Automatic Generation of Platform Independent Built-in Contract Testers

Helton S. Lima[1], **Franklin Ramalho**[1], **Patricia D. L. Machado**[1], **Everton L. Galdino** [1]

[1]Formal Methods Group, Federal University of Campina Grande
Campina Grande, Brazil

{helton,franklin,patricia,everton}@dsc.ufcg.edu.br

Abstract. *Automatic generation of built-in contract testers that check pairwise interactions between client and server components is presented. Test artifacts are added to development models, i.e., these models are refined towards testing. The refinement is specified by a set of ATL transformation rules that are automatically performed by the ATL-DT tool. We focus on the KobrA development methodology and the Built-In Testing (BIT) method. Therefore, development and testing artifacts are specified by UML diagrams. To make transformation from development to testing models possible, a UML 2 profile for the BIT method is proposed. The overall solution is part of the MoBIT (Model-driven Built-In contract Testers) tool that was fully developed following Model-Driven Engineering (MDE) approaches. A case study is presented to illustrate the problems and solutions addressed.*

1. Introduction

Component-based software development practices can promote the development of high quality software that can be more effectively verified and validated. This is due to the principles behind the practices such as uniformity, encapsulation and locality. Also, since component interfaces are usually specified, for instance, using UML diagrams, functional test cases can be derived from them favoring an effective test coverage of component services and improving reliability and productivity in the testing process, particularly if testing activities are automated.

Deriving functional test cases from functional model specifications to check conformity between an implementation and its specification is commonly known as Model-Based Testing (MBT) [El-Far and Whittaker 2001]. Test suites can be constructed as soon as specifications are produced, favoring reuse of development artifacts and also providing a strong link between development and testing teams. Test cases are related to the intended functional requirements.

Research has been conducted towards effective model-based testing approaches for component-based software. Notable progress has already been made, including methods, techniques and tools [Atkinson and Gross 2002, Briand et al. 2006, Barbosa et al. 2007]. However, there are still challenges to be faced to cope with the increasing complexity and diversity of systems. For instance, apart from test case generation from UML artifacts, a number of artifacts are needed to support test execution such as test drivers, context dependencies and concrete instances of method arguments. These artifacts can also be constructed from development artifacts. Also, as components are developed to be reused and because applications may run in different platforms, it is

crucial that test artifacts can be easily modified to reflect development models variabilities. In this sense, there is a growing interest in combining MBT with Model-Driven Engineering (MDE) strategies, where test artifacts generation is defined through model transformations that can be automated. This is know as Model-Driven Testing (MDT).

The expected benefits of using a MDE approach for testing are twofold. The first is to lower the cost of deploying a given component and corresponding testers on multiple platforms through reuse of the models. The second is to reach a higher level of automation in the test case generation process, through the specification of transformations among those models. In addition, the separation of concerns inherent of MDE added by the level of abstraction and orthogonality deals perfectly with MDT perspectives.

This paper presents a model-driven testing solution for component-based software that supports the automatic implementation of testers according to the Built-In contract Testing (BIT) method proposed in [Atkinson and Gross 2002]. BIT is based on testing components functionality on the client side and testing interfaces on the server side of a pairwise contract. This method is integrated with the KobrA component development methodology [Atkinson et al. 2002]. UML is the central notation in both methods. To allow the automatic mapping from KobrA component specifications to BIT artifacts, we propose a BIT profile and rules for transforming pure UML 2 Kobra diagrams to UML 2 Kobra diagrams fully annotated with BIT profile. Transformations are automated by using the ATL-DT tool [AMMA Project 2005], a MDE based tool that provides ways to produce a set of target models from a set of source models. As a result, the Model-driven Built-In contract Testing (MoBIT) tool was developed.

The reasons for choosing KobrA and BIT are the following. On one hand, KobrA provides a reasonable coverage of the main activities of component-based software development. Also, it is independent of a specific platform. Moreover, it has a clear separation of concerns and is fully based on UML models and OCL specifications, that are the pivotal elements within the MDE vision. Moreover, KobrA is defined towards MDE guidelines: UML models, OCL specifications, profiles, transformations between them (step-by-step described in natural language) and meta-models. On the other hand, BIT is the testing method prescribed to KobrA applications, also based on UML artifacts. In addition, it is flexible in the sense that different test case generation algorithms can be easily incorporated. Moreover, it deals with different aspects of modelling (structural, functional, behavioral diagrams) and generates test artifacts based on them. Furthermore, it covers all aspects concerning test specification and realization.

The paper is organized as follows. Section 2 presents basic terminology on MDE and MDT and briefly describes the KobrA and BIT methods. Section 3 presents the architecture of the MoBIT solution. Section 4 introduces the BIT profile whereas Section 5 introduces the transformation rules. Section 6 presents a case study that illustrates the application of the MoBIT solution focusing on the application of the proposed transformation rules. Section 7 discusses related works. Finally, Section 8 present concluding remarks and pointers for further work.

2. Background

This section briefly presents the main concepts and methods addressed in this paper.

2.1. MDE

Model-Driven Engineering (MDE) is a software engineering approach defined by the OMG. Its main goal is to speed-up the development process by providing the highest possible level of automation for the following tasks: implementation, migration from one implementation platform to another, evolution from one version to the next, integration with other software.

The key idea of the MDE is to shift the emphasis in effort and time during the software life cycle away from implementation towards modelling, meta-modelling and model transformations. In order to reach this goal, MDE prescribes the elaboration of a set of standard models. The first one is the Computational Independent Model (CIM) that captures an organization's ontology and activities independently of the requirements of a computational system. The CIM is used as input for the elaboration of the Platform Independent Model (PIM), which captures the requirements and design of a computational system independently of any target implementation platform. In turn, the PIM serves as input for the elaboration of the Platform Specific Model (PSM), a translation of the PIM geared towards a specific platform. Finally, the PSM serves as input to the implementation code.

The expected benefits of this MDE approach are twofold. The first is to lower the cost of deploying a given application on multiple platforms through reuse of the CIM and PIM. The second is to reach a higher level of automation in the development process, through the specification of transformations among those standard models.

2.2. KobrA

KobrA [Atkinson et al. 2002] is a component-based software development methodology where engineering practices are supported mostly by the use of the UML notation for modelling. The main focus is on Product-Line Engineering (PLE) with the development cycle split into two parts: one deals with the development of a framework and the other with the development of an application as a concrete instance of the framework. KobrA frameworks and applications are all organized in terms of hierarchies of components. Also, the method adopts MDE principles in order to allow core assets to be developed as PIMs as well as to provide a highly cost-effective and flexible technique for implementing KobrA components. The methodology presents guidelines on how to break large models into smaller transformable models; approaches for organizing CIMs and PIMs as components; and what diagrams to use.

With a strict separation of concerns between product (what to produce) and process (how should it be produced), KobrA is aimed at being simple and systematic. Also, it is independent of specific implementation technologies. Moreover, quality assurance practices such as inspections, testing and quality modelling are integrated with development activities. Furthermore, systems and components are treated uniformly: all significant elements should be viewed as components.

KobrA supports the principle of encapsulation in which a component is defined in terms of a specification that describes its provided services and a realization that describes how it provides these services. In the sequel, we focus on component modelling and specification in the KobrA methodology which is the basis of our proposal.

The basic goal of component specification is to create a set of models that collectively describe the external view of a component. A component specification may contain the following sets of artifacts: structural model, functional model, behavioral model, non-functional requirements specification, quality documentation and decision model. The first three artifacts encompass the functional requirements of the component by capturing distinct aspects of the component properties and behavior. The remaining artifacts describe information related to quality and product line engineering and are out of the scope of this paper.

The structural model is composed of UML class and object diagrams that describe the structure of a component that is visible from its interface. Also, it presents classes and relationships by which the component interact with its environment. On the other hand, the functional model describes the visible effects of the services supplied by the component by a set of operation specifications following a standard template. Finally, the behavioral model shows how the component behaves in response to stimuli. This is specified by behavioral state machine diagrams and tables.

2.3. BIT

The correct functioning of a component-based system is dependent on the correct interaction between components. This is usually specified by contracts between components. A contract is a set of rules that governs the interaction of a pair of components and characterizes the relationships between a component and its clients, expressing each part's rights and obligations [Meyer 1997].

Assembling reusable components to build applications requires validation at deployment time of the interfaces and contracts between these components: the ones that requires some services (clients) and the ones that provide the services (servers). For instance, a client can only properly deliver its services if the components it depends on fulfill their contracts. To address this problem, [Atkinson and Gross 2002] proposed the Built-In contract Testing (BIT) method integrated with KobrA where the behavior of the servers are made explicit through a specific embedded testing interface. Using this interface, the clients are able to verify if the servers fulfill their contracts. Contracts are basically the component specifications produced by the KobrA methodology.

This testing method is an approach for reducing manual effort to system verification within component-based development at the moment the components are integrated. The components are generally implemented and tested without the concern of environment faults. By equipping components with the ability to check their execution environments, it is possible to test the client-server integration at runtime.

According to the BIT method, component specification and realization artifacts are augmented by test artifacts. These artifacts consists in testing components and interfaces to be embedded in a client component. Testing components are composed of methods that define and support the execution of test cases.

Following this idea, it is necessary to create a testing interface at the server side, based on the behavior state machine diagram, that is, according to the KobrA methodology, the main artifact that models the behavior of the component. This testing interface is quite simple, consisting on two different types of operations: the first one, sets the component to a specific desired state, whereas the second returns a boolean that informs if

the current state of the component is the one desired or not. This interface is accessible through a testing component that is a server's subclass, generally named *Testable Server*.

Using the servers testing interface, the client implements test cases that access this interface and checks the conformity between required and provided services. These test cases are held on a testing component on the client side, generally named *Server Tester*. The execution of the test cases are also responsibility of the client. This is defined in the *Testing Client* component.

To assemble the components, there is a specific component called *Context* that is responsible for the configuration task, creating the instances of the components and passing the references to the components that depend on other components.

This paper presents a solution to the automatic generation of these testing components: the *Testable Server*, the *Server Tester* and the *Testing Client*, from the structural and behavioral specifications of each server component following the KobrA methodology. The generation of the test cases inside the *Server Tester* are also addressed but are out of the scope of this paper.

2.4. MDT

Model-driven Testing (MDT) [Heckel and Lohmann 2003] is a MBT approach, based on MDE practices, in which test artifacts are automatically generated according to predefined transformation rules that derive, from development models, test cases and their test oracles and the structure of the necessary execution support for different platforms. Development models are required to be abstract and independent of platform.

MBT test cases have been traditionally derived from development artifacts. But this has proven to be ineffective in practice since such artifacts are usually incomplete, particularly concerning information that is vital for testing such as alternative cases, context dependency and constraints. In this sense, MDE practices can play an important role (in addition to the ones already mentioned): the necessary test artifacts can be automatically identified and generated from development artifacts, making it easier to pinpoint missing information.

In general, UML is the central modelling notation for current MDT approaches. Test cases are usually defined as sequences of activation of the system under test that are derived from behavioral models such as state diagrams. Oracles - responsible for deciding whether a test has passed or failed - are decision procedures that are usually based on constraints such as pre- and post-conditions and invariants, possibly written in OCL. Context information is usually identified from class diagrams and method functional specification.

The outcome of a MDT process is a set of testing components that implement drivers for the selected test cases along with all execution support that is needed.

3. MoBIT Architecture

MoBIT is a CASE tool for MDT that was itself built using MDE. It consists in model transformations specified and executed in the Atlas Transformation Language (ATL) [AMMA Project 2005] applied on component models, such as KobrA models instantiating the OMG meta-models of UML 2 [Object Management Group 2006] and OCL 2 [Object Management Group 2006].

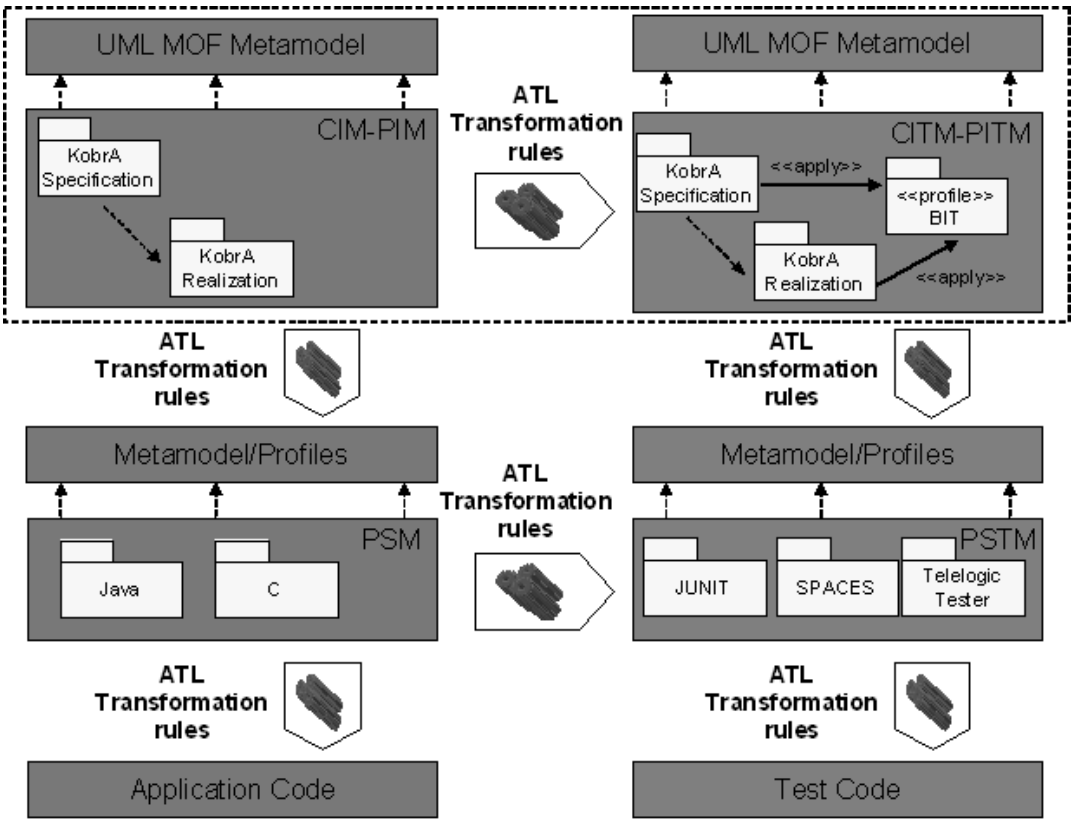

Figure 1. MoBIT Architecture

The internal architecture of MoBIT is shown in Figure 1. It consists of a set of modules containing models, meta-models and profiles. At the top level, there are two modules concerning the CIM-PIM and CITM-PITM. The CIM-PIM are computational and platform independent KobrA models – specification and realization – that do not include information either about the platform on which their components are implemented or about the testing artifacts built to annotate these models. These testing artifacts are present at CITM-PITM that refine CIM-PIM by adding elements according to the BIT method. In fact, the CITM-PITM are KobrA components fully annotated with stereotypes and UML elements defined in the BIT profile and UML metamodel, respectively.

At the middle level there are two modules concerning the PSM and the PSTM. The former includes details about the specific platform on which the KobrA application is implemented, whereas the latter refines the former (together the CITM-PITM) with details about the testing specific platform. The bottom level includes the component code and its testing code.

As shown in Figure 1, there is a separation of concerns in the MoBIT architecture. On the one hand, at the left side of the figure (CIM-PIM, PSM and application code), we find the modules containing models concerning the KobrA application and its implementation independently of any testing approach to be pursued during the development process of the components. On the other hand, at the right side of the figure (CITM-PITM, PSTM and test code), we find the modules containing models concern-

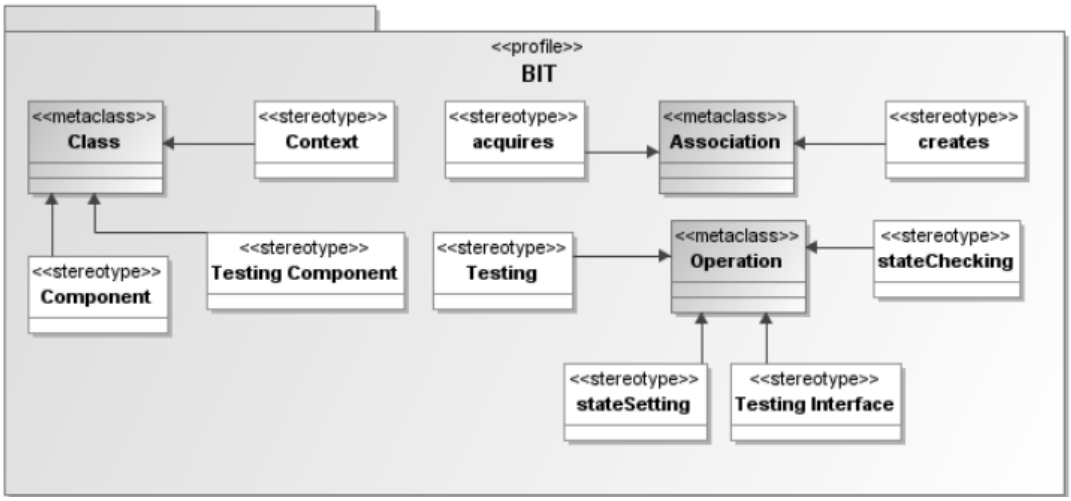

Figure 2. BIT Profile

ing the testing method, artifacts and platform to be incorporated and pursued during the development process of the components.

It is important to highlight that following a MDE approach, transformations may be defined between any pair of the MoBIT modules. This paper is focused on the current state of the work, consisting of transformations from CIM-PIM to CITM-PITM modules, emphasized by the elements inside of the dotted rectangle of Figure 1. We illustrate and discuss them in the next sections.

4. BIT Profile

UML profiles are mechanisms to extend UML with specific domain semantics, allowing any metaclass of the UML specification [Object Management Group 2006] to be annotated with stereotypes representing a specific concept of the domain. In order to formalize the BIT concepts and make the transformation from KobrA diagrams to KobrA diagrams annotated with BIT elements possible, we propose a UML 2 profile, that defines a set of stereotypes which identifies BIT elements inside KobrA artifacts.

In this profile, we specify each element prescribed by the BIT method as stereotypes to be applied to KobrA diagrams. The set of stereotypes and the UML 2 metaclasses that may be extended with them can be seen in Figure 2. The semantics associated with each stereotype is briefly explained below.

- $<< Context >>$: The class responsible for creating both client and server instances and configuring the connection between them, passing the reference of the server to the client.
- $<< Component >>$: A component of the system, responsible for performing functionalities of the system. Can be either a client (requiring a service) or a server (offering a service) or even both (a client of a component but, at the same time, server of another).
- $<< TestingComponent >>$: A class created specifically with testing purposes, responsible for testing activities. On the client side, this is the component that contains the test cases to check the compatibility with the server and/or executes

the test. On the server side, this is the component that makes public the internal states of the server through an specific interface.

- $<< acquires >>$: An association between client and server, where the client receives a reference of the server on configuration time.
- $<< creates >>$: An association between the context and a client or a server, indicating the creation of an instance of a component by a context.
- $<< Testing >>$: Operations with specific testing proposes. It annotates all operations of the *Testable Server* (see Section 2.3).
- $<< TestingInterface >>$: Operations that is part of an interface with testing proposes. This stereotype is applied on the operations of the *Testable Server*. These are public operations that deals with the existing states of the server (defined in the server's state machine diagram): they can change the states or just inform the current state of the server.
- $<< stateSetting >>$: An operation responsible for setting the component to a specific state.
- $<< stateChecking >>$: An operation responsible for checking if the component is in a specific state, returning true or false.

As it can be seen in Figure 2, the three first stereotypes must be applied to the *Class* metaclass since this is the UML metaelement used to model components in KobrA models. The four last ones must be applied to the *Operation* metaclass, identifying the testing methods that form the testing interface of the *Testable Server* element and the remaining ones must be applied to the *Association* metaclass, annotating the relationships between the components. An example of use of this profile can be seen in the case study, detailed in Section 6.

5. Transformation Rules

Although QVT [Object Management Group 2006] is the current OMG's purpose for specifying transformations in the MDE vision, we have adopted ATL for specifying the MoBIT transformations. While QVT tools have still low robustness and its use is not widely disseminated, ATL has a framework widely used by an increasing and enthusiast community, with full support to model operations through a plug-in developed on the Eclipse [Budinsky et al. 2003] framework, where, in an integrated way, one can specify and instantiate metamodels as well as specify and execute transformations on them. In addition, ATL has a wide set of transformation examples available at the literature and discussion lists, in contrast of QVT, whose documentation is poor and not didactic.

The transformations from the CIM-PIM patterns to the CITM-PITM patterns are implemented as ATL matched rules. In essence, these are rewrite rules that match input model patterns expressed as UML elements on the input metamodel and produce output patterns, also expressed as UML elements with a few additional, ATL-proper imperative constructs that cut and paste the matched input model elements in new configurations in conformance to the output metamodel.

Due to lack of space, we show below only an excerpt from one MoBIT ATL transformation rule named CIMPIM2CITMPITM (Figure 3). This rule specifies how to transform a pure KobrA application into one fully refined with testing artifacts conformant to the BIT method. Lines 1-2 defines the name of the ATL transformation module (one ATL module may contain one or more ATL rules) and the input and output model variables,

respectively. Line 3 defines the rule named CIMPIM2CITMPITM whose specification is between lines 4-50. Lines 4-6 state that the Class element (annotated with the stereotype $<< Context >>$) is that one from the source UML meta-model to be transformed, whereas lines 10-50 specify which UML elements have to be transformed to. This rule creates several new UML elements, such as one subclass for the client component (lines 11-16), a tester component for testing the server component (lines 30-33) and one association outgoing from the former and targeting the latter, annotated with the stereotype $<< acquires >>$ (lines 41-49). In addition, all remaining testing artifacts prescribed by the BIT method are generated during the transformation. For instance, the generated server tester component (lines 30-33) must have an operation named 'executeTest' (lines 34-37) and be extended with the stereotype $<< TestingComponent >>$ (lines 38-40).

6. Case Study

An illustrative KobrA specification structural model is shown in Figure 4. It is a simplified and extended version of the library system (LS) presented in [Atkinson et al. 2002]. LS manages enrolled users and available items for loaning as well as records all performed loans, returns and reservations executed in a library environment. In this model we find some prescribed KobrA artifacts whose semantics is defined as follows.

In Figure 4, we focus on the structural model of the LoanManager class representing the component under specification, whose offered services range the management and record of item loans. In order to offer these services, the LoanManager requires a set of other services. Therefore, it plays dual role in the system: (1) as server, offering to the Library component a set of services identified by some of its operations, such as loanItem(), reserveItem() and printLoanInformation(); (2) as client, requiring a set of other services, such as printing services offered by the ReportWriter component or item querying services offered by the ItemManager component.

The creation and integration of each pair of client-server components is executed by the Assembler component. This component is responsible for creating the component under specification (LoanManager) and its client (Library). Moreover, it configures all server components in its clients, integrating each pair of client-server components.

The behavior of the LoanManager component is partially illustrated in Figure 5. It is an extension of that presented in [Atkinson et al. 2002] to the LS. In this figure, we show: (1) the behavioral state machine describing the major states of the LoanManager; (2) the operations that can be executed in each one of theses states; and (3) the events that lead to state transitions. Starting with the initial state, control passes to the unconfigured state. Then, the state of the LoanManager changes from the unconfigured to neutral when the Assembler component configures and integrates it with the remaining ones. The LoanManager remains on this state until the user account is identified, when the control passes to the accountIdentified state. On entering the accountIdentified state, any of the actions contained in this state may be executed by the LoanManager component. Then, whenever the user account is closed, there is a transition back to the neutral state.

From these two models we are able to fully apply the BIT method to the LS application. In Figure 4, the classes filled in grey color and its elements as well as the associations involving them are generated by applying the BIT method on the aforementioned two models.

```
1 module UML2BICT;
2 create OUT : UML2 from IN : UML2;
3 rule CIMPIM2CITMPITM{
4    from cont:UML2!Class(
5        cont.extension
6            ->exists(e | e.ownedExtension.stereotype.name = 'Context') )
7    using{
8        server : UML2!Class =  …;
9        client : UML2!Class …; }
10   to
11       subc:UML2!Class(
12          superClass <- Set{client},
13          generalization <- genc,
14          name <- 'Testing' + client.name,
15          ownedOperation <- oo,
16          extension <- cExt),
17       genc:UML2!Generalization(
18          specific <- subc,
19          general <- client),
20       oo:UML2!Operation(
21          name <- 'setTester',
22          visibility <- #public,
23          ownedParameter <- param ),
24       param:UML2!Parameter(
25          name <- 'tester',
26          type <- sTester),
27       cExt:UML2!Extension(ownedExtension <- cExtEnd),
28       cExtEnd:UML2!ExtensionEnd(type <- cSte),
29       cSte:UML2!Stereotype(name <- 'TestingComponent'),
30       sTester:UML2!Class(
31          name <- server.name +  'Tester',
32          ownedOperation <- ooTester,
33          extension <- cExtTester),
34       ooTester:UML2!Operation(
35          name <- 'executeTest',
36          visibility <- #public,
37          ownedParameter <- param),
38       cExtTester:UML2!Extension( ownedExtension <- coeTester  ),
39       coeTester:UML2!ExtensionEnd(type <- ct),
40       ct:UML2!Stereotype(name <- 'TestingComponent'),
41       assoc1: UML2!Association(
42          extension <- aExt,
43          memberEnd <- Set{prop11, prop12},
44          navigableOwnedEnd <- Set{prop12}),
45       prop11: UML2!Property(type <- subc),
46       prop12: UML2!Property(type <- sTester),
47       aExt:UML2!Extension(ownedExtension <- aExtEnd),
48        aExtEnd:UML2!ExtensionEnd(type <- aSte),
49        aSte:UML2!Stereotype(name <- 'acquires')
50 }
```

Figure 3. CIMPIM2CITMPITM Transformation Rule

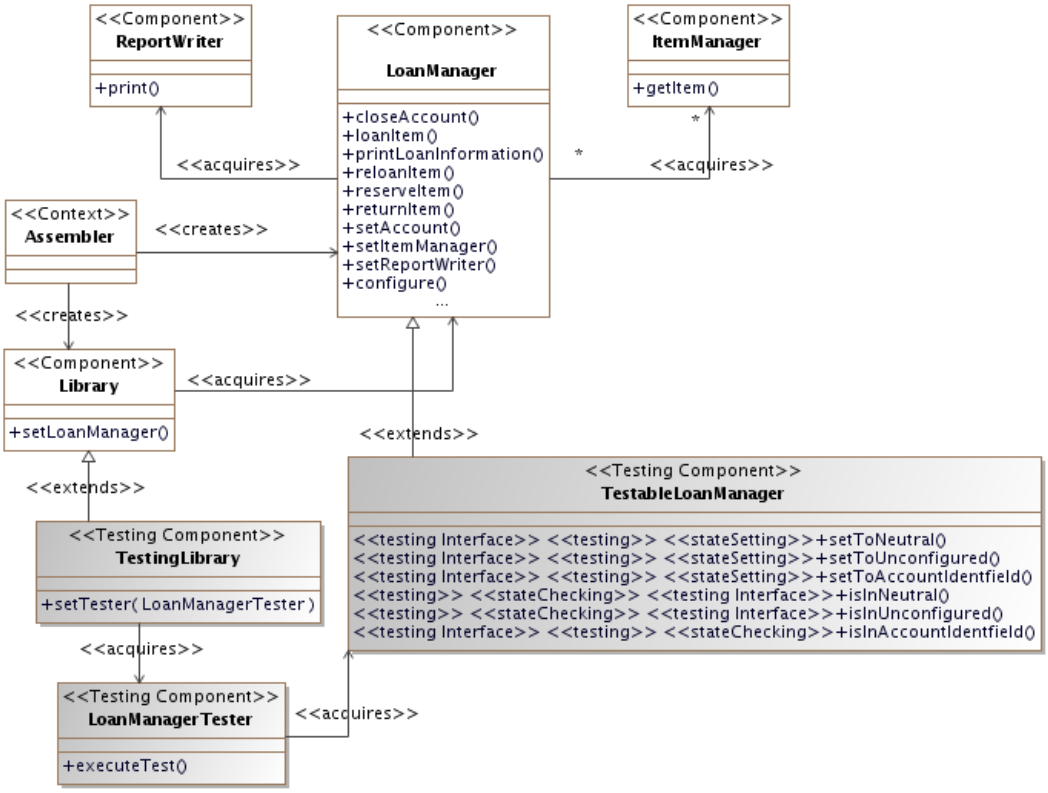

Figure 4. LoanManager Structural Model

The MoBIT tool through its architecture, metamodels, profiles and transformation rules is able to automatically incorporate these artifacts to the original KobrA structural model, *i.e.*, automatically generates the elements filled in grey color from those unfilled. For instance, the MoBIT automatically generates the filled elements by applying the CIMPIM2CITMPITM rule shown in Section 5 as follows:

- Lines 11-16 and 27-29 generate the TestingLibrary component;
- Lines 17-19 generate the hierarchy between Library and the TestingLibrary components;
- Lines 20-26 generate the operation setTester() inside the TestingLibrary component;
- Lines 30-33 and 38-40 generate the LoanManagerTester component;
- Lines 34-37 generate the operation executeTest() inside LoanManagerTester;
- Lines 41-49 generate the association between TestingLibrary and LoanManagerTester components.

The TestableLoanManager component and its state checking and state setting operations deriving from the behavioral state machine shown in Figure 5 as well as its relationships are also automatically generated by the MoBIT tool. However, due to the lack of space, we do not illustrate in this paper the transformation rule concerning this refinement.

Although our case study covers only one client-server pair, the tool is able to generate built-in testers for any number of pairs since: (1) a state machine is specified to

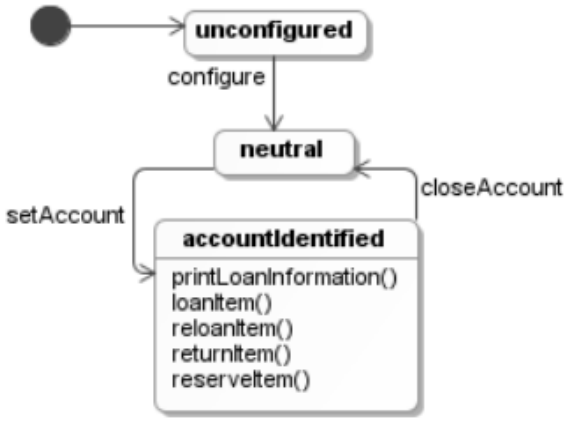

Figure 5. LoanManager Behavioral State Machine Diagram

the server; (2) a *creates* relationship is specified from the client to the *Context* class; (3) a *creates* relationship is specified from the server to the *Context* class; and (4) an *acquires* relationship is specified from the client to the server.

7. Related Works

A number of attempts have already been made towards the theory and practice of MDT. For instance, [Berger et al. 1997] discusses the use of MDT for automatic test case generation and briefly describes a general environment defined at DNA Enterprises that is generally based on the use of existent tools. The underlying MDT concepts are preliminary and no details are given, for instance, concerning the models used. Also, [Heckel and Lohmann 2003] focuses on models and processes for testing distributed components. Even though MDE is taken into account, automation through transformation rules is not considered as well as the use of MDT is only motivated. Moreover, [Engels et al. 2006] presents a testing process based on models and design by contract using the JML language for unit testing. Although automation is considered, this is not based on the use of the MDE technology.

A definite step towards the practice of MDT has been made by OMG by defining the UML 2.0 Testing Profile (U2TP). However, this is not directly integrated with methodologies such as KobrA and BIT since the concepts handled may differ, mainly regarding component based concepts and testing artifacts. The BIT profile presented in this work is a crucial step to make automatic transformation from KobrA models following the BIT method to the U2TP format.

Some approaches have been developed towards MDT based on U2TP. [Born et al. 2004] proposes a MDT approach for telecommunication domains. For instance, they propose a metamodel for validating specific telecommunication domain models on which they work. However, they do not adopt a full MDE architecture as that proposed in our work. In particular, they do not adopt any transformation definition language (such as QVT or ATL) nor explicitly refer to any MDE models (CIM, PIM or PSM). Moreover, although they have adopted the KobrA methodology, the testing artifacts are

manually modeled in separation from the development models.

Another U2TP based approach is presented by [Dueñas et al. 2004]. They propose a metamodel for software testing of product families based on the U2TP profile. The focus is on state diagrams from which test cases are devised.

Finally, [Dai 2004] presents a methodology for applying the U2TP profile to both development and testing models. The general ideas are very similar to ours. However, it is not mentioned in the paper whether the methodology and, particularly, transformation rules have been developed and experimented. Also, the focus is on object-oriented applications, not dealing with the additional complexity of component based systems, *i.e.*, its approach is not eligible for KobrA applications.

8. Concluding Remarks

In this paper, we overviewed MoBIT, a testing tool that automatically generates KobrA components fully annotated with BIT artifacts from pure KobrA component structural and behavioral models that consists of UML class and behavioral state machine diagrams. Because KobrA and BIT are independent of a specific platform and are fully towards MDE guidelines, MoBIT supports the synergetic synthesis of principles from two software engineering approaches, MDE and MDT. In addition, MoBIT was built by pursuing a MDE approach itself. The expected benefits of MoBIT by pursuing these approaches are twofold: (1) to lower the cost of deploying a given component and corresponding testers on multiple platforms through reuse of CIM-PIM and CITM-PITM; and (2) to reach a higher level of automation in the development and test case generation processes, through specification of transformations among those standard models.

The current implementation of MoBIT consists of several ATL rules that cover 100% of UML class properties and behavioral state machines that have a direct translation as BIT artifacts on the KobrA specification models according to the BIT method. As a result, the tool is an Eclipse plugin, and the user can perform the transformation (from models created in the ATL-DT tool itself or imported as a XMI file from another tool) through an intuitive interface. Since XMI is the OMG standard for exchanging metadata information, the integration with other tools, including other test case generation tools, just requires from these ones the support to XMI.

Although KobrA is fully defined towards MDE principles, its current version remains on previous version of UML instead of the current UML 2. However, in MoBIT, we are handling UML 2 with no onus because particularly the UML class and behavioral state machines artifacts prescribed by BIT are in conformance between these two UML versions.

The proposed BIT profile and the MoBIT transformations may be used together with any other methodology than KobrA. To achieve this, the adopted methodology must only prescribe the artifacts required by the BIT method.

In future work, we plan to extend MoBIT to cover all UML diagrams and OCL expressions. We also intend to systematically support each pair of module translations present in the MoBIT architecture: CIM-PIM to PSM, CITM-PITM to PSTM, PSM to PSTM, PSM to application code, PSTM to test code, and application code to test code. In addition, we will study how ATL transformation rules involved in the functional vertical

transformations (CIM-PIM, PSM, application code) may be reused in the test vertical transformations (CITM-PITM, PSTM, test code).

References

AMMA Project (2005). Atlas transformation language. http://www.sciences.univ-nantes.fr/lina/atl/.

Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., and Zettel, J. (2002). *Component-based Product Line Engineering with UML*. Component Software Series. Addison-Wesley.

Atkinson, C. and Gross, H.-G. (2002). Built-in contract testing in model-driven, component-based development. In *ICSR Work. on Component-Based Develop. Processes*.

Barbosa, D. L., Lima, H. S., Machado, P. D. L., Figueiredo, J. C. A., Juca, M. A., and Andrade, W. L. (2007). Automating functional testing of components from uml specifications. *Int. Journal of Software Eng. and Knowledge Engineering*. To appear.

Berger, B., Abuelbassal, M., and Hossain, M. (1997). Model driven testing. Technical report, DNA Enterprises, Inc.

Born, M., Schieferdecker, I., Gross, H.-G., and Santos, P. (2004). Model-driven development and testing - a case study. In *First European Workshop on Model Driven Architecture with Emphasis on Industrial Application*, pages 97–104, University of Twente.

Briand, L. C., Labiche, Y., and Sówka, M. M. (2006). Automated, contract-based user testing of commercial-off-the-shelf components. In *ICSE '06: Proc. of the 28th Int. Conference on Software Engineering*, pages 92–101, New York, NY, USA. ACM Press.

Budinsky, F., Brodsky, S. A., and Merks, E. (2003). *Eclipse Modeling Framework*. Pearson Education.

Dai, Z. R. (2004). Model-driven testing with uml 2.0. In *Second European Workshop on Model Driven Architecture (MDA) with an Emphasis on Methodologies and Transformations*, Canterbury, England.

Dueñas, J. C., Mellado, J., Cerón, R., Arciniega, J. L. ., Ruiz, J. L., and Capilla, R. (2004). Model driven testing in product family context. In *First European Workshop on Model Driven Architecture with Emphasis on Industrial Application*, University of Twente.

El-Far, I. K. and Whittaker, J. A. (2001). Model-based software testing. *Encyclopedia on Software Engineering*.

Engels, G., Güldali, B., and Lohmann, M. (2006). Towards model-driven unit testing. In *Workshops and Symposia at MoDELS 2006*, volume 4364 of *LNCS*.

Heckel, R. and Lohmann, M. (2003). Towards model-driven testing. *Electronic Notes in Theoretical Computer Science*, 82(6).

Meyer, B. (1997). *Object-oriented Software Construction*. Prentice Hall.

Object Management Group (2006). Catalog of omg modeling and metadata specifications. http://www.omg.org/technology/documents/modeling_spec_catalog.htm.