Model–driven System Testing of Service Oriented Systems

Michael Felderer*, Philipp Zech*, Frank Fiedler†, Joanna Chimiak–Opoka*, Ruth Breu*

*Institute of Computer Science
University of Innsbruck
Innsbruck, Austria
Email: michael.felderer@uibk.ac.at

†Softmethod Gmbh
Munich, Germany
Email: frank.fiedler@softmethod.de

Abstract—With the increasing number of service oriented system implementations, new challenges concerning their development and testing are emerging. This paper presents an approach for model–driven system testing of service oriented systems. The approach offers a systematic testing methodology and it is based on tightly integrated system and test models with a formal metamodel. The test code generation itself is supported by automatic consistency and coverage checks and has a flexible adapter concept that allows different target technologies to be integrated.

Index Terms—Model–Driven Testing, System Testing, Service Oriented Architecture, Requirements Specification, Quality of Services.

I. INTRODUCTION

Service orientation is currently the most discussed concept for structuring and implementing IT systems. From the system point of view, the main components of such service oriented systems are services and their orchestration. Service oriented systems have some specifics, i.e. the implementation of some components is not available in an initial phase and dynamically evolving, the systems are complex, highly adaptive and integrate different platforms, and there is a strong linkage between the application domain and the technology of such systems. These features have not been addressed adequately in existing system development and testing methodologies.

We define a test–driven approach to requirements specification and specify test models before or at least intertwined with the system specification. A test–driven approach to requirements specification of service oriented systems has to fulfill some constraints. First, the test cases have to be business oriented which allows to define test cases together with domain experts as in FIT [16]. Test activities have to be separated from the concrete test data. There has to be traceability between the system and the test model to link them. This enables automatic consistency and coverage checks but also test case generation when starting from requirements. When starting from the test cases, it is possible to generate service orchestrations. The framework has to support frequent system adaptations, e.g. services have not been implemented or services are replaced. The specified tests have to be executable in an automatic way which implies automation of the test generation process.

Based on the requirements above, we have defined a novel methodology and a framework for model–driven system testing of service oriented systems. Our approach is based on tightly linked system and test models specified in the Unified Modeling Language (UML) providing support for test–driven development. To make the test models executable, we have to transform them to executable test code based on a flexible adapter concept to invoke services on arbitrary platforms. In this paper we present our test model transformation and adapter concept.

Our paper is innovative in the following respects. We provide a novel approach for model–driven system testing of service oriented systems. In our approach test–models can be statically checked and directly transformed into executable test code via a metamodel. The connection to the system services under test is established via flexible adapters that can be automatically generated in many cases. This flexible adapter concept allows for defining executable test–model integrated with the requirements specification in a very early stage of the system development process.

The paper is structured as follows. In Section II, we give an overview of our testing methodology and its main artifacts. In Section III we define the metamodel of our system and test model. Section IV presents our test code generation approach and related activities. In Section V we provide related work and finally in Section VI, we summarize and present some future work.

II. BASIC CONCEPTS

In this section we give an overview of our system testing methodology and its underlying artifacts following [11].

A. System and Testing Artifacts

Figure I shows the basic structure of the TTS artifacts. The artifacts are categorized along two orthogonal classifications: Model and Implementation on the one hand and System and Test on the other hand.
The specification of developed in an incremental process. This process starts with services or by a link to surrogate services like manual input providing the glue between the service calls and the execution of test stories executable by the test controller. Service adapters make called which transforms test story files into source code files, so that system model and system implementation are traceable. This contains services representing business logic and configuration services for test purposes.

b) The system implementation: also called the system under test (SUT) provides services callable by the test implementation. This contains services representing business logic and configuration services for test purposes.

c) The test model: contains the test case specifications developed in an incremental process. This process starts with the specification of test stories. Test stories are structured sequences of service calls at business level exemplifying the interaction of actors with the system. Test stories may be generic in the sense that they do not contain concrete objects but variables which refer to test values provided in tables. For testing purposes, test stories are enhanced by assertions, i.e., conditions to be checked within the execution of the test story. For completely specifying tests, each test story has a corresponding initial state and test table. Test stories can be seen as high level descriptions of the test requirements. Their formal metamodel is discussed in Section III-B.

d) The test implementation: is generated by a compiler which transforms test story files into source code files, so called test code, of the execution language. These files are then executed by the test controller. Service adapters make the abstract service calls of the test stories executable by either providing the glue between the service calls and the executable services or by a link to surrogate services like manual input, mock services or external test services. Our prototypical test controller implementation is in Java, and the test code is Java source code. The methodology itself is not restricted to Java (see Section IV-C for more details on the architecture).

B. Testing Methodology

Figure 2 shows the workflow of our testing methodology. The activities Consistency/Coverage Checking, Test Code Generation, Test Execution, and Test Analysis are executed automatically by the TTS framework, and the activities System Model Design, Test Model Design, Data Pool Definition and Test Sequence Definition have to be done manually (with tool support). Note that the Adapter Implementation can be done manually or automatically based on the system model and the SUT if a stub generator for the underlying service technology is provided. The four round boxes group elements and relate them to the four corresponding artifacts, i.e., system model, test model, system implementation and test implementation of Figure 1.

The methodology of our framework supports test–driven development of systems on the model level. The first step in the development process is the iterative design of a test and a system model. The test design additionally contains a data pool definition, i.e., the definition of test data and of initial states for the test stories, and the test sequence definition, i.e., the sequence of test stories together with states and data to be tested. The system model and the test model, including the test stories, the data and the test sequences, can be checked for consistency and completeness. Completeness corresponds to coverage checks in the domain of testing, independently of the system or test implementation itself at any point in time (more in Section IV-A). This allows for an iterative improvement of their quality and supports model–driven system and test development. Our methodology does not consider the system development itself but is based on traceable services offered by a SUT. As soon as adapters which may be generated automatically or implemented manually are available for the system services, the process of test code generation can take place (more in Section IV-B). The generated test code is then automatically compiled and executed by a test execution engine which logs all occurring events into a test log. The test evaluation is done offline by a test analysis tool which generates a test report and annotations to those elements of the system and test model influencing the test result.

The main focus of this paper is the test code generation and its underlying consistency and completeness check mechanism. According to [28] model–driven testing can be defined as testing–based Model–driven Architecture (MDA) [17], either...
by transforming a platform independent test design model (PIM) directly to test code or to a platform specific test design model. Our approach directly transforms the PIM to test code. This part of the overall methodology is condensed and structured according to the platform dependency (Platform Independent, Platform Dependent) and the abstraction level (Metamodel, Model, Implementation, Execution) in Figure 3.

![Fig. 3. Model Transformation](image)

The metamodel provides the abstract syntax for the system and test model. Additionally, the rules for checking consistency and completeness, resp. the rules for test code generation are defined based on the metamodel. The system and the test model are created iteratively with integrated consistency and completeness checks. Then the test code is generated. Additionally the adapters for accessing the SUT services are implemented or generated by extending the abstract adapter depending on the SUT implementation. Finally the test code can be executed. Compared to Figure 3 two rectangles for the artifacts Metamodel and AbstractAdapter have been added.

III. METAMODEL

As mentioned in the previous section the metamodel defines the abstract syntax and the checking resp. transformation rules are defined on it. The artifacts of the metamodel have been defined in [11]. In this section we give an overview of the metamodel and provide a running example of a basic ticket reservation system. The metamodel for system and test modeling of service oriented systems has been defined as UML profile [22] following the principles of domain-specific language design [24]. Defining a UML profile has the advantage that UML itself has been widely used for system modeling and existing system models resp. tools for designing them can therefore be reused for testing purposes based on our profile. Additionally, it is possible to integrate our UML profile with existing standards in the area of SOA modeling such as SOAML [23] which may be used for system modeling.

A. System Metamodel

In Figure 4 the stereotypes and tagged values for system modeling are depicted.

![Fig. 4. System elements of TTS Profile](image)

The defined stereotypes and their relationships are as follows:

- The System stereotype represents the whole test system including services and actors.
- The abstract stereotype Service generalizes SystemService, i.e. a service with a business functionality, and ConfigurationService, i.e. a service needed for testing purposes. A Service refers to actors corresponding to the list of all roles which can call a service, and it has InputParameter elements (input), an OutputParameter element (output) and a precondition (pre) and a postcondition (post) specified in the Object Constraint Language (OCL) [19].
- The stereotype Actor represents the roles which can invoke a service.
- The abstract stereotype Parameter generalizes the stereotypes for input and output parameters, i.e. InputParameter and OutputParameter and has a type which may be of an arbitrary class.

The stereotypes extend the UML metaclasses Actor, Class, Classifier resp. Package (see Figure 4 for details) and can therefore be used for modeling a system within use case diagrams or class diagrams. Services can itself be composed of other services which can be modeled by compositions. Business services resemble the behavior of a system and are therefore like use cases [10].

The following example models the actors and services of a ticket reservation system in a use case diagram which refers to an additional class diagram modeling the domain classes of the system such as Event and Reservation. Figure 5 depicts a use case diagram which defines four services (Login, BookReservation, SearchEvent, RegisterUser) callable by two types of actors (Customer, User).
Each service has tagged values for the actors, the input parameters, the output parameter, its name, a precondition and a postcondition. The parameters itself refer to data types of the UML Infrastructure Specification [21] or to domain classes which have been modeled in a separate class diagram.

B. Test Metamodel

In Figure 6 the stereotypes and tagged values for test modeling are depicted.

![Diagram of Test Metamodel](image)

Fig. 5. Actors and services of ticket reservation system

For combining test stories to a sequence, the following stereotypes have been defined. The stereotype Testsequence encapsulates a sequence of SequenceElement elements each having an initial setup state (setup), a final teardown state (teardown), a test story to execute (story), a table with test data (data) containing objects for all input values and result values of assertions, a timeout (timeout) and an arbitration (arbitration) for computing global verdicts [8]. The setup and teardown states are itself sequences of service calls, especially of calls to ConfigurationService elements.

The test stories and the test sequences are depicted as activity diagrams with its stereotypes extending the UML metaclasses Activity and Action.

For the ticket reservation system example defined above, a test story and a test sequence may be defined as in Figures 7 and 8.

![Diagram of Test Stories](image)

Fig. 7. Test story for Login service and Register service

A basic test story for testing the services Login and Register is depicted in 7. First, the test story tries to invoke the service Login. If the return value provides an concrete input and output values of type Value. It also has a timeout after which it is canceled.

- The stereotype Reference refers to other stories which can be called within a test story.
- The stereotype Assertion allows for defining assertions for computing the test verdict. The expression whether a testcase passes (pass), fails (fail), is inconclusive (inconclusive), or is an exception (exception), e.g. for negative tests.
- The stereotype ParallelTask allows for executing the same flow a specific number of times (tasknumber) in parallel.

For combining test stories to a sequence, the following stereotypes have been defined. The stereotype Testsequence encapsulates a sequence of SequenceElement elements each having an initial setup state (setup), a final teardown state (teardown), a test story to execute (story), a table with test data (data) containing objects for all input values and result values of assertions, a timeout (timeout) and an arbitration (arbitration) for computing global verdicts [8]. The setup and teardown states are itself sequences of service calls, especially of calls to ConfigurationService elements.

The test stories and the test sequences are depicted as activity diagrams with its stereotypes extending the UML metaclasses Activity and Action.

For the ticket reservation system example defined above, a test story and a test sequence may be defined as in Figures 7 and 8.

![Diagram of Test Stories](image)

Fig. 7. Test story for Login service and Register service

A basic test story for testing the services Login and Register is depicted in 7. First, the test story tries to invoke the service Login. If the return value provides an concrete input and output values of type Value. It also has a timeout after which it is canceled.

- The stereotype Reference refers to other stories which can be called within a test story.
- The stereotype Assertion allows for defining assertions for computing the test verdict. The expression whether a testcase passes (pass), fails (fail), is inconclusive (inconclusive), or is an exception (exception), e.g. for negative tests.
- The stereotype ParallelTask allows for executing the same flow a specific number of times (tasknumber) in parallel.
object of type Customer then Assertion1 is evaluated, else the service Register is invoked and Assertion2 is evaluated. In a test sequence testing two stories, Login and LoginRegister, both with a test table, a timeout value and an arbitration, the first passing if all tests pass and the second if more than 70% of all tests pass. The test table Table2 for the SequenceElement LoginRegister contains the columns VName, VPassword for the input parameters and column Result for the assertions, e.g. the headings and one data row in Table2 looks as follows.

In the table above name_1, password_1 and success_1 are object ids referring to elements in an object repository.

IV. TEST CODE GENERATION AND RELATED ACTIVITIES

This section is about the test code generation and all related activities as depicted in Figure 3, i.e. consistency and completeness checks for ensuring quality of the underlying models, the adapter implementation or generation to access concrete services and test execution for the execution of the test code. We also link the test code generation to our system architecture.

A. Consistency and coverage checks

The test model is designed manually. Therefore automatic consistency and coverage checks are needed to guarantee the quality of the test model. As previously mentioned, the consistency and coverage checks are defined in OCL.

Consistency is the extent to which no conflicting information is contained. In our respect, consistency rules ensure that e.g. each test story can be mapped to an executable Java class. In Listing 1 an OCL constraint hasServiceCall and its helpers isTeststory and isServicecall are defined which guarantee that every service is called in at least one test story.

context Element
def isTeststory(): Boolean =
  self.getAppliedStereotypes().name
  ->includes('Teststory')

def isServicecall(): Boolean =
  self.getAppliedStereotypes().name
  ->includes('Servicecall')

context Activity
def hasServicecall:
  hasServiceCall(): Boolean =
  self.isTeststory() implies
  self.allOwnedElements()->
  select(a | a.isServicecall())->
  ->notEmpty()

In the testing domain, a special and very important kind of completeness between a system and a test model is coverage. Coverage rules ensure that the set of test cases specified in the test model satisfy some kind of completeness criterion with respect to the set of all executions specified in the system model. In Listing 2 an OCL constraint allServicesCoverage and its helpers allSystemServices and allServicecallServices are defined to test whether all services are invoked in at least one teststory.

class Package
def allSystemServices:
  allSystemServices():
    Set(TTSProfile::SystemService) =
    self.allOwnedElements() ->select(a | a.isSystemService()).
    oclAsType(Classifier)
    .extension_SystemService
    ->any(true) ->asSet()

def allServicecallServices:
  allServiceCallServices(): Set(String) =
  Action.allInstances().
  .extension_ServiceCall
  ->any(true).service.name ->asSet()

def allServicesCoverage:
  allServicesCoverage(): Boolean =
  allSystemServices()->
  ->forAll(s | allServiceCallServices()->
  ->includes(s.name))

Listing 2. Coverage Example

The performance and scalability of our OCL evaluation technique has already been investigated. Consistency and completeness of the test model have a high impact on the quality of the generated Java code. The test model has to be consistent before the test code generation procedure can be invoked.

B. Adapter

In three approaches for connecting abstract test cases, i.e. test stories in our respect, and the SUT are distinguished, namely an adaptation approach, a transformation approach and a mixed approach. We follow a mixed approach, by implementing or generating adapters called by test code which has itself been generated by a transformation. This raises the abstraction level for invoking services in the test code classes and makes the generation of these classes easier.

The role of an adapter in our framework corresponds to the role of an adapter pattern as defined in [12]. The core idea of an adapter is to provide an interface realization for that classes with incompatible interfaces can communicate. An adapter provides an implementation to a client, i.e. the test code, to use the services of a class whose interface is unknown to the client itself. Adapter implementations enable the test controller to communicate with arbitrary SUT technologies. Our adaptation approach is based on an abstract adapter and on concrete adapter implementations which extend the abstract

<table>
<thead>
<tr>
<th>id</th>
<th>VName</th>
<th>VPassword</th>
<th>Result</th>
<th>allServicecallServices</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_1</td>
<td>name_1</td>
<td>password_1</td>
<td>success_1</td>
<td></td>
</tr>
</tbody>
</table>
adapter and which can be automatically generated in many situations. Semantically, an adapter corresponds to an actor calling services on the SUT.

1) Abstract Adapter: The abstract adapter is the abstract superclass of all adapter classes implementing common methods for all concrete adapter implementations. The abstract adapter provides a method `invoke` in pseudo code printed in Listing 3 for calling services on the SUT. Hereby the abstract adapter enables the test controller to abstract from any underlying SUT. The controller can invoke any method possibly provided by the SUT. Furthermore the abstract adapter makes the test controller independent of the underlying communication technique. Concrete adapter implementations can use Java RMI, web services, sockets or any other communication protocol. It is also possible to start the SUT as child process.

The implementation of the abstract adapter is based on reflection technology. It has the service name, a timeout value and an array of input objects as parameters. The method `invoke` on a concrete adapter is then called in the test code of a test story for accessing services. Therefore there has to be a method for all callable services within a concrete adapter. Otherwise the test case reports an invocation exception. Every service call is executed asynchronously in a separate thread which tries to call the service and waits for a response or until the timeout is fully consumed. That enables the test controller to handle multiple requests in parallel and to run load tests.

Listing 3. The invoke method of the abstract adapter

```java
public void invoke
(serviceName, timeout, parameters) {
    Set methods = adapter.getAllMethods();
    for (method : methods) {
        if (method.getName() = serviceName)
            method.execute(parameters, timeout)
            break;
        end if;
    end for;
}
```

2) Adapter Implementation: The concrete adapter implementation for accessing the SUT can be generated or implemented automatically depending on the service technology. For every possible actor interacting with the SUT, the corresponding adapter has to be implemented providing adaptations to all the services callable by the actor.

All adaptation methods are declared as private to prevent from violating the adapter concepts. In fact, the SUT must only be accessed abstractly via the `invoke` method and not by directly calling any of the adaptation methods. The implemented adapter extends the abstract adapter. This enables the abstract adapter to invoke the concrete methods for accessing the SUT. An exemplary definition of the adaptation method for the service `Login` is printed in Listing 4.

Listing 4. Concrete adapter implementation

```java
public class CustomerAdapter extends AbstractAdapter {

    private CustomerLogin(String PName, String PPassword) {
        return TicketReservationStub.login(PName, PPassword);
    }

    . . .
}
```

Listing 4 defines the adapter implementation for the actor `Customer`. Its service `Login` can be called by the statement `customer.invoke("Login", timeout, args);` in a test code class. Here, the communication is done via web services as implemented in the `TicketReservationStub`.

Depending on the technology, adapters for accessing services can be generated based on the system model and SUT artifacts. The system model provides the service interfaces, their calling actors and domain classes. For web service technology, a WSDL file provides the technical interfaces, data types and bindings. We can then automatically generate adapter implementations from the system model and a WSDL file. The services in both artifacts are linked via unique service names.

If the SUT on the one hand has a public interface, its adapters can mostly be generated using the system model and the interface file. Non-standalone systems can easily be integrated by generated adapter implementations. But also for standalone SUTs like application servers some adapter implementations are available. The integration of CORBA, RMI or web services are just example solutions.

On the other hand, if a SUT does not provide a public interface, an adapter implementation is needed in order to connect the test controller to the SUT. This might have to be implemented by hand, or using one of the client abstract adapters. Those adapters which are normally implemented manually encapsulate the network communication between the test controller and the SUT.

C. Test code generation

The test code generation itself can be implemented straightforward in our framework. Our test code generator is based on a template and metamodel approach [26], i.e. defining model–to–text transformation rules for metamodel elements in a template file. More precisely, each node in the diagram is visited by following the references defined in the model and executing the appropriate template defined by the metaclass of the node. Therefore our approach is also very scalable and allows to modify the code generator easily and efficiently.

At first the code generator retrieves all the activity nodes from a test story as list. Further on the initial node is searched and from that node on, the model is parsed by following the internal references until the final node is reached. For each specific node in the model, e.g. of type `ServiceCall` or `Assertion`, a custom template is called and the according code is written to a Java source file. The generator has to follow the references because the list of nodes retrieved from the model file (usually XMI [20]) is serialized and does not
represent the control flow as visualized in the diagram of the test story.

D. Test execution

In Figure 9, the various artifacts, i.e. TestSequence, TestCode, Adapters, SUT and TestData necessary for test execution and their relationships are depicted.

At first, the execution engine parses the test sequence, which defines the ordering in which the test stories, more exactly the generated Java classes corresponding to test stories, are executed with concrete test data. Internally, for each declared task in the workflow, a task object is created, which further on is processed by the execution engine. The task object contains a reference to the concrete test code file that has been generated from a test story by a compiler. The story itself makes use of the adapter by calling the send method, declared in the abstract adapter and realized in the concrete adapter needed to access the SUT. For each service call a task is created which asynchronously invokes the service on the SUT. At the same time, a thread is started to check the timeout constraint. Now, either the service invocation completes within the timeout and the result can be received by calling deliver, or otherwise, a timeout exception is thrown. If, due to some illicit or erroneous code editing of the adapter, the service adaption method cannot be invoked or found, the adapter will throw an exception, which later on will be evaluated by the reporting component of the framework.

The activity Test Execution in Figure 9 is executed by the TestController component, depicted in Figure 10. We discuss the architecture of the framework in greater detail in the next section.

E. Architecture

In this section we give an overview of the architecture following [11] and explain how it is related to the test code generation. The architecture of our framework is depicted in Figure 10.

The TSRepository stores and versions all object nodes depicted in Figure 2.
- SystemModel holds the system model,
- TestModel holds the test model as a collection of test stories,
- Datapool holds the test tables corresponding to test stories and the initial states for executing test stories,
- TestSequence holds sequences of test stories for direct execution,
- TestLog holds the log files generated within test runs,
- TestReport holds test reports.

The TSChecker implements model checks within and between the test model and the system model as explained in Section IV-A. The TSCompiler translates each test story into TestCode, i.e. Java code in our implementation which contains one top-level story method parameterized with the input parameters during test execution. In the test code Adapter objects for accessing the available services are instantiated. The adapters which may be generated automatically or manually encapsulate the communication with the available services which may be implemented e.g. in CORBA or web service technology and are able to handle synchronous and asynchronous service calls. The SUT provides executable services and may additionally provide system services for testing, e.g. for resetting the internal database. The TSTestController executes a sequence of test stories. For every test story an initial state is set up and the stories top-level method is invoked for every line of its corresponding data table. The TestController has a TimeLogger component supporting timeout monitoring needed for handling asynchronous service calls and a component for event handling (EventHandler) processing the events which occur during the test execution such as errors, timeouts or test verdicts. The Controller generates a test log for one test story execution. Test logs are used by the ReportManager to produce test reports.

We have implemented our framework based on the Eclipse platform. Therein the repository component corresponds to a versioned Eclipse workspace where all artifacts are stored as files. The test controller itself has been implemented in Java. Based on that the test code and the adapters are also written in Java. The models are in EMF UML2 [11] and for the test code generation Xpand [4] has been used. Our implementation is available online as Eclipse Update Site [3].
V. Related Work

We have defined a framework for model-driven system testing of service oriented systems, its UML profile and the test code generation process. Many approaches to the generation of executable tests from annotated non-UML models [15] and UML models [14] have been developed. But only a few approaches define separate test models based on a UML profile as metamodel such as [13] or the most prominent and standardized approach based on the UML Testing Profile [18]. According to [5] system level testing is based on the formalization of use cases using interactions and their mapping to UML Testing Profile (UTP) test specifications.

The aim of our approach is a business oriented view on the test models. Test models can be created together with domain experts in a test-driven way even before the system model has been completed. Therefore the system and test model share concepts, are very abstract – not considering the test architecture and the underlying technology – and support the tabular description of data.

The generation of executable test code from UTP specifications has already been addressed in the specification itself [18] which provides mappings to JUnit [2] and TTCN-3 [27]. In [28] transformation rules are defined mapping UTP to TTCN–3 which is afterwards compiled to executable test code in Java. In our approach, the test models are directly transformed to test code in Java which is also the language for adapter implementation. Due to traceability of services this emphasizes the feedback cycle to the model elements.

VI. Summary and Future Work

We have presented a novel approach for test code generation of service oriented systems. The validation ensures the quality of the system and test model on an implementation independent level and allows for generating high quality test code directly from test models. Adapters provide flexible ways to access executable services on a system under test.

Our approach is the first one that makes requirements for service oriented systems executable by linking them via system models, test models and adapters to executable system services. We have developed the framework based on the needs of our industrial partners. Future improvements include the automatic generation of test data and the extension of the system model by a variety of sub–models and specifications such as service orchestrations, object life cycles or rules for specifying constraints on the order of the service execution. We also plan to integrate the whole test cycle by annotating the test results into the system and test model and defining patterns and the technical infrastructure for testing non–functional properties.

Acknowledgement

The research herein is partially conducted within the Telling TestStories project funded by TransIT (www.transit.ac.at). Moreover part of the research is conducted within the competence network Softnet Austria (www.soft-net.at) funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirisches Wirtschaftsförderungsgesellschaft mbH (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

References