Inf Syst Front (2012) 14:789–808
DOI 10.1007/s10796-011-9303-3

Agile test framework for business-to-business interoperability

Jungyub Woo · Nenad Ivezic · Hyunbo Cho

Published online: 12 April 2011

Abstract Business-to-business (B2B) applications are tested routinely for conformance and interoperability against a set of data exchange standards before deployment. However, the existence of many data exchange standards, planned utilizations, deployment environments, and testing scenarios makes it difficult to develop reusable testing tools. To address this challenge, we propose the Agile Test Framework (ATF), which consists of a test case design and test execution model. Test case is defined at two levels: abstract and executable. The abstract level addresses issues related to human understanding and the executable level addresses issues related to machine processing. The test execution model addresses issues related to both reusability and plug-compatibility. The ATF allows the test engineer to generate test cases for a variety of standards and scenarios. Thus, it increases reusability, extensibility, and efficiency compared to other test frameworks.

Keywords Agile Test Framework (ATF) · Business-to-business (B2B) integration · Conformance testing · Interoperability testing · Standards · Test automation · Test case generation

1 Introduction

Business-to-business (B2B) applications continue to have data exchange problems. One reason is that the number of data exchange standards and software solutions that purport to implement those standards continue to grow. Consequently, the same standards are rarely adopted by all B2B applications. Even when B2B applications do implement the same standard, exchange problems still occur. They occur because the B2B standard (1) may contain ambiguous requirements leading to different interpretations, (2) may contain errors or omissions, (3) may be only partially implemented, and (4) may be designed to be flexible by allowing specializations and implementations (Shaw et al. 2000; Moseley et al. 2004; Gosain 2007).

B2B interoperability testing has become a popular means to discover and correct data exchange problems before applications get deployed. To maximize the likelihood of discovery, two types of B2B interoperability testing are typically performed: conformance and interoperability. Conformance testing verifies that an application has implemented the standard correctly (Heckel and Mariani 2005; Site Test Center 2009). Interoperability testing verifies that two or more conformant applications can exchange information using the standards correctly (ETSI, 2009). In general, testing will be repeated several times before all problems have been resolved. Therefore, efficient facilities (test beds) have been designed and developed as a practical means to carry out this testing (Kindrick et al. 1996; Heckel and Mariani 2005).

Test bed designs are usually based on a test framework. A test framework includes test case specifications and a testbed architecture. A test case specification—for both conformance and interoperability testing—enumerates the set of conditions or parameters that define what it means to be correct. Testers use them to determine whether an application is conformant or interoperable (Kindrick et al. 1996; Tsai et al. 2003; Caruso and Umar 2004; Smythe 2006). Generally, the specifications are in machine-readable formats to support automated testing. Test case developers follow specific schemas and grammars in developing these formats. Once the test case schemas and grammars are defined, a specific test bed is designed to execute the test cases. That design is based on the test bed architecture.

J. Woo · N. Ivezic (✉)
Manufacturing Engineering Laboratory,
National Institute of Standards and Technology,
Gaithersburg, MD, USA
e-mail: nivezic@nist.gov

H. Cho
Department of Industrial and Management Engineering,
Pohang University of Science and Technology,
Pohang, Korea

Traditionally, designing and implementing such test beds has been very expensive. The reason is that each standard has different test components, test materials, and test harnesses (i.e., configuration specifications). This variability makes test case development expensive, because the developer must learn a variety of test case schemas and architectures for each standard. It also makes test execution process expensive, because solution providers must develop a variety of supporting test components and interfaces containing different artifacts for each standard (Tsai et al. 2003; Caruso and Umar 2004).

In this paper, we propose a new conceptual architecture for test frameworks, Agile Test Framework (ATF). The ATF includes (1) a systematic test case design for designing and generating modular test cases and test metadata, and (2) a reusable and reconfigurable test execution model that allows for re-configurability of test beds. We describe both in this paper and we demonstrate the ATF's agility using actual use-case studies in a B2B testing context.

The paper is organized as follows. Section 2 introduces basic concepts and provides an analysis of the existing test frameworks for B2B systems. Section 3 describes the overall Agile Test Framework architecture. Section 4 details the design of the test case, and Section 5 describes the design of the execution model. Section 6 uses a specific B2B example to compare the ATF to an existing framework. Finally, Section 7 concludes the paper.

2 Analysis of existing test frameworks for B2B systems

In this section, we consider the existing test frameworks and summarize their problems. Initially, however, we provide basic concepts and terminology for this analysis.

2.1 Basic concepts and terminology

Business-to-business data exchanges may be performed over any communication channel. They allow enterprise application systems to support a collection of defined trading partner protocols; and, they use a composite B2B communication stack profile. That protocol includes the following types of standards:

- *Messaging infrastructure standards*, ranging from transport level (including HTTP[1] and SMTP[2]) to higher-level messaging protocols and quality of service (such as reliability and security) and queries (such as those defined as SOAP[3] extensions).
- *Message choreography standards*, sequences of possible messages including business choreographies (such as UMM business transaction patterns) as well as lower level exchange patterns that are often necessary to establish quality of service.
- *Business document standards*, usually industry-specific (such as UNeDOC,[4] UN/CEFACT,[5] or OAG[6]). These are defined in UML,[7] XML-Schema,[8] RDF,[9] or OWL[10] and include syntactic as well as semantic specifications.

Conformance and interoperability problems can occur at any level of this stack. Before describing our proposed solution, we provide several important definitions.

Test Framework describes the test bed architecture, its software components, and the ways in which these components can be combined using a test harness configuration script (see below) for B2B testing. The *Test Framework* includes not only design of testing tools but also schemas for test materials like test cases. It also describes the test materials that may be processed by that architecture, a mark-up language, formats for representing test requirements, and test case scripts. The *Test Framework* also includes the methodology and process descriptions used to automate testing use profiles, test cases, test requirements, and a message store, as well as other supporting schemas (IIC, 2001; TaMIE 2008; Zhong Jie Li et al. 2008; Namli et al. 2009).

Test Framework Implementation includes tools and documents that are implemented based on a test framework. It consists of test case scripts and test beds (or testing tools) (IIC, 2001; Zhong Jie Li et al. 2008; Namli et al. 2009). A *Test Item* is an information object acquired during testing activities. Test Items include messages, documents, events, and statuses obtained during testing.

Test Case Script specifies a set of test inputs, execution conditions, and expected results. It is used in making an evaluation of some particular aspect of a target system under test (SUT). It is composed in compliance with a formalized format provided by the test framework (Astra Infotech, 2009; Namli et al. 2009). Test case may, in general, be thought of as consisting of procedure and verification information. The procedure information describes how to get the test items from the SUT during the test. The verification information determines whether or not the SUT satisfies the test requirements by way of processing the test items.

Test bed is an execution environment configured for testing. It consists of the specific hardware, operating system, network topology, and configuration of the system under test,

[1] Hypertext Transfer Protocol (HTTP), http://www.w3.org/Protocols/
[2] Simple Mail Transfer Protocol (SMTP), http://tools.ietf.org/html/rfc5321/
[3] Simple Object Access Protocol (SOAP), http://www.w3.org/TR/soap/
[4] UNeDOC, http://www.unece.org/contact/UNECE404.htm
[5] UN/CEFACT, http://www.unece.org/cefact/
[6] Open Applications Group (OAG), http://www.oagi.org/
[7] Unified Modeling Language (UML), http://www.uml.org/
[8] XML Schema (XSD), http://www.w3.org/XML/Schema
[9] Resource Description Framework (RDF), http://www.w3.org/RDF/
[10] Web Ontology Language (OWL), http://www.w3.org/TR/owl-features/

test components, or related documents. It is constructed based on a formalized architecture provided by the test framework and using a test harness specification (Astra Infotech, 2009).

Test Harness Configuration Script defines a test harness specification for a test bed configuration, derived from the test framework, which meets the test objectives and requirements (IIC, 2001; Namli et al. 2009). In other words, for given test objectives and requirements, a test harness configuration script describes the ways in which a specific test bed may be configured prior to execution of a certain test case. That is, the test bed may be automatically configured by the test harness configuration script before the test execution for a specific test case.

Test Infrastructure, as a new concept introduced in the ATF architecture, is a permanent, invariable functional module that can be reused without modification and for any standard and Test User environment. Test infrastructure is designed to allow assembly of arbitrary, pluggable test components that implement test services. (See Section 5.2 for detailed description of the Test Infrastructure.)

The following roles (i.e., types of participants) are typically encountered in the process of test bed development:

- □ *Standard Developer* develops a standard specification, which is an explicit set of requirements for a document, component, system, or service. This role requires the applications and documents used in the domain to be validated by a test system to promote and certify conformance and interoperability of the standard-based solutions.
- □ *Test User* requires a software application to be validated against a specific standard or quality guideline.
- □ *Test Case Developer* develops test cases to validate the system under test (SUT) based on the Test User requirements, which entails use of one or more standard specifications.
- □ *Test Bed Architect* implements a specific test bed system to provide test functions to the Test User. The test bed may be constructed by combining test services within a test infrastructure. The test bed interprets and executes test cases to give the Test User a test report including potential problems and performance metrics for the SUT.

In addition, within the new ATF architecture, the role of *Test Bed Architect* is broken down into essentially three distinct new roles (as discussed later in the paper):

- □ *Test Bed Builder* is primarily responsible for assembly of a new test bed. He or she searches and selects the pluggable test components and generates the test harness configuration script, which specifies the configuration of the test components within the test bed.
- □ *Test Service Provider* implements a test service, which may be utilized within a specific test bed. For example, a verification engine or a messaging handler may be offered as a test service. Once such a test service is designed, it is registered within a test service registry, allowing the Service Provider to advertise the service. A Test Bed Architect may search such a test service registry to identify test services that may be deployed for a specific testing need.
- □ *Test Framework Provider* manages, maintains, and supports the use of the test framework, including its components, such as the test case schema, interface specification for pluggable test components, and test infrastructure specification. The role of the Test Framework Provider is to implement and support the test infrastructure, which is readily reusable in a test bed.

2.2 Problems with existing test frameworks

A number of test frameworks exist. Examples include (1) the IIC test framework for testing ebMS (ebXML Messaging Specification) implementations (IIC, 2001), (2) the RossettaNet self-test toolkit for testing conformance to RossettaNet PIP systems (RosettaNet, 2004), and (3) the WS-I tool which checks the data integrity of Web Service request/response messages (Foster et al. 2003; Bertolino et al. 2006; WS-I, 2009; Durand 2007a). These frameworks, like most others, are limited in terms of reuse because their purpose is to meet testing requirements for a specific standard and a specific Test User's usage only. Therefore, their scope is too narrow to reuse their functionality for other, even similar, testing requirements. For example, the IIC framework requires significant changes before other related messaging standards, such as WS messaging, can be tested. A few test frameworks, such as TTCN (TTCN-3, 2009; Baker et al. 2001) and eTSM (TaMIE, 2008; Durand 2007b), have been developed to overcome such a narrow scope of application. But, even they lack the modularity and extensibility needed to easily reuse them for other purposes (Caruso and Umar 2004).

Figure 1 shows the typical usage of existing test frameworks—from test requirements to test bed execution. The Test User first introduces specific requirements for the intended usage of the standard specification of interest. Typically, this consists of a specific business process or pattern of usage. Then, the Test Case Developer composes test cases based on the grammar and structure of test case scripts designed for that particular test framework. The figure indicates this situation by situating the 'Compose Executable Test Case' and 'Implement Test Bed' activities within the Implementation Phase, supported by the Platform Specific Model (PSM). On the other hand, as shown

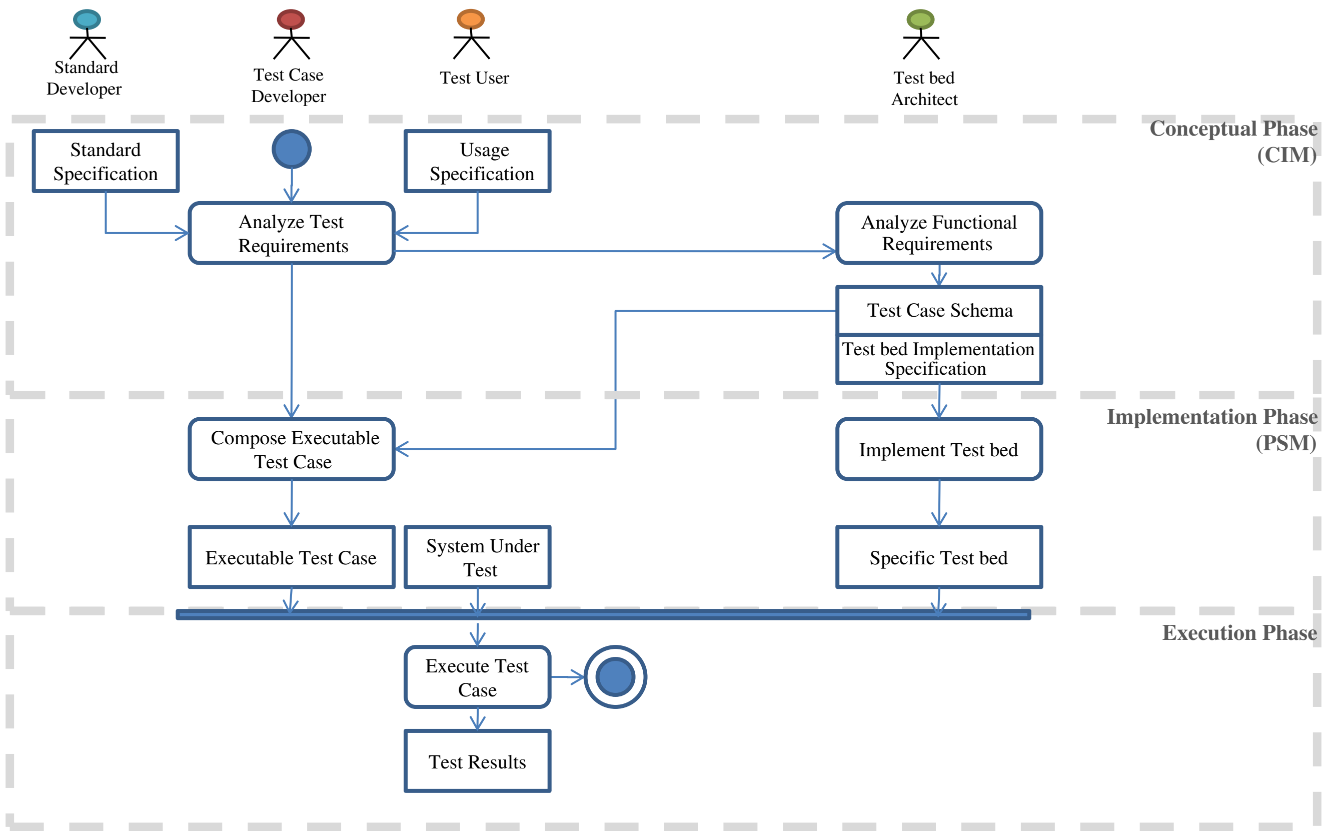

Fig. 1 Conceptual activity diagram for standards-based testing in existing test frameworks

on the right side of Fig. 1, the Test Bed Architect implements a specific test bed based on extracted test bed functional requirements from the Test User's test requirements. At the Execution Phase, the Specific Test Bed uses the Executable Test Cases and verifies the system under test.

Test frameworks designed in this way have three specific problems that limit their reusability.

- **Problem I**. *Test case dependency on the test bed and test requirements*: The existing frameworks promote test case designs that depend on specific test bed design decisions. For example, an IIC test case cannot represent SUT activities during a test because the test cases do not utilize information from the SUT activities. This obviously prevents reuse even in the simple case where the test bed role changes (say, from a customer to a supplier). Also, most test case designs depend on the specific test items' requirements, such as a standard specification. For example, the WS-I test bed is designed only for XML-based web service testing. Therefore, it cannot be used to test other, even the XML-based, standards. Obviously, these two dependencies make test case development a time-consuming, costly, and one-of-a-kind process.
- **Problem II**. *Low configurability of test beds*: Most test bed designs only consider a specific testing purpose and take into account only specific test bed design decisions. It is, therefore, difficult to adapt one test bed to another, even similar, testing situation. Such test beds cannot be re-configured and do not allow new components to be introduced in a plug-compatible manner to address new functional requirements.
- **Problem III**. *Low modularity and extensibility of test cases:* Most test cases contain both procedure and verification instructions for the data. However, the existing test framework designs do not treat the two types of information separately. In other words, the existing test cases mix the test procedure information and test verification information. Consequently, it is difficult to extract verification data from a test case in order to reuse a specific part of the test case in a related, even similar, testing situation.

3 Agile test framework

The agile test framework is designed for agility to generate test cases and to implement test bed. To achive agility, it is important to efficiently reuse existing test scripts and components.

3.1 Overall approach

To address the identified problems, we introduce specific design decisions within a proposed test framework. To resolve **Problem I**, we remove the dependency of the test cases by introducing a two-layer test case design consisting of abstract and executable test case definitions.

- □ **ATF Design Decision I**. *Two-level test case design*: The ATF test case structure consists of an abstract test case and an executable test case. The abstract test case is a human-readable, high level description of a test case. To enhance reusability, an abstract test case does not contain specific, implementation-level information. On the other hand, an executable test case is a machine-readable, low-level description of a test case. An executable test case can be derived from an abstract test case when additional information is provided such as a messaging protocol specification, a specific verification component, and a communication channel between the SUT and the test bed.

The existing test frameworks include only executable test cases. The addition of abstract test case descriptions is expected to have three benefits. First, the abstract test case can be developed early in the design process, providing a clear specification of intent. Second, the abstract test case can be reused easily when similar test requirements arise. Third, the abstract test case helps the tester to understand the executable test case, its procedure, and its rules. That is, the abstract test case provides a kind of three-way bridge between the Standard Developer, who provides test requirements, the Testing User who verifies the system under test, and the Test Case Developer, who interprets the requirements to create the actual test case.

To resolve **Problem II**, we propose the use of *pluggable test components* and *infrastructure designs* and *event-driven test execution*.

- □ **ATF Design Decision II.** *Pluggable test components and infrastructure designs*: Similar to previous research, such as the TestBATN test framework for eHealth standards testing, the ATF adopts a modular and reusable design for pluggable test components (Namli and Dogac 2010). Also, such a modular design approach is adopted by the Global e-Business Interoperability Test Bed (GITB) project which specifically addresses e-business and enterprise interoperability (GITB, 2010). However the ATF is distinguished from the existing works by the permanent infrastructure model. The ATF test bed execution model consists of a test infrastructure that is independent of any specific standard and/or Test User's environment. That is, the infrastructure is designed to be used for all testing situations without requiring any modification. In addition, the test infrastructure is designed to allow assembly of the arbitrary, pluggable test components that implement testing services. The objective is for the components to be provided by any test service provider.
- □ **ATF Design Decision III.** *Event-driven Test Execution design*: Since test components are only loosely coupled to the test infrastructure, direct interfacing between the test components is not feasible. Consequently, ATF relies on a generic transaction handler built around an event board. When a component attempts to interact with another component, it sends data to this event board instead of the target component. The event board stores the various types of interaction data as events so that every component can inquire and retrieve a specific event. The result is that all major activities within the ATF are coordinated via events.

When a service provider designs a new testing service, its design is registered with the Test Service Model Repository. It then can be discovered, integrated into the test infrastructure, and reused whenever necessary. These capabilities will provide three benefits. The test infrastructure may be consistently reused for many types of testing. The pluggable test components may be conditionally reused for the same testing requirements. Automatic configuration makes the test bed more extensible and reusable.

To address **Problem III**, we introduce two test case design decisions: modular and event-centric.

- □ **ATF Design Decision IV**. *Modular test case design*: The abstract test case has two design modules: assertion and procedures. This design makes the abstract test case easier to read. Also the Test Bed Builder may use each module separately when the abstract test case is adapted for different test requirements. Additionally, procedure and assertion content scripts will be separated for the associated executable test case—making them easier to reuse as well.
- □ **ATF Design Decision V**. *Event-centric test case design*: Every type of event—from low-level protocol signals to business document receipts—can be captured by the event board and wrapped into a standard event envelope. Virtually all testing events are captured, stored, and may be used to trigger an arbitrary testing activity or to be fetched and correlated (past events) by such an activity, e.g. a data extraction procedure or verification action.

Because every assertion or procedure module in the test case contains triggering conditions, each module can be introduced and managed independently.

By introducing a modular and event-centric design for test cases, we expect two benefits. First, procedure and assertion modules may be independently reused by another test case. Second, because the test case module is not affected by other changes in the module, a Test Case Developer may easily manage large scale test suites.

3.2 The agile test framework process

The ATF architecture is illustrated in Fig. 2. There are several important differences between Fig. 2 and 1. First, the design phase (corresponding to Platform Independent Model (PIM)) is positioned between the conceptual and implementation phases. Actions in this phase are abstract and independent of a specific test bed implementation. The previous Test Bed Architect role now is broken into the three new roles: Test Service Provider, Test Bed Builder, and Test Framework Provider. The new roles and their activities reflect the drive towards greater modularity and re-usability. Each test case has separate modules for verification rules and procedural data.

The following are descriptions of the key processes supported by the ATF architecture.

- ***Analyze Test Requirements***: The Test Case Developer investigates the requirements given by a standard specification and the intended use of the standard and then determines the functions needed to verify the identified Test User's SUT (System Under Test).
- ***Compose Abstract Test Cases***: An abstract test case consists of usage and assertion scripts. Usage scripts represent the required scenarios that extract test items from the SUT. Assertions scripts provide the predicates needed to determine whether the test item is true or false. Since they are generated without concern for a specific test harness configuration script, these scripts are not machine-readable. Therefore, an ab-

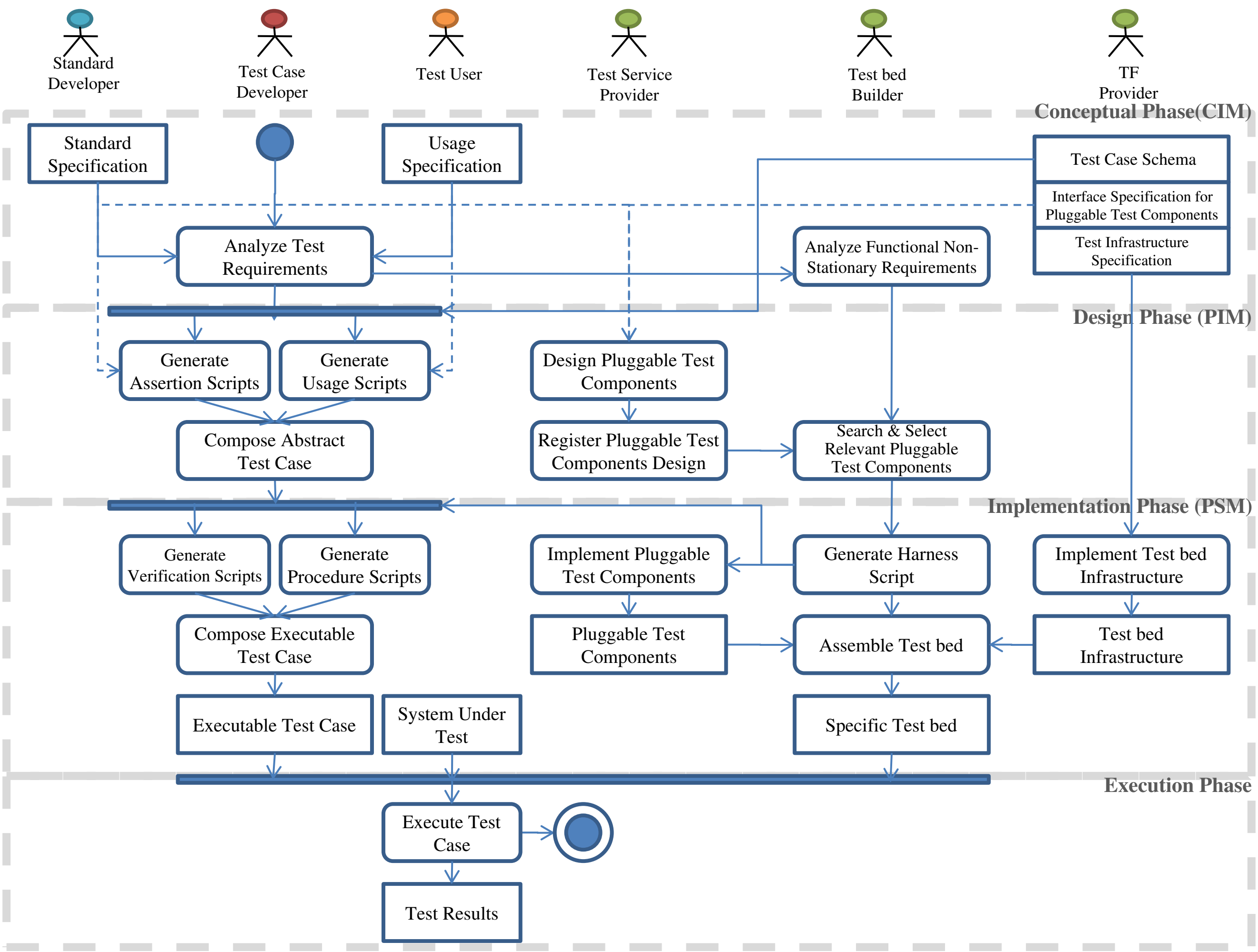
Standard Developer
Test Case Developer
Test User
Test Service Provider
Test bed Builder
TF Provider
Conceptual Phase(CIM)
Standard Specification
Usage Specification
Test Case Schema
Interface Specification for Pluggable Test Components
Test Infrastructure Specification
Analyze Test Requirements
Analyze Functional Non-Stationary Requirements
Design Phase (PIM)
Generate Assertion Scripts
Generate Usage Scripts
Design Pluggable Test Components
Compose Abstract Test Case
Register Pluggable Test Components Design
Search & Select Relevant Pluggable Test Components
Implementation Phase (PSM)
Generate Verification Scripts
Generate Procedure Scripts
Implement Pluggable Test Components
Generate Harness Script
Implement Test bed Infrastructure
Compose Executable Test Case
Pluggable Test Components
Assemble Test bed
Test bed Infrastructure
Executable Test Case
System Under Test
Specific Test bed
Execution Phase
Execute Test Case
Test Results

Fig. 2 Procedural architecture of the agile test framework

stract test case is independent of a specific test bed implementation and is a kind of meta-model for an executable test case.

- □ ***Compose Executable Test Cases****:* An executable test case consists of verification and procedural scripts. These scripts are derived for the corresponding abstract test case using a specific test harness configuration script. The test harness configuration script includes information about test environment and configuration, the partner(s) and target SUT(s), the required test components, and, protocols and schema to be used for business documents, among other things. Using all of this information, the Test Case Developer generates a machine-readable executable test case from an abstract test case. This test environment and configuration information is encoded as a test harness configuration script by the Test Bed Builder. The generated executable test case is registered into the test case repository.
- □ ***Analyze Functional Non-stationary Requirements***: The Test Bed Builder investigates functional, non-stationary requirements based on the test requirements. A non-stationary requirement is a specific test service requirement for a specific testing situation. That is, the test service is required for the testing scenario at hand but it may not be used in other testing situations without additional adaptation.
- □ ***Design and Register Pluggable Test Components***: The pluggable test component provides a specific test service. A Test Service Provider designs the pluggable test component according to the standard specification and the interface specification for the pluggable test components. Once the pluggable test component is designed, it should be registered in a public service registry. At that time, the Test Service Provider registers the information, such as technical capabilities, service interface, and index of service categorization, which are described in Section 5.4.
- □ ***Search and Select Pluggable Test Components***: The Test Bed Builder searches for the relevant pluggable test components that are needed for the specific test bed. The Test Bed Builder decides the optimal set of pluggable test components among the alternative components.
- □ ***Generate Test Harness Configuration Script***: The Test Bed Builder composes the test harness configuration script, which contains the test environment and configuration information describing the partner (s) and target SUT(s), pluggable test components to be used for the testing, the protocol and schema to be used for business documents, and other information necessary to assemble the required test bed.
- □ ***Implement Pluggable Test Component***: The Test Service Provider implements and customizes a pluggable test component based on the pluggable test component design and specific interface requirements described in the test harness configuration script provided by the Test Bed Builder.
- □ ***Assemble Test Bed and Execute Test Case***: A test bed is automatically configured on the basis of information contained in the test harness configuration script. That is, a test bed is assembled from the test infrastructure and the selected loosely coupled pluggable test components. Finally the test bed interprets and executes the executable test case to verify the system under test.

4 Test case design for agile testing

The purpose of the ATF test case design is to address the dependency issues of traditional test case designs that affect the reusability and manageability of test cases.

4.1 Two-layer design of a test case

Most existing test case designs depend on both a standard specification and a specific test bed implementation. These two dependencies make test case development a time-consuming and costly process because test case developers must simultaneously consider both of these constraints when developing test cases. To remove the two dependencies (and resolve **Problem I**), we designed a test case architecture containing two layers: an abstract test case and executable test case. Figure 3 illustrates the test case design.

An abstract test case is derived, in general, from standard specifications and the intended usage patterns for the system under test. Its purpose is to specify the validation rules and testing procedure at an abstract level. Validation rules are written using logical conditions; they describe the normative requirements given under the standard specifications. The testing procedure describes the usage patterns that are simulated for the system under test (SUT). As noted above, abstract test cases are intended for human consumption and may be thought of as a meta-model for the executable test cases. This implies that the abstract test case is independent of a specific test bed. On the other hand, an executable test case is an implementation of the abstract test case that actually executes the validation process. Consequently, the executable test case contains machine-readable content that reflects a specific test bed.

4.2 Modular design of the ATF test case

Typical test case designs embed verification rules as an integral part of the testing procedure. These verification

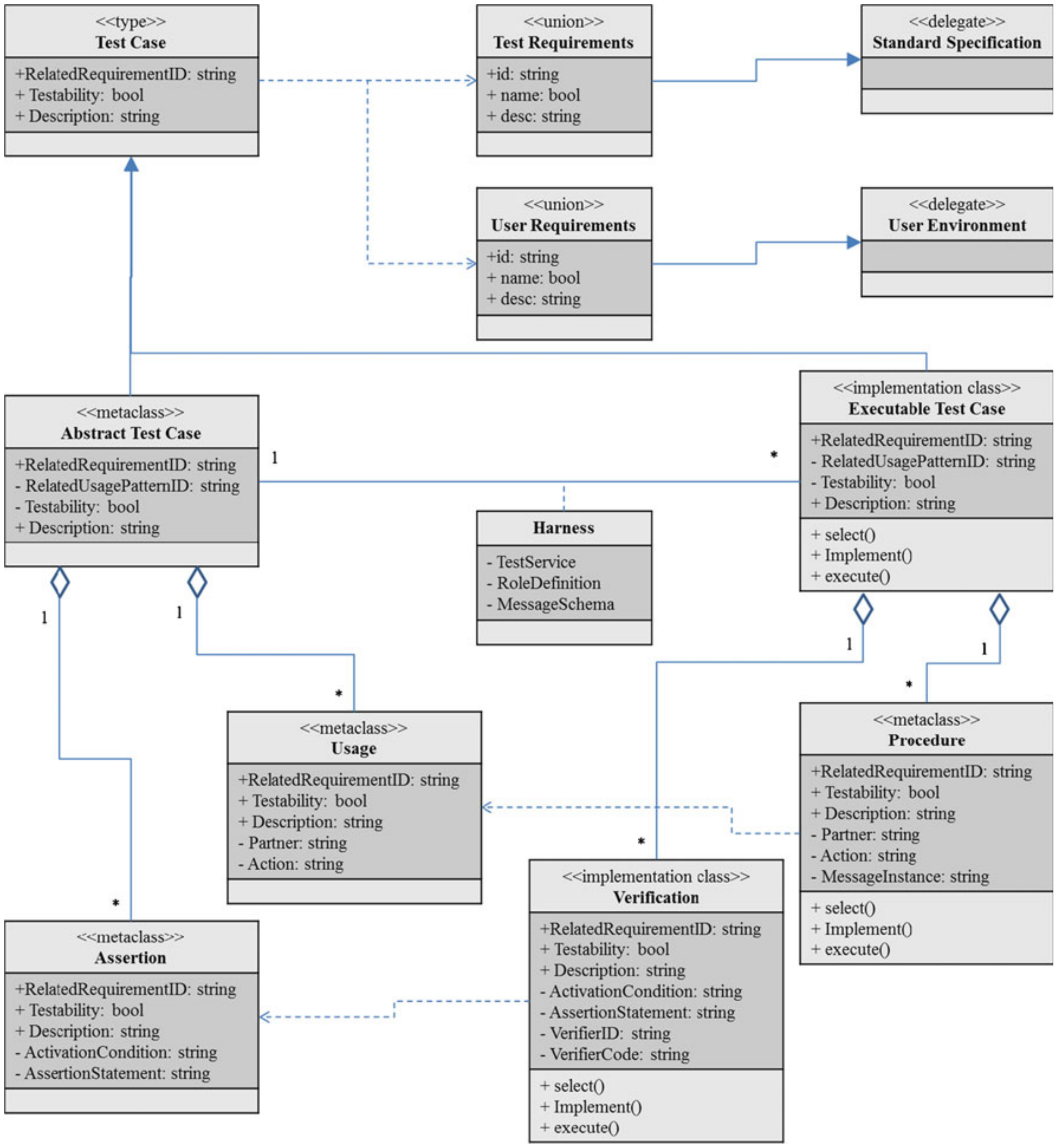

Fig. 3 Data model for the agile test case

rules are used to ascertain whether the test items are true with respect to the test requirements. In this way, these two parts of the test case are closely coupled, because the verification rules will be executed at a specific point in time within the testing procedure. This approach, however, gives rise to two types of problems. First, test cases tend to be monolithic, large, and difficult to maintain. Also, test case design is difficult to modify when the underlying standards change. The second problem is low reusability. Since verification rules are based on the SUT test requirements and testing is based on the business scenarios in which the SUT participates, numerous combinations are possible. The tight coupling means that each such combination will require significant changes to the test cases.

To overcome these problems (and resolve **Problem III**) we propose a modular design for test cases, in which the test cases consist of procedural content and verification rules. As Fig. 3 shows, each test case (either the abstract or executable test case) is composed of two scripts. One script contains procedural content: a usage script for the abstract test case and a procedure script for the executable test case. The other script contains verification content: an assertion for the abstract test case and a verification script for the executable test case.

The two procedural scripts are distinguished by their intent and time of specification. The usage script in the abstract test case represents the testing-related business process, which includes the partners' life cycles and actions during testing. Actions are abstract descriptions and contain no specific message instances. For example, the usage script might say "Buyer sends a purchase order message to a Supplier". The specific buyer, purchase order, and supplier instances are not yet specified. On the other hand, the procedural script in the executable test case represents a business transaction that will be executed at testing time and contains specific test item instances. Additionally, both the usage script and procedural script contain event-driven pre-conditions, meaning that when the pre-condition is satisfied, the procedure step is executed. However, if all preconditions in the usage and procedural scripts only define a fixed time of execution (e.g., next step will be executed after 5 s of wait time), these scripts would encode conventional sequence-based testing and test cases.

Verification scripts also contain event-driven conditions, meaning that an activation condition is defined before the verification script is activated (triggered). When the activation condition is satisfied, a test item is verified against an assertion. These activation conditions render the verification rule independent of the testing procedure, since the rule is not activated at a specific step of testing procedure. Consequently, verification scripts may be readily reused within a new testing procedure because the verification script is independently executed in response to the events referenced in the test procedure. Using this event-driven concept, the verification script can be also reused without any modification when the procedural script is changed. Verification scripts are also distinguished by their intent and time of specification. The assertion script in the abstract test case is human-, not machine-, readable because a specific verifier may not be known at development time. When that verifier is known, the Test Case Developer can add assertion codes using a specific executable language. This assertion code is the verification script in the executable test case.

If the target test item is changed, the verification script should be re-generated because the assertion script in the abstract test case is developed on the basis of the standard specification of the specific test items. However the procedural script could be partially reused because if it form follows a reusable procedural pattern (e.g., Query pattern, Acknowledgement pattern, or Notice) that is independent from the specific test items.

The concept of assertion script is similar to the test assertion concept in the OASIS Test Assertion Guidelines (TAG) (TAG, 2010). A test assertion is a testable or measurable expression for evaluating the adherence of a part of an implementation to a normative statement in a specification, for which TAG defines a formal form. While the assertion script has same concepts as the TAG's test assertion concept, the assertion script also introduces additional components that support extension of the assertion script into the verification script within the executable test case. One type of component is a "Partner Information" which will be extended to a specific pluggable test component or SUT in the verification script. For example, the partner information in the assertion script indicates the specific partner like Buyer or Supplier which produces a target test item to be tested. Then, this information will be replaced for the verification script with the specific partner identification like SUT001 (Buyer) for the executable test case generation according to the test harness configuration script.

4.3 Additional design decisions for agile test case

We made the following additional modifications to the ATF test case design to increase reusability.

- □ *XML-based test case design*: XML facilitates communication between the various participants in heterogeneous systems. Using a single XML-based test case specification, Test Case Developers can format and distribute newly formatted test cases with minimal effort. Using the eXtensible Stylesheet Language (XSL[11]), developers can easily separate content from formatting instructions for various test participants.
- □ *Self-describing test case design*: The test case design is self-describing, meaning that the test cases carry sufficient information to describe all activities during testing. This overcomes s significant limitation of existing test case designs in which a significant amount of essential information is omitted or hidden. For example, the IIC test case does not include how to generate test items at the SUT side because the test service defined by the IIC test framework is embedded in the SUT and generates the test items automatically. However, such limited information content makes the IIC test case difficult to understand and to reuse in different testing scenarios.
- □ *Business process-based test representation*: Existing test cases are represented as related activities between a testing system (a test bed) and a SUT(s). The ATF test cases, however, are represented as related activities between business partners (applications). For example, the IIC test case will describe the specific responsibilities of the buyer and supplier applications. The ATF test case will describe the responsibilities of buyer and supplier in the business process. A focus on business processes allows test cases to be reused

[11] Extensible Stylesheet Language (XSL), http://www.w3.org/Style/XSL/

without modification when the roles of the testing system (a test bed) and SUT are reversed.

5 The agile execution model

As noted, to resolve **Problem II**, we are concerned with two design factors: Pluggable test components & infrastructure design and event-driven test execution design. These are described in more detail below.

5.1 Conceptual architecture of the agile execution model

Figure 4 illustrates the proposed agile execution model that verifies documents and applications implemented in a variety of e-business standards. The model is based on concepts from the Implementation Interoperability and Conformance (IIC) test framework (IIC, 2007). The primary similarity is between our Test Infrastructure module and IIC's test driver. The major difference is the introduction of Pluggable Test Components. The Test Infrastructure module is present in any type of testing; its functionality is invariable and independent of specific testing requirements. The module is developed in advance by the Test Framework Provider according to stationary requirements for B2B testing.

The functionalities of the Pluggable Test Components may change and require re-implementation as a result of changes to the specific test requirements. The proposed ATF provides a bundle of standard interfaces that simplifies the re-implementation process.

A typical test procedure consists of six steps. First, select an executable test case and the related test harness configuration script for processing. Second, process the test harness configuration script to select and deploy identified pluggable test components via the Test Configuration Engine (TCE). Third, process the test case by the Test Sequence Engine (TSE), which, in turn, calls the appropriate pluggable test components or test infrastructure. Fourth, send (or receive) test messages to (or from) the target SUT through the discovered pluggable component (messaging service) of the Test Messaging Interface (TMI). Fifth, verify received messages using the pluggable component (e.g., a reasoning service) of a Test Verification Interface (TVI). Last, generate a test report by invoking a reporting component (implementing a reporting service) of the Outbound Event Interface (OEI).

5.2 Standard interface definition of the pluggable test components

The agile execution model enables the Test Bed Builder to assemble relevant pluggable test components on top of the test infrastructure to provide a specific test bed to the Test User. This model provides standard interfaces for a Test Service Provider to design and implement the pluggable test components. The Test Bed Builder discovers a design model for a relevant pluggable test component in the Test Service Model Repository (discussed below) and requests its imple-

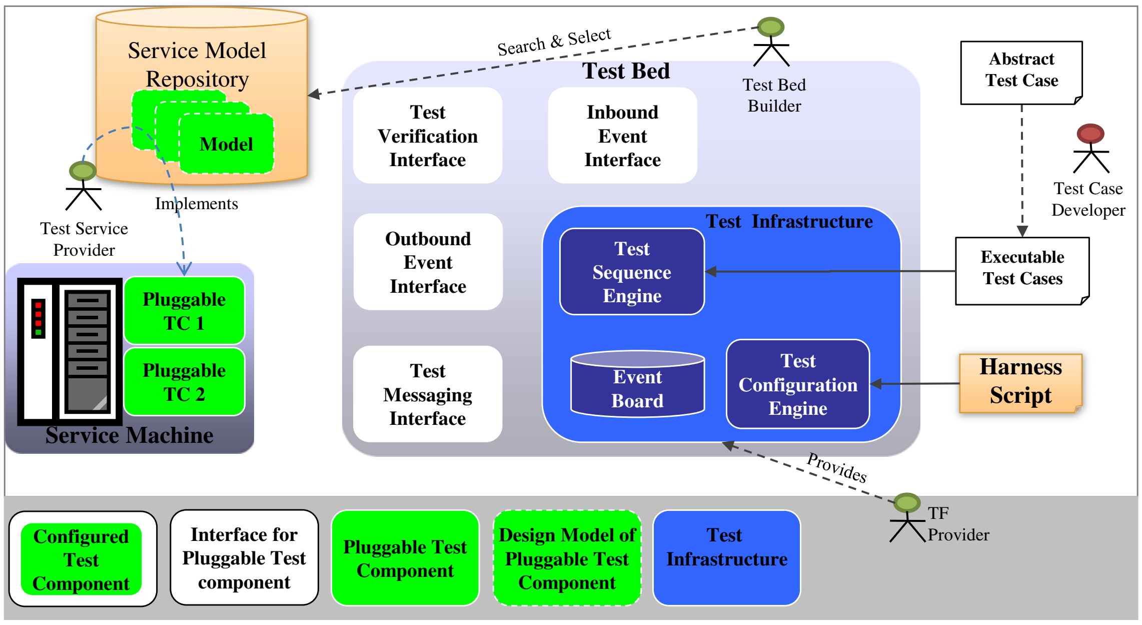

Fig. 4 Conceptual architecture of the agile execution model

mentation from the Test Service Provider. Once the Test Service Provider implements or customizes the pluggable test components, the Test Bed Builder composes the test harness configuration script for a specific test bed with the information necessary to bind these components into a test bed and to connect them to specific SUT(s). The test bed is configured according to the test harness configuration script automatically by the TCE. The test bed is then ready to verify the system under test by running selected test cases.

The Test Infrastructure is independent of any specific standard and/or test environment. It is, in essence, the brain of a test bed. It controls the whole test process in that it reads and interprets an executable test case, orchestrates other test components, and, consequently, drives the execution of test cases that have an SUT in the loop. It has two central engines—the Test Configuration Engine (TCE) and Test Sequence Engine (TSE)—and the Event Board data store. The TCE is responsible for dynamically setting up a test bed in accordance with a test harness configuration script. The TSE drives the testing process by interpreting a selected executable test case. The Event Board collects all significant events generated by the pluggable test components and SUT(s) and supports event logging, search, and correlation.

The standard interface of a pluggable test component contains a data model specifying the data exchange between the test infrastructure and the pluggable test component. The interface is used to guide the Test Service Provider in implementing a specific pluggable test component and to help the Test Bed Builder discover the pluggable components. The TCE dynamically inserts and invokes these pluggable components according to the test harness configuration script and the interface protocols. A test bed may have multiple pluggable test components for each interface type. These interface types are described below.

- □ The Test Verification Interface (TVI) is the interface of a verification engine that determines whether a message from the SUT is valid for a given assertion. The assertions may be scripted in a variety of languages, such as XPath, Schematron,[12] XQuery, JESS,[13] or OWL,[14] thereby necessitating different validation services at execution time. If an XPath assertion script is used, for example, TCE must discover and virtually integrate an XPATH validation service.
- □ The Test Messaging Interface (TMI) is the interface for a messaging engine that delivers messages between a test bed and the SUT. A messaging engine is implemented based on a required messaging protocol, such as ebXML, RosettaNet, or Web Service. At execution time, a particular messaging engine may be selected according to the protocol used by the SUT. For example, ebXML-compliant SUTs need an ebMS-messaging engine.
- □ The Outbound Event Interface (OEI) is the interface for an outbound event handler that provides an event from a test bed to external entities. The reporting service is one of the more important outbound event handlers. It provides a formatted summary of test traces and results to the Test Users. The format depends on both the information and the receiver. For example, a messaging engine developer who seeks to verify message transactions between messaging engines may request a view of the message transactions record. On the other hand, a business document developer who performs a content integrity test may request the list of syntactic errors, conflicts against canonical semantic models, and suggested correction guidelines.
- □ The Inbound Event Interface (IEI) is the interface for an inbound event handler that generates an event for consumption by a test bed during test execution. Triggers are probably the most important inbound events. They control general execution flow and timing of test cases.

The Test Framework Provider should provide a single technical document interface, such as WSDL, for each standard interface: TVI, MEI, OEI, and IEI. A Test Service Provider implements a specific pluggable test component based on this standard interface. At design time, the Test Bed Builder deploys several relevant test components into the test infrastructure, as specified in the test harness script.

Figure 5 shows that several different TVI pluggable test components, which are implemented by Test Service Providers, may be deployed into the test infrastructure. Each of the various verifier engines, which may use varied languages such Jess, Schematron, and XSLT, implement the same interface to be deployed into the test infrastructure.

We encounter a trade-off problem here, as we need to decide how to provide a practical interface definition for each interface type. If the Test Framework Provider designs the interface by providing very detailed, fixed set of parameters, it will cause difficulty to the Test Service Providers who implement pluggable test components based on the single standard interface as there are likely to be many aspects of these test components that are incompatible with such a detailed, fixed interface definition. If the Test Framework Provider, however, defines the interface by providing very broad, extensible set of parameters (like a general string type parameter), no useful structure or semantics of the interface will be exposed for the collaborative parties to base their designs and, eventually, achieve pluggable components. Consequently, reusability of the pluggable test components in this situation is virtually non-existent, because it is necessary that the test case script become tightly coupled

[12] Schematron, http://www.schematron.com/
[13] JESS, http://herzberg.ca.sandia.gov/jess/
[14] Web Ontology Language (OWL), http://www.w3.org/2004/OWL

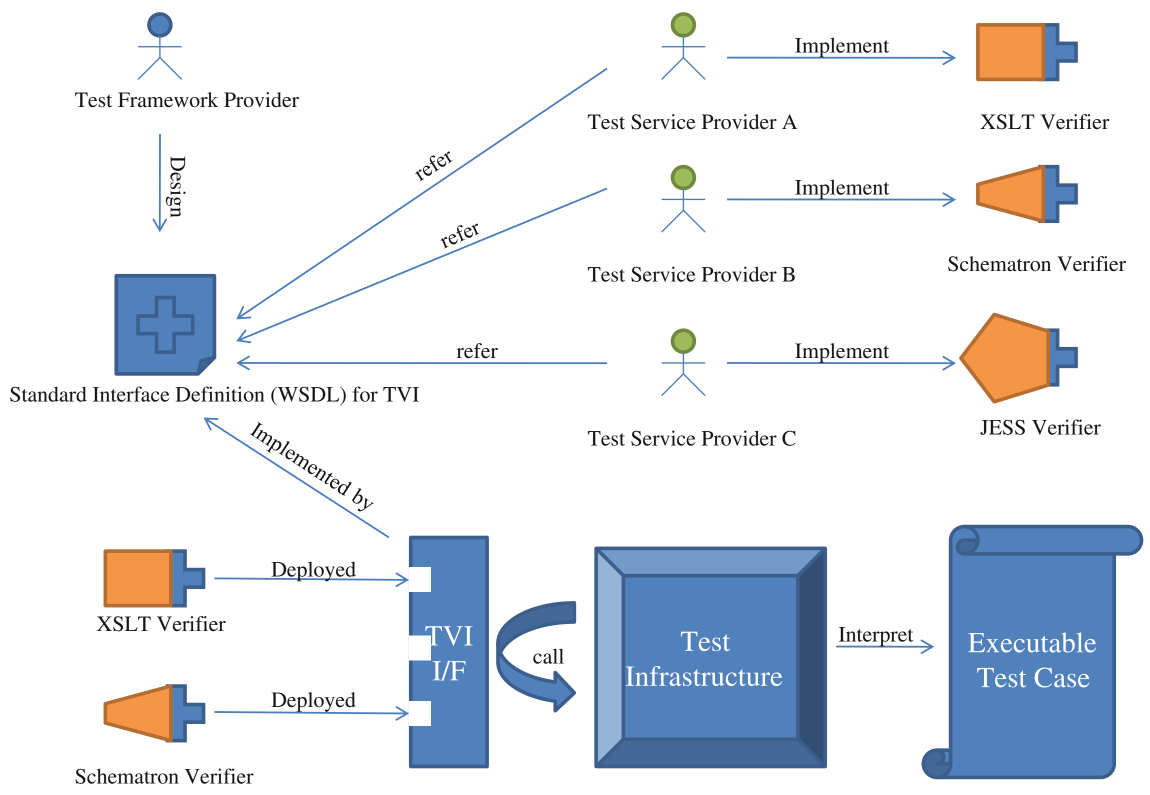

Fig. 5 An example of TVI pluggable test components deployment using the standard interface definition

with a specific pluggable test component, which is the container of the script. So, the tightly coupled pluggable test components really cannot be reused for the other test cases.

A potentially interesting research direction to enhance reuse of testing components is to develop a practical basis for standardization of semantics for testing interfaces. For example, one property that may be usefully included for the TVI in this basis is a kind of reasoning language. Another property of interest in designing TVI is often a kind of test result format. Once these and other useful properties are identified, a taxonomy of the Test Verification Interfaces (TVI) may be created and the classes from this taxonomy may be associated with specific interface types to provide for standardization of such interfaces. For example, a specific Test Service Provider may try to develop own Schematron verifier which has a specific interface class "TVI/AA" because the components uses Schematron for the reasoning language and its result format should include the true/false result and potential problem's location. Similar standardization may be useful for the other interfaces—the TMI, OEI, and IEI.

The test harness configuration script defines what pluggable test components are combined into each interface. However, for a specific testing situation, some interfaces may not have any pluggable test components. For example, if the test bed does not need to communicate messages with SUT, the TMI pluggable test component would not need to be included in the test harness configuration script. For a specific testing situation, if more than one communication protocol is used, multiple TMI pluggable test components are considered in the test harness configuration. That is, multiple pluggable test components can be specified in the test harness configuration script to be deployed using same interface type.

5.3 Event management of the agile execution model

Inbound and outbound events are time-stamped data objects stored on the Event Board and generated by a SUT or test component within a test bed. The test bed uses these events to expedite processing and reporting of test execution according to a test case. An event may serve as a triggering mechanism for test cases, as a synchronization mechanism, or as a proxy for business messages. When an event takes place, it must be posted on the Event Board wrapped within an envelope, as specified in the event schema. The event schema includes meta-information, such as time, type, owner, and identification of the event. Listing 1 shows an event schema using RelaxNG notation

```
<define name=”Event”>
<element name=”event” datatypeLibrary="http://www.w3.org/2001/XMLSchema-
datatypes”>
 <attribute name=”id”><data type=”integer”/></attribute>
 <attribute name=”timepost”><data type=”dateTime”/></attribute>
 <attribute name=”evtype”><text/></attribute>
 <attribute name=”temp”><data type=”boolean”/></attribute>
 <optional>
 <attribute name=”caseid”><text/></attribute>
 <attribute name=”minstanceid”><text/></attribute>
 <element name=”evproperties”>
  <zeroOrMore>
  <element name=”property”>
  <attribute name=”name”/>
  </element>
  </zeroOrMore>
 </ele ment>
 </optional>
 <element name=”content”> <!-- a wrapper for the original event content -
->
 </element>
</element>
</define>
```

Listing 1 – The ATF Event Schema (in RelaxNG Notation).

5.4 Pluggable test component development, registration, search, and deployment

The Test Service Provider designs a model of pluggable test components for specific test requirements. A component can be implemented in any programming language, but the model is described in a neutral language, such as WSDL,[15] an XML-based language that provides a means for describing Web services.

Figure 6 illustrates tasks of the Test Service Provider and Test Bed Builder. The Test Service Provider analyzes the B2B standards and standard interfaces of pluggable test components from the ATF, and then designs a model for a specific component. For example, to implement a Schematron engine for verification of XML documents, the service provider will select a Test Verification Interface provided by the ATF framework and design the model for the Schematron engine. The interface is described in WSDL and includes input/output message protocol specifications. The Service Provider registers the model with the Test Service Model Repository.

Then, the Test Bed Builder searches for, and eventually discovers, the relevant model required for the specific test bed. First, the Test Bed Builder analyzes the non-stationary requirements, and then searches for a relevant model for the pluggable component within the Test Service Model Repository. Next, he or she selects a specific model that will be used in the test bed, and then requests the Test Service Provider to implement the selected model. Finally, the Test Service Provider implements the pluggable test component and provides the service to the Test Bed Builder.

When the necessary pluggable test components are implemented, the Test Bed Builder composes a test harness configuration script to configure the specific test bed. The test bed may be configured according to the test harness configuration script automatically. Figure 7 illustrates a specific test bed configured with a number of pluggable test components and a SUT.

6 Initial evaluation

In this section, we use a simple B2B scenario to compare the proposed ATF architecture to the previously developed eTSM test framework (TaMIE, 2008). We will show how ATF resolves the problems present in the eTSM framework, as identified in this paper.

6.1 Simple use case: Purchase order scenario

In this scenario, the supplier offers the buyer a purchase order (PO[16]) service. The interface to this service is a

[15] Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/wsdl

[16] Example of Purchase Order Scenario (PO), http://sws-challenge.org/wiki/index.php/Scenario:_Purchase_Order_Mediation_v2

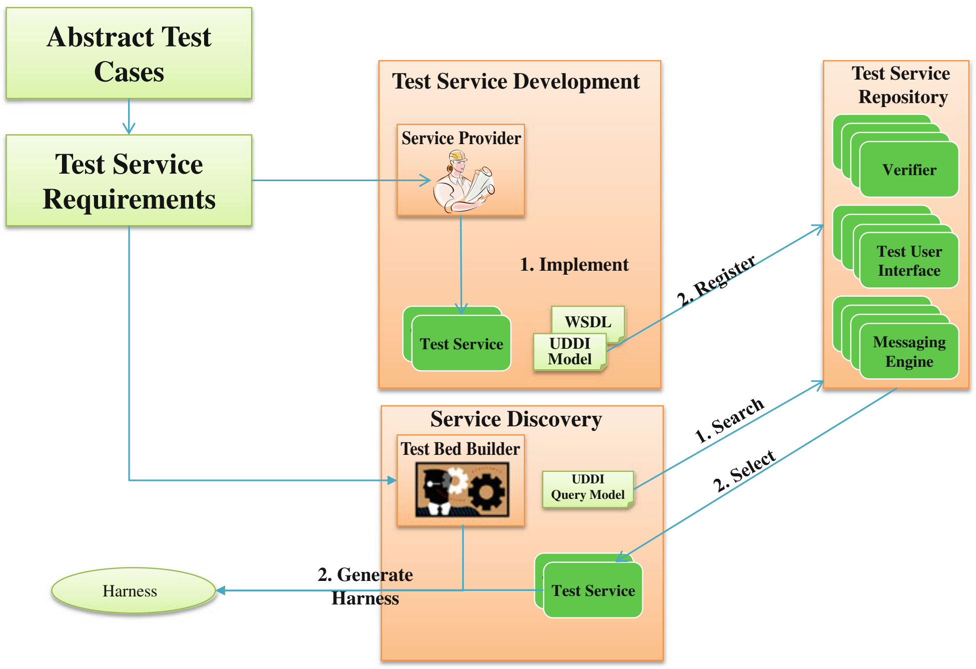

Fig. 6 Pluggable test component development, registration, and search

CRUD (create, retrieve, update, and delete) interface for POs. The buyer's client uses this interface to create a new PO or retrieve (get status), update, or delete (cancel) an existing PO. When a customer submits a new PO, the supplier assigns a new, unique order number. The PO order number can be used to get the status of, update, or cancel the PO. To create a new PO, the buyer and supplier exchange messages as follows.

- □ Step 1: Buyer client sends a PO message to Supplier's service.
- □ Step 2: If Buyer does not get a response message from the supplier within 150 s, then Buyer sends a PO to Supplier again.
- □ Step 3: Sender sends and Buyer receives a PO Confirmation or Rejection, correlated with the PO, within 300 s.

The Test Case Developer analyzes the above scenario and extracts test requirements, which represent constraints to both buyer and supplier systems. In this case, there are two such constraints.

- □ Total time for steps 1–3 is bounded by an upper limit of 300 s.
- □ The correlation is based on a PO reference number provided in the payload of the message

Before composing the test cases, the Test Case Developer defines success conditions and failure conditions to judge both the buyer and supplier systems. When the test results satisfy all success conditions and no failure conditions, the systems under the test are operating normally.

The two success conditions are:

- □ S1: Within 300 s, receive either a single PO Confirmation or Rejection.
- □ S2: Both messages are correlated as expected.

The four failure conditions and one warning condition are:

- □ F1: An error message is immediately received, which correlates with the initial message (either a SOAP Fault or an ebMS Error received on the back-channel of the PO message).
- □ F2: No response is received within 300 s.
- □ F3: There were two contradicting response messages within 300 s (Confirmation AND Rejection).
- □ F4: a PO response was received, but its payload correlation was wrong.
- □ W1: a PO response, correlated with the PO, was received between 150 s and 300 s

6.2 eTSM test case vs. ATF test case

Listing 2 shows an example of the corresponding eTSM test case script. By inspecting the script, one can see that steps 3, 5, and 6 are verification steps, while the remaining steps may be treated as parts of the test procedure script. In step 1, the test bed sends a PO message to SUT and waits for 300 s. In step 2, the test bed attempts to retrieve the PO response message from the SUT. Between step 1 and step 2, the SUT should send a PO response message, but there is no description for this event in the eTSM test case.

```
<!-- ========== step 1: send the purchase order message -->
<call step=”step1” adapter=”sendsoap” type=”event”>
<param arg=”1”>
<template document=”purchaseorder”>
<set var=”POref” copy=”mPOref”>P1234</set> <!-- set Poref in payload -->
<set var=”convId” copy=”mconvId”>100</set>
<set var=”mesgId” copy=”mmesgId”>111444777</set>
</template>
</param>
</call>
<!-- ========== : sleep 300 sec -->
<sleep duration=”300”/>
<!-- ========== step 2: get related error message(s) if any -->
<find step=”step2”>
<selector>
<condition
language=”xpath1”>content/soap/Header/messageInfo[type=”Error”][RefToMess
ageId=$mmesgId]</condition>
</selector>
</find>
<!-- ========== step 3: check exit failure case A (exist an error
message?) -->
<cad step=”step3”>
<condition language=”xpath1”>$output/event</condition>
<do>
<exit value=”fail”/>
</do>
</cad>
<!-- ========== step 4: get related response message(s) if any -->
<find step=”step4”>
<selector>
<condition
language=”xpath1”>content/soap/Header/messageData[POReference=$mPOref][ac
tion=”confirm” or action=”reject”]</condition>
</selector>
</find>
<!-- ========== step 5: check success case (1 response) -->
<cad step=”step5”>
<condition language=”xpath1”>
count($output/event) = 1
</condition>
<do>
<exit value=”pass”/>
</do>
</cad>
<!-- ========== step 6: check exit failure case B (too many responses) --
>
<cad step=”step6”>
<condition language=”xpath1”>
count($output/event) &gt; 1
</condition>
<do>
<exit value=”fail”/>
</do>
</cad>
<!-- ========== step 7: exit failure case C (no responses) -->
<exit step=”step7” value=”fail”/>
```

Listing 2 – The eTSM Test Case Script Corresponding to the Purchase Order Scenario.

The eTSM test case is intended to be machine-readable. It describes a sequence of test bed execution steps during testing. Consequently, if the Test User has no knowledge of the framework, a correct interpretation of the eTSM test case is not possible without additional assistance (**Problem I**). It is also intended specifically to verify a supplier service. If we want to verify a buyer–service, we must modify the verification part of script. Even though we do not change the test procedure, the procedure script must be changed also to reflect the buyer’s perspective. That is, the Test Case Developer cannot reuse the eTSM test case (**Problem III**).

```
Usage script:
UsageID: U0001
Partner: 'Buyer'
Partner: 'Supplier'
Description: "To check the PO Transaction"
Action 1: (Precondition: Test starts) Buyer sends a PO to Supplier
Action 2: (Precondition: Supplier gets PO) Supplier sends a PO response
to Buyer
Action 3: (Precondition: Time(150s) && No PO response at Buyer side)
Buyer re-sends a PO to Supplier

Assertion scripts:
AssertionID: A0001
Description: "Messaging Error"
ActivationCondition: When Buyer gets messaging error from the Supplier
AssertionStatement: HTTP response != 200 || is(SOAP_Fault)
TrueResult: "Fail", "F1: An error message is immediately received, that
correlates with the initial message"

AssertionID: A0002
Description: "Total time for messages 1-2 is bounded by 300 sec"
ActivationCondition: 300 seconds later after Buyer sends a PO to Supplier
AssertionStatement: No PO response (Confirmation or Rejection)
TrueResult: "Fail", "F2: No response received within 300 sec"

AssertionID: A0003
Description: "There were two contradicting response messages within 300
sec"
ActivationCondition: 300 seconds later after Buyer sends a PO to Supplier
AssertionStatement: Confirmation and Rejection messages
TrueResult: "Fail", "F3: There were two contradicting response messages
within 300 sec"

AssertionID: A0004
Description: "PO Correlation"
ActivationCondition: 300 seconds later after Buyer sends a PO to Supplier
AssertionStatement: PO Response header correlation is wrong.
TrueResult: "Failure", "F4: a PO response was well received, but its
header correlation was wrong."

AssertionID: A0005
Description: "unreliable messaging"
ActivationCondition: 300 seconds later after Buyer sends a PO to Supplier
AssertionStatement: 150s < PO Response Time < 300s
TrueResult: "Warning", "W1: a PO response, correlated with the PO, was
received at time t between 150 seconds and 300 seconds."
```

Listing 3 – The ATF Abstract Test Case Scripts Corresponding to the Purchase Order Scenario.

Listing 3 shows the corresponding ATF abstract test case. The usage script describes what the buyer and supplier should do during the testing process. The assertion scripts are independent from the usage script; they have activation conditions that indicate when they are executed. Using this mechanism, they can be more readily reused for new usage scripts.

This ATF test case was also composed to allow specific verification of the supplier service. If the Test Case Developer wants to verify the buyer's client system, the usage script could be reused and he or she must only add new assertion scripts for the buyer's test requirements. If the Test Case Developer wants to modify the usage script, the assertion scripts can be reused if there is no change to the test requirements. In this example, the five assertion scripts (A0001–A0005) are related to the four failure conditions and the one warning condition listed in Section 6.2.

6.3 Test execution of ATF

To overcome **Problem II**, the ATF execution model provides interface definitions for pluggable test components. When the Test Service Provider implements a pluggable test component, the Test Bed Builder may discover and deploy the

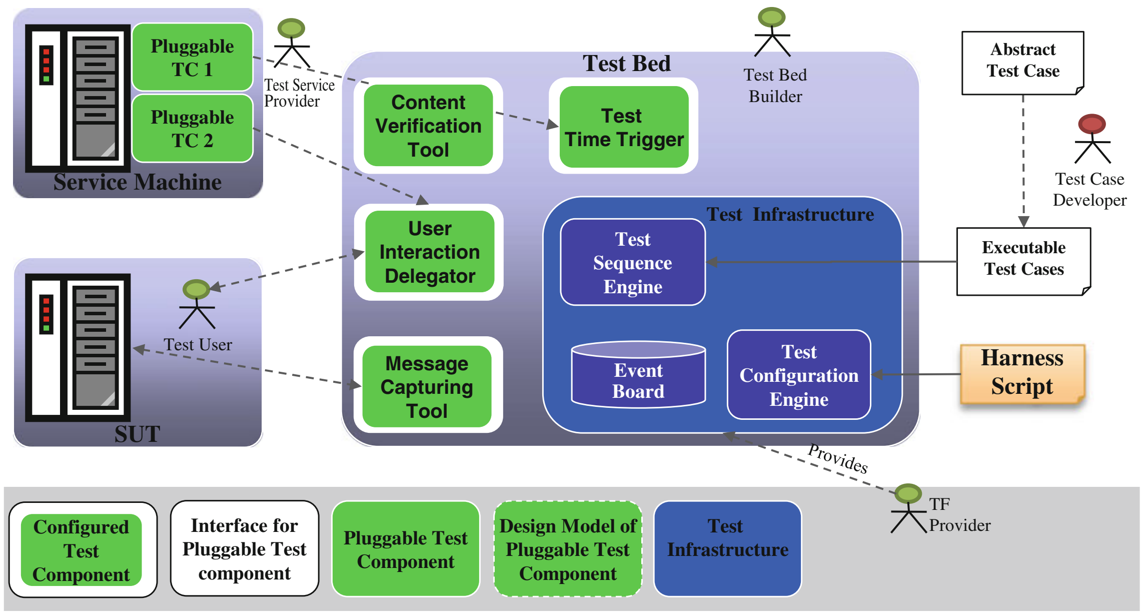

Fig. 7 An example of a configured test bed

component in the test bed. Listing 4 shows the test harness configuration script, and Fig. 8 illustrates the resulting test bed configuration for the Purchase Order scenario. Messaging is handled by configuring the test bed to use a WS-I messaging handler. Had the SUT required ebMS messaging capability, the ebMS messaging handler would have been specified in the test harness configuration script and the ebMS handler would have appeared in the test bed.

In addition, when the assertion scripts in the abstract test case specification require alternative validation functionalities, the Test Bed Builder may search for alternative verification tools that provide such reasoning or verifying functionalities. Such functionalities are then deployed in the test bed via the TVI interface. In this manner, the ATF test execution model addresses the issue of scalability for configurable components of varied functionalities.

```
Test bed: Buyer (NIST TB: http://nist.gov/ATF/testbed)
SUT 1: Supplier (PO Service: http://exmaple.com/PO)
Messaging Standard Specification: WS-I Basic/Reliable Profile
Document Schema: OAGi Purchase Order, Purchase Order Confirmation,
Purchase Order Rejection
Test Services:
TMI: WS-I_Messaging_Handler
TVI: SchematronEngine_1, XPATHEngine_1
IEI: TimeClock
```

Listing 4 – The Test Harness Configuration Script for the Purchase Order Scenario.

As discussed in Section 4, the ATF architecture allows generation of executable test cases by the Test Case Developer once the test harness configuration script specifies the test bed configuration. Listing 5 illustrates the ATF executable test case scripts that the Test Case Developer could generate from the corresponding abstract test cases for the Purchase Order scenario.

Essentially, the above procedural and verification scripts can be seen to be based on the concept of Event Condition Action (ECA). That is, the event part specifies the signal that triggers the invocation of the rule, and the condition part is a logical test that, if satisfied or evaluates to true, causes the action to be carried out. Finally, the action part consists of updates or invocations on the local data.

This characteristic is a clear distinction with a more procedural model like eTSM (TaMIE, 2008). There is a trade-off between the event-driven test case scripting and the procedural-driven test case scripting. The event-driven

Procedure script:

```
ProcedureID: P0001
ReferenceID: U0001
Partner: 'Buyer'
Partner: 'Supplier'
Description: "To check the PO Transaction"
Action 1: (Precondition: Test starts) Buyer:WS-I_Messaging_Handler sends
a PO message to Supplier:SUT_1
Action 2: (Precondition: Supplier:SUT_1 gets PO) Supplier:SUT_1 sends a
PO response to Buyer:WS-I_Messaging_Handler
Action 3: (Precondition: Time(150s) && No PO response at Buyer:WS-
I_Messaging_Handler side) Buyer:WS-I_Messaging_Handler re-sends a PO to
Supplier:SUT_1
Message: [PO message]
  <etsm:function name="purchaseorder">
  <etsm:param name="POref"/>
  <etsm:param name="cust"/>
  <etsm:param name="amount"/>
  <etsm:return>
   <PO>
    <POReference><etsm:eval expr="POref"/></POReference>
    <Customer><etsm:eval expr="$cust"/></Customer>
    <Amount><etsm:eval expr="$amount"/></Amount>
   </PO>
  </etsm:return>
 </etsm:function>
```

Verification scripts:

```
VerificationID: V0001
ReferenceID: A0001
Description: "Messaging Error"
ActivationCondition: When Buyer gets messaging error from the Supplier
ActivationConditionRule(XPATH,XPATHEngine_1):
condition="emam:content/soap/Header/messageInfo[type='Error'][RefToMessag
eId=$mesgId]"
AssertionStatement: HTTP response != 200 || is(SOAP_Fault)
VerificationStatement(XPATH,XPATHEngine_1): (True)
TrueResult: "Fail", "F1: An error message is immediately received, that
correlates with the initial message"

VerificationID: V0002
ReferenceID: A0002
Description: "Total time for 1-2 is bounded by 300 sec"
ActivationCondition: 300 seconds later after Buyer sends a PO to Supplier
ActivationConditionRule(XPATH,XPATHEngine_1):
condition="emam:content/timeClockEvent/messageInfo[time='300']"
AssertionStatement: No PO response (Confirmation or Rejection)
VerificationStatement(XPATH,XPATHEngine_1):
"Count(emam:content/soap/Header/messageData[POReference=$POref][action='c
onfirm' or action='reject']) = 0"
TrueResult: "Fail", "F2: No response received within 300 sec"

VerificationID: V0003
ReferenceID: A0003
Description: "There were two contradicting response messages within 300
sec"
ActivationCondition: 300 seconds later after Buyer sends a PO to Supplier
ActivationConditionRule(XPATH,XPATHEngine_1):
condition="emam:content/timeClockEvent/messageInfo[time='300']"
AssertionStatement: Confirmation and Rejection messages
VerificationStatement(XPATH,XPATHEngine_1):
"Count(emam:content/soap/Header/messageData[POReference=$POref][action='c
onfirm'])=1                                                           and
Count(emam:content/soap/Header/messageData[POReference=$POref][action='Re
ject'])=1"
TrueResult: "Fail", "F3: There were two contradicting response messages
within 300 sec"


VerificationID: V0004
ReferenceID: A0004
Description: "PO Correlation"
ActivationCondition: 300 seconds later after Buyer sends a PO to Supplier
ActivationConditionRule(XPATH,XPATHEngine_1):
condition="emam:content/timeClockEvent/messageInfo[time='300']"
AssertionStatement: PO Response header correlation is wrong.
VerificationStatement(Schematron,SchematronEngine_1):
       InputMessage1: PO Message
       InputMessage2:emam:content/soap/Header/messageData[POReference=$PO
ref][action='confirm' or action='reject']"
       SchematronCase: CheckCorrelation
TrueResult: "Warning", "F4: a PO response was well received, but its
header correlation was wrong."

VerificationID: V0005
ReferenceID: A0005
Description: "unreliable messaging"
ActivationCondition: 300 seconds later after Buyer sends a PO to Supplier
ActivationConditionRule(XPATH,XPATHEngine_1):
condition="emam:content/timeClockEvent/messageInfo[time='300']"
AssertionStatement: 150s < PO Response Time < 300s
VerificationStatement(XPATH,XPATHEngine_1): 150s <
Time(emam:content/soap/Header/messageData[POReference=$POref][action='con
firm' or action='reject']) < 300s
TrueResult: "Warning", "W1: a PO response, correlated with the PO, was
received between 150 seconds and 300 seconds."
```

Listing 5 – The ATF Executable Test Case Generated from the Corresponding Abstract Test Case.

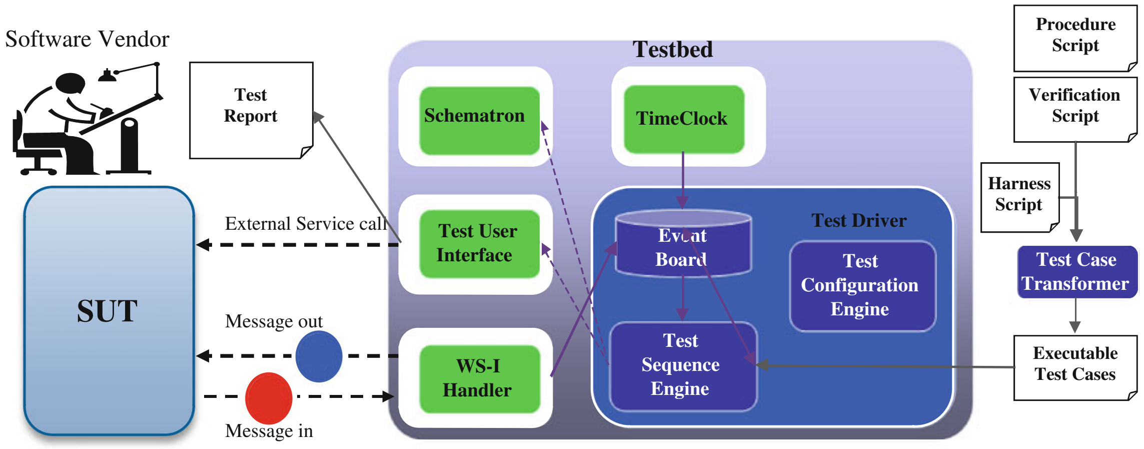

Fig. 8 Resulting test bed configuration for the purchase order scenario

test case could represent more various testing environments and needs, but anything that requires some consolidation of results (e.g. a Service-Level Architecture requirement-stating that at least 90% of transactions must execute within 10mn, or a need to calculate average response time) might be better served by a procedural approach like eTSM, allowing stateful computations.

7 Conclusion

In this paper, we have described a novel framework for B2B testing, the Agile Test Framework (ATF). The ATF addresses core deficiencies found in traditional testing frameworks. These deficiencies prohibit the reuse of testing facilities based on these frameworks without major modifications. Just like these other frameworks, the ATF includes a test case design and a test execution model; but there are significant differences. First, traditional test case designs rely on specific test bed and test case implementation decisions. The ATF introduces the concept of an abstract test case, which insulates the test case specifications from the implementation details and from the specific testing platform requirements.

Second, traditional test execution models rely on specific testing requirements and do not consider the reuse of testing modules in similar situations. The ATF introduces three new concepts: test infrastructure, pluggable test components, and event-driven execution. The test infrastructure contains readily reusable core functionalities that allow assembly pluggable test components into novel implementations of desired testing services. The ATF uses event-driven execution, which is based on a generic transaction handler and an event board. Together, they coordinate and facilitate communication between the infrastructure and pluggable components.

Third, traditional test case designs do not recognize and separate procedural and verification aspects of the test case, which makes test cases impossible to reuse. The ATF introduces modular and event-centric test case design. The modular design separates two types of information: assertions and procedural information. The event-centric design allows assertion and procedure modules in the test case to contain triggering conditions, which allow each module to be managed and reused independently.

In this paper, we detail these differences and their impacts on reuse. We then use a simple B2B scenario to compare the ATF with the existing eTSM test framework architecture. We illustrated how the ATF architecture successfully addresses the reuse issues encountered in the eTSM approach. This preliminary evaluation indicates the potential benefits of the proposed ATF in more complicated B2B testing situations.

References

Astra Infotech (2009). Software Testing Glossary: http://www.astrainfotech.com/software-testing-glossary.html.

Baker, P., Rudolph, E., & Schieferdecker, I. (2001). Graphical test specification—the graphical format of TTCN-3. Lecture Notes in Computer Science.

Bertolino, A., Frantzen, L., Polini, A., & Tretmans, J. (2006). Audition of web services for testing conformance to open specified protocols. In R. Reussner, J. Stafford & C. Szyperski (Eds.), Architecting Systems with Trustworthy Components, number 3938 in LNCS.

Caruso, F., & Umar, A. (2004). Architectures to survive technological and business turbulences. *Information Systems Frontiers, 6*(1), 9–21.

Durand, J. (2007a). Will your SOA systems work in the real world? STAR-East, Software Testing Analysis and Review Conference.

Durand, J. (2007b). OASIS ebXML IIC TC., Event-driven Test Scripting Language. http://kavi.oasis-open.org/committees/download.php/22445/eTSL-draft-082.pdf.

ETSI (2009). Web Site of European Telecommunications Standards Institute, http://www.etsi.org.

Foster, H., Uchitel, S., Magee, J., & Kramer, J. (2003). Model-based Verification of Web Service Compositions, ase, pp.152, 18th IEEE International Conference on Automated Software Engineering (ASE'03).

GITB (2010). Global e-Business Interoperability Test Bed project, http://www.ebusiness-testbed.eu/home/.

Gosain, S. (2007). Realizing the vision for web services: strategies for dealing with imperfect standards. *Information Systems Frontiers, 9*(1), 53–67.

Heckel, R., & Mariani, L. (2005). Automatic conformance testing of web services. In Proc. FASE, Edinburgh, Scotland, Apr., 2–10.

IIC (2001). ebXML IIC Test Framework Version 1.0., OASIS, http://www.oasis-open.org/committees/download.php/1990/ebXML-TestFramework-10.zip.

Kindrick, J. D., Sauter, J. A., & Matthews, R. S. (1996). Improving conformance and interoperability testing. *StandardView, 4*(1), 61–68.

Namli, T., & Dogac, A. (2010). Testing conformance and interoperability of eHealth applications. *Methods of Information in Medicine, 49*(3), 281–289.

Namli, T., Aluc, G., & Dogac, A. (2009). An interoperability test framework for HL7-based systems. *IEEE Transactions on Information Technology in Biomedicine, 13*(3), 389–399.

Moseley, S., Randall, S., & Wiles, A. (2004). In pursuit of interoperability. *International Journal of IT Standards and Standardization Research, 2*(2), 34–48.

TAG (2010). OASIS Test Assertions Guidelines (TAG) TC, http://www.oasis-open.org/committees/tag/.

RosettaNet. (2004). RosettaNet Ready Self-Test Kit (STK) User's Guide Release Version 2.0.7. RosettaNet.

Shaw, M., Blanning, R., Strader, T., & Whinston, A. (2000). *Handbook on electronic commerce*. Berlin: Springer.

Site Test Center (2009). Web site of Software Testing Glossary, http://www.sitetestcenter.com/software_testing_glossary.htm.

Smythe, C. (2006). Initial Investigations into Interoperability Testing of Web Services from their Specification using the Unified Modeling Language, Web Services - Modeling and Testing Proceeding.

TaMIE (2008). OASIS Testing and Monitoring Internet Exchanges (TaMIE) TC, http://www.oasis-open.org/committees/tamie/.

TTCN-3 (2009). Testing and Test Control Notation Version 3, http://www.ttcn-3.org.

Tsai, W. T., et al. (2003). Scenario-based web service testing with distributed agents. *IEICE Transaction on Information and System, E86-D*(10), 2130–2144.

WS-I (2009). Web Service Interoperability Organization, http://ws-i.org.

Zhong Jie Li, Z. J., Sun, W., & Du, B. (2008). BPEL4WS unit testing: framework and implementation. *International Journal of Business Process Integration and Management, 3*(2), 131–143.

Jungyub Woo is a researcher at the Software and System Division of the National Institute of Standards and Technology (NIST), the technology laboratory of the US Department of Commerce. He received his B.S. and M.S. degrees in Industrial Engineering from Postech (Pohang University of Science and Technology) in 2001 and 2003, respectively, and his Ph.D. in Industrial Engineering with a specialization in Manufacturing Systems Engineering from Postech (Pohang University of Science and Technology) in 2007. Dr. Woo has spearheaded development of the Business-to-Business test bed at NIST, which has been used by a number of industry and standards development organizations. Also he is researching for the Health Level Seven International (HL7) Test bed. His areas of expertise include Supply Chain Management, Manufacturing Management and Strategy, and Automated Test. He is an active member of OASIS TaMIE.

Nenad Ivezic is a project manager at the Enterprise Systems Group of the National Institute of Standards and Technology (NIST), the technology laboratory of the US Department of Commerce. He received his MS and PhD degrees from Carnegie Mellon University and BS degree from University of Belgrade. His interests include Supply Chain Management, and Enterprise Systems Integration. Dr. Ivezic has spearheaded development of the Business-to-Business test bed at NIST, which has been used by a number of industry and standards-development organizations. Also, he led the effort in advanced semantic-based interoperability methods development and evaluation to assess potential impact of these methods in automotive industry. He is currently working on advanced information systems for supply chain integration and participates actively in standards testing for e-business technical specifications.

Hyunbo Cho is a professor of department of industrial and management engineering at the Pohang University of Science and Technology. He received his B.S. and M.S. degrees in Industrial Engineering from Seoul National University in 1986 and 1988, respectively, and his Ph.D. in Industrial Engineering with a specialization in Manufacturing Systems Engineering from Texas A&M University in 1993. He was a recipient of the SME's 1997 Outstanding Young Manufacturing Engineer Award. His areas of expertise include Supply Chain Management, Manufacturing Management and Strategy, and Open Business Model. He is an active member of IIE and SME.