Towards a Software Test Process Framework

Jens R. Calamé
CWI, P.O.Box 94079, 1090 GB Amsterdam
jens.calame@cwi.nl
Erika Horn
Univ. Potsdam, P.O.Box 900327, 14439 Potsdam
ehorn@soft.cs.uni-potsdam.de

Abstract

Testing modern software-intensive systems is a complex process that often lacks proper means for management and documentation. In this paper, we propose an approach to specifying and documenting a test process framework that facilitates automatization of testing processes. The approach bridges the gap between product requirements, software under test, and the actual testware. It also allows to specify stages of a specific test process. We use Meta Object Facility-based meta models as the specification basis for our approach. Furthermore, we provide an application of the approach to testing connector-based multi agent systems.

1 Introduction

In a software test process, the implementation of a software is validated with regard to its model. In order to do so, the test is planned and test cases are developed which focus on certain properties of the software model. These test cases stimulate the software components under consideration (system under test, SUT) and assert the results. A problematic aspect of software testing is the complexity of the software test process.

In this process, many documents, e.g. software models, and software components are involved. These documents have a certain structure and they build up a complex web of relationships. In these relationships, every document or software component plays a certain role. The relationships and roles form the static aspects of the process, which is difficult to overview and to understand. Thus, they must be modeled consistently, so that testing engineers are able to keep track of documents and software components over the test process.

Furthermore, the test of software must not happen randomly, but has to be repeatable to ensure the quality of test results. This means, that a test engineer has to perform the same actions on the same objects over and over again. This is necessary to ensure that a failure in a software component is not found once by chance, but can be found again, if it has for instance not been corrected in a new version of the same component. To assure repeatability, the model of static aspects has to be supported by the description of

actions on the elements of this model. These are the dynamical aspects of the software test process.

Finally, the implementation of software testing requires a good knowledge management. This is a crucial aspect for the test process, since it has to be documented for the customer and also internally, for instance to be able to repeat the process. Another factor, which requires a good knowledge management, is the introduction of new members into a particular project or the training of new employees on a company-wide project scheme.

In this paper, we introduce a test process framework to support the recommended structurization of test processes. In this framework, we model the static and dynamical aspects of such a process.

The test process framework is modeled with meta models, which are based on Meta Object Facilities (MOFs, [9]), and with patterns. MOFs are a framework for the definition of meta models using the means of description of the Unified Modeling Language (UML, [11]). The meta models can be instantiated to the models of concrete test processes. These models can then be used to perform such a process (process instance).

With the means of MOF-based meta models, we describe three stages of a test process focusing on the static aspects. The stages form combinations of the test phases as they are defined in the ANSI/IEEE standard 829 [8]. The three stages cover the definition of models, their implementation and the test execution. With this tripartition, our process framework supports software tests in a software development process based on the Model-Driven Architecture (MDA, [10]). Dynamical aspects for each of the stages can be described by patterns.

In this paper, we work out the meta models of the test process framework and apply them to the design of a framework of test components for multi agent systems (MAS). The paper is organized as follows: The meta models of the process framework are discussed in section 2. In section 3, we describe the application of our test process framework to a case study from the domain of MAS. Therefore, we describe the type of MAS, which we have taken as a case study. Then we show the design of a class library to support the implementation of test cases for MAS as an exemplary application of the defined meta models. We conclude with section 4.

2 Meta Models for the Test Process Framework

In this section, we describe the meta models for the test process framework. We will first introduce an ANSI/IEEE standard for software processes. Then, we give an overview of the MDA, before we combine both concepts to the test process framework, which we propose in this paper.

A software test process consists of several phases. The ANSI/IEEE Standard 829 [8], for instance, proposes the following sequence of phases: *test planning*, *test design*, *test case specification*, *test procedure implementation*, *test setup*, *test execution* and *test analysis*. In the test planning phase, the aspects are defined, which the test process is founded on: the scope of the test, the approach to be taken, the project resources, which have to be assigned to testing, and finally the test schedule. In the test design, the test approach is worked out and coverage criteria are defined. The test case specification finally defines the behavior of the single test cases. These test cases are then implemented

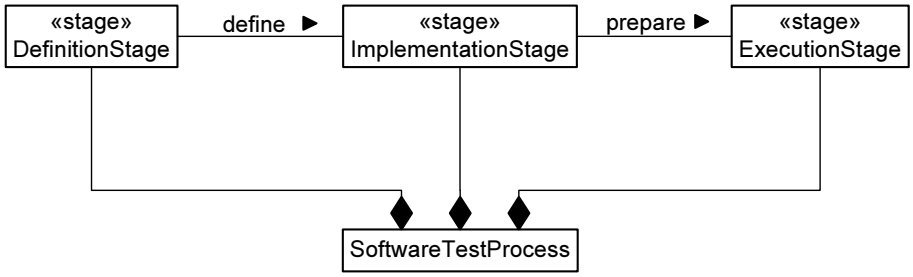

Figure 1: The structure of the test process framework

and, after the setup of the test environment, executed. The outcome of test execution is a set of test results, which are analyzed in the test analysis phase.

The MDA [10] is an approach to develop software systems with a systematical basis on models throughout all project phases. Therefore, several abstraction levels for models have been defined. Three of these levels are of interest for our test process framework: the *Platform-Independent Model* (PIM), the *Platform-Specific Model* (PSM) and the *Code* level. A PIM defines a software system on a level of abstraction, which makes it independent from the platform, which the system is later implemented on. This platform-dependency is introduced in the PSM and finally realized as an implementation of the software system as code.

In this paper, we combine the ideas of both existing standards for software test processes and the MDA in a test process framework. Therefore, we introduce the term *stage*, which describes a combination of standardized test phases and corresponding software models according to the MDA. Considering test phases strongly following IEEE 829 has the disadvantage that model elements, which contextually belong together, can be spread over several such phases. To avoid this, our stages form combinations of test phases. Furthermore, we only want to focus on software in this paper, so we do not model any hardware aspects into the process framework. Thus, we define the following three stages: (1) Definition Stage, (2) Implementation Stage and (3) Execution Stage.

The *Definition Stage* covers planning, design and test case specification phases of the test process following the ANSI/IEEE standard. From the viewpoint of MDA, we can find – clearly separated from each other – the PIMs (business process definitions) and PSMs (component definitions) of both the system under test (SUT) and the testware in this stage. The term *testware* covers all documents (specification documents as well as actual software), which are developed to test a certain product. The SUT is the product which is tested.

The *Implementation Stage* covers the implementation phase of the standardized test process and that part of the test setup, which has to be implemented into the testware. From the viewpoint of MDA, this stage covers the code level.

Finally, the *Execution Stage* covers the test execution phase of the standard. Furthermore, it covers those parts of the test setup, which are related to the deployment of software components in a test environment. The Execution Stage prepares the test analysis phase by collecting the test results in a test protocol.

The complete test process framework with its three stages is depicted in figure 1. In our approach, each stage is described by one meta model with a consistent modu-

lar structure. The core is a package `TestingSystem` which contains the whole testware produced in that particular stage. This package imports both the SUT in a package `SystemUnderTest` and basic test libraries or tools contained in a package `TestFramework`. The packages `SystemUnderTest` and `TestingSystem` form the test environment and together with the package `TestFramework` the test infrastructure. In the following, we describe the stages in more detail.

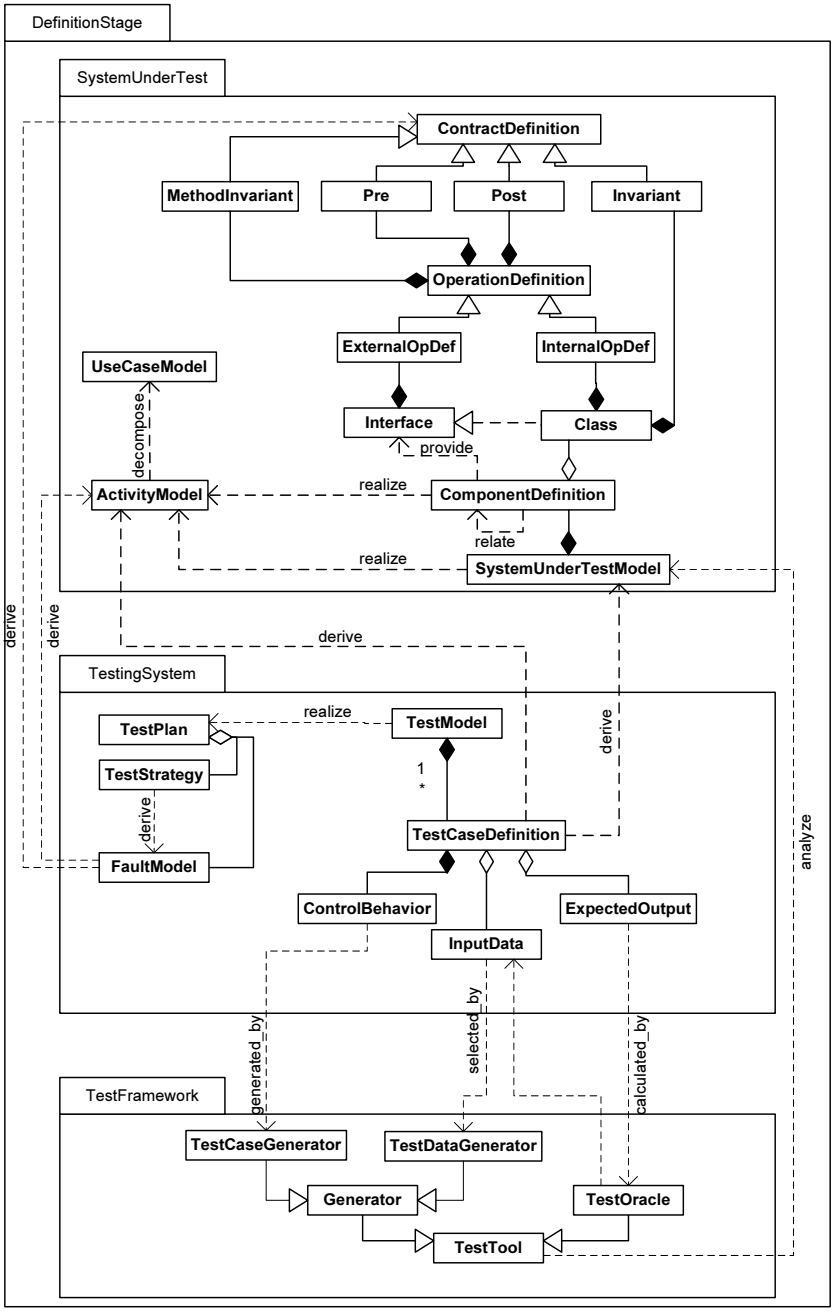

Figure 2: The Definition Stage

2.1 Definition Stage

In the *Definition Stage*, the PIMs and PSMs of both the SUT and the testing system are specified. The detailed refinement of this stage is given in figure 2. The meta classes of this stage make it possible, to describe the conceptual level of testing.

In the package `SystemUnderTest`, we provide elements for the description of the PIMs defining the SUT (the business models given as classes `UseCaseModel` and `ActivityModel`) and for a detailed description of the SUT's PSMs (`SystemUnderTestModel` and the related classes).

The package `TestingSystem` provides elements to describe test models. Here, we distinguish between different parts of the model. On the one hand, there are documents, which cover strategic decisions, represented by the classes `FaultModel`, `TestStrategy` and `TestPlan`; they can partially be considered as the PIMs of the testing system. On the other hand, we have those parts, which model the test cases to be applied to the SUT. The latter ones are represented here by the class `TestModel` and its related classes and can be considered as the testing system's PSMs.

In the package `TestFramework`, we leave the purely conceptual level of test modeling. With the elements of this package, a toolset can be described, which works on the modeling elements of both other packages. Given sufficiently formal models of the SUT, such a toolset can generate test cases automatically or derive useful test data from the SUT specification. In our work on the development of a test framework for multi agent systems, we have used informal models, so that automatic test case generation was not applicable.

2.2 Implementation Stage

By automatic generation of code or by manual implementation from the models of the Definition Stage, the *Implementation Stage* is entered. The meta model of this stage is given in figure 3.

In the package `SystemUnderTest`, all meta classes defining operations are taken directly from the previous stage. The meta class `TestObjectImpl` is the implementation of the `ComponentDefinition` from that stage, while class-internal attributes are newly introduced during implementation.

The interface `Instrumentation` has been introduced in this stage to allow access to internals of the test object, which can be internal operations (test object manipulation) or attributes (test object monitoring). This can be neccesary for a whitebox test, but is not allowed for blackbox testing. The instrumentation can be generated by tools or be implemented manually on the transition from the Definition Stage to the Implementation Stage.

The core component of the package `TestingSystem` is the test execution unit (class `TestExecutionUnitImpl`), which is either a test case implementation or a complete test suite. The meta class `TestCaseImpl` corresponds to `TestCaseDefinition` of the previous stage. A test case implementation consists of one or more test operation implementations, which realize the control behavior of the Definition Stage together with the input data and expected results. A test operation stimulates the test object and compares results coming from this object to expected results.

This comparison happens, for instance, with the help of comparators as they are

defined in the package `TestFramework`. A comparator is a prefabricated operation or a class, which provides the functionality to check, whether certain expectations (e.g. the equality of a result and and expected value) are met, and to set an appropriate test verdict (like test *passed* or *failed*, resp.). Examples for comparators are the `assert...()` functions provided by unit testing frameworks like *JUnit* [2].

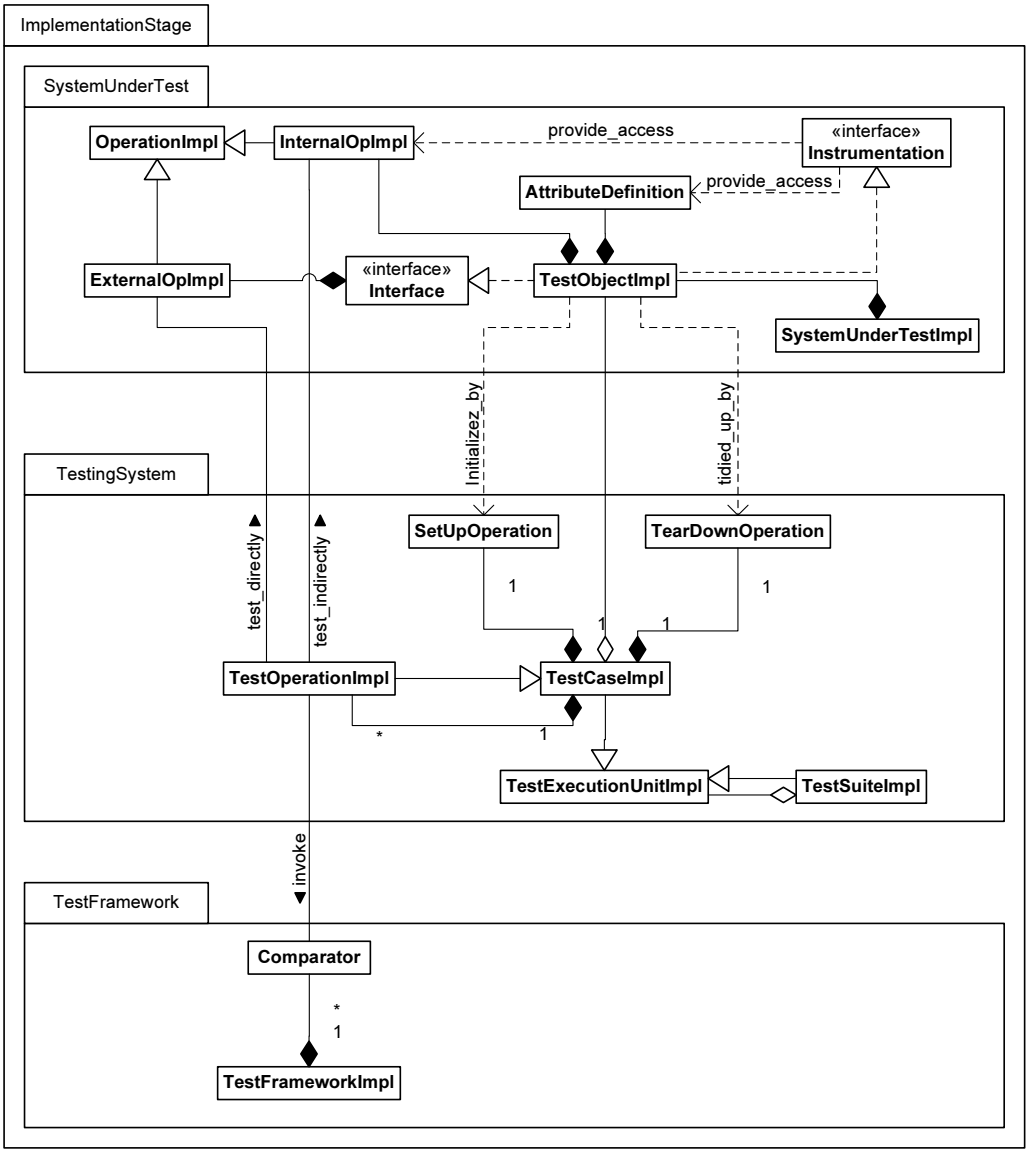

Figure 3: The Implementation Stage

2.3 Execution Stage

By compiling and deploying the components, which have been implemented in the previous stage, the *Execution Stage* is entered. In the meta model for package `Execution-Stage` (figure 4), the test of a test object on the functional level is depicted. In this stage, both the SUT and the testing system are executed in parallel.

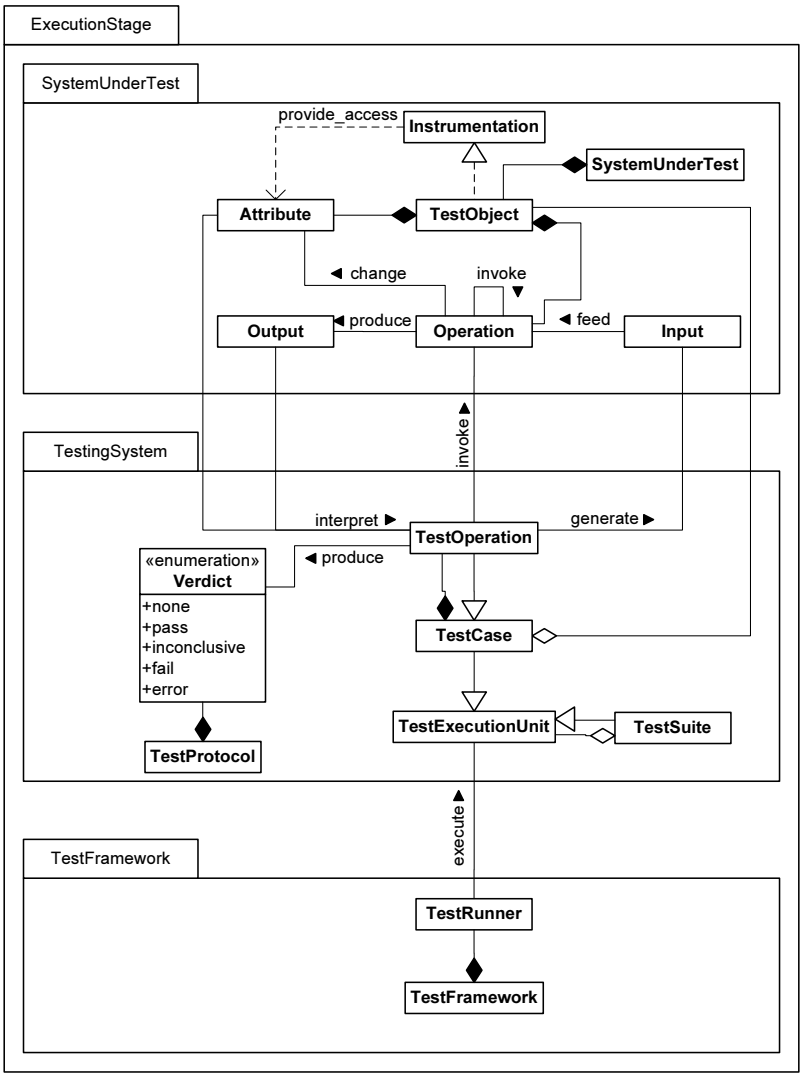

Figure 4: The Execution Stage

The description here is based on the idea of regarding a test object as a state-machine, as it has been worked out in [3]. Inputs are generated by the test operations, implemented in the previous stage. By interpretation of results, test verdicts are derived and both results and test verdicts are logged in a test protocol as a preparation for a subsequent test analysis.

The prefabricated test framework has to provide support to execute test cases or test suites. This support is modeled as the class `TestRunner` in package `TestFramework`.

3 Case Study: Test of Multi Agent Systems

Connector-based Multi Agent Systems Multi agent systems are a technology for the development of distributed component-based systems [4]. A multi agent system is based on *agents*, autonomously acting components which solve problems in cooperation with each other. This cooperation can happen beyond the limits of one computer, not only by remote communication, but also by migrating agents (*mobile code*). Agents

communicate technically by invoking interface functions or by sending events. On the logical level, inter-agent communication can be based on standard protocols, for instance the FIPA *communicative acts* [6].

In the agent platform *Software Architecture type-based Development Environment for Multi Agent Systems* (ADE) [7], communication protocols are not implemented in the agents themselves, but in separate *connectors*. Thus, agents do not communicate directly, but provide interfaces at so-called *ports* to which a connector is bound during communication. The connector actively executes the communication protocol by invoking the agents' interface functions, while the agents themselves stay passive. When the communication has finished, the connector is unbound and released again. This architecture for multi agent systems has been developed following the ideas of the component/connector architecture by Shaw and Garlan [14].

Testing a multi agent system following this architecture is different from testing one which only consists of agents. Since a direct communication between testing agents and tested agents is not possible, and since connectors also implement functionality which must be tested, the test process must be adapted for these systems. Furthermore, a test infrastructure must be developed, to support the test of connectors as well as the test of agents. An appropriate test process has been developed as a set of patterns based on the meta models defined earlier in this paper. On the same basis, the test infrastructure has been designed and implemented.

A Test Infrastructure for Connector-based Multi Agent Systems As we described earlier, testing a multi agent system following the component/connector architecture requires two different approaches. First, the connectors of the system must be tested. This provides a basis for the subsequent agent component test, because already tested connectors can then be used to control the SUT. Thus, a specialized test component framework must be developed, which supports these two concepts. However, there must still be one central instance to control test execution. This is done by a test runner, described in the previous section.

Testing a connector requires a relatively complex setup of testing components. The test runner instantiates a connector test agent. This agent serves as a manager for the test case and as a collector for test results. It instantiates two stub agents which start communicating using the connector under test. While doing so, the actual course of the communication is compared to the expected course with respect to both the protocol's control flow and the transfered data. Results of the test case are sent back via the connector test agent to the test runner and are logged there.

The agent test setup is less complicated. The test runner instantiates an agent test agent which itself connects to the agent under test using all connectors which are necessary to run the actual test case. Results are again sent back to the test runner. Unlike the connector test, the agent test can also carry whitebox test characteristics. These are related to the interface `SystemUnderTest::Instrumentation` given in sections 2.2 and 2.3. This instrumentation requires the implementation of an additional connector according to the interface. This mechanism is only available for agent testing because directly invoking interface functionality on a connector is not possible in ADE.

In figure 5, we depict the design of a test infrastructure for ADE. The package `TestFramework` covers several classes, for which a common basic implementation can be provided. All test case implementations, no matter if they test connectors (class

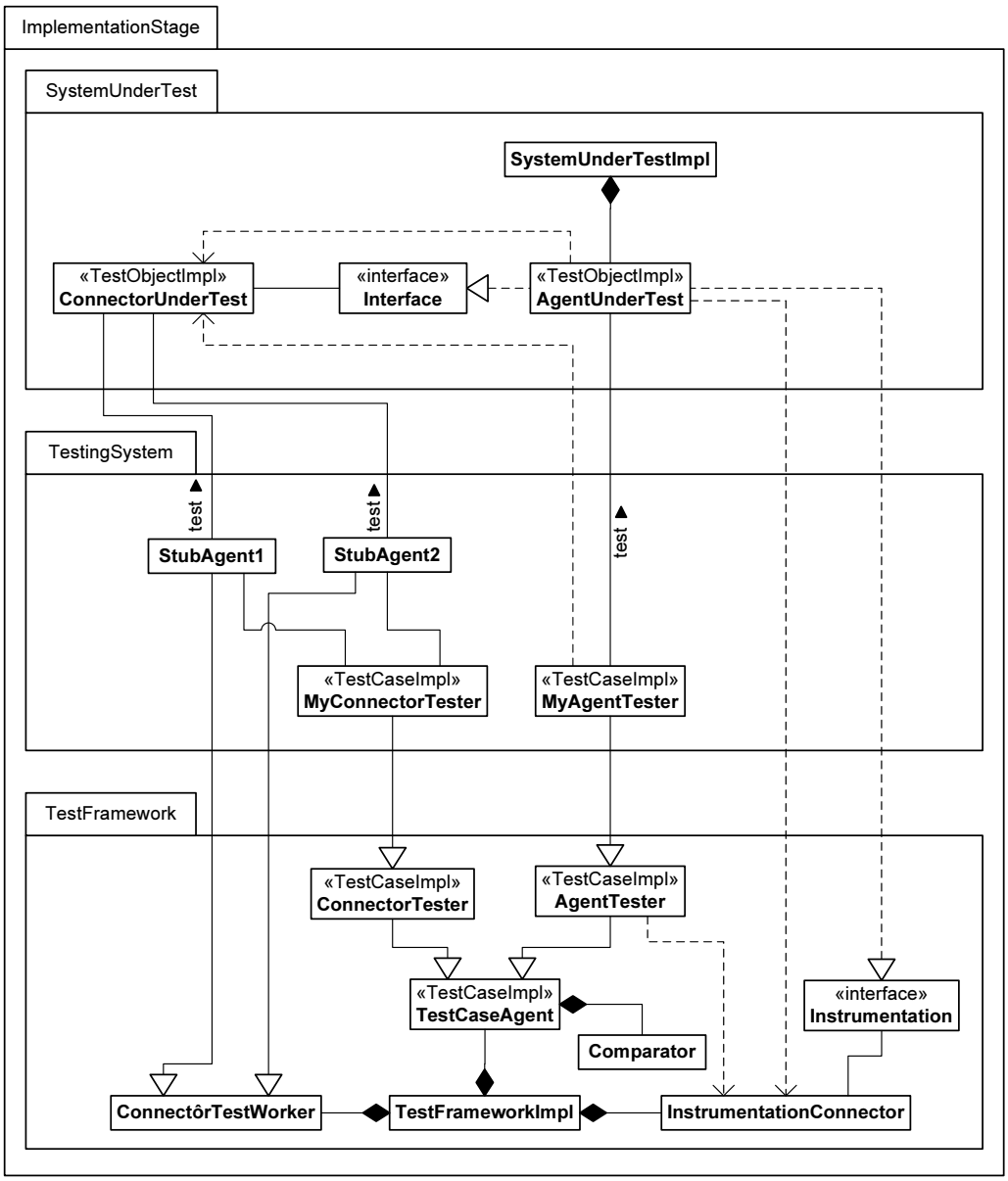

Figure 5: Implementation of an agent test infrastructure

ConnectorTester) or agents (class AgentTester), inherit from one common class TestCaseAgent, which implements the communication with the test runner during test execution. The test runner is not depicted here, since it is part of the Excution Stage. The implementation of actual test cases for connectors inherit from class ConnectorTester. It is bound to two connector stubs (classes StubAgent1 and StubAgent2), which play through the communication protocol with the tested connector (ConnectorUnderTest). Both stub agents are inherited from a class ConnectorTestWorker from the test framework, which implements the communication between connector stub and connector tester. An agent tester inherits from the basic class AgentTester of the test framework and is connected to the agent under test (class AgentUnderTest) by the connectors specific for the particular SUT. The agent under test can optionally implement the interface Instrumentation to provide direct access to its state. Since not only the interface has to be implemented, but also a connector has to be given, we provide both a generic interface as well as the appropriate connector as parts of the framework (interface Instrumentation and class InstrumentationConnector). So, the instrumentation interface changes its position from the package SystemUnderTest to TestFramework.

Benefits The test infrastructure presented in this section was implemented for the ADE multi agent platform. Designing such an infrastructure for the ADE component/connector architecture is very complex and was well supported by the meta models presented in this paper, especially the one for the Implementation Stage. Also the test process for a system based on a strict distinction of components and connectors is intricate, too. To be able to better support this process, we developed a set of test patterns for multi agent systems on the ADE platform. The structure of this pattern catalog could be well founded on the test process framework, which we have proposed in this paper.

4 Conclusions

In a software test process, many documents like models and software components are involved. They have a certain structure themselves and build up a complex web of relationships between each other. Furthermore, they are transformed over several phases of the process, which leads to a second dimension of relationships. The requirement of repeatability of a test process and test-related tasks like knowledge management make it necessary to structurize the documents related to a test process.

In the main body of this work, we provide MOF-based meta models that form a basis for a test process framework that allows to manage and document software test processes in a structured way. Such meta models are also applied on this area by the *UML 2 Testing Profile* (U2TP, [1, 12, 13]). This profile, part of the new UML 2.0 Standard, defines a UML-based language for modeling the artefacts of test systems. Therefore, elements for the description of the architecture, behavior and data of a test system as well as time concepts are defined. However, U2TP does not define the appearance of documents over several phases in a test process. Furthermore, in this paper we do not only describe the architecture of a testing system and of the system under test (SUT), but also that of a framework of tools and prefabricated components supporting the software test process. These three system parts are in contrast to the U2TP clearly modularized in our meta models, so that the presented work goes beyond the U2TP.

The documents, defined in the meta models, cover the testing system, system under test and test framework structures and their development and usage phases along definition, implementation and execution. The structural considerations of the test process can be completed with behavioral considerations in terms of a pattern catalog. Such a pattern catalog has been developed specifically oriented towards the testing of agents and connectors in multi-agent systems.

We have applied this test process framework to a case study from the area of multi agent systems. Thereby, we have concentrated on the implementation stage of the process framework. In result, a test component framework for multi agent systems on the ADE agent platform was realized which can be used to evaluate the functionality and performance of multi agent systems.

An extension of this work would be the application of the test process framework to a test process, in which the models of an SUT are checked automatically and also the test cases are designed in a language, which is – like TTCN-3 [5] – platform-independent, so that the test case models can be used as the implementation of the test cases immediately. While automatic model checking would affect the package `TestFramework` in the Definition Stage by adding further tools like model checkers, the platform-independent realization of test cases would probably affect the design of the whole Implementation Stage. The step from the Definition Stage to the Execution Stage would become smaller in this case, since the dedicated implementation of test cases either becomes obsolete or at least highly automated.

References

[1] P. Baker, Z. R. Dai, J. Grabowski, Ø. Haugen, S. Lucio, E. Samuelsson, I. Schieferdecker, and C. Williams. The UML 2.0 Testing Profile. In *Conquest 2004*, Nuremberg, Germany, September 2004. ASQF Press.

[2] Kent Beck. *Test Driven Development*. Addison-Wesley Professional, 2002.

[3] Robert V. Binder. *Testing Object-Oriented Systems – Models, Patterns, and Tools*. Object Technology Series. Addison-Wesley, 1999.

[4] Hans-Dieter Burkhard. Software-Agenten. In Görz, G., Rollinger, C.-R., and Schneeberger, J., editors, *Handbuch der Künstlichen Intelligenz*, chapter 24, pages 941–1018. Oldenbourg, München, Wien, third edition, 2000.

[5] European Telecommunications Standards Institute, Sophia Antipolis Cedex. *Methods for Testing and Specification; The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*, 2003. ETSI Standard ES 201 873-1 v.2.2.1.

[6] Foundation for Intelligent Physical Agents, Genf. *FIPA Communicative Act Library Specification*, Dezember 2002.

[7] Erika Horn and Thomas Reinke. *Softwarearchitektur und Softwarebauelemente*. Hanser, München, Wien, 2002.

[8] IEEE. *ANSI/IEEE Standard 829 – Standard for Software Test Documentation*. International Standards Organization, 1998.

[9] OMG. *Meta Object Facility (MOF) Specification*, April 2002. version 1.4.

[10] OMG. *MDA Guide*, June 2003. version 1.0.1.

[11] OMG. *UML 2.0 Superstructure Specification*, August 2005.

[12] OMG. *UML 2.0 Testing Profile Specification*, July 2005.

[13] I. Schieferdecker. Integrated System and Test Development with the UML 2.0 Testing Profile. In *Eurostar 2004*, Cologne, Germany, December 2004.

[14] Mary Shaw and David Garlan. *Software Architecture – Perspectives on an Emerging Discipline*. Prentice Hall, 1996.