

Testing Event-Driven Business Processes

Gerd Saurer, Josef Schiefer
Senactive IT Dienstleistungs GmbH, Vienna, Austria
Email: {gerd.saurer, josef.schiefer}@senactive.com

Alexander Schatten
Institute for Software Technology and Interactive Systems, Vienna, Austria
Email: aschatt@ifs.tuwien.ac.at

Abstract—Today’s business climate requires you to constantly evolve IT strategies responding to new opportunities or threats. While the fundamentals of IT - reliability, availability, security and manageability – are still crucial, rapid results are mandatory for business success. These business challenges can be solved by acting with agility – striking the proper balance between the introduction of leading-edge technology and the pragmatic application of IT. In this paper, we introduce a testing framework for business solutions dealing with complex and dynamic IT environments. Our framework supports test-driven development which facilitates an incremental construction of reliable business solutions that can be adapted to changes of a business environment easily. We compare our testing framework with model-driven development approaches and show how we applied it to an event-driven process management platform called SARI (Sense And Respond Infrastructure).

Index Terms—Testing, Test-Driven Development, Business Process Management, Business Process Engineering, Event Management.

I. INTRODUCTION

The emergence of e-business has dramatically changed the context in which information systems are used for providing a better service to customers. With the increased rate of change possible in e-business, it has become essential that business processes can be adapted according to the changing business environment more quickly than in the past. Static processes that cannot adapt to changing needs are a liability. Enterprises are scrutinizing the effectiveness of their business and IT operations to identify opportunities for greater efficiencies. Sense & Respond [5] and Event Management [1] have emerged as disciplines to enable enterprises to manage time-sensitive and dynamic business processes which are highly integrated with information systems.

A major challenge for systems in continuously changing business environments is to keep them robust and operational for critical business processes [14]. In this paper, we introduce a framework for testing the functionality of event-driven and time-sensitive business processes. With our testing framework we address the following issues:

- Validation of the quality and correctness of a process model
- Automated testing of business process execution within a distributed environment
- Testing side effects from modifications of a process model
- Detecting and diagnosing performance problems of process deployments
- Facilities for investigating discovered malfunctioned processing steps

Traditional business process management solutions use simulation for analyzing the quality of process models. Process simulation is a top-down approach for process reengineering programs and provides valuable insight in variations of process models. However, process simulation is only as good as the underlying assumptions for the simulation parameters. In this paper, we propose a testing framework which follows a bottom-up approach for gaining insight in existing business processes or business process solutions in development. Starting from an existing process model, we use our framework to validate it with tests bottom up. For the testing, we not only consider the description of the process model, but also runtime aspects as

- Verifying the outcomes of processing steps by actually invoking services and components,
- Measuring the throughput of processing engines,
- Testing the robustness of the process management system during failover scenarios,
- Testing changes for configuration settings such as parameters for processing engine, business rules, or resource assignments.
- Creating simulation tools and mockup strategy to replace complex business environments like SAP during test scenarios

The remainder of this paper is organized as follows. In Section II, we discuss test-driven development of business solutions and compare it with model-driven development. Section III describes problems and issues of process testing. In Sections IV – VI, we give an overview of our testing framework for testing Sense &

Respond processes, show the testing environment we used, and demonstrate a testing scenario with our proposed framework. In Section VII, we describe some extensions of our testing framework. Finally, with Section VIII, we conclude our paper and give an outlook for future work.

II. TEST-DRIVEN DEVELOPMENT FOR BUSINESS SOLUTIONS

Looking back the way software has been developed – particularly considering the last years – dramatic changes can be observed. Starting 2001 with the “Agile Manifesto” [6] agile software developing processes became commonly accepted development strategies, hence having significant impact on the way software projects are planned, managed and implemented nowadays. Agile methodologies like XP [2], Scrum [3] or Crystal [4], which formed the foundation for the Manifesto, strongly emphasize the role of software tests [17] by integrating them early into the software development instead of just “accepting” it as part of software production at the end of iteration or the whole process. Parallel with the new software development processes, also new software architectures appeared on the scene, driven by the widespread usage of Internet related technologies. Particularly noteworthy are so-called Service Oriented Architectures (SOA) [9]. SOA has gained popularity as new software engineering paradigm and arose from the necessity of creating components providing clearly defined small pieces of functionality that later on can be assembled into complex (usually distributed) applications. Splitting up complex applications into smaller components and services aims not only at increased reusability but should also leverage composing and refactoring (business) processes. Evidently, this approach fit well for fast changing environments, where business systems have to continuously adapt to new business needs.

Service-oriented systems usually don’t try to substitute systems that are already installed in companies; much

more they try to provide an infrastructure that enables to utilize older applications as services. Taking all this into account, it appears to be obvious, that such systems often result in an operational environment with complex dynamical behavior which is hard to predict and evaluate [13]. Hence (automatic) testing of an infrastructure is a difficult issue. An additional challenge for such environments is, to bring the requirements defined by a user or several stake holders into a model that can be understood from technical architects, developers and of course testers (which is the main focus of this work).

Using models to design systems help us understanding complex problems and its potential solutions through abstraction. Selic proposes in [12] the following five characteristics of engineering models for building complex systems (see Table I).

With our testing framework, we want to address these characteristics in different ways as traditional model-driven development approaches. Figure 1 shows a comparison of a model-driven and test-driven development of business solutions.

TABLE I.
CHARACTERISTICS OF ENGINEERING MODELS

Abstraction: A model is always a reduced rendering of the system that it represents. By removing or hiding detail that is irrelevant for a given viewpoint, it lets us understand the essence more easily.
Understandability: Is a direct function of the expressiveness of the modeling form used (expressiveness is the capacity to convey a complex idea with little direct information). A good model provides a shortcut by reducing the amount of intellectual effort required for understanding.
Accuracy: A model must provide a true-to-life representation of the modeled system’s features of interest.
Predictive: You should be able to use a model to correctly predict the modeled system’s interesting but non-obvious properties, either through experimentation (such as by executing a model on a computer) or through some type of formal analysis, which depends greatly on the model’s accuracy and modeling form.
Inexpensive: It must be significantly cheaper to construct and analyze than the modeled system.

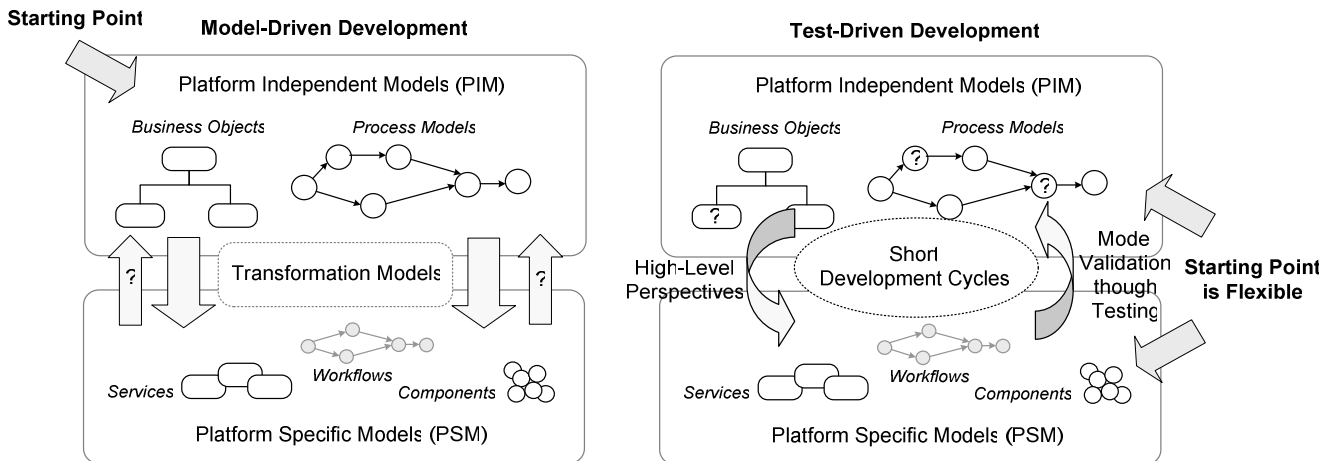


Figure 1. Model-Driven Development vs. Test-Driven Development

The focal point of model-driven development is to describe business solutions ideally with models without any consideration of platform specific details (= Platform Independent Models) and automating the transformation of these models into executable software components. Model-driven development holds promise of being the first true generational leap in software development since the introduction of the compiler.

One of the biggest challenges for model-driven development is the transformation between the platform independent and platform form specific models. Roundtrips between these models are very difficult and sometimes it is even not feasible to maintain the traceability between model artifacts. In particular, the traceability links from platform specific models back to platform independent models are often missing which makes it difficult to create a true-to-life representation of the modeled system. Model-driven development approaches often only predict how platform independent models will map into an execution environment.

With our testing framework, we make this process also reversible and further consider how an existing execution environment conforms to a platform independent model. Test-driven development does not assume that all platform specific models (which include for instance also code) are generated in a top-down fashion. A software engineer can decide whether to start with a model which provides a high-level perspective of the system, or with platform specific artifacts such as software components.

A software engineer can use software tests to validate the behavior of an existing system with platform dependent and independent models. Although it would be desirable to validate all details of a platform independent model, this is often too time consuming and not cost-efficient. Software tests facilitate users to test out various aspects of the system behavior that are relevant to them. As we will show in the next sections, they can do this a various abstraction levels.

III. RATIONAL FOR PROCESS TESTING

Many business process solutions are nowadays deployed in production systems with little or no testing. One of the reasons why process solutions are not tested thoroughly is the fact that testing business process solutions is challenging. Business process solutions are complex, distributed, heterogeneous and asynchronous in nature and there are very few tools available to help in these testing efforts. In this paper, we want to discuss the challenges of functionally testing business processes and propose a testing framework to tackle the inherent complexities. With our testing framework we address the following issues:

- Verification of the quality and correctness of process behavior when executing the business process
- Transforming process requirements into test cases
- Testing side effects from modifications of a process model

- Detecting and diagnosing performance problems of process deployments
- Facilities for investigating discovered malfunctioned processing step

If we take all this into account we see the need for testers, but also developers to have test framework to execute business processes in a test environment where configurations and parts of a business process can be easily changed and tested against requirements.

A major challenge for process management environments is, to bring the requirements defined by a user or several stake holders into a valuable and working business process solution. Figure 2 shows the typical stages for such a process. In the following we want to discuss the stages of a business process lifecycle and the role of testing within these stages.

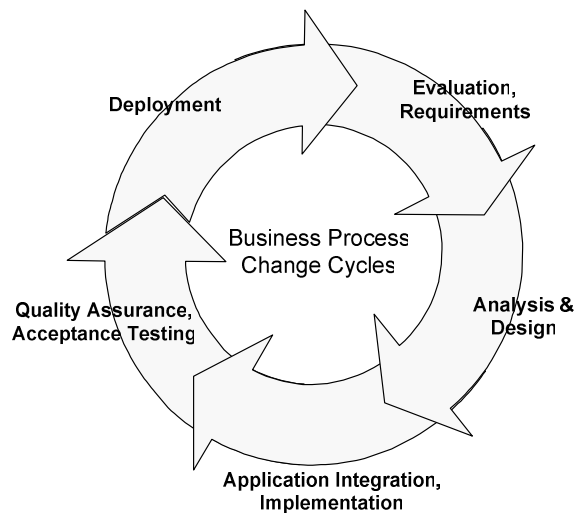


Figure 2. Business Process Lifecycle

Evaluation, Requirements. In order to capture the requirements for a process, we need to understand it. This usually involves interviewing people with expert knowledge about the process. When sufficient knowledge about the process is obtained, it can be used to create a new process model or to adapt an existing one. Evaluation of an existing process model is necessary to determine if the process model actually represents the intended process. The outcome of this stage is a description and evaluation of an existing business process as well as new requirements for improving it.

Analysis & Design. When changes are proposed to business processes in order to improve performance, the projected improvements can be simulated and optimized artificially using business process analysis. Analysis of a modeled process is performed to identify the process strengths and weaknesses, and determine if the process should be redesigned or proceed to its implementation.

Application Integration. Implementation: The next step is to operate business processes by integrating the underlying applications, thereby streamlining process execution and facilitating real-time information sharing.

In order to do so, very often plumbing code and the implementation of additional software components is required.

Quality Assurance. Acceptance Testing: Involves a planned and systematic process which includes all actions necessary to provide adequate confidence that the business process implementation conforms to established technical and business requirements. Acceptance tests determine whether or not a business process satisfies its acceptance criteria, such as a certain predictable process behavior or performance goals. They generally take the form of a suite of tests designed to be run on the target systems for executing the business process. Each individual test exercises a particular operating condition of the user's environment, or a function of the process, known as a case or business scenario.

Deployment. A deployment is the distribution of a business process solution to many users. The deployment process puts a business process change into production and makes sure that it can most appropriately support the users with their business activities.

Process change typically happens in cycles with some form of the above mentioned stages. By passing through the stages, new knowledge is gained about a business process. During the lifecycles, it is important that old knowledge and obsolete requirements from business processes are replaced by process innovations. In a complex environment, it is nearly impossible to ensure a smooth and stable transition after implementing process innovations and adaptations without automated process testing.

IV. TESTING FRAMEWORK FOR SENSE AND RESPOND

Before we introduce our testing framework, we want to briefly introduce the target system. The testing framework is tailored to an agile business process

management platform called SARI (Sense And Respond Infrastructure). We want to briefly give an overview of the SARI system in order to better understand how the testing framework was applied.

A. Brief Introduction of SARI

SARI's main objective is to help organizations to control and monitor their business processes and IT systems in order to respond to business situations or exceptions with minimal latency. SARI is able to continuously receive, process and augment events from various source systems, and transforms in near real-time these events into performance indicators and intelligent business actions. SARI is a distributed, scalable platform, which allows modeling and executing various forms of Sense and Respond processes (see Figure 3).

The data processing is controlled by "Sense & Response loops" which can be divided into 5 stages. During the "Sense" stage events are continuously captured and transmitted to the SARI system where they are initially unified before the actual data processing starts. In the "Interpret" stage, the captured events are transformed into meaningful business information, such as key performance indicators, business situations and exceptions. The next stage is the "Analysis" of the generated business information in order to determine root causes for business situations and exceptions. The analysis has the goal to find the best possibilities to improve the current situation of the enterprise. The analysis also allows predicting the business performance and risks for making changes to the business environment. In the following "Decide" stage, the best option for improving the current business situations and determines the most appropriate action for a response to the business environment is chosen. Finally, in the "Respond" stage, the decision made has to be

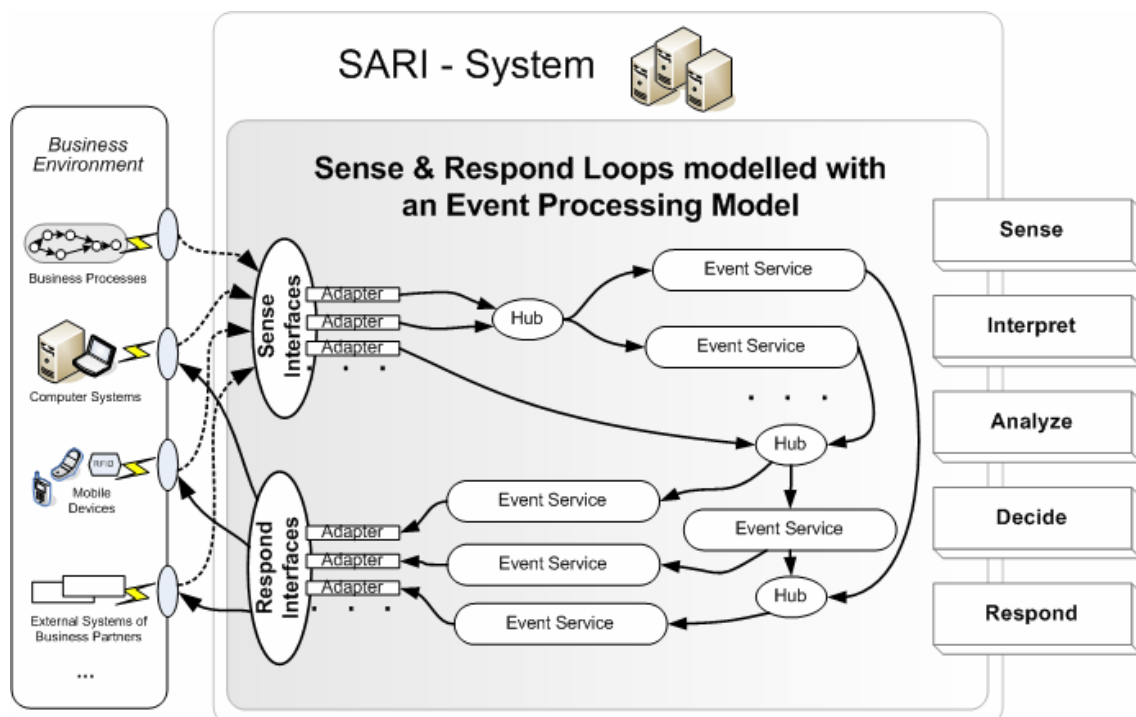


Figure 3. SARI Sense & Respond Loops [11]

implemented in the business environment by communicating the decision as a command or suggestion (e.g. by e-mail), or by directly adapting and reconfiguring business processes and IT systems. During the processing in the 5 stages, business information is continuously generated and decisions are made to which a response follows. The response has an effect on the source systems (from which SARI originally might have received events) and consequently also on the performance and the success of the organization.

Using the SOA approach, we model in SARI Sense & Respond loops with a pool of services and establish the infrastructure that enables a robust communication and interaction between them. SARI uses an event processing model (EPM) for modeling Sense & Respond loops. Similar to a construction kit, the EPM offers various building blocks which can be used to construct a Sense & Respond loop. Dependent on the requirements and the business problem, these building blocks can be flexibly conjoined or disconnected. Links between the building blocks represent a flow of events from one service to the next. Typical building blocks are event services for various data processing tasks in each stage. For more details please see [11].

B. Overview of the Testing Framework

In this section, we introduce a testing framework for Sense & Respond which was tailored to the SARI system. Our testing framework includes classes of tests with different abstraction levels (see Figure 4).

Starting with “Code & Unit Tests” at the lowest abstraction level, we use a “bottom-up” approach for testing a wide range of characteristics and functionality of the system. This is different to the “top-down” testing approaches (e.g. used for model-driven development [12]) which use high-level models and the customer requirements as starting point for the testing. The problem with “top-down” approaches is that a complete model and sufficient details for customer requirements have to be available in order to start the testing. On the other hand, with a “bottom-up” approach, we first test available parts of the system and we work up the ladder

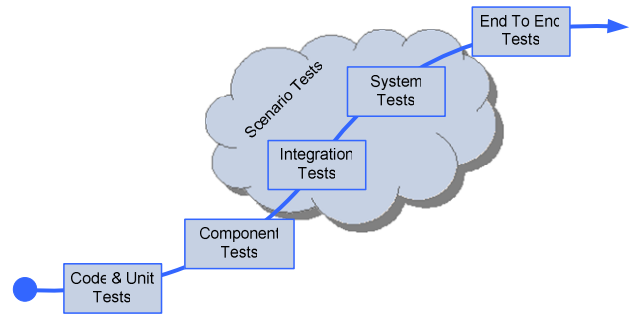


Figure 4. SARI Framework Tests

until we finally reach the end-to-end tests for testing a full business process. With each new abstraction level, the complexity for tests is increasing since the scope of the test scenarios is expanding [16].

Code & Unit Tests. These are the foundation of our testing framework and ensure that each implementation unit of the application works correctly. Usually, an implementation unit consists of a class. The aim of unit tests is to ensure the correct behavior and results of implemented functionality on code level. Unit tests facilitate significantly the code changes, since they are the “insurance” for a developer that a code is working correctly. If a unit test doesn’t continue to work, a developer immediately knows that a change within the system “broke” the code base.

Component Tests. SARI is a modular system and is decomposed into functional or logical components with well defined interfaces used for communication across the components. Examples for such components are Event Adapters and Event Services (see also Figure 1). Component tests only consider the interfaces of components and test typical communication scenarios with these interfaces.

Integration Tests. The next level is to test the integration of different parts of the system. At this level of abstraction, we can observe significantly higher complexity of the test cases, due to the possible

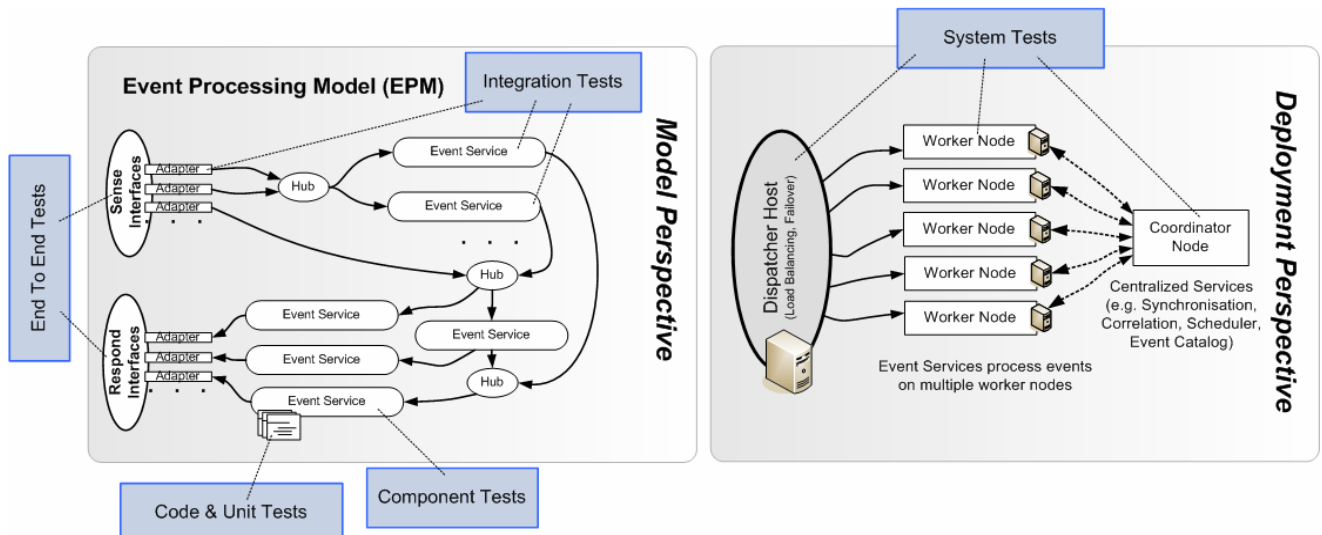


Figure 5. SARI Testing Environment

interactions of a larger number of components (which is typical for SOA).

System Tests. The previous tests focused on testing the functionality of components their interactions, however didn't consider the operational environment in which they are executed. System tests close this gap and further consider the deployment environment, the target platform as well as involved external system that the SARI system is dependent on (e.g. a database management system or queuing system).

End-To-End Tests. End-to-end tests consider complete business cases [15]. The goal is to guarantee a proper integration with other business systems (e.g. an ERP system) besides the SARI system. All services and applications of the SARI system as well as of other business systems are taken into consideration.

C. Test Fixtures and Mock

There are two key concepts that the SARI framework makes heavy use of, namely test fixtures and mocks. In the following we want to briefly introduce these concepts. Test fixtures are a skeleton for defining unit tests and provide an environment that allows the unit tests to perform its job properly. Any part of the environment needed by a unit test besides the testing framework itself is considered a test fixture. Figure 6 shows a typical test fixture. A test fixture is used by testers to describe and implement test scenarios. Test fixtures help to mount any required system functionality or data into the test environment, such as data from a database or proxies to real system services. Sometimes, it is time-consuming to include real services into a test scenario and dummy placeholders are sufficient to perform the tests. Such placeholders are also called "mocks". The test fixture contains a list of tests which 1) initialize the test environment, 2) invoke the tested unit (such as a service) and 3) assert the results. The last step is the most crucial one since the assertions check the current behavior of the tested unit.

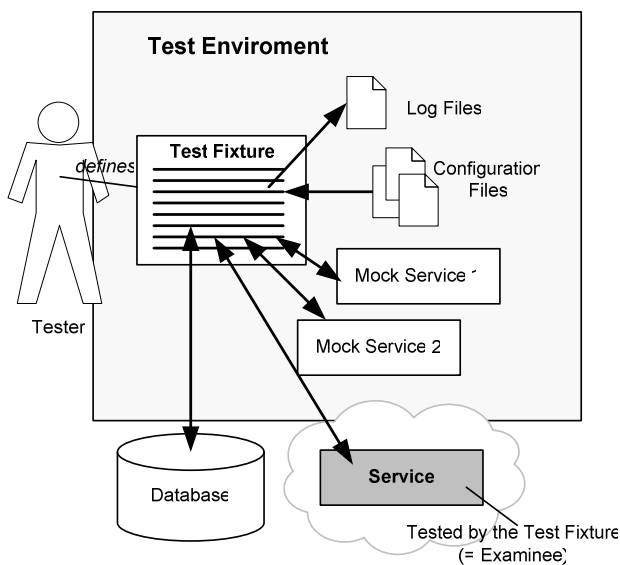


Figure 6. Test Fixture for a Service

V. SARI TESTING ENVIRONMENT

In this section, we introduce the testing environment for the SARI testing framework. The idea of the sense and response paradigm is to capture business events, and build processing steps (following business processes) that sense situations relevant to the business. If such a "situation" takes place the system intelligently responds.

As shown in the previous sections, SARI uses a modular approach for constructing Sense & Respond loops. This is achieved by using an EPM, which allows flexibly conjoining and disconnecting elements (e.g. event services) of the model, dependent on the requirements and the business problem. Links between the elements represent a flow of events from one service to the next. Figure 5 shows an EPM and the deployment environment of SARI in context of our testing framework. Table II summarizes some elements of the SARI system that we want to use later in our discussion.

TABLE II.
ELEMENTS OF AN EVENT PROCESSING MAP (EPM)

Event Service: A component for processing events. Event service have input ports and output ports which describe the interfaces for events that can be processed and that are generated as result of the event processing.
Adapter: Component for receive an event from or respond to an external system.
Hub: A component for merging and splitting event streams.
Worker Node: A computation node which executes the event services of EPMs.
Coordinator Node: A computation node which coordinates the worker nodes in order to ensure a consistent and reliable event processing.

We used the testing framework NUnit [8] as a foundation for the code & unit tests, component tests, integration tests and system test. We extended the NUnit framework with comprehensive test fixture and mocks (as will be explained later) in order to support complex tests. For the end-to-end test a simulation module was developed.

Code & Unit Tests. In SARI, data is collected and received through adapters and continuously processed by event services. Each adapter and event service is a component that can consist of multiple classes. For instance, when implementing an event services for discovering event patterns based on ECA (Event/Condition/Action) sets, a set of classes for describing the ECA sets, a class for processing the ECA sets (the ECA engine), and an event service hosting the ECA engine needs to be implemented. Code and unit tests are fully supported by NUnit; hence an extension of this testing framework was not required at this level of testing.

Component Tests. Interfaces of a component and services are validated using component tests. In SARI interfaces are defined by event types describing the structure of specific event data. The event types are used for input- and output ports of event services and adapters with EPMs. E.g., when testing an event service, a tester

sends a set of events to the input ports of the event services and tests the result of the event processing by investigating the results at the output ports of the event services. Figure 7 illustrates this process. Within a test fixture a tester uses an event injector to send events to the input ports of the event service. During the event processing, the event service might use various kinds of system services (such as event correlation [10], transaction management, and queuing) which are also provided by mock objects. These mock objects are lightweight implementations of system services which results in shorter execution times for the component tests (the initialization of component tests typically occurs in less than a second). After the event processing, an event collector mock object is used to capture the events published on the output ports of the event service. The tester can use the collected events for assertions within the test fixture.

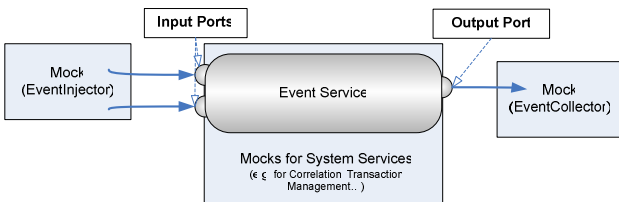


Figure 7. Testing Event Services

Integration Tests. An event service can depend on the results of other event services or adapters. The aim of integration tests is to consider a larger set of components and services within EPMs and test whether they work correctly in concert.

Figure 8 shows a set of dependent event services. Event injector mocks are used to send events to the event services from a test fixture. Similar as in component tests,

event collector mocks collect events from output ports of an event service. The collected events can be used for assertions within the test fixture. The difference to component tests is as follows:

- Testing of the outcome of an event processing problem including multiple event services
- Multiple event injectors and event collectors are allowed
- Additional map elements such as hubs (shown in Figure 8) or synchronization blocks
- Event services can use “external” parameters such as map parameters
- A comprehensive set of heavy-weight mocks for system services is available

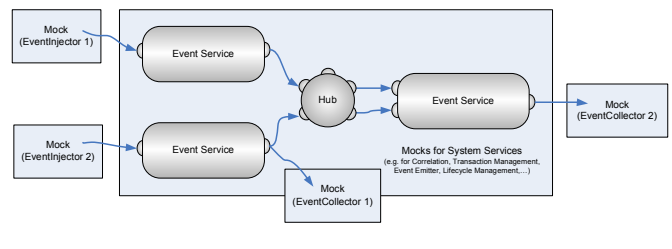


Figure 8. Integration Test for Event Services

With integration tests, users can test parts of an EPM or an EPM in its entirety. Integration tests include a larger set of mocks for system services. Some of the mocks are heavy-weight because they are built on top of SARI system services. Examples for such heavy-weight services are the correlation service, the event emitter for forwarding events within an EPM, and the lifecycle management for event services and adapters. The executing of an integration test takes about 5 to 10

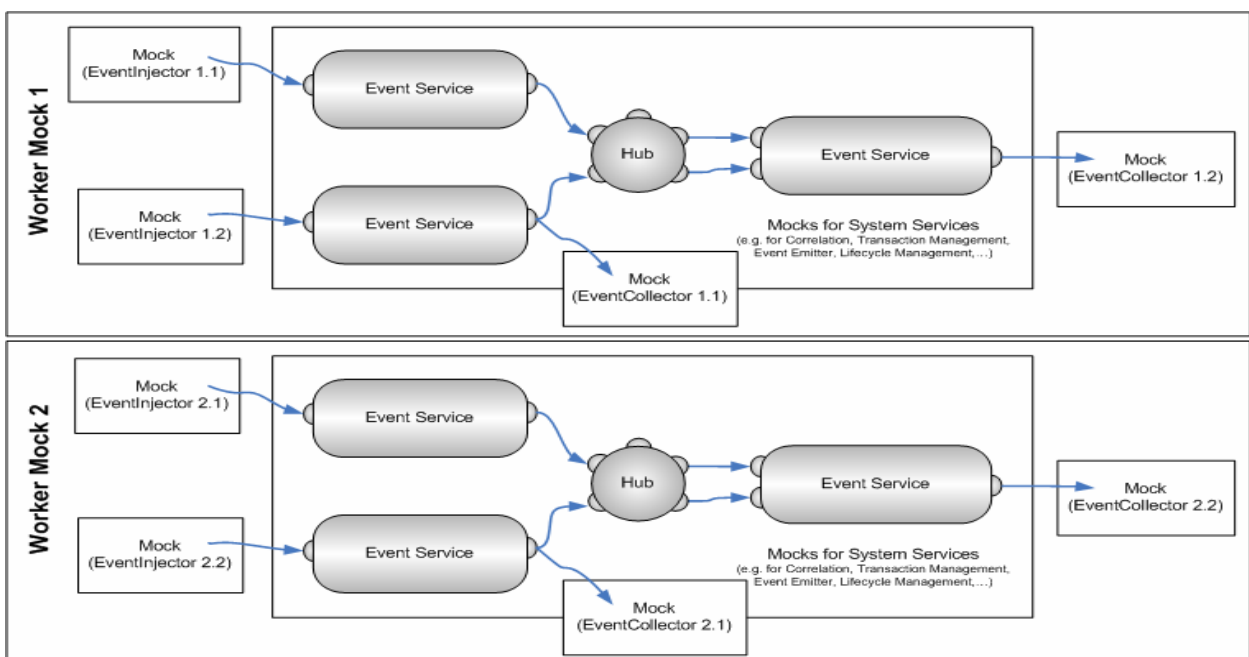


Figure 9. Testing Worker Nodes

seconds on our testing system, due to the longer instantiation times of the complex mocks for the system services.

System Tests. The SARI system is a distributed platform for event processing (see Figure 5 – deployment perspective). It includes various types of nodes that perform dedicated processing tasks. E.g., the dispatcher host performs load balancing, the worker nodes host the event services and process events and the coordinator node provides centralized services such as a timer service and synchronization. System tests allow to test the processing tasks within this complex distributed environment. Test fixtures for system tests can assert processing results for EPMs as well as system states (e.g. lifecycle state of an event services) in case of exceptional situations such as failures of a working unit. The repetition of such failure situations is extremely difficult in real-world scenarios, as the system state is difficult to preserve. Using test fixture that prepare the system state of worker nodes and other working units elegantly solve this problem since the tester has full control and access to the working units (e.g. worker nodes) and can initialize and verify the system state. Figure 9 shows an example of testing an EPM with two mocks for worker nodes.

End-To-End Tests. End-to-end tests are used to test complete sense & respond business scenarios in order to verify their correct behavior from an end user perspective. In order to facilitate the testing of end-to-end business scenarios, we developed a simulator for simulating event sequences of real-world scenarios. End users can either generate events by initiating state changes in the real business environment (which result in events that are sent to the SARI system) or by using a

simulator to generate the events for a test. These types of tests address the need to see test cases as requirements specification as described in Section III and differ in several ways from the other tests:

- They use the original process instead of parts of adapted processes and can therefore test full business scenarios instead of a limited set of components or process parts.
- They aim to address a higher abstraction level of the business process than any other type of tests.

Based on an existing business process model, a user defines simulation models for generating runtime data which is used as external trigger for the process, parameterises it for a configuration that should be tested (e.g. failover of services or special event handling) and makes some assertions the process and the information sent through the process should fulfil. After providing this information, the testing framework should be able to evaluate the assertions defined by the tester. If we adapt the end-to-end tests as soon as new information is provided (this may be an error that occurred in the production system or new requirements) we can define our tests as additional process requirements.

The following table provides some statistics about discovered defects in BPM solutions build with SARI. We measured the number of defects found and the actual costs for fixing the defects. Table III shows the distribution of the defects found and the costs for fixing them. The costs for fixing defects increase with the abstraction level of tests. The highest costs were caused by defects discovered by integration and end-to-end tests.

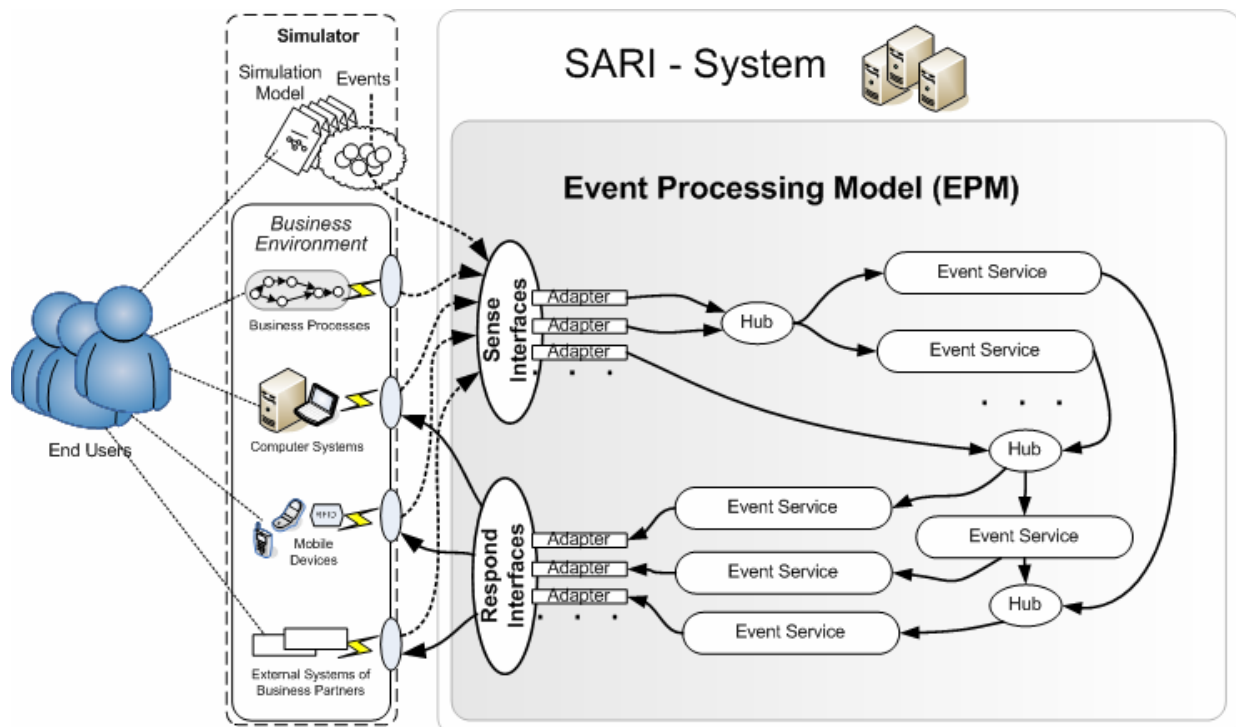


Figure 10. End-to-End Tests

TABLE III.
FOUND DEFECTS VERSUS COSTS OF DEFECTS

Level	% of Defects Found	Costs of Defects
Code & Unit Tests	37 %	15 %
Component Tests	11 %	8 %
Integration Tests	33 %	42 %
System Tests	5 %	11 %
End-To-End Tests	14 %	24 %

VI. EXAMPLE

In this section, we show a short example for integration tests. The starting point for our example is a simple EPM for calculating the cycle time for business activities (see Figure 11). In this EPM, the time difference between two events “Activity Started” and “Activity Completed” is calculated and stored in a database. Additionally simple exception handling is implemented. Table IV lists the components used in Figure 11.

To test the proper functionality of this EPM, several issues have to be taken into consideration:

SARI has to provide a testing environment for executing this application which includes in SARI’s case worker and coordinator nodes. The test environment must be able to instantiate and control these nodes and to simulate the event processing.

- Metadata about the event types “Activity Started” and “Activity Completed” (e.g. the list of attributes for the event type) as well as the configuration for adapters and services must be deployed from the testing environment.
- The EPM has to be controllable from a test fixture such as starting and stopping the event processing. Furthermore, testers must be able to send sample events to the EPM and evaluate the results of the event processing.
- A tester should be able to reduce the complexity of external systems (such as a SAP system) by mocking these event sources.
- The EPM has to be controllable from a test fixture such as starting and stopping of the event processing. Furthermore, testers must be able to send sample events to the EPM and evaluate the results of the

event processing.

- Simple definition of tests and analyzing of test results.
- A tester should be able to reduce the complexity of external systems (such as a SAP system) by mocking these event sources.

TABLE IV.
COMPONENTS FOR CYCLE TIME EXAMPLE

SAP Adapter. Is used to deliver “ <i>Activity Started</i> ” events from an SAP system into the system.
MQSeries Adapter. The “ <i>Activity Completed</i> ” events are received from a different data source – in the example from an MQSeries messaging system.
Cycle Time Calculation Service. Receives the “ <i>Activity Started</i> ” and “ <i>Activity Completed</i> ” events and calculates the cycle time. The event service has to identify the corresponding event pairs and use their timestamp for the calculation of the metric.
Cycle Time Storage Service. Stores the calculated cycle time metrics in a database.
Exception Handling Service. Handles exception events that can be sent by the other event services if an exception situation occurred (e.g. database was not available).

Figure 12 shows the interaction between the SARI system and the testing environment in which the tests of a test fixture are executed.

In the following, we will show snippets of a test script from a test fixture, which show how a tester can setup the testing environment and communicate with the SARI runtime environment.

During the start of a test, the Coordinator and Worker nodes are initialized and the configuration settings are provided. Every required configuration is stored as XML file and can be provided by testers within the test. The following implementation of a test in C# shows how a tester can provide a configuration:

```
// Initializes the test (part 1/4)
MyServiceTest() {
    Deployment.AddEventTypeLibrary("EventTypes.xml");
    Deployment.AddComponentLibrary("ComponentDescriptions.xml");
    Deployment.AddComponentLibrary("Resources.xml");
    ...
}
```

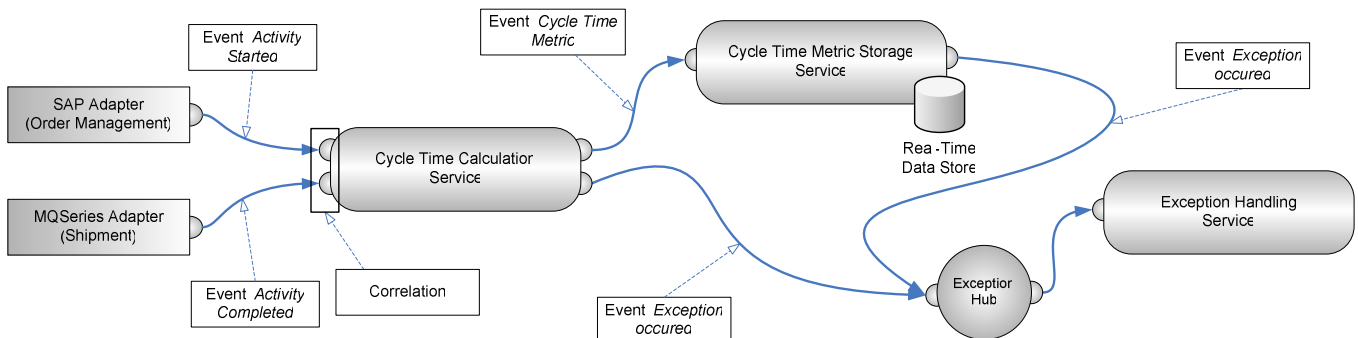


Figure 11. Cycle Time Example

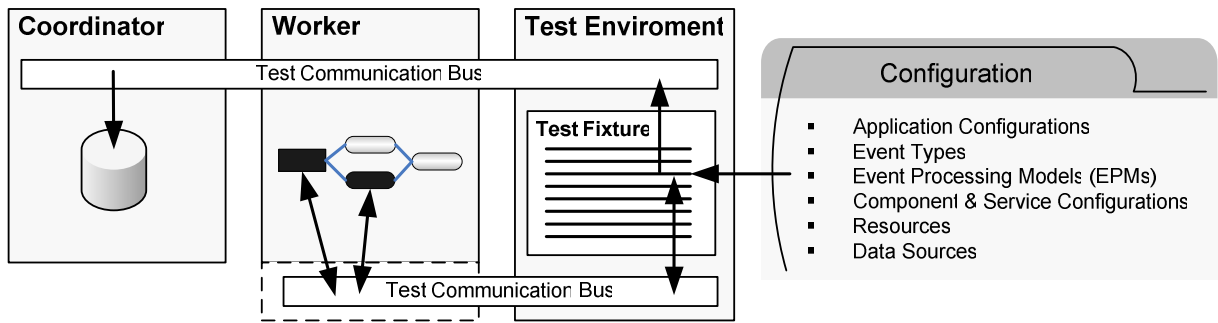


Figure 12. Test Communication Bus of SARI

In the next step, the tester adds the application description which contains also the EPM. After providing all configuration files and the application description, the application can be deployed and started on the Coordinator and Worker node.

```

...
// Deployment of an application (part 2/4)
Deployment.AddApplicationDescription("Application.xml");
Deployment.DeployAndStartApplication();
...

```

```

...
// Sending events to event injectors (part 3/4)
for (int i = 0; i < 100; i++)
{
    Event activityStarted = CreateEvent("Activity Started");
    TestCommunication.PublishEvents("EventInjector for
    SAP Adapter", activityStarted);
    Event activityCompleted = CreateEvent("Activity Completed");
    TestCommunication.PublishEvents("Event Injector for MQSeries
    Adapter", activityCompleted);
}
...

```

After these steps, the application is running and can be tested. For our example, we want to show how to run the test scenario without executing transactions in real business systems. By replacing the event adapters with Event Injectors, a tester can directly send events from a test fixture to the EPM. Furthermore, tester can add Event Collector services to the EPM in order to collect event that are generated during the event processing (see Figure 13).

After sending the events, the test fixture waits with `TestCommunication.WaitOnCallback()` until 100 "CycleTime Metric" events have been received in the Collector Event Service. After successfully waiting until the "Cycle Time Metric" events have been calculated and received, the collected events can be accessed within the test fixture and used for assertions.

```

...
// Wait for result and make assertions (part 4/4)
Assert.IsTrue(TestCommunication.WaitOnCallback(
    "CollectorServiceForCycleTimeMetrics", 100, Timeout));
List cycleTimeEvents = TestCommunication.GetCollectedEvents(
    "CollectorServiceForCycleTimeMetrics");
// Check whether the collected events in cycleTimeEvents are correct...
...
}

```

The Event Injectors (representing test adapters) can be used to publish events generated in the test fixture instead of executing a real transaction from a SAP system. Collector Services can be used for collecting event processing results. The following code fragments illustrate a test of the example process. The next code fragment sends 100 "Activity Started" and "Activity Completed" events to the Event Injectors in the EPM.

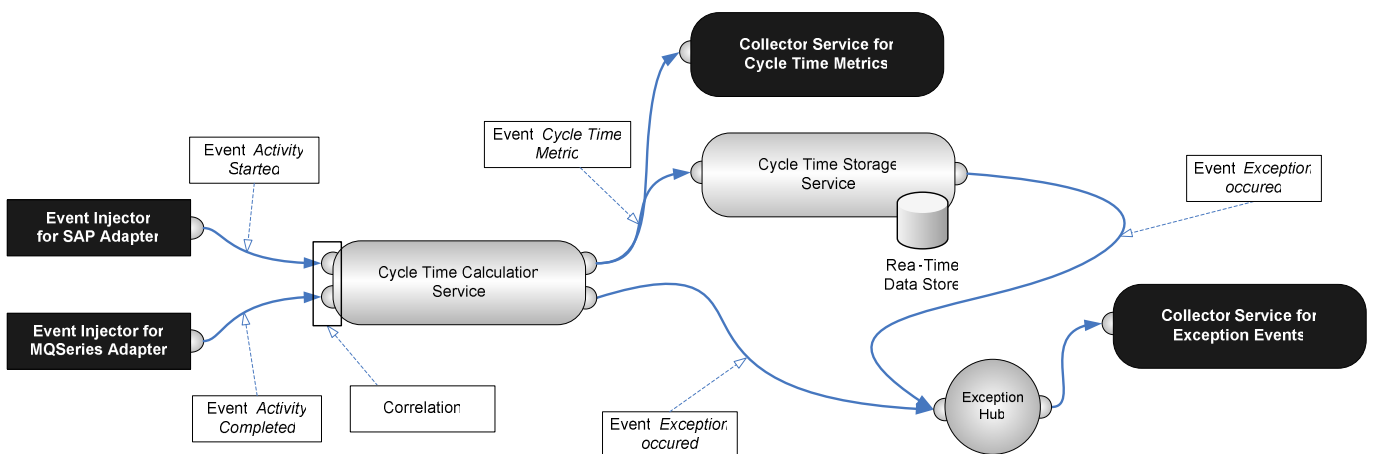


Figure 13. Cycle Time Example with Event Injectors and Event Collector Services

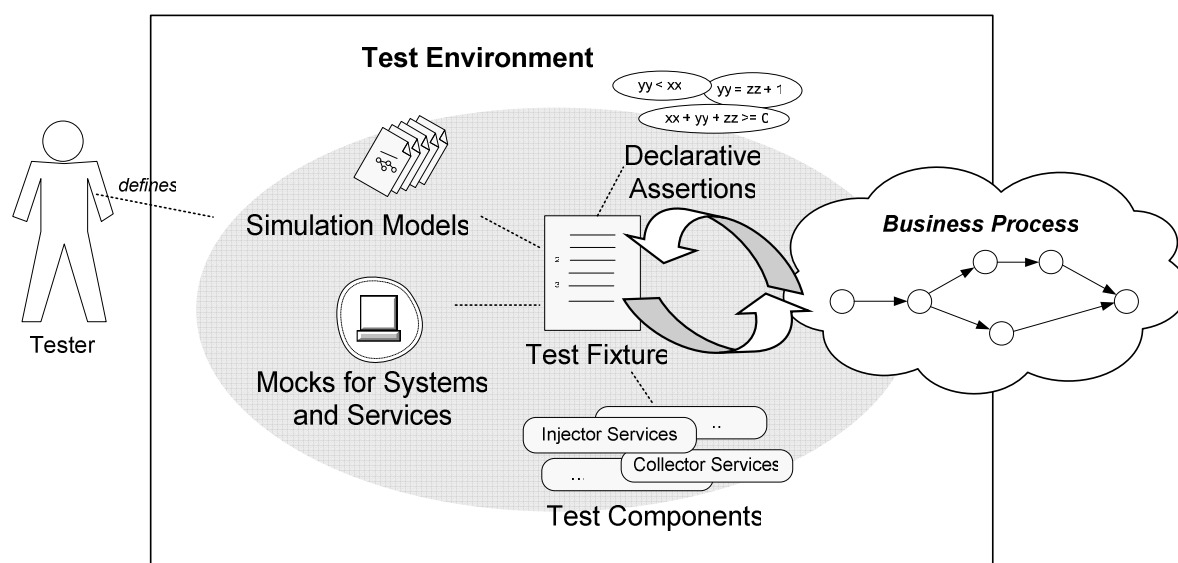


Figure 14. Extensions for Test Fixtures

VII. TESTING FRAMEWORK EXTENSIONS

After showing you an example of how we applied the testing framework to the SARI system we want to propose some further extensions which are part of future work. The goal of the extensions is to further facilitate the testing of business processes by replacing parts of a test script with declarations and models.

In order to achieve these goals, we extend our testing framework with additional artifacts which are used within a text fixture. Figure 14 provides an overview of these artifacts. The goal of these artifacts is to declaratively describe information for testing scenarios, which can be reused and embedded in tests and thereby avoid the writing and maintenance of complex test scripts.

Based on an existing business process model, a user should be able to define 1) *simulation models* for generating runtime data which is used as an external trigger for the process, 2) *mocks for system services* as long as the original system is not available or too complex to set up, 3) arbitrary *test components* which can be easily integrated within the testing framework and perform specific aspects of a test, and finally 4) *assertions* for testing assumptions about the business process. After providing all these artifacts, the testing framework will be able to automatically generate a process description which is testable. In the following, we describe the artifacts in more detail:

Simulation Models. A simulation model is used to generate real-world input data for the business process. It allows defining the data of a sequence of events for typical and exceptional business scenarios. A key requirement of the simulation model is that it generates consistent data sets for entire such scenarios. For instance, if we want to simulate our previous cycle time example, the simulator has to generate consistent

“Activity Started” and “Activity Completed” event pairs in order to ensure that the process engine receives correct input data.

Mocks for Systems and Services. Instead of invoking the real source systems, the tested process calls a mock object that merely makes sure that the correct function were called, with the expected parameters, in the correct order without actually contacting a real source system. Often this is needed on an abstraction level. For example, to test an access to the SAP system, it may be burdensome to install, configure, and start a local copy of SAP, run your tests, then tear the local SAP installation down again. Sophisticated mock objects which are built into the testing framework provide a way out of this dilemma. A mock object conforms to the interface of the real object, but has just enough code to fool the tested process and track its behavior. For example, a SAP mock object might record the system request while always returning the same hardwired result. As long as the process being tested behaves as expected, it won't notice the difference, and the process test can check that the proper SAP request was emitted.

Test Components. We want to extend our testing framework with user-defined components which cover new testing aspects. For the SARI framework, we want to add components for tracing event patterns which occurred during process execution. Event patterns describe selected events for an execution path of a business process. Test fixtures can use the events of the discovered event patterns in order to evaluate assertions.

Declarative Assertions. Testers should be able to declaratively define assertions within the test fixture. Thereby, an assertion is an expression that enables you to test your assumptions about a business process. Assertions contain a boolean expression that you believe will be true when the assertion executes. If it is not true,

an error will be thrown. Declarative assertions have the additional benefit of serving as documentation for the behavior and constraints of a business process.

VIII. CONCLUSION AND FUTURE WORK

With new paradigms for software development and software architectures, companies are able to respond faster to changes in a business environment. In this paper, we introduced a testing framework which supports these paradigms by being able to test the correct behavior of running business solutions. We compared our approach with model-driven development, and showed how software tests can be used to make model assertions for business processes. We applied the testing framework to the SARI system and proposed extensions which are part of our future work.

The work presented in this paper is part of a larger, long term research effort aiming to make testing an integral part for the management of business processes. Part of this work is the development of a test bed for testing, simulating and benchmarking business process solutions. One of our key ideas to improve our current testing framework is to use more abstract languages for the definition of a test fixture. At the moment, testers have to use programming languages such as C# for writing tests. In the future, we want to allow testers to use higher level languages for writing tests.

REFERENCES

- [1] E. Albek, E. Bax, G. Billock, K. Chandy, and I. Swett, "An event processing language (epl) for building sense and respond applications," in Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), 2005.
- [2] K. Beck and D. Andres, *Extreme Programming Explained*, 2nd ed., Addison-Wesley Professional, November 2004.
- [3] M. Beedle and K. Schwaber, *Agile Software Development with Scrum*, Prentice Hall, October 2001.
- [4] A. Cockburn, *Crystal Clear*, Addison-Wesley, October 2004.
- [5] S. Haeckel, *Adaptive Enterprise: Creating and Leading Sense-And-Respond Organizations*. Harvard Business School Press, 1999.
- [6] Manifesto for Agile Software Development, <http://www.agilemanifesto.org>
- [7] L. Meade, A. Presley, and J. Rogers, "Tools for engineering the agile enterprise," in Proceedings of the International Conference on Engineering and Technology Management (IEMC 96), IEEE, 1996.
- [8] P. Hamill, *Unit Test Frameworks*, O'Reilly Media, Cambridge, 2004.
- [9] M. P. Papazoglou. Service-oriented computing: concepts, characteristics and directions. In Proceedings of the Fourth International Conference on Web Information Systems Engineering, pages 3–12, December 2003.
- [10] J. Schiefer, C. McGregor, Correlating Events for Monitoring Business Processes, in Proceedings of the 6th International Conference on Enterprise Information Systems (ICEIS), Porto, 2004.
- [11] J. Schiefer, A. Seufert, Management and Controlling of Time-Sensitive Business Processes with Sense & Respond, International Conference on Computational Intelligence for Modelling Control and Automation (CIMCA), Vienna, 2005.
- [12] B. Selic, The Pragmatics of Model-Driven Development, *IEEE Software* 20 (5) (2003) 19-25.
- [13] C. Stevenson, A. Pols, An Agile Approach to a Legacy System, 5th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2004), 123-129, Garmisch-Partenkirchen, Germany, 2004.
- [14] S. H. Kan, J. Parrish, D. Manlove, In-process metrics for software testing, *IBM System Journal*, Volume 40, Pages 220 - 241, 2001.
- [15] W. T. Tsai, X. Bai, R. Paul, W. Shao, V. Agarwal, End-To-End Integration Testing Design, 25th Annual International Computer Software and Applications Conference, 2001.
- [16] X. Bai, W.T. Tsai, R. Paul, T. Shen, B. Li, Distributed End-to-End Testing Management, 5th IEEE International Enterprise Distributed Object Computing Conference, 2001.
- [17] D. C. Kung, P. Hsia, and J. Gao, *Testing Object-Oriented Software*, IEEE Computer Society Press, Los Alamitos, CA, 1999.