EA. This integration was, as defined by Siomon [12], "*satisficing*", a sufficing solution within the problem constraints. In other words, it achieved all the important integration goals, with low adaptation costs, without the costs of full integration of the tools. The analysis of this integration shows that the reduced effort was a consequence of the following factors: the facilities provided by current desktop technology advancements, the adoption of a common data model and protocol based on SQL and UML, and the extensibility supported by both tools. We also observed that by prioritizing efforts on data integration over control and presentation, we can achieve high degree of integration between model-driven tools, without the need of full integration.

This paper is organized as follows. In the next section, we perform a literature review providing the background on tool integration concerns, advancements and current issues. Section III discusses the motivation of our case study (the integration of TDE/UML with EA) and our approach. In Section IV we provide an overview of TDE/UML, highlighting its main design decisions that support its integration with other tools; while Section V presents the main characteristics of EA, the tool used in our tool integration case study. In Section VI we discuss our tool integration approach and its design trade-offs. Section VII discusses key implementation details, and in Section VIII we evaluate our approach with respect to tool integration dimensions proposed by [2] discussing the lessons learned. Finally, Section IX presents related work, followed by concluding remarks in Section X.

II. BACKGROUND

A. Tool Integration

"Tool integration is about the extent to which tools agree. The subject of these agreements may include data

formats, user-interface conventions, use of common functions or other aspects in tool construction." [2]

In 1990, Wasserman [1] characterized the tool integration problem in a Software Engineering Environments (SEE) into five aspects: *platform*, *presentation*, *data*, *control*, and *process*. Focusing on data, presentation and control, Wasserman presented different integration mechanisms, characterizing them into a spectrum of technologies, in each dimension, ranging from the weakest to the strongest. The strength of the integration is measured as the extent to which the tools agree on data, control and presentation representations. Wasserman proposes standardization along these dimensions as a way to support the generalized integration between software engineering tools.

In a later work, Thomas and Nejmeh [2] proposed a refined framework for tool integration assessment in Software Engineering, this time, focusing not on mechanisms, but on the problem itself. They propose a tool integration evaluation framework based not only on abstract *presentation*, *data*, *control* and *process* dimensions, but also on particular goals that must be achieved in each one of these concerns. They also consider two user perspectives – environment users and environment builders. In this approach, the software integration goals are evaluated by means of questions, used to inspect the tool integration design with respect to different concerns. These questions, when properly answered, can better assess how well two or more tools are integrated with respect to a given set of properties. In particular, we apply this approach in the evaluation of our tool integration approach, as discussed in section VIII.

While the foundation work of Wasserman, Thomas and Nejmeh's characterized the main concerns involving Software Tool integration, few have studied the application of these concepts on real-work applications. In particular, there is a lack of guidance on the importance of each one of these factors is, as well as in which priority they should be considered. For example, one may assume that control, presentation, process and data integration are equally important, whereas our experience shows that data integration plays a major role in software tool integration. Finally there is a lack of studies that analyze the impact of current advances in technology and standardization on tool integration.

In the following sections we discuss some of the current advancements in technology that facilitate tool integration, and further analyze the existing challenges in this field.

B. Recent Advancements on Tool Integration Technology

Different technological advances have facilitated the integration of software tools. These include standardized operating systems and desktop integration, UI guidelines, standard protocols and data repositories.

With respect to presentation, existing desktop technology provides a standardized User Interface (UI) layer, supporting a common set of widgets, UI guidelines and keybindings. Moreover, it is the case that applications belonging to the same domain (*e.g.*, modeling, coding, and configuration management) usually share very similar user interaction patterns and ontology. For instance, most modeling tools have a palette, a canvas for the working diagram, and panels for specific functions. This standardization reduces cognitive load from the users.

With respect to control integration, software tools can now better communicate and synchronize their operation through a wide range of technologies including: Copy and paste, drag and drop, OLE, and inter-process communication mechanisms based on network protocols such as TCP/IP and integrated notification services.

Finally, data integration has been facilitated by the adoption of industry standards, as originally suggested by Wasserman in his work [1]. For example, XML data representations, the SQL query language and JDBC database compatibility layers [11] are widely used. These standards have provided the foundation for the development of domain-specific data formats in the software engineering domain, such as UML and SysML; or even Open Document (http://www.oasis-open.org/specs/) for electronic documents interchange.

In sum, while standard information retrieval protocols, UI interfaces and inter-process communication approaches allow tools to better integrate with one another, domain specific standards provide a common vocabulary for data interchange in an application domain.

C. Current Challenges in Tool Integration

While existing advances in technology have reduced the data, control and presentation gaps between software tools, the integration of new tools into existing software engineering environments still faces major technological challenges.

The first problem we observed is the **lack of design for extensibility**. Software tools are many times designed with single purposes and processes in mind, being adopted into separate niches. As a tool becomes popular and widely adopted in an organization, the need for interoperability increases. At that point, the lack of design for extensibility results in high costs of development to adapt existing models and processes to the formats adopted by the tool.

Moreover, the integration between different tools is also jeopardized by the **lack of standardization** in a domain, and the need to support **application-specific concerns**. This is particularly true to innovative approaches implemented by research prototypes, where existing standards are usually extended to support novel approaches. The result is a tension between specificity and generality: to define new data representations, better fit to the new approach supported by the tool, or to abide by existing data representations, that usually cannot express application-specific concerns.

Lack of domain-specific protocols. The same problem is also found in protocols. Existing protocols and inter-process communication approaches such as network protocols, file system and database integration are usually too generic. They do not support domain-specific semantic, required to express control and data interaction between different tools. This factor, together with the lack of extensibility, contribute to the integration costs.

User interface inconsistencies. Finally, in spite of standard UI guidelines and toolkits, tools still adopt different semantics and UI features. While this need may come from the specific needs in each domain and application, the lack of common presentation may create a usability barrier between different tools.

As a result, novel approaches must be either integrated into an existing tool or environment, or must be provided in a stand-alone way, supporting existing functionality. Both approaches are usually costly and result in duplication of functionality. Moreover, from the point of view of the users, tools become isolated niches, requiring extra work of integration.

D. Existing Integration Approaches

On the face of the problems discussed above, common strategies are adopted in the integration of tools.

Import/export. Maybe the most popular approach to tool integration is the import and export of data representations between tools. Users manually import and export data files by using data adapters implemented by each tool. While simple and efficient, this approach is not always adequate. First, it requires the adaptation of tools in order to read and write proprietary formats. It relies on tool vendors' ability to implement tool specific filters. In this approach, data formats or not so popular tools are usually not supported. Another major problem in this approach is the lack of control integration between the tools, requiring the manual intervention of users in the synchronization of data.

Data standardization. A second strategy is the adoption of standard data representations, where tools agree on a common data representation that is equally supported by every tool within a domain. This approach, however, usually results in "common denominator" representations that do not account for tool-specific concerns. For example, in a model-based testing tool, one requires information about data and path constraints, that are not found in general-purpose UML editors such as Enterprise Architect. As a result, ways must be found to represent these concerns in a common data format such that it is compatible with both tools. Another common approach is the adoption of "one-size-fits-all" data representations that try to incorporate specific characteristics of each tool by means of profiles, optional elements and data types. The results are usually overly complex formats that lack general acceptance [10].

Vertical integration. Another common approach is the adoption of common vertical platforms that integrate standard data control and presentations. Software engineering platforms such as IBM Rational (www.ibm.com/software/rational) which provide a full stack of tools including configuration management, requirements, modeling, coding and validation and verification is an example. This approach however can be expensive and sometimes challenging to integrate with third party tools.

Standard protocols. Another approach that has been gaining momentum in the industry has been the adoption of standard Integration protocols. For example, IBM's new Jazz platform (www.jazz.net) and the OSLC (Open Services for Lifecycle Collaboration initiative - http://www.open-services.net/) interfaces provide assistance to address these issues by defining a standard RESTFUL protocol to support data, control and presentation integration of heterogeneous tools. Even though promising, this approach requires tools to speak this common protocol, providing a OSLC interface, and is still not fully supported by many off the shelf tools as Enterprise Architect.

III. APPROACH

Based on the problems and solution trade-offs described in the previous section, and the limitations of existing software tools, we sought to devise a set of design recommendations and principles that could guide developers in better supporting the integration of their software tools. In particular, considering all the technological advances in the last decade, we sought after a way to achieve a high degree of integration while minimizing the adaptation effort of each tool. The result was a set of design recommendations as follows.

i. **Design for extensibility**. Tools must provide mechanisms that support developers in writing their own data adapters, allowing models to be read and written in different formats. In particular, we propose the use of an abstract data persistence layer that goes beyond data import/export and also supports control.

ii. **Representation of tool-specific concerns in a non-intrusive, backward compatible way**. A standard data representation should be adopted, supporting common domain model elements, while providing mechanisms that allow tool-specific concerns to be defined in an optional, backward compatible way.

iii. **Support for bottom-up, data centric integration** data integration should be prioritized over control and presentation. Once data integration is achieved, a thin control layer can be provided supporting the automatic synchronization of common data representations. Finally, while tools can have independent UI representations, these must be aligned, supporting similar layouts and interaction mechanism.

Figure 1 summarizes our approach showing the different integration layers, indicating a bottom-up priority, and their main strategies.

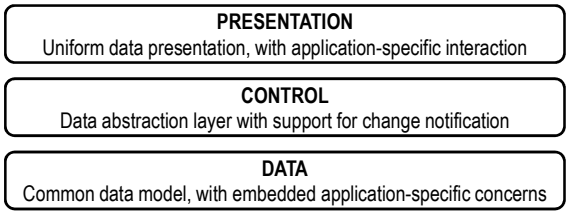

Figure 1. Data, Control and Presentation integration approaches adopted in TDE-EA integration

We applied the recommendations above in the design of TDE/UML, and in the integration of TDE/UML with Enterprise Architect (EA). In sections VI and VII, we analyze the effectiveness of these design decisions in the integration of these two tools. First, however, we will

describe both tools used in our case study in the next two sessions.

IV. TDE/UML

TDE/UML is a model-driven tool [6] that supports the automatic generation of test cases, according to different data and control coverage algorithms. In particular, TDE/UML uses the category partition method [7] to generate tests based on activity and sequence UML diagrams that are annotated with control and data constraints as shown in Figure 2 screen shot. By automating the process of test generation, TDE/UML can better support software testers in the process of software quality assurance, and the development of high assurance systems. Moreover, it achieves high levels of test coverage with reduced development effort.

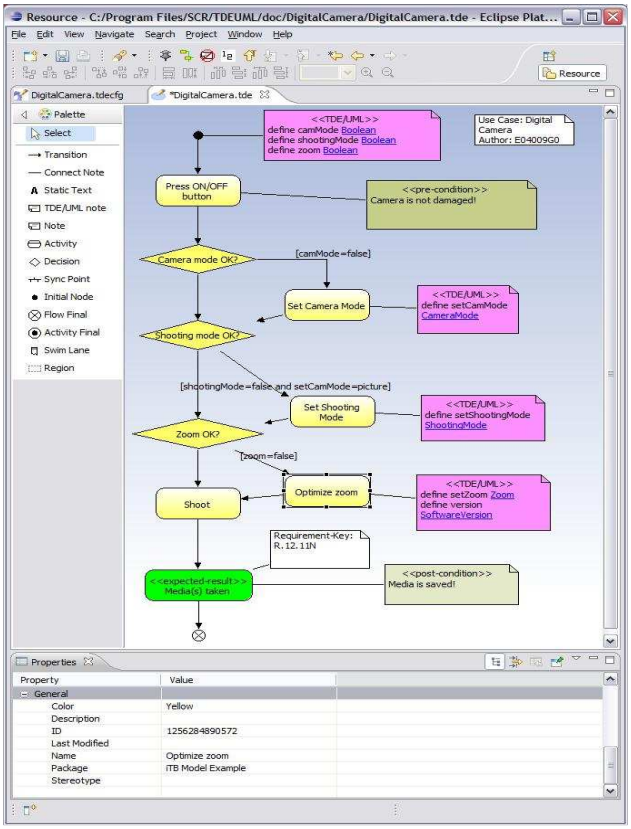

Figure 2. TDE model editor screenshot. Note the use of <<TDE/UML>> notes with data and control constraints

For such characteristics, TDE/UML goes beyond the simple edition of UML models, commonly found in model-driven environments. In particular, it supports model-based testing specifics such as the ability to define data input categories that are used in expressions attached to transitions and decision nodes in UML diagrams. These extensions are key to the generation of tests that cover different paths and data inputs in the model.

Extensibility is one of the cornerstones of the TDE/UML design. It supports extensibility along the main model-driven development concerns including: model edition, model checking, test generation, and code generation. In particular, it supports custom persistence components (design principle i.), and standard UML representation with application-specific constraints being represented as backward compatible UML notes (design principle ii).

By applying the strategies above, we increased the adaptability of TDE, facilitating its integration with existing software tools. Note, that our approach relies not only on data but also in control and profiles defined in each tool. It leverages on extensibility approaches that allow the customization of each tool for the common data representation, and relies on standard protocols to synchronize the access to common resources.

A. *TDE/UML Detailed Architecture*

TDE/UML is based on Eclipse RCP and is designed as a service-oriented architecture, where its major concerns in modeling and generation are extensible by means of plug-ins as shown in Figure 3.

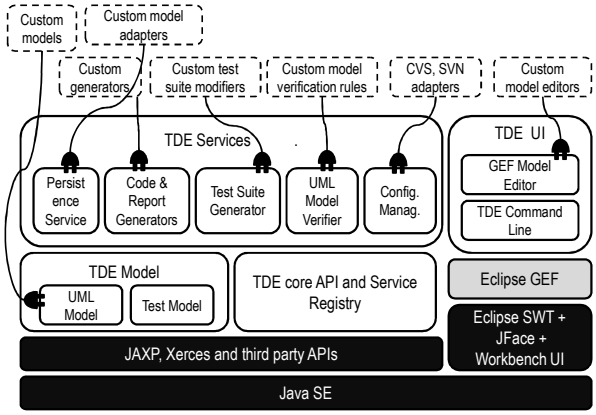

Figure 3. TDE/UML model-based architecture overview

Elements painted in black are reused platform components; solid white blocks represent TDE/UML main services, while dashed boxes represent plug-ins used to extend these services. Whereas general purpose plug-ins are already supplied, *e.g.*, the HTML model report generator, third party plug-ins can be developed using TDE/UML's public API. The main services and elements supported by TDE/UML are:

UML Model: TDE/UML test models include package, use case, activity, sequence and class diagrams. All of them contained in the UML 2 specification. Extra information used for test generation and other features are added by using custom stereotypes, element properties and OCL-like annotations [14].

Note that this extended notation does not break backward compatibility with UML, as they are all contained in regular UML elements. In particular, we rely on UML support for stereotypes.

Persistence Service: The use of UML diagrams allows TDE/UML to interoperate with existing model formats. The persistence service allows models to be saved in different formats, and also supports the interoperability with existing

data representations such as XML, XMI and relational databases via JDBC [11].

In particular, the design for extensibility, and explicit support for persistence plug-ins facilitates TDE/UML interoperability.

Model Editor: Different UML diagrams can be supported. For such, custom model editors, supporting specific UML diagrams and their respective editing commands (e.g. create activity, create note, add guard, etc.) can be defined.

Model Rule Verification: During its development, models can be checked for different consistency and style rules. In particular, TDE/UML supports syntax and semantic checking of its own constraint language defined within notes in the model. Constraints specify data-driven guards for activities and transitions in the model.

Model Report Generation: Provides support for the process of exporting UML models to different formats, and the generation of model documents. For example: HTML and word processing documents reports, or formats compatible to other UML tools.

Test Generation: During the test generation, the annotated UML model is used to produce a *Test Suite*. This process is configurable and supports different data and path modifiers, which implement coverage algorithms. For example: *happy path* (user-defined critical path), data coverage, path coverage, path-data coverage and others.

Test Suite: Is a data structure representing a set of test procedures derived from UML models. Test procedures are the basic product of test generation. They describe a set of test steps, operating over specific data bindings, as well as generic template code to be used in code generation.

Code Generation: The code generation is based on the test procedures described in *Test Suites*, and on the traceability links to the model. Based on that information, generators (each specific to a programming language) are used to produce executable test procedures.

Test Report Generation: *Test Suites* can also be used as a basis for generating more detailed test reports, for example, summarizing coverage information.

The separation of concerns achieved in TDE/UML supports its integration with other tools. In particular, the separation between model edition and test and code generation allows models to be developed in other tools, while the process of test generation can be carried out independently.

In order to reap the benefits of model-based testing in an organization the model based testing support must be integrated with the modeling tools and other tools in the software process. In our study we choose Enterprise Architect as the modeling tool to integrate with our own model based testing solutions.

V. ENTERPRISE ARCHITECT

Enterprise Architect (EA) is a popular off-the-shelf modeling tool, used by several software development companies. Its main purpose is to support the model-driven software design by means of UML 2 diagrams. It also supports:

- Requirements modeling and tracing amongst all model elements;
- Relational database design, generation and reverse engineering compatible with many vendors;
- Reverse engineering and source code generation in many programming languages;
- Project management;
- Concurrent use through supporting relational databases as shared repositories;
- HTML model reporting.

A. *Interoperability*

EA delivers integration mechanisms to interoperate with third-party tools such as:

- XMI (XML Metadata Interchange) [9] importer and exporter to allow interoperation with third party tools;
- Rational DOORS (http://www-01.ibm.com/software/awdtools/doors/) integration using a separately sold add-in;
- ActiveX interface (Automation API) to support external commands from batch scripts or other tools;
- Java API to provide access to EA models within third party java programs – unfortunately, this feature was found to be very slow and unstable as of Enterprise Architect version 7.1.

B. *UML Profile*

The UML Profile feature of EA provides a generic extension mechanism for building UML models customized to particular domains. Examples of these domains include Business Process Modeling (BPM) and Web Applications.

These profiles are specified in XML, and can be created in EA, defining custom stereotypes and properties particular to each domain. When building models, users can import and use several UML profiles as needed.

This feature is particularly interesting in the TDE/UML integration context, as we built an UML Profile for TDE/UML test models, which will be discussed in Section VII.A.

C. *Lack of Model-Based Testing*

Enterprise Architect lacks model-based testing features. Even though it supports *JUnit* and *NUnit* code generation, this generation is limited to creating method stubs. In particular, it lacks the ability to generate test procedures based on different data and path coverage algorithms, which is the main focus of TDE/UML.

As a consequence, EA users in need of model-based testing capability must choose other tools for this purpose. This lack of model-based testing support motivated our study for different ways to integrate TDE functionality within EA. The analysis of the options considered in this integration is presented in the next section.

VI. THE TRADE-OFF OF INTEGRATING EA WITH TDE/UML

In the previous sections we described Enterprise Architect, a powerful general purpose UML modeling tool, and TDE/UML, an extensible model-based testing tool.

The goal of our study was to integrate TDE/UML's model-based testing approach with EA's model design capabilities, thus supporting model-based testing in the existing environment supported by EA, a popular tool within SIEMENS.

By using both tools separately, test models stay apart from system design models, which compromises the traceability between these models. To adapt one of the tools in order to implement all the desired functionalities from the other is unfeasible, because:

- The extension hotspots provided by EA are not flexible enough to allow the incorporation of model-based testing, since EA focus is on model editing the tool is in itself a commercial closed-source software;
- To implement every feature of EA in TDE/UML would mean a huge development effort; additionally, teams were already using EA, which would extra costs of migration from an existing tool to TDE/UML.

Analyzing all the constraints, we opted for the integration of both tools.

Initially, we planned on a full integration of those tools i.e. involving data, control and presentation. We planned for the development of an add-on for EA with a full model-based testing profile, that would call TDE/UML test generation services from within EA. As the cost of integration revealed itself, we search for a way to achieve the initial goal of generating tests based on EA model, by minimizing the integration costs of those two tools. In particular, we adopted the integration approach described in section III. In this section, we justify the adoption of our approach by comparing the cost-effectiveness of other integration strategies given the characteristics of both tools and the application domain. We perform this comparison by discussing different scenarios.

A. Integration Scenarios

In section II.A, we briefly discussed the integration concerns leveraged by Wasserman. It is clear that having two tools fully integrated with respect to data, control and presentation concerns is the ideal scenario. However, the full integration of tools is usually not cost effective. In this section we seek to answer the following question: Can we devise a non-ideal solution that maximizes the integration benefits of the tools, while keeping the integration costs low?

In order to answer this question, we analyzed four integration possibilities, considering loose, medium, and tight integration levels, with respect to the three major concerns proposed by Wasserman. In our analysis we consider both the pros and cons of each strategy. A summary of the results is presented in Table I.

In our evaluation, we consider three levels of integration: loose, medium and tight.

A loose integration level means that the tools don't agree to almost any extent in the given concern. For example, two tools that don't share any data that they produce one with the other are loosely integrated in the data concern.

Medium integration means some agreement in a given concern. For example, two tools have some control integration such as a change notification mechanism, but don't share functionalities one with another.

Tight integration means a high level of agreement. In the presentation concern, tools are tightly integrated with respect to a concern when very few or no adaptation is necessary to integrate the tools. For example, two tools are tightly integrated with respect to presentation if their look and fell, key binding and ontology are similar. As a result, the cognitive load required to use one tool after learning the other one is very low.

Table I. Analysis of integration scenarios costs

-	DATA	CONTROL	PRESENTATION
SCENARIO 1	Loose	Loose	Tight
SCENARIO 2	Loose	Tight	Loose
SCENARIO 3	Tight	Tight	Tight
SCENARIO 4	**Tight**	**Medium**	**Medium**
SCENARIO 5	Loose	Loose	Loose
SCENARIO 6	Medium	Medium	Medium
SCENARIO 7	Loose	Medium	Medium
SCENARIO 8	Loose	Medium	Loose

1) Scenario 1: Presentation focused integration

In this scenario, two tools are tightly integrated only with respect to presentation. In other words, they provide a very similar user experience pattern. Meanwhile, no integration exist with respect to control and data.

For instance, consider two model-driven tools sharing the same look-and-feel and interaction paradigm, but that cannot communicate with each other nor use each other's data.

Pros: optimum cognitive load from the users, as learning how to use one tool means to quickly getting up to speed with the other one. If the tools don't need to share data or functionalities one with another, this strategy is sufficient.

Cons: if model sharing or traceability between the two tools is necessary, this approach does not suffice as there are no means to share data and control between both tools.

2) Scenario 2: Control-focused integration

If two tools operate over the same type of data, but provide completely different presentation, sharing functionalities can be a good integration approach. Say that tool 1 provides external APIs for some of its functionalities. If tool 2 needs to use one of these functionalities over the data it is operating, it does so programmatically, using the public interface of tool 1.

These interfaces can be wrapped as services, being developed using several technologies, such as RPC (remote procedure call), Microsoft OLE, TCP/IP sockets or even command line executables. Still, the tools need to agree on a common set of parameters and return types.

In the TDE/UML and EA integration case study, this approach would require the interchange of subsets of the working model between tools. For example, the invocation of a *TestSuite generateTestSuite(SubModel m)* method from EA to TDE/UML. Also, changes would have to be made to EA in order to create visualizations in order to show the results of such operations.

Pros: reusability of the test generation capability of TDE/UML by means of public interfaces solution, with no changes in the presentation and data concerns of TDE/UML.

Cons: both presentation and data representations from EA must be adapted and aligned in order to comply with TDE/UML internal representation, and in order to present the resulting tests to EA users.

3) Scenario 3: Full Integration

This scenario represents a full integration of the tools with respect to data, control and presentation. A solution like that is common in integrated software environments such as IBM Rational (www.ibm.com/software/rational), where all the tools come from the same vendor and even share plenty of internal components, making the integration even easier. Interoperability is planned since the beginning of the development of the tools.

However, this approach can be costly. They also depend on the tools to be extensible enough to support new features required by each organization.

Pros: benefits of tight integration in the aspects of data, control and presentation.

Cons: high initial development costs, and lack of extensibility towards non-planned concerns.

4) Scenario 4: Data-focused Integration

When full integration is too costly or constraints make it impractical, how to effectively integrate model-driven tools without demanding too much effort? We find that a data-focused approach is useful. In this scenario, the connection between the two tools is first established through the model they share. Control and presentation concerns are integrated in a second step, as means to synchronize the data shared by the tools.

In the particular case of TDE and EA integration, the data-focused solution was designed as follows:

- The EA model was elected as the main data repository. i.e. EA is used as the primary tool for UML design, including test models;
- TDE is used for test generation using the test models created in EA. For such, TDE was extended with a persistence plug-in, allowing it to seamless operate over EA data models;
- To facilitate the creation of test models in EA, an UML profile for TDE test models was created ;
- In this configuration, both tools are running, at the same time, under a common data abstraction layer, so the user can make changes to the model using EA, switching to TDE only during test generation;
- The models are kept synchronized through a JDBC layer in TDE/UML that periodically pools the model for changes, guaranteeing that during test generation, the earliest model is used. In other words, although there are no explicit calls between the tools, there is pool notification mechanism, performing a medium control integration;
- No adaptations to the user interface were necessary, as they already shared a similar user interaction pattern, and the models are presented in similar layouts, using standard UML notations – medium presentation integration already existed.

Pros: We find this approach to be "*Satisficing*" [12], [4], reaching all our initial goals (integrate test generation with EA), with relatively low integration effort, if compared to a full (ideal) integration, and with adequate control integration, without the extra adaptation costs of scenario 2.

Cons: lack of tight presentation integration can be considered a seam, as the users still had switch between tools with different user interfaces.

5) Scenarios 5 trhrough 8: Medium to loosely related application domains

The integration provided by scenario 5 is applicable to domains where the applications are highly unrelated, e.g. running a command line compiler together with a desktop calculator, where both tools have no common data, presentation and control characteristics that facilitate or justify the effort on full integration. In this situation, copy-and-paste integration is usually the only available and/or desirable integration approach.

Scenarios 6 through 8 represent situations where the differences between the applications prohibit a full data, control or presentation integration, for example, when integrating a word processor and a notepad text editor. In this case, the text editor data (ASCII text) does not have the markup capability of the word processor, nor the layout options of the latter. In these situations, the selection of medium integration indicates a cost/benefit compromise, whereas loose integration indicates the existence of high costs.

All these situations are out of the scope of our case study, which focus on the integration of model-based tools. These options are listed here for completeness.

VII. IMPLEMENTATION

In this section, we present the details of our data-focused tool integration approach. We first extended EA with custom test models, and used TDE/UML functionalities achieving tight data integration, medium control integration, and medium presentation integration, with very little effort in the last two concerns.

Figure 4 summarizes the data integration solution we adopted, where EA, customized with a TDE/UML profile, was used together with TDE/UML persistence plug-ins. The resulting blackboard architecture provides a shared data model, with periodic pool control integration.

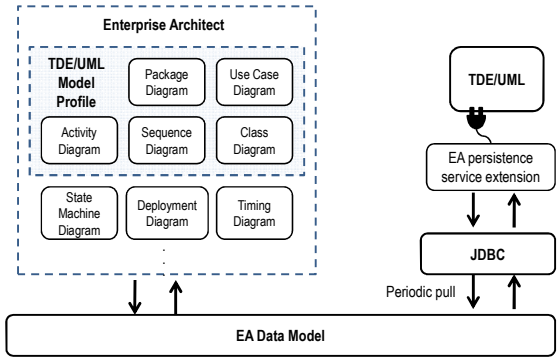

Figure 4. Data integration using EA profiles and TDE/UML persistence plug-ins

A. *Data Integration*

One particular challenge in data integration is the choice of data representation that supports both TDE/UML model-based testing approach, and that is backward compatible with existing standards.

A TDE/UML test model uses a subset of elements and diagrams from UML 2 specification. These diagrams are annotated with testing expressions and custom stereotypes. Transitions are also annotated with logical expressions between []'s written in OCL, an object-constraint language [14]. Hence, TDE/UML relies on UML 2 backward compatible annotations and expressions. A key insight is the use of notes and stereotypes. Constraints are posted in the UML diagrams using variables declared in UML notes (see Figure 2), tagged with special <<TDE/UML>>. Stereotypes support TDE/UML model checking, which analyzes these notes, looking for syntax and semantic errors. This approach also allows the models to be read by EA without any compatibility issue.

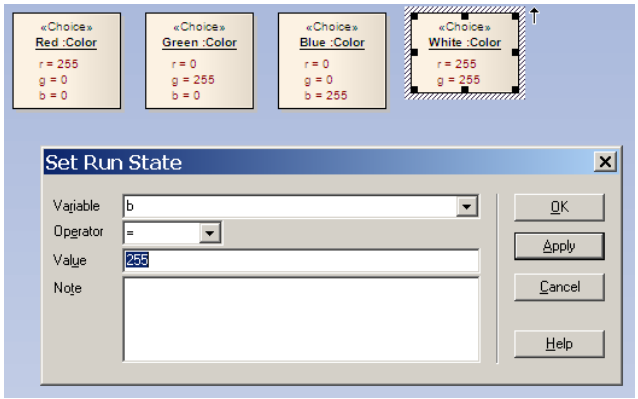

Figure 5. Setting the choice values in EA

A particular challenge in the use of the category partition method (algorithm used by TDE/UML to generate tests) is the definition of data inputs. In our approach, data inputs are represented by means of classes in static UML diagrams. These classes are annotated with stereotypes representing data categories as shown in Figure 5. Class instances in object diagrams are used to represent choices (unique data inputs within categories) in the model. Again, this approach shows how one can customize existing standards for application-specific proposes, keeping backward compatibility.

TDE/UML was then extended with a persistence service for EA model files or repositories. This approach allows TDE/UML to work over .eap (EA's local file format) files or remote EA model database the same way it does with .tde (TDE/UML's model format) files, but in a read-only mode. The custom stereotypes and element properties used by TDE/UML were added to EA using the Enterprise Architect UML Profile feature, discussed in section V.B and shown in Figure 6. This provide a convenience set of predefined diagram elements that are compatible with the TDE/UML model-based testing approach. The XML file containing the TDE/UML profile for EA was added to TDE's distribution, so it can be imported into EA.

By making adaptations only to the persistence layer, no changes had to be done in the other layers of TDE/UML. Once the layer is installed in TDE/UML, EA models behave just like a regular TDE model. Since we elected EA as the primary modeling tool, TDE access to EA models is read-only.

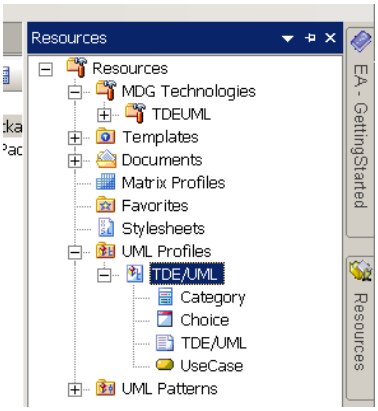

Figure 6. TDE UML profile in EA

TDE/UML's architecture made this task easy, as there is a clear separation of the presentation layer from other concerns, and explicit support for extensibility through TDE/UML plug-in model.

B. *Periodic pull control integration*

While TDE/UML extensible architecture supports model importers and EA has its own XMI export feature, the integration via import/export of models would not be very user friendly. These manual steps would require users to generate data representations every time the model changed. By relying on a data adaptation layer, we supported control integration by periodically polling the model for changes. Every time TDE/UML is used to generate tests, the newest version of the model is automatically accessed. This approach also allows the concurrent access of the model by the two tools, preventing synchronization problems.

VIII. EVALUATION

We further analyze the TDE/UML-EA integration using Thomas and Nejmeh [2] set of concerns. These concerns include presentation concerns such as: appearance and behavior, and interaction paradigm; data integration concerns such as interoperability, non-redundancy, and data consistency, as well as presentation and process integration concerns.

We evaluated our solution focusing on data, control, and presentation integration properties, as follows:

A. Data integration

1) Data Interoperability: How much work must be done for a tool to manipulate data produced by another?

The tools use the same data schema (UML 2), but with different storage formats. EA uses a relational database – either local, stored in a Microsoft Access database file, or any SQL-compatible remote database –, and TDE/UML uses XML files with a proprietary XML-schema. The custom persistence service queries data from an EA database using SQL and extract a TDE/UML model in memory. With that, the data managed by EA became interoperable with TDE/UML.

With respect to development effort, the TDE/UML EA persistence layer required 1232 LOC, a relatively low development effort.

2) Data Nonredundancy: How much data managed by a tool is duplicated in or can be derived from the data managed by the other?

This question evaluates the amount of duplication in the data that the tools independently store and manipulate. Duplication creates consistency problems that increase the complexity of the integrated solution.

With the implementation of the persistence service for EA models, no redundancy existed because both tools operated over the same data.

3) Data consistency: How well do two tools cooperate to maintain the semantic constraints on the data they manipulate?

The semantic constraints in both TDE/UML and EA were inherited from the UML specification. But TDE/UML uses only a subset of UML diagrams and elements. As a consequence, any model in TDE/UML is semantically correct to EA, but not every EA model is semantically acceptable by TDE/UML. The implementation decision was to ignore any elements or diagrams that did not belong to TDE/UML's schema when reading an EA model. Plus, the UML Profile feature added the custom stereotypes and properties to EA, avoiding potential typing errors due to manual operation.

Semantic check of the EA model in the TDE/UML context was done normally using the model rule verification feature explained in section IV.A, as it is independent from the TDE/UML model editor.

The following factors facilitated the data integration design and implementation:

- The adoption of standard formats. TDE/UML model is backwards compatible with UML 2;
- The use of standard data query protocols. EA models are stored in a relational database, which made it possible to reverse engineer its data schema and to query data using SQL via Java's JDBC [11];
- TDE/UML's modular architecture. In TDE/UML, the model is clearly separated from other concerns, making it easy to change between model representations.

B. Control Integration

By sharing functionality one with another, tools are integrated with respect to control. Thomas and Nejmeh propose the analysis of *Provision* and *Use* properties to check how well two tools are integrated with respect to control.

1) Provision: To what extent are the tools services used by other tools in the environment?

A tool is said to be well integrated with respect to provision integration if it offers services other tools in the environment require and use.

2) Use: to what extent does a tool use the services provided by other tools in the environment?

A tool is well integrated with respect to use if it appropriately uses the services offered by other tools in the environment.

In the particular case of EA-TDE/UML integration, our goal was to minimize the development effort while maximizing provision and use. Instead of directly utilizing each other's features, the features are reused indirectly. EA is used as an editor, while TDE/UML is used as a test generator. This blackboard integration approach, based on a common data representation, was able to achieve these goals.

C. Presentation Integration

Two tools are well integrated in the presentation concern when there is small or no cognitive load for a user of one tool to use the other one [2]. In particular, Thomas and Nejmeh propose two questions to assess presentation integration:

1) To what extent do two tools use similar screen apperance and share similar interaction behavior? 2) To which extent do two tools use similar metaphors and mental models?

The fact that the two tools we used belong to the same domain was key to their integration. First, both tools are based on the same editing metaphors, based on canvas, palettes and diagrams. Second, they both share the same model, with common metaphors and representation: standard UML.

The differences between the tools were addressed by the use of TDE/UML profile in EA, whereas the lack of model-checking semantic in EA as addressed by the side-by-size use of TDE/UML. The result was a similar look and feel in both tools.

During test generation, however, the specific model checking and generation concerns are handled by TDE/UML interface. This step of the integration requires users to be familiar with TDE/UML. The learning costs of TDE/UML

test generation approach, however, is much lower than the costs of extending EA with model-based testing interface, and has not been a big issue with users up to this point.

IX. Related Work

On the light of these problems, the Open Services for Lifecycle Collaboration (or OSLC) (http://open-services.net) strive to provide a common data and control representation for the integration of different tools. It relies on a common RESTFUL protocol, and a set of customizable user-defined adapters deployed to web services. These services, allied with existing tool extensibility, can better support the interaction between tools with respect to data, control and presentation (via web interface). Our approach goes in line with this initiative by studying the combined impact of tool extensibility, common data representation and custom adapters in support of tool integration.

The ModelBuss [15] provides tool integration by means of SOA interfaces and custom adapters build around a versioned shared repository. It moves the extensibility burden from tool developers, to a shared data repository. Distributed development is achieved by the use of check-in/check-out protocols, and data integration is achieved by adapters attached to the model bus. These adapters translate the data from/to existing UML model formats. While the concept of adapters allow tools to read/write shared models as if they were using their own native files on a regular file system, check-in/out semantics needs to be build within existing tools. Situations such as model updates and handling of conflicting changes and merges should be manually performed. While approaches as OSLC support different types of data, ModelBuss works with standard UML models.

The design for extensibility principle has been applied in different software tools such the Eclipse platform (http://www.eclipse.org) and the InterLisp environment [13]. These systems leverage on a common data representation and a fully extensible environment, providing a platform that can be extended for different needs. In spite of its benefits provided by these environments, there is still a need for integration of heterogeneous tools in large organizations. To the best of our knowledge, few works have focused on the use of extensibility in these situations. Our work shows the benefits of extensibility, in particular, the role of model abstraction layers, in support of the integration of heterogeneous tools.

X. Conclusion and Future Work

Model-driven testing and design tools support the automation of software quality assurance tasks as test generation. The integration of these tools into traceable and accountable processes is central to the development of high quality software in large organizations, where different tools must coexist. In order to achieve higher levels of traceability and assurance between different development phases, and to facilitate the incorporation of new features to existing tools and processes these tools need to be integrated

In this paper, we contextualized the classic tool integration frameworks in current technology scenario. Although many have changed during the past two decades, the definitions of the problem still apply, and some solutions proposed in the past have gone mainstream, such as the adoption of standards for data representation, and common look-and-feel.

Our integration case study shows that by leveraging on current standardization on both presentation and control, a data-focused integration can be achieved if simple design principles are adopted including: design for extensibility, application of standard data representations in an application-specific way, and the adoption of bottom-up data, control and presentation integration.

As a future work, we plan on further analyzing the trade-offs of existing tool integration approaches by means of more detailed user studies. We also plan on studying the efficacy of data-focused integration approach to other application domains, with different types of tools involved.

References

[1] A. I. Wasserman, “Tool Integration in Software Engineering Environments”, in *Software Engineering Environments*, Springer Berlin / Heidelberg, 1990 pp. 137–149.

[2] I. Thomas, Brian A. Nejmeh, “Definitions of Tool Integration for Environments,” in *IEEE Software*, vol. 9, pp. 29–35, March 1992.

[3] David B. Leblang, Robert P. Chase Jr., “Computer-Aided Software Engineering in a distributed workstation environment”, in Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments, pp. 104–112, 1984.

[4] Eric S. Raymond, “The Art of UNIX Programming”, Chapter 1, Pearson Education, 2003.

[5] B. Hasling, H. Goetz, and K. Beetz, "Model Based Testing of System Requirements using UML Use Case Models," in *International Conference on Software Testing, Verification, and Validation*: IEEE Computer Society, 2008.

[6] L. Apfelbaum, J. Doyle, “Model based testing,” in *Software Quality Week Conference*, May, 1997.

[7] T. Ostrand, M. J. Balcer, “The Category-Partition Method for Specifying and Generating Functional Tests”, in *Comm. ACM*, vol. 31, no. 6, pp.676-686, 1988.

[8] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan, and J. Kazmeier, "Automation of GUI Testing Using a Model-driven Approach," in *International Workshop on Automation of Software Test* Shanghai, China: ACM, 2006.

[9] S. Brodsky, “XMI Opens Application Interchange”, *IBM White paper*, http://www-3.ibm.com/software/awdtools/standards/xmiwhite0399.pdf, 1999.

[10] M. Henning, "The rise and fall of CORBA," *Commun. ACM*, vol. 51, pp. 52-57, 2008.

[11] G. Reese , A. Oram, “Database Programming with JDBC and Java,” Second Edition, O'Reilly & Associates, Inc., Sebastopol, CA, 2000.

[12] H. A. Simon, *The Sciences of the Artificial (3rd edition)*. Cambridge, MA: MIT Press, 1996.

[13] W. Teitelman and L. Manister, “The Interlisp Programming Environment”, in *IEEE Computer*. vol. 14, 1981, pp. 25-33.

[14] M. Richters , M. Gogolla, “On Formalizing the UML Object Constraint Language OCL”, in *Proceedings of the 17th International Conference on Conceptual Modeling*, p.449-464, November, 1998.

[15] Hein, C., Ritter, T., Wagner, M.: Model-Driven Tool Integration with ModelBus; Workshop Future Trends of Model-Driven Development, 2009.