CEN/WS GITB3

Date: 2015-xx-xx

CWA XXXXX:2015

Secretariat: NEN

Draft CEN Workshop Agreement:
Global eBusiness Interoperability Test Bed (GITB)

Phase 3: Implementation Specifications and Proof-of-Concept

Status: Draft CWA for Public Comment

Contents

Foreword

CWA XXX was developed “CEN/CENELEC Workshop Agreements – The way to rapid agreement” and with the relevant provisions of CEN/CENELEC Internal Regulations - Part 2. It was agreed on YYYY-MM-DD in a Workshop by representatives of interested parties, approved and supported by [CEN and/or CENELEC] following a public call for participation made on YYYY-MM-DD. It does not necessarily reflect the views of all stakeholders that might have an interest in its subject matter.

The final text of CWA XXX was submitted to [CEN and/or CENELEC] for publication on YYYY-MM-DD. It was developed and approved by”:

- Colinde
- Compliance Test Pty Ltd
- EHIBCC (European Health Industry BarCode Council
- ENEA
- European Commission, DIGIT unit B6
- FernUniversität in Hagen
- Netsend Limited
- Nexus IT
- RBS
- Sonnenglanz Consulting B.V.
- TeleTrusT – IT Security Association Germany
- TNO

The GITB Project Team was chaired by Asuman Dogac, SRDC (Software Research & Development and Consultancy Limited) and composed of:

- University of Lausanne, Faculty of Business and Economics, Department of Information Systems
- Midran ehf
- Invinet sistemes
- IHE Service
- SRDC, Software Research & Development and Consultancy Limited
- Engisis S.r.l.

Other contributors included:

- ISA Action 4.2.6 - Interoperability Test Bed

1 Executive Overview

Motivation

The work on GITB is motivated by the increasing need to support testing of eBusiness scenarios as a means to foster standards adoption, achieve better compliance to standards and greater interoperability within and across the various industry, governmental and public sectors. Without testing, it is cumbersome to reach interoperability of eBusiness implementations and to achieve conformance with standards specifications. More advanced testing methodologies and practices are needed to cope with the relevant set of standards for realizing comprehensive eBusiness scenarios (i.e. business processes and choreography, business documents, transport and communication protocols), as well as Test Beds addressing the specific requirements of multi-partner interactions.

GITB intends to increase the coordination between the manifold industry consortia and standards development organizations with the goal to increase awareness of testing in eBusiness standardization and to reduce the risk of fragmentation, duplication and conflicting eBusiness testing efforts. It thereby supports the goals of the European ICT standardization policy[1][2] to increase the quality, coherence and consistency of ICT standards and provide active support to the implementation of ICT standards.

Vision

The long-term objective is to establish a shared and Global eBusiness Interoperability Test Bed (GITB) infrastructure to support conformance and interoperability testing of eBusiness Specifications and their implementation by software vendors and end-users.

Objectives

The GITB project aims at

- developing the required **global Testing Framework, architecture and methodologies for state-of-the-art eBusiness Specifications and profiles** covering all layers of the interoperability stack (business processes, business documents, transport and communication);

- supporting the realization of GITB as a network of multiple Test Beds, thereby **leveraging existing and future testing capabilities from different stakeholders** (for example standards development organizations and industry consortia, Test Bed Providers, and accreditation / certification authorities);

- establishing under EU support and guidance, a **setup of a comprehensive and global eBusiness interoperability Test Bed infrastructure** in a global collaboration of European, North American and Asian partners.

GITB focuses on the architecture, methodology and guidelines for assisting in the creation, use and coordination of Test Beds. It is not intended to become an accreditation/certification authority or to impose a particular Test Bed implementation.

[1] COMMUNICATION FROM THE COMMISSION TO THE EUROPEAN PARLIAMENT, THE COUNCIL AND THE EUROPEAN ECONOMIC AND SOCIAL COMMITTEE A strategic vision for European standards: Moving forward to enhance and accelerate the sustainable growth of the European economy by 2020 COM(2011)311 final

[2] Regulation (EU) 1025/2012 on European Standardisation.

Benefits

Overall GITB benefits are two-fold: First, GITB raises the **awareness of testing** as a prerequisite to standards adoption and interoperable eBusiness implementations. It ensures that advanced testing methodologies and services will be available for state-of-the-art eBusiness Specifications. Second, GITB promotes a **shared, international testing infrastructure realized as a network of Test Beds** that leverages synergies between existing and future testing activities. Compared to stand-alone Test Beds covering only one or a few eBusiness Specifications, the GITB network saves costs and increases speed in developing and providing high-quality testing services for eBusiness Specifications.

End-users will benefit from the advanced testing methodologies, architectures and services for realizing comprehensive eBusiness scenarios more quickly and with less project risks. They also avoid costs implied by investments in low quality, non-interoperable standards.

By putting more emphasis on testing, **standards development organizations (SDOs) and industry consortia** ensure developing high quality, timely eBusiness Specifications in support of the industry needs; and enable straightforward and effective approaches for standards' implementation assessment, piloting, and deployment.

Based on advanced testing methodologies and services, **software vendors, eBusiness consultants and integrators** are able to develop and integrate enterprise applications in a demonstrably conformant and interoperable manner. Missing implementation guidelines and missing testing facilities increase their implementation efforts and the risks that their software applications do not conform to eBusiness Specifications and / or are not interoperable with other implementations

With GITB, **standards development organizations, test service providers, and software vendors** benefit from a joint approach for developing Test Beds across different world regions and sectors, which positively affects development cost, capability, and compatibility of future testing facilities by leveraging best of class expertise and shared resources. They could benefit from sharing the work load, agreeing on the interpretations of the standards, and working in a synchronized manner.

National governments and the European Union benefit by providing industry, including the SMEs, with high-quality ICT standards in a timely manner to ensure competitiveness in the global market while responding to societal expectations. By providing active support to the implementation of ICT standards using a standard testing approach, they increase the quality, coherence, and consistency of ICT standards.

GITB Phases and Approach

GITB objectives are planned to be achieved in three phases (Table 1-11-1). This CWA summarizes the GITB third phase which develops implementation specifications, an open source Test Bed and a prototype for Test Registry and Repository as proof-of-concept for the GITB architecture. It builds on the results of the first phase (*feasibility study*) and second phase (*testing framework and architecture*).

In all three phases of the GITB project, **eBusiness use cases** played an important role to capture real-world eBusiness testing requirements and to develop and validate the GITB Testing Framework and the specifications. GITB relies on the following industry communities which contribute their requirements and experience in eBusiness testing:

- Public Procurement: CEN Business Interoperability Interfaces (BII), Pan European Public Procurement Online (PEPPOL / openPEPPOL), Electronic Simple European Networked Services (e-SENS).

- Healthcare: HL7 Clinical Document Architecture (CDA), IHE Cross-Enterprise Document Sharing (XDS).

- Automotive Industry: MOSS (Materials Off-Shore Sourcing).

Table 1-1: The Three Phases of the GITB Project

Phase	**Phase 1: Feasibility study**	**Phase 2: Conceptualization of the GITB framework and architecture**	**Phase 3: Realization**
Main activities	An analysis of the benefits, risks, tasks, requirements, required resources for a GITB based on business use cases; current state of eBusiness testing facilities.	Analysis of alternative approaches to architecting and implementing a GITB. A recommended architecture and process to implement the Test Bed that follows from the requirements and architectural analysis with clear rationale. Assessment requirements from international stakeholders.	Refinement and detailed specification of GITB Testing Framework. Proof-of-concept implementation comprising • Test Bed • Test Registry and Repository • Test artifacts for business use cases
Main results	Assessment of testing requirements from three use cases Comparison to existing testing facilities and gap analysis	GITB Testing Framework (architecture, methodology and guidelines) for assisting in the creation, use and coordination of Test Beds Validation based on three use cases	GITB specifications: • Core Test Bed implementation specifications • Test Registry and Repository Proof-of-Concept: • Open source Test Bed • Test Registry and Respository prototype Sample Test Artifacts
Published as CWA	CWA 16093:2010	CWA 16408:2012	To be announced

During the *initial phase*, the feasibility analysis was performed by gathering the requirements from three use cases with regard to Verification Scope ("what to test") and operational requirements ("how to test"). The comparison between these requirements and the existing eBusiness Testing Capabilities revealed a set of functional and non-functional gaps. The assessment of these gaps demonstrated that a shared, operational Test Bed infrastructure is desirable and feasible to complement eBusiness standards development efforts.

The *second phase* further conceptualized and elaborated the suggested approaches to architecting and implementing GITB. Its main result is the **GITB Testing Framework** which comprises architecture, methodology and guidelines for assisting in the creation, use and coordination of Test Beds. The GITB Testing Framework has been instantiated and validated for the use cases, and a pilot implementation has been done in one case.

The GITB Testing Framework forms the basis for the realization of the GITB Platform in the *third phase*. The aim of this third GITB phase is to elaborate implementation specifications, to develop a proof-of-concept

(POC) Test Bed that implements these specifications, and to apply this test bed to one or more real use case test scenarios.

GITB Phase 3 comprises **three lines of work** (see Figure 1-2):

(1) **Implementation specifications:** this line of work is about the development of detailed, machine-processable specifications for the Test Artifacts and interfaces functionally described in Phase 2, so that interoperable test bed implementations can be developed from these. These specifications include formal document structures (e.g. XML schemas and rules), formally defined remote service interfaces (e.g. WSDL) and internal APIs (e.g. as written in Java). These specifications also include the profiling of existing standards or technologies that are considered supportive of Phase 3, called here "supportive standards / technologies".

(2) **Test Bed proof-of-concept development:** this includes all development and integration work for the POC test bed. In particular: (a) development of a core test bed platform and some plug-in components necessary for targeted test scenarios, i.e. at least one of each plug-in category: message adapter, test suite engine, document validator. (b) development of a prototype TRR based on a supportive Registry/Repository standard. This may also include (c) integration of an existing "legacy" test bed following the "GITB-service compliance" approach described in Phase 2.

(3) **Test Scenario development:** this line of work is about demonstrating actual use of the test bed POC for one or more industry domain(s). It includes (a) development or migration of a test suite for the domain, including a document validator, (b) deployment of these test artifacts in the TRR and in the test bed POCs, and (c) an end-to-end test demonstration over test material and/or SUTs used in real use case test scenarios, followed by with feedback from domain experts and users.

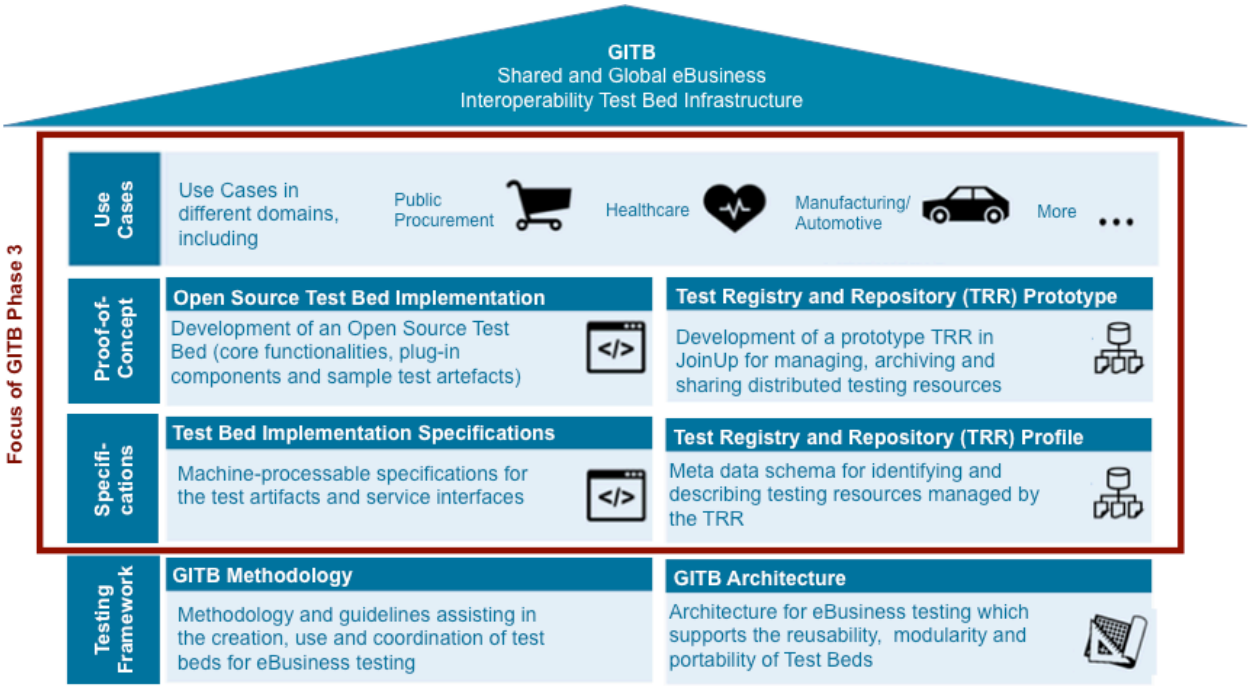

Figure 1-2: Focus of the Third Phase of GITB

How to Read This CWA Document

This CWA presents the GITB implementation specifications for Test Bed as well as the Test Registry and Respository. It describes prototype implementations for both, which serve as proof-of-concept. It also applies the GITB results in use cases. In order to improve readability, the report is structured in four main sections addressing different target groups and their view on GITB project results (Table 1-2).

Table 1-2: Guidelines of How to Read the CWA Document

Main Sections	Content	Relevant for …
Part I: Motivation for eBusiness Testing and GITB Testing Framework Chapters 3 to 4	Why eBusiness testing matters? • Motivation for eBusiness Testing How GITB envisions eBusiness testing • GITB Architecture Vision • GITB Testing Framework	**eBusiness users, standard development organizations, industry consortia, testing experts and all other stakeholders** interested in the general motivation for GITB and an overview of the proposed solution
Part II: GITB Test Bed Core Implementation Specifications Chapters 5-11	• Test Presentation Language (TPL) • Test Reporting Format • GITB Test Service Specifications • GITB Test Description Language (TDL) • GITB Proof-of-Concept Test Bed Implementation • GITB Compliance	**Testing experts and architects** that are interested in the detailed Test Bed Architecture and Specifications
Part III: GITB Test Repository and Registry (TRR) Specifications Chapters 12-13	• Application profile for TRR based on the Asset Description Metadata Schema (ADMS) • Prototype Implementation based on JoinUp	**Testing experts and architects** that are interested in registries and repositories for sharing Testing Resources
Part IV: Testing Scenarios from Public Procurement, Healthcare and Automotive / Manufacturing Industries Chapters 14-23	How to Use GITB for eBusiness testing? • Test Scenarios definitions and workflow. • Test Artifacts related to Test Scenarios (Test Suite, Test Assertions) For selected industries • Public Procurement • Healthcare • Automotive and Manufacturing Industry	**eBusiness users, standard development organizations, industry consortia** that are interested in applying the Test Bed Architecture to their eBusiness scenarios

2 Definitions and Abbreviations

2.1 Definitions

The following definitions are intended to address the most commonly recurring terms about testing in this report. They are general definitions that may be refined in later sections of the document. Other terms relating to specific areas (e.g. architecture, artifacts), will be listed in the related sections.

For the purpose of the present document, the terms and definitions given in ISO/IEC 9646-1:1994 "Information technology - Open Systems Interconnection - Conformance Testing methodology and framework - Part 1: General concepts" apply.

Most of the definitions below are capitalized, even when involving common terms – e.g. "Test Bed". When the capitalized version is used in this document, it should be understood as having the particular meaning defined in this section (or as defined in a further section), usually more precise or specific to the GITB context than the common meaning for the term.

2.1.1 eBusiness Specifications (→ see Section 3)

eBusiness Specification: An eBusiness Specification is any agreement or mode of operation that needs to be in place between two or more partners in order to conduct eBusiness transactions. An eBusiness Specification is associated with one or more of three different layers in the eBusiness interoperability stack: transport and communication (Messaging) layer, Business Document layer, and Business Process layer. In many situations, an eBusiness Specification comprises a set of standards or a profile of these.

Profile (of eBusiness Specifications): A Profile represents an agreed upon subset or interpretation of one or more eBusiness Specifications, intended to achieve interoperability while adapting to specific needs of a user community.

Business Process: A Business Process is a flow of related, structured activities or tasks that fulfill a specific service or business function (serve a particular goal). It can be visualized with a flowchart as a sequence of activities. The term also includes the resulting exchanges between business partners, which is also named "public process". The public process makes abstraction of the back-end processes driving these exchanges.

Business Document: A Business Document is a set of structured information that is relevant to conducting business, e.g., an order or an invoice. Business Documents may be exchanged as a paper format or electronically, e.g. in the form of XML or EDI messages.

2.1.2 Testing Purposes and Requirements (→ see Section 3.4)

System Under Test (SUT): An implementation of one or more eBusiness Specifications, which are part of an eBusiness system which is to be evaluated by testing.

Conformance Testing: Process of verifying that an implementation of a specification (SUT) fulfills the requirements of this specification, or of a subset of these in case of a particular conformance profile or level. Conformance Testing is usually realized by a Test Bed connected to the SUT. The Test Bed simulates eBusiness protocol processes and artifacts against the SUT, and is generally driven by the means of test scripts.

Interoperability Testing: A process for verifying that several SUTs can interoperate at one or more layers of the eBusiness interoperability stack (see "eBusiness Specification"), while conforming to one or more eBusiness Specifications. This type of testing is executed by operating SUTs and capturing their exchanges. The logistics of Interoperability Testing is usually more costly (time, coordination, set-up, human efforts) thanConformance Testing. Conformance does not guarantee interoperability, and Interoperability Testing is no substitute for a conformance Test Suite. Experience shows that Interoperability Testing is more

successful and less costly when Conformance of implementations has been tested first. The interoperability test process can also be piloted by a Test Bed, using test scripts as in Conformance Testing.

Operational Testing Requirements: An operating environment requirement specifies the concerns of defining, obtaining, and validating Test Items within a specific testing environment. It answers the question *what is the specific testing environment?*

Verification Scope: The Verification Scope specifies the subject of testing. It answers the question: *What type of concern to test for?* A type of concern is defined by (1) a specific aspect or quality of SUT to be assessed and (2) an eBusiness Specification or Profile.

2.1.3 Testing Roles (→ see Section 4)

Test Designer: A Test Engineer who develops Test Suites, Test Cases and Document Assertions. This includes interpreting the B2B specifications, understanding – or writing – Test Assertions if any in order to derive test Cases from these.

Test Manager: A role responsible for executing Test Suites or for facilitating their execution, including related organizational tasks such as coordination with Test Participants.

Test Participant: The owner or operator of an SUT, typically the end-user, an integrator or a software vendor. This role defines the Verification Scope and Testing Requirements.

Test Bed Provider: A general role that applies to anyone offering a Test Bed to Test Participants / Managers / Designers.

Testing Capability Provider: A general role that applies to anyone offering a Testing Capability for use in a Test Bed (e.g. offering an HL7 conformance Testing Capability that can be plugged-in a Test Bed platform).

2.1.4 Testing Framework and Architecture (→ see Section 3.4)

Document Assertions Set: A Document Assertions Set (DAS) is a package of artifacts used to validate a Business Document, typically including one or more of the following: a schema (XML), consistency rules, codelists, etc. These artifacts are generally machine-processable.

Document Validator: A processor (a software application) that can verify some aspects of document requirements, i.e. some validation assertions about a document such as an XML schema or some consistency rules. A Document Validator may be specialized for some type of validation assertion (e.g. XML schema validation, or semantic rules).

GITB Architecture: An architecture for a testing infrastructure comprising Test Beds and a Test Registry and Repository. It comprises the following elements:

(a) Test Artifacts that are processed by test beds

(b) Test Services for supporting testing activities,

(c) Test Bed Components and their integration,

(d) Test Registry and Repository for managing, archiving and sharing various Test Resources.

GITB Compliant Test Bed: GITB-compliance means either GITB-framework compliance or GITB-service compliance. A GITB-framework compliant Test Bed follows the GITB recommendations with regard to its functional scope. A GITB-service compliant Test Bed is only required to follow the GITB recommendations for its Service interfaces, and the Test Artifacts it produces.

GITB Methodology: provides guidelines for eBusiness testing. It assists in specifying the subject of testing and the type of concern to test for ("what to test"). It also defines the means by which the testing goal is

achieved ("how to test") and outlines typical testing scenarios (Standalone Document Validation, SUT-Interactive Conformance Testing, Interoperability Testing).

GITB Testing Framework: The architecture, methodology and guidelines for assisting in the creation, use and coordination of Test Beds. The GITB Testing Framework comprises the **GITB Methodology** and the **GITB Architecture** for a modular testing infrastructure comprising Test Beds and a Test Registry and Repository.

Legacy Test Bed: An existing Test Bed that has been developed prior to GITB recommendations. A Legacy Test Bed can be made "GITB-service compliant" by extending it with a subset of the service interfaces described in this report.

Test Agent: A processor – either a simple software testing application or a complete Test Bed – that plays a secondary role in the execution of a Test Suite, i.e. is interacting either with a Test Bed or with a Web browser for the purpose of assisting Test Suite execution. A Test Agent may simulate one party in the execution of a Test Suite (e.g. send messages to an SUT or wait for messages from the SUT), or may be specialized for the execution of some Test Case, or for executing a Document Validator. A Test Agent may be either one or both of: (a) An interacting [Test] Agent if it is able to directly interact with an SUT e.g. to execute parts of the Test Suite (e.g. simulates a business party in some Test Case), (b) A validating [Test] Agent if it is able to verify conformance of some Test Items to an eBusiness Specification or to a profile.

Test Artifact: A Test Artifact is a document used as input or output of Test Beds. These documents may represent various data objects, e.g. Test Cases, Test Assertions, Test Suite scripts, Test Reports, test logs. A Test Artifact should be machine-readable (e.g. formatted in XML).

Test Assertion (Cf. OASIS Test Assertion Guidelines [TAG]): A Test Assertion is a testable or measurable expression - usually in plain text or with a semi-formal representation - for evaluating the adherence of an implementation (or part of it) to a normative statement in a specification. Test Assertions generally provide a starting point for writing a conformance Test Suite or an interoperability Test Suite for a specification.

Test Bed: An actual test execution environment for Test Suites or Test Services. In the context of this document, this generic term applies by default to various operational combinations of components provided by or developed according to the (GITB) Testing Framework.

Test Bed Architecture: A particular combination of components and relationships among the components, in a software system design based on Testing Framework resources and definitions and intended to perform testing operations in accordance with use case requirements.

Test Bed Component: A component of a Test Bed that executes a function required for conformance and interoperability testing. Either a core Test Bed platform component (performing an internal test Bed function, e.g. Test Suite deployment) or a user-facing component (e.g. a Test Suite editor), or a component providing a specific Testing Capability (e.g. a Document Validator).

Test Case: A Test Case is an executable unit of verification and/or of interaction with an SUT, corresponding to a particular testing requirement, as identified in an eBusiness Specification. Each test case includes: (1) a description of the test purpose (what is being tested - the conditions / requirements / capabilities which are to be addressed by a particular test), (2) the pass/fail criteria, (3) traceability information to the verified normative statements, either as a reference to a test assertion, or as a direct reference to the normative statement (Cf. OASIS Test Assertion Guidelines definition [TAG]).

Test Scenario: A Test Scenario is an abstract definition of the testing process to perform a specific set of validations needed to perform conformance or interoperability testing based on a specification.

Test Description Language: In the eBusiness domain, Test Description Language (TDL) is a high-level computational language capable of expressing Test Case and Test Suite execution logic and semantics.

Test Execution Log: (A specific kind of Test Artifact). Message capture or other trace of observable behavior that results from SUT activity. It is a collection of Test Items, subject to further verification or analysis.

Testing Capability: A general term to designate the set of resources (Test Bed Components, test logic or test configuration artifacts) supportive of a particular test function or of a Test Suite execution, typically related to an eBusiness standard. All Testing Capabilities (plug-in components and/or artifacts) are typically add-ons to a Test Bed platform. They may be added to or removed from a Test Bed depending on the testing needs without modifying the code of the Test Bed but instead via a configuration change – they do not represent core functions of such a platform. Examples are:

- Testing Capabilities that relate to a particular eBusiness Specification, e.g. an “HL7 document Testing Capability” involves a set of resources necessary to validate HL7 documents: an HL7 document assertion set (test logic definition) combined with a Document Validator component (the processor of this test logic). An “ebMS2.0 messaging adapter” Testing Capability is an ebMS2.0 Adapter Test Bed Component that will enable Test Suites to use ebMS2.0 messaging during execution. A Test Bed with HL7 validation capability will be said to be “HL7 validation-capable”, or with ebMS2.0 messaging capability to be “ebMS2.0 messaging capable”.

- Some Testing Capability components are not associated with a specific eBusiness Specification or standard, but rather with a specific test logic standard such as XML schema or a particular TDL. Processors for such standards (e.g. XML schema validator, TDL script interpreter) are also considered as Testing Capability components.

Testing Resource: A generic term to designate any part of a Test Bed (Test Artifact, Test Service interface, core or plug-in Test Bed Component), or a combination of these.

Testing Framework (see GITB Testing Framework).

Test Item: A unit of data to be verified, e.g. a document, a message envelope, an XML fragment. In the B2B or eBusiness environment, Test Item can be message instance, event, or status report that is obtained from an SUT for the purposes of assessing conformance or interoperability of the SUT (see Conformance Testing,Interoperability Testing).

Test Registry and Repository (→ see Part III): A component for managing, archiving and sharing distributed testing resources.

Test Report: documents the result of verifying the behavior or output of one or more SUT(s), or verifying Test Items such as Business Documents. It is making a conformance or interoperability assessment (see Conformance Testing and Interoperability Testing). It is generally intended for human readers (although possibly after some rendering, e.g. HTML rendering in a browser or after a translation XML to HTML).

Test Services: These services allow for managing Test Artifacts (design, deploy, archive, search) as well as controlling the major Test Bed functions (test execution and coordination).

Test Step: A unit of test operation(s) that translates into a controllable, separate unit of test execution.

Test Suite: (A kind of Test Artifact). A Test Suite defines a workflow of Test Case executions and/or Document Validator executions, with the intent of verifying one or more SUTs against one or more eBusiness Specifications, either for conformance or interoperability.

Test Suite Engine: A Test Suite Engine (or "Test Suite Driver") is a processor that can execute a Test Suite, or has control of the Test Suite main process execution in case it delegates part of the execution - e.g. some Test Cases or some validation tasks - to specialized Test Agents or to a Document Validator.

2.2 Abbreviations

AIAG	Automotive Industry Action Group
B2B	Business-to-Business
B2C	Business-to-Consumer
B2G	Business-to-Government
CDA	Clinical Document Architecture
DAS	Document Assertion Set
eAC	ebXML Asia Committee
ebBP	ebXML Business Process
eBIF	eBusiness Interoperability Forum (eBIF)
EDI	Electronic Data Interchange
EIRA	European Interoperability Reference Architecture
GITB	Global eBusiness Interoperability Test Bed
GUI	Graphical User Interface
HL7	Health Level Seven
HTML	Hypertext Markup Language
IDEI	Integrated Development Environment
IHE	Integrating the Healthcare Enterprise
MOSS	Material Off-Shore Sourcing
NHIS	National Health Information System
PEPPOL	Pan-European Public Procurement
PoC	Proof-of-Concept
SDO	Standards Development Organization
SOAP	Simple Object Access Protocol
SUT	System Under Test
TAG	Test Assertion Guidelines
TAPM	Test Artifacts Persistence Manager
TDL	Test Description Language
TPL	Test Presentation Language
TRR	Test Registry and Repository

XML	Extensible Markup Language

Part I: Motivation for eBusiness Testing and Overview of GITB Testing Framework

Part I summarizes the motivation for eBusiness testing and provides an overview of the GITB Testing Framework. It is relevant for the following target groups: eBusiness users, standard development organizations, industry consortia, testing experts and all other stakeholders.

3 Motivation

3.1 Testing as a Key Prerequisite to eBusiness Interoperability

In the move towards globally networked enterprises, eBusiness scenarios are to support increasingly complex interactions among a larger number of organizations from industry, governmental and public sectors. While eBusiness scenarios are implemented and adopted at a global level, interoperability has become a major concern. Consequently, organizations from private and public sectors as well as technology and software providers are engaged in cooperation for the development of vertical industry standards. However, it can be noticed that it is still cumbersome for software vendors and end-users to demonstrate full compliance with the specified standards and to achieve interoperability of the implementations[3]. This is due to a number of facts:

(1) Many standards development organizations (SDOs) and industry consortia are only in the process of conceptualizing how they will ensure interoperability of standards' implementations. They are unsure how to provide adequate testing and certification services.

(2) eBusiness interoperability typically requires that a full set of standards – from open internet and Web Services standards to industry-level specifications and eBusiness frameworks – are implemented. We denote this set of standards as eBusiness Specifications that underlie the electronic business relationship.

(3) As of today, there are only limited and scattered Test Beds. If Test Beds are provided by one of the standards development organizations, they have a rather narrow focus on a particular standard. In particular, they might not encompass testing the entire set of relevant eBusiness Specifications from a company perspective, i.e. a "Profile", and interactions in more complex Business Processes with several partners.

The following section outlines the demand for eBusiness testing from the perspective of the relevant stakeholders.

3.2 Stakeholders and their Interests in eBusiness Testing

The relevant stakeholders in eBusiness testing comprise end-users from private and public sectors, industry consortia and SDOs, technology and software vendors, testing laboratories as well as public authorities and governments.

End-users comprise all organizations – from private and public sectors – which implement eBusiness scenarios. Their ultimate goal is to increase the efficiency and effectiveness of their organizations and to keep up-to-date in solutions for enhanced customer experiences. eBusiness testing is of interest for them as they:

(1) realize the benefits of eBusiness solutions more quickly, with less project risks, and

(2) avoid costs implied by investments in low quality, non-interoperable standards.

For end-users, the lack of eBusiness testing has negative impacts on project duration for on-boarding business partners and is one of the root causes of significant B2B integration costs. While the ability of an enterprise to quickly add new business partners is a key factor in determining the level of its business agility,

[3] eBusiness W@tch Report on e-Business Interoperability and Standards: A Cross-Sector Perspective and Outlook, 2005

most companies need 3 to 10 days or more to on-board new business partners[4]. The most negative effects of a lack of testing, however, are errors that occur in productive eBusiness scenarios, i.e. if supply chain operations are slowed down or customer requirements cannot be fulfilled as planned.

Industry consortia and formal SDOs are communities of end-users, public authorities and other interested parties that act to achieve the following objectives:

(1) Maintain cohesive community acting on key set of industry issues leading to industry-driven, voluntary standards development;

(2) Develop high quality, timely industry standards specifications in support of industry needs;

(3) Effect efficient implementation of the developed standards by the vendors to provide a rational basis for the standards assessment;

(4) Enable straightforward and effective approaches for standards' implementation assessment, piloting, and eventual deployment.

For industry consortia, the lack of testing increases the risks that implementations of the specified standards are not interoperable.

Software vendors that act to achieve the following objectives:

(1) Develop enterprise applications that are standards-compliant, and

(2) Effectively support their client base by achieving functional and interoperable dBusiness solutions.

Software application vendors are struggling with the pure number and complexity of standards as well as the low quality of eBusiness Specifications with regard to their consistency. Missing implementation guidelines and missing Testing Capabilities increase their implementation efforts and the risks that their software applications do not conform to eBusiness Specifications and / or are not interoperable with other implementations.

Testing laboratories act to achieve the following objectives:

(1) Increase efficiency and reliability of interoperable implementation of standards;

(2) Assure unbiased and objective nature of the standards implementation assessment process.

From the perspective of national governments and the European Union lacking interoperability and poor-quality standards harm innovation and competition, burn investments, and drain the growth potential of markets. In their current efforts to modernize the EU ICT standardization policy, the European Commission states the following policy goals:

- To provide industry including SMEs, with high-quality ICT standards in a timely manner to ensure competitiveness in the global market while responding to societal expectations;
- To increase the quality, coherence and consistency of ICT standard, and
- To provide active support to the implementation of ICT standards.

eBusiness testing provides the necessary means to achieve these goals, as it contributes to solve quality issues in standards development and addresses implementation issues which currently hamper the adoption

[4] Forrester Research Inc. (2009): The Value of a Comprehensive Integration Solution, Forrester Research Inc., Cambridge, 2009

of eBusiness standards. Consequently, eBusiness testing needs to be a cornerstone of EU ICT standardization policy.

3.3 Categories of eBusiness Specifications

Doing business electronically requires that certain agreements are in place between two or more partners in order to conduct eBusiness transactions. We denote these agreements as the eBusiness Specifications governing an electronic business relationship. An eBusiness Specification is associated with one or more of three different layers in the eBusiness interoperability stack[56] and often relies on standards that have been developed or are still under development (Table 3.1)

1. Transport and Communication (Messaging) Layer: How do organizations communicate electronically?
   This layer addresses *technical interoperability*. Relevant specifications cover the range from transport and communication layer protocols like HTTP to higher level messaging protocols such as Simple Object Access Protocol (SOAP) or ebXML Messaging. Furthermore, security, reliability and other quality of service protocols and extensions over the transport and communication protocols are also considered in this layer.

2. Business Document Layer: What type of information do organizations exchange?
   This layer addresses the *semantic interoperability* and specifies the form and content of Business Documents which are exchanged electronically. Specifications may relate to:

   - Document structure, i.e. definition of the document syntax (e.g. XML), the naming and design rules (e.g. rules for generic Business Document structure, as specified by OAGIS BOD architecture) and the assembly of the document (e.g. rules for the assembly of Business Documents, as defined by OAGIS BOD architecture);

   - Document semantics, i.e. the definition of document and fields (e.g. an XML document definition) and their meaning including reference to external code lists, taxonomies and vocabularies (UN/CEFACT Core Component Library, UBL Component Library), and

   - Business rules that define restrictions or constraints among data element values.

3. Business Process Layer: How do the organizations interact?
   Business Processes address *organizational interoperability*. Specifications at this level describe how Business Processes are organized across organizational boundaries. The Business Process layer, either presented in a formal Business Process specification standard such as ebXML Business Process Specification Schema (BPSS) or with an informal workflow definition like flowcharts or interaction diagrams, provides a message choreography, exception flows (error handling) and other business rules for the eBusiness application roles participating in the process.

In addition to these layers, an eBusiness Specification may rely on profiles which define cross-layer dependencies and further restrictions on the single layers.

[5] CEN ISSS: eBUSINESS ROADMAP addressing key eBusiness standards issues 2006-2008.

[6] Legner, C.; Vogel, T. (2008): Leveraging Web Services for Implementing Vertical Industry Standards: A Model for Service-Based Interoperability, in: Electronic Markets, 18, 1, 2008, pp. 39-52.

Table 3-1: eBusiness Specifications

1. Business Process Layer describes the different actors, their roles and interactions, as well as the information flow			
	Cross-org. process flow	**Specifications (example)**	**Machine-readable representation**
	Actors and roles	Text descriptions of roles, UML Use case diagrams	
	Activities and interactions	flow charts, text, UML activity diagrams, UN/CEFACT UMM diagrams, …	BPSS, (BPEL), …
	Message choreography	**Specifications (example)**	**Machine-readable representation**
	Sequence of messages	flow charts, UML sequence diagram, BPMN diagram, …	WS-Choreography Language , BPSS …
2. Business Document Layer addresses the structure and content of business documents which are exchanged electronically			
	Content / Semantics	**Specifications (example)**	**Machine-readable representation**
	Document definition	XML message (XSD), EDIFACT document	XML schema (XSD), EDI
	Optional / mandatory fields		
	Dictionary / vocabulary	External code lists, UN/CEFACT Core Component Library (CCL), RosettaNet Business Dictionary, UBL Component Library, HR-XML Core Components, other dictionaries …	Data models, Ontologies
	Business rules	Consistency rules for content validation; consistency rules for cross-content validation	Rules, data models
	Structure / Syntax	**Specifications (example)**	**Machine-readable representation**
	Document assembly	UN/CEFACT Core Component Business Document Assembly (CCBDA), OAGIS BOD Architecture…	…
	Naming & design rules (NDR)	UN/CEFACT XML NDR, OAGIS NDR, …	…
	Header definition	UN/CEFACT Business Document Header (BDH), …	…
	Document syntax	XML, EDI guidelines, …	…
3. Messaging Layer covers the range from transport and communication layer protocols to higher level messaging protocols			
	Messaging protocols	ebXML Messaging (ebMS), SOAP, EDIFACT X12 RosettaNet Implementation Framework (RNIF),	…
	Transport & communication	HTTP, TCP/IP, OFTP, FTP, X.400 …	…
	Others (Security, etc.)	WS-Security, WS-Reliability, WS-Policy, …	…

3.4 eBusiness Testing

3.4.1 Conformance and Interoperability Testing

From a general perspective, two types of testing are relevant in the context of eBusiness:

- Conformance testing involves verifying whether an eBusiness implementation conforms to the underlying eBusiness Specifications. This is the first step toward interoperability with other conformant systems as prescribed by the specification.

- Interoperability testing is verifying that two or more eBusiness implementations actually are able to intercommunicate based on some exchange scenarios. This form of testing is generally more difficult to automate than Conformance Testing, and is more effort intensive in terms of human involvement and coordination.

Experience shows that only through conformance and Interoperability Testing, correct information exchange among eBusiness implementations can be guaranteed and software implementations can be certified. Conformance Testing is no substitute for Interoperability Testing, and vice-versa.

Experience also shows that the type and quality of the eBusiness Specifications impact whether conformance and Interoperability Testing can easily be performed. If eBusiness Specifications comprise substantial text descriptions, with some flow-charts or diagrams, these narrative or semi-formal representations often leave many degrees of freedom for interpretation to the users. The efforts to prepare test scripts and Test Cases are much higher than in the case of an eBusiness Specification which comprises machine-readable representations, such as XML schemas, code lists, data models or formal representations (e.g. in the Web Services Definition Language, or in ebXML Business Process (ebBP) – some examples for machine-readable specifications are depicted in the right column of Table 3-1).

As of today, the existing testing tools, Test Suites and testing committees individually address a specific standard or one of the above layers. However, integrated Testing Frameworks which do not hard-code a

specific standard at any layer (because different communities may use different standards) and are capable of handling testing activities at all layers of the interoperability stack are necessary for conformance and Interoperability Testing.

3.4.2 Testing Context and Stakeholders

eBusiness testing is performed in different contexts with different business rationale and stakeholders:

(1) **Standardization initiated by a standard development organization (SDO) or an industry consortium** (Figure 3-13-1):
An SDO or industry consortium develops an eBusiness Specification (or Profile) and deploys it to the community of users, software vendors etc. In this case, testing occurs during standard development (in order to test conformance with other specifications, such as Naming and Design Rules) for quality assurance of the developed eBusiness Specifications. Testing also occurs during standard deployment to ensure the quality and the interoperability of the implementations. Testing may lead to certification of software or productive implementations.

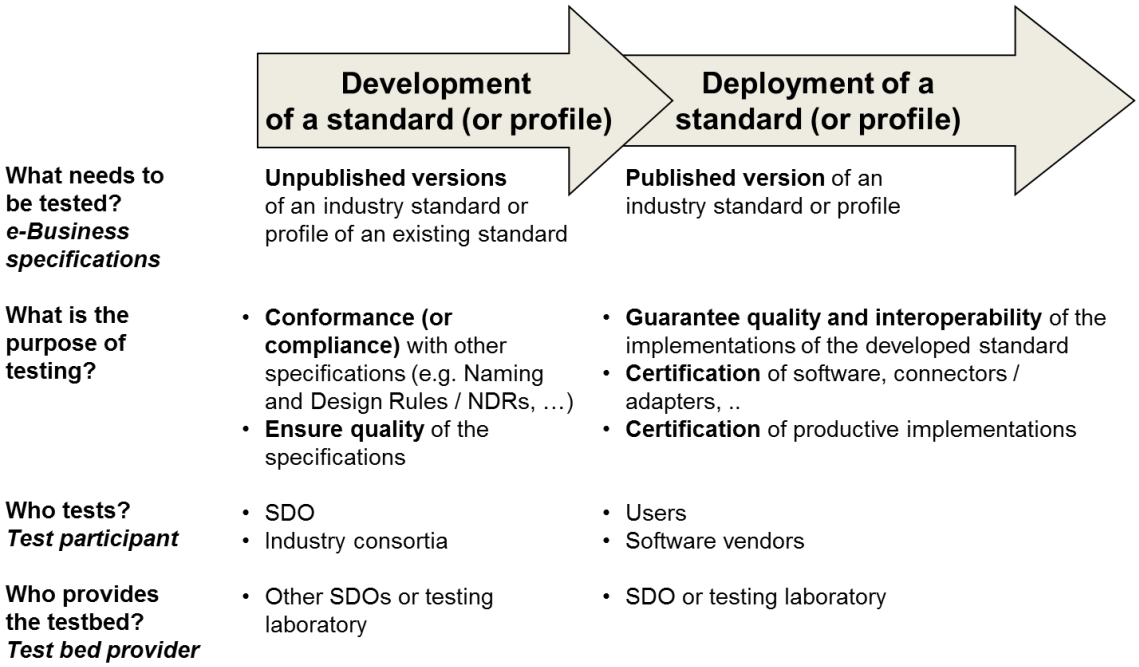

Figure 3-1: Testing Context "Standardization"

(2) **"Onboarding" of new business partners initiated by user company** (Figure 3-2Figure 3-23-2)**:**
In this case, a company defines eBusiness Specifications and imposes their implementation on all business partners. Testing is performed as part of the so-called "onboarding process" of partners.

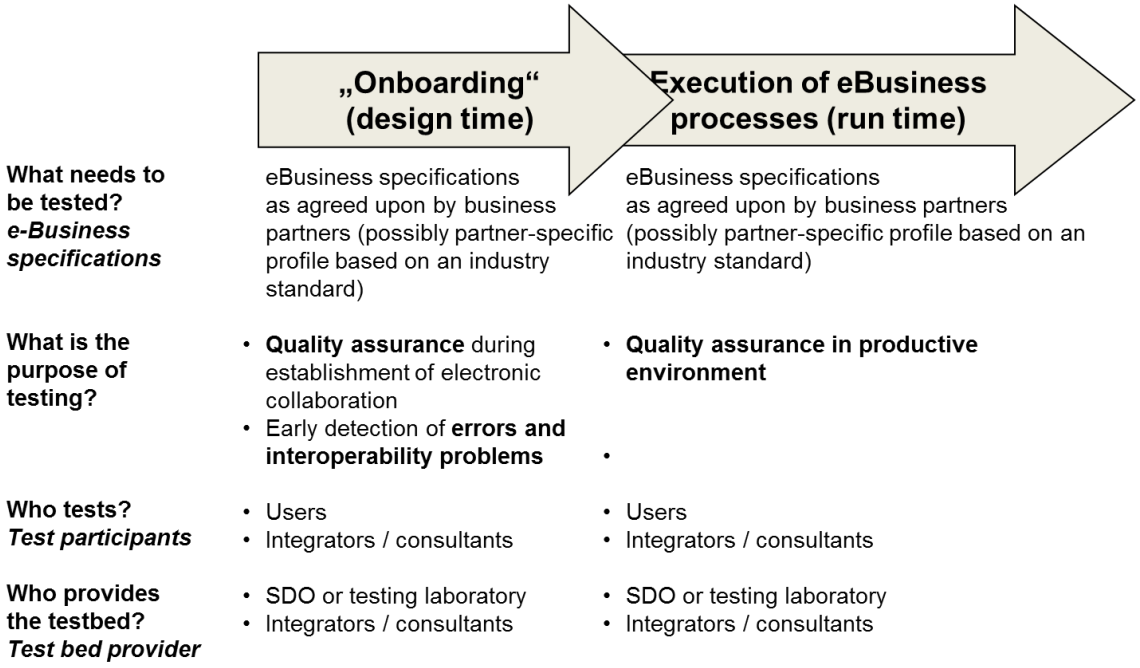

Figure 3-2: Testing Context “Onboarding”

3.5 Benefits of a Global eBusiness Interoperability Test Bed

To summarize, without eBusiness testing the potential of standard setting is not fully exploited and the wide-spread adoption of eBusiness standards will not be possible. Hence, the rationale for GITB can be summarized as follows:

(1) Efficient allocation of resources and efforts in eBusiness implementation projects (less resources will be spent to overcome low quality, conflicting or fragmented standards),

(2) Higher quality of eBusiness standards and mitigation of systemic risks in the eBusiness community, and

(3) Improvement of the eBusiness standards development and diffusion process.

More attention to testing, visibility of outcomes and feedback from testing to industry consortia and SDO's will imply increased attention to quality of standards and their implementation, and to a crisp boundary between commons (standards as public resources) and proprietary assets.

4 GITB Principles and Testing Framework

GITB emphasizes the modularity and reusability of a Test Bed design and the easy plug-in of existing and future Testing Capabilities for state-of-the-art eBusiness Specifications. In proposing a Testing Architecture, GITB enables the coordination of multiple collaborating Test Beds in a network of Testing Resources, offering Testing Capabilities for eBusiness Specifications that can be used either directly by Test Participants, or by other Test Beds.

4.1 Objectives and Principles

The following objectives and principles were guiding the GITB work and should be met by the GITB Architecture, the underlying Testing Framework and the Test Beds:

- **Coverage of all eBusiness Interoperability Layers:** In view of the increasing number of eBusiness Specifications that are implemented and adopted at a global level, testing has to address all interoperability layers (i.e. business processes and choreography, business documents, transport and communication protocols) as well as profiles of them.

- **Testing Anywhere, Anytime**: Interoperability and Conformance Testing should not be restricted in time and place. Software vendors and end-users should be able to test their implementations over the Web anytime, anywhere and with any parties willing to do so. Interoperability Testing is expected to be repeated on a regular basis, as B2B networks and systems evolve continuously due to new versions of eBusiness Specifications, upgrades of eBusiness systems, changing business communities, and changing business requirements.

- **Reduction of Time Spent in Testing**: Considering the amount of Test Cases necessary to cover the conformance or Interoperability Testing requirements of eBusiness Specifications, the time spent by participants during the testing process should be significantly reduced by a testing methodology that favors reuse, automation and test integration. Partial coverage of the eBusiness stack by using disparate, unrelated tools for each layer is error prone and costly in terms of integration efforts and skills. The GITB Testing Framework aims to provide a comprehensive approach to eBusiness testing by integrating configuration management and other preliminary Test Steps into the testing process.

- **Ease of Design and Use**: The Test Bed will aim at the “low cost of entry” for its users and hence provide a graphical environment where a Test Designer can assemble the reusable Test Cases for conformance and Interoperability Testing.

- **Independence of Test Bed design from the eBusiness Specifications:** “Hard-coded” test logic in one-off Test Bed implementations is not desirable due to opacity, maintenance difficulties, non-reusable skills and platforms. The Test Bed design(s) have to be independent from eBusiness Specifications to be tested for.

- **Modularity:** Current eBusiness Specifications specify a variety of messaging protocols, business document formats or choreographies. In order to support all of these and test them, the Test Bed should be adaptable and modular. Therefore, it is necessary to define interfaces for several layers and facilitate plug-in modules supporting different protocols or formats implementing the specified interfaces.

- **Reuse of existing Test Beds and Test Suites:** A “Service” approach allows for reuse and leverage of existing Test Beds (legacy or not) and Test Suites. This reuse can be accomplished at design time – by creating Test Suites from existing components, assembling and deriving Test Cases from existing ones, reusing similar design patterns – or at run-time be enabling a distributed execution of a Test Suite over cooperating Test Beds.

- **Flexibility in Architecture:** The Testing Framework should allow for flexibility in architecture Test Bed designs. It may be instantiated, e.g. as centralized Test Bed or as distributed Test Bed using a service-oriented approach.

- **Standardized and Innovative Testing Methodologies** will ensure the successful development of testing of comprehensive eBusiness Specifications and Profiles.

4.2 Synthesis of GITB Testing Framework

The GITB Testing Framework focuses on the architecture, methodology and guidelines for assisting in the creation, use and coordination of Test Beds.

The GITB Testing Framework's constituents are two fold:

1. The **GITB Methodology** provides guidelines for eBusiness testing. It assists in specifying the subject of testing and the type of concern to test for ("what to test"). It also defines the means by which the testing goal is achieved ("how to test") and outlines typical testing scenarios (Standalone Document Validation, SUT-Interactive Conformance Testing, Interoperability Testing).

2. the **GITB Architecture** allows for a network of Test Beds to share Testing Resources and Testing Capabilities by means of services, yet also recommends an internal Test Bed design that promotes modularity and reuse.

The objectives in focusing on the definition of a Testing Framework and Architecture – as opposed to defining a specific Test Bed design – are:

- To define a general methodology and best practices related to all of the above, so that a common set of skills in designing tests and operating them, may be shared and applied across eBusiness disciplines.

- To promote reuse of functional components across eBusiness Test Beds while allowing variability in Test Bed architectural options,

- To allow for the portability and reuse of Test Artifacts across Test Beds by defining some level of standardization of these, and by facilitating their archival and discovery,

- To ensure the use of common design concepts across Test Beds, thus promoting a common understanding across eBusiness communities, and the same governance options,

4.3 Roles within the Testing Framework

The following roles, which generally correspond to different categories of Test Bed users and providers, are identified and supported by the Testing Framework:

- **Test Designer**: this role involves all tasks related to the creation of a Test Suite or of its parts (Test Cases, document assertion sets, configuration artifacts). The Test Designer may also be responsible for the creation of the set of Test Assertions from which Test Suite/Cases or Document Assertion Sets will be derived. S/he must have a good understanding of the eBusiness domain and specification(s) addressed by the Test Suite. S/he must also understand the testing conditions and constraints under which the Test Suite and Test Bed will be used, and the variability that the Test Suite must offer with respect to its reuse. The Test Designer is expected to be familiar with the Testing Framework methodology and best practices.

- **Test Participant**: The owner or operator of an SUT, typically the end-user, an integrator or a software vendor. This role defines the Verification Scope and Testing Requirements. This role is generally held by someone responsible for an eBusiness implementation, and having business domain expertise.

- **Test Manager**: A role responsible for executing Test Suites or for facilitating their execution, including related organizational tasks such as coordination with Test Participants. The Test Manager is an expert in Test Suites, and in the logistics involved in running tests. S/he is generally using the Test Bed on behalf of the Test Participants, or assisting the Test Participant in using the Test Bed,

e.g. for configuring and deploying a Test Suite before execution, and for searching/discovering the appropriate Test Suite in the Test Repository. S/he is also is familiar with the Test Suite logic and related eBusiness domain. Test Participants may act as Test Manager, if they are knowledgeable in testing.

- **Test Bed Provider**: This role is about operating the Test Bed itself as a server or an application service. It also may extend to the actual development and evolution of the Test Bed from Testing Framework resources and components (as obtained from the Test Repository). The Test Bed Operator is responsible for keeping the Test Bed functionally operational, and represents the Test Bed owning party for any contractual relationship with users, i.e. all other roles.

4.4 GITB Methodology

4.4.1 Using Test Assertions

Ideally, a set of Test Assertions have been defined for an eBusiness Specification before a Test Suite and Test Cases are developed. Test Assertions provide a way to bridge the narrative of an eBusiness Specification and the Test Cases for verifying conformance (or interoperability). Test Assertions help to interpret the specification statements from a testing viewpoint. Test Cases should then be derived from such Test Assertions, as illustrated in Figure 4-14-1.

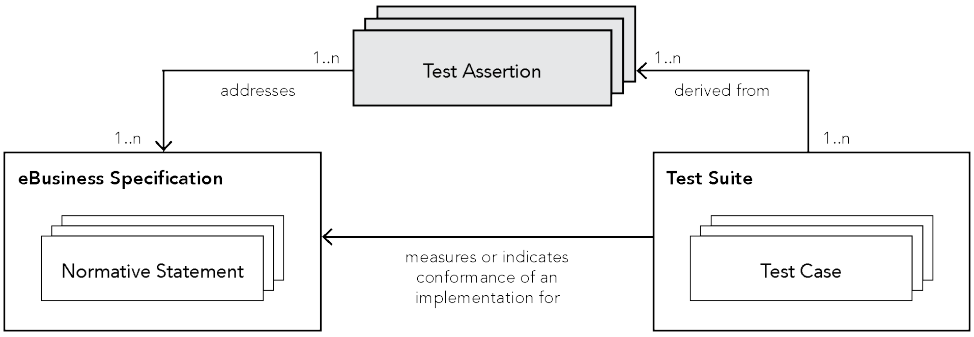

Figure 4-1: The Role of Test Assertions

Test Assertions provide a starting point for writing conformance and interoperability Test Suites. They simplify the distribution of the test development effort between different groups: often, Test Designers are not experts in the specification to be tested, and need guidance. By interpreting specification statements in terms of testing terms and conditions, Test Assertions improve confidence in the resulting Test Suite and provide the basis for coverage analysis (estimating the extent to which the specification is tested). OASIS has developed Test Assertions Guidelines (TAG) that can be used to help developing Test Assertions.

4.4.2 Standalone Document Validation

Document validation – also sometimes called “Instance” or “conformance/unit” testing – is a particular form of Conformance Testing which verifies a Test Item (e.g., an HL7 V2 message) against the rules defined in the specification. This form of testing does not directly involve a System Under Test (SUT), but rather a testing artifact (Test Item) that was produced by the SUT. Examples of such testing include validating a Clinical Document Architecture (CDA) document instance against the CDA general rules and document type rules, and validating an HL7 V2 message instance against an HL7 V2 conformance profile.

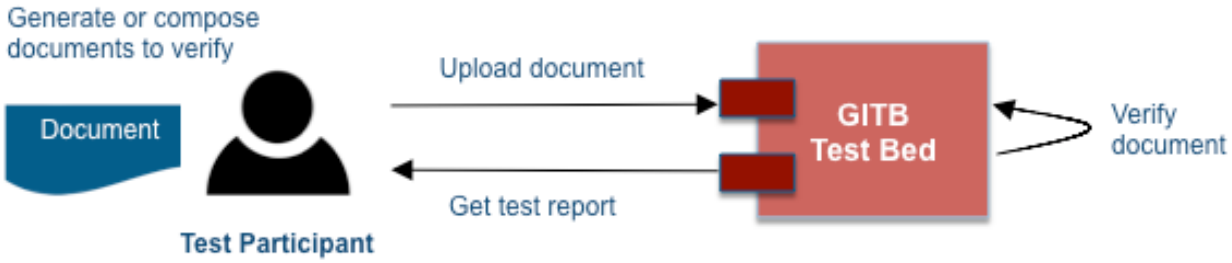

Figure 4-2: Workflow of a Standalone Document Validation

In "standalone" document validation, the document under test is obtained by a Test Participant, who directly submits the document to and gets the Test Report from the Test Bed. This document validation is then disconnected from any SUT communication, or larger Test Suite execution, as the Test Participant directly controls all inputs to the Test Bed.

4.4.3 SUT-Interactive Conformance Testing

Conformance Testing is defined as verifying an artifact (e.g., an HL7 V2 message) against the rules defined in the specification. Interactive Conformance Testing involves direct interaction between Test Bed and SUT, combined with dynamic validation of SUT outputs (document validation). The document validation is usually delegated by the Test Suite engine to a Document Validator.

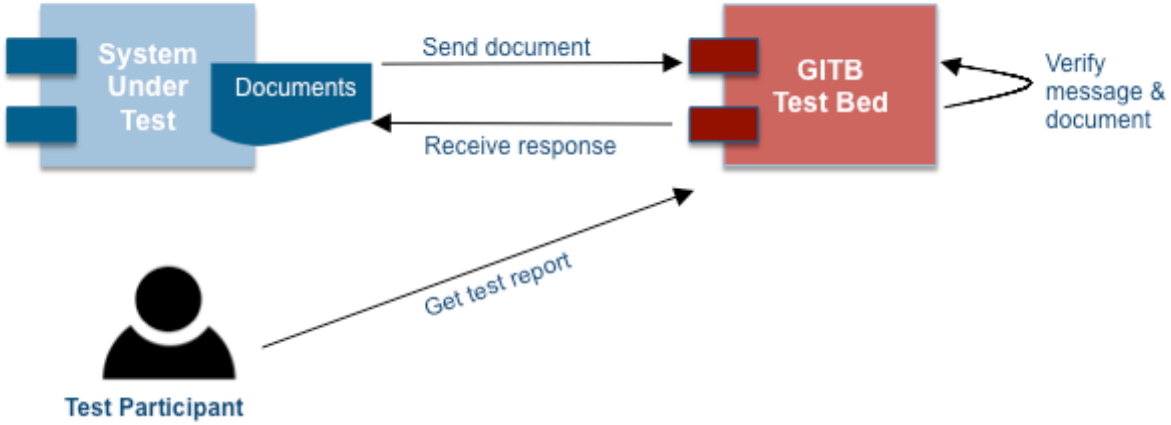

Figure 4-3: Sample Workflow in Interactive Conformance Testing

In such interactive Conformance Testing, the Test Participant (or Test Manager) only needs to interact with the Test Bed to control the overall execution and get the final report.

4.4.4 Interoperability Testing

Interoperability is defined as the ability of two SUTs to interact with each other in compliance with the specification. This interaction usually involves data artifacts (e.g. messages) produced by one SUT and consumed by the other. Interoperability Testing (see definition in section 2.1.2) can be conducted in different modes:

(1) Passive Interoperability Testing: in this mode, the SUTs are not controlled by the Test Suite, i.e. by a Test Bed. The SUTs interact on their own or under regular business activity. The interoperability Test Suite only verifies captured traffic: it is a validating Test Suite.

(2) Directly driven Interoperability Testing: in this mode, the interoperability Test Suite actively drives one or more SUTs in order to cause them to interact: it is an interacting Test Suite. In addition, the Test Suite (or another one, in case of "two-phase testing" – see next section) does the verification of captured traffic.

(3) Indirectly driven Interoperability Testing: in this mode, the SUTs are controlled indirectly by the Test Bed. The interoperability Test Suite interacts using a different channel with an entity controlling the SUT – e.g. sends an email to a Test Participant asking for initiation of a message from or to the SUT. In addition, the Test Suite (or another one, in case of "two-phase testing" – see next section) does the verification of captured traffic.

Ideally, the message capture should not interfere with the way the SUTs interoperate as they would under real business conditions. The three most common ways to capture message traffic between SUTs are:

a) Using a "man-in-the-middle" system operating and re-routing messages at transport level (e.g. an HTTP proxy or a TCP intermediary). This is typically the least intrusive approach, although it imposes restrictive conditions (the messages and sessions should not be encrypted).
b) Instrumenting of one of the SUT so that message capture is performed at the endpoint, e.g. on the message handler of the SUT. Later on this message capture can be consolidated in a Test Execution Log.
c) Configuring the sending SUT(s) so that they duplicate messages sent and forward a copy a Monitoring component or directly to the Test Bed.

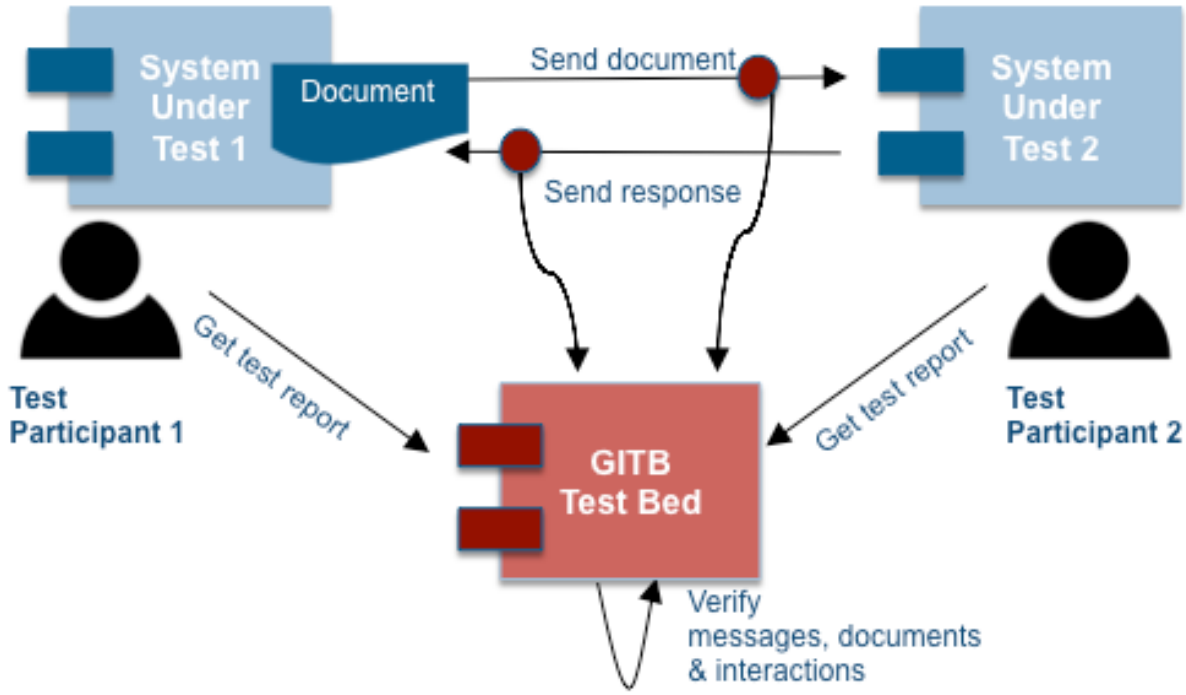

Figure 4-4: Basic Interoperability Testing

4.4.5 Proposed Testing Practices for SUTs

If possible, first perform Document/Message Instance Testing: The Document/Message Instance testing eliminates the problems within a single document. The structure and the business rules are checked. After passing the Document/Message Instance Testing, the SUT can guarantee that can generate valid documents/messages.

Always perform Conformance Testing: The Document/Message Instance Testing can ensure that a SUT can generate valid documents/messages. However, it cannot guarantee the SUT can send/receive these messages/documents as defined in the standard. Therefore, through the Conformance Tests, a SUT is tested to check whether it can send/receive messages in the order defined by the standard. In the Conformance Tests, all the other roles that the SUT communicates according to the specific standard are

simulated by the testing applications or Test Beds. Therefore, the SUT is expected to behave as if it is in real life settings. The business rules that should be applied across documents are also controlled in Conformance Tests.

Perform Interoperability Testing after the Conformance Testing, if possible design interoperability Test Suites so that they are not redundant with tests already done during Conformance Testing: Sometime fatal errors can be found during the Interoperability Testing. If so, Test Suites must be designed in such a way that those fatal errors are detected in the Conformance Testing. Through the interoperability tests more than one SUT is tested. Their ability to act with real-life settings is tested. In the certification process, most of the time, passing the Conformance Testing is sufficient. However, through Interoperability Testing, the interoperability with other real-life SUTs is tested.

4.5 GITB Architecture

For further portability and reuse, the Testing Framework defines a modular architecture based on standard Test Bed Component interfaces that allow for reusability of certain Testing Capabilities, and extensible plug-in design. A key tenet of interoperability and reuse across Test Beds is an information model that standardizes at appropriate levels the Test Artifacts to be processed (Test Cases, Test Suites, Test Reports, test configurations, etc.).

Figure 4-5Figure **4-5** provides an overview of the GITB Testing Architecture and its key elements:

- **Test Artifacts** that are processed by a Test Bed:

  (a) test logic documents (Test Suite definitions, document Test Assertions),

  (b) test configurations documents (parameters and message bindings for Test Suites, configuration of messaging adapters), and

  (c) test output documents (test logs and Test Reports).

- **Test Services** definitions and interfaces. These services are about managing the above Test Artifacts (design, deploy, archive, search) as well as controlling the major Test Bed functions (test execution and coordination).

- **Test Bed Components** and their integration. These components are functionally defined. They are of three kinds:

  (a) Core Test Bed platform components providing basic features, integration and coordination support to be found in any GITB-compliant Test Bed,

  (b) Testing Capability components, that directly enable the processing of Test Suites (e.g. a Test Suite engine, a Document Validator) and related tasks (e.g. send/receive messages),

  (c) User-facing components, through which the users interact with the Test Bed for various functions (e.g. Test Suite design, test execution).

- **Test Registry and Repository** for managing, archiving and sharing various Testing Resources. This component is not considered as part of a Test Bed, as it is a Testing Resource that can be independently deployed, managed and accessed. It supports the archiving, publishing and sharing of various Test Artifacts (e.g. Test Suites to be reused, Test Reports to be published). It also provides for storing and sharing Testing Capability components to be downloaded when assembling or upgrading a test Bed (e.g. the latest version of a Test Suite engine, of a Document Validator).

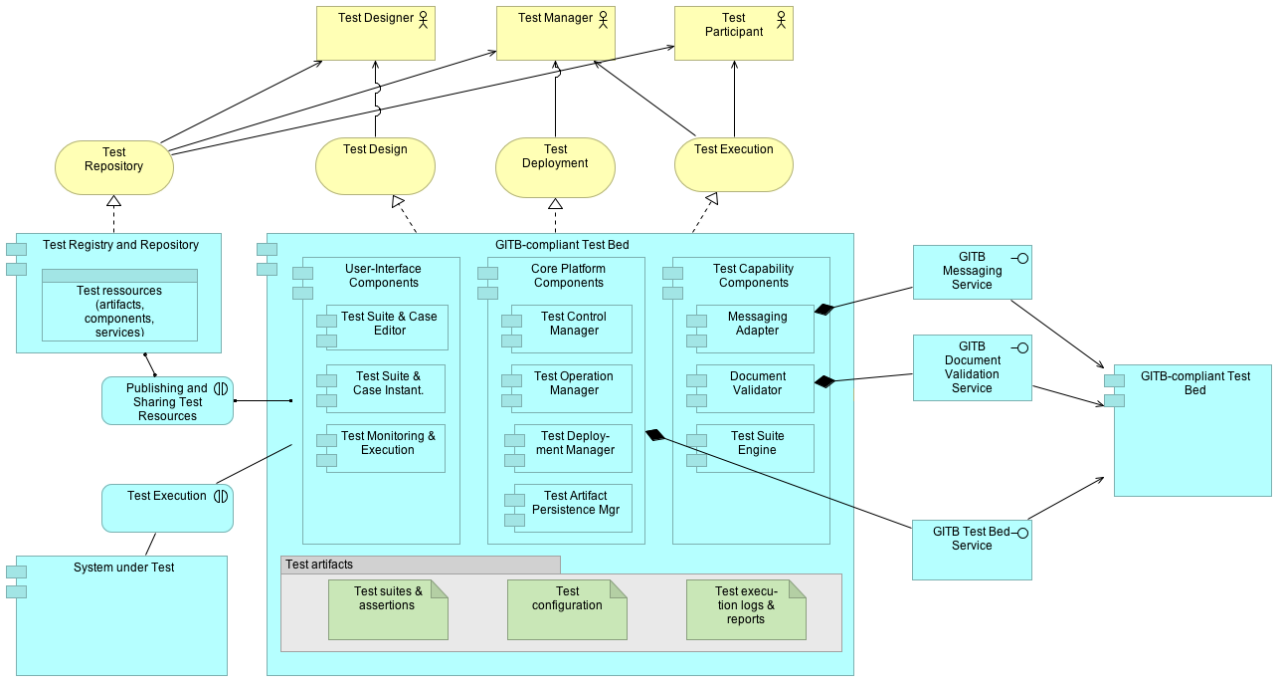

Figure 4-5: **Overview of the GITB Architecture**

In the proposed architecture, the GITB Test Bed is perceived by its users (either persons with specific roles or other Test Beds) as a set of Test Services. The Test Bed in itself allows for plug-in Testing Capabilities (for example, Test Suite engines, specialized validation components, message adapters, etc.). These Testing Capabilities can be supported either by existing (legacy) Test Beds, by remote services or by future test components to be developed. The Test Bed is also a platform where various Test Suites or Document Assertions can be deployed, i.e. it is not tied to a particular eBusiness Specification and its Test Suites.

The proposed architecture enables the coordination of several Test Beds specialized for the testing of different eBusiness Specifications, or for different testing procedures. Some of these Test Beds will be developed according to GITB recommendations, while others are Legacy Test Beds that have been augmented with GITB-compliant Service interfaces. Both types of Test Beds can then be integrated in the same network by providing access via similar Service interfaces. These Service interfaces can either be directly accessed by users, e.g. a Service Manager accessing the test services from a public interface (Web for instance), or they can be accessed by some other Test Beds, e.g. when a Test Suite executing on a Test Bed needs to delegate some document validation to another specialized Test Bed.

The Testing Capabilities (either provided by local components or by remote services) support the conformance and Interoperability Testing of any eBusiness Specification. For example, the purpose of a Document Validation capability is to validate a given document according to a set of syntactical or semantic restrictions specified in an eBusiness Specification. Such capability can be implemented as a local, pluggable (and reusable) component, or as a remote service from another Test Bed. Similarly a Messaging Adapter capability aims to communicate with the SUTs based on specified transport and communication protocols and to provide some level of messaging validation. Such a capability can be provided as a component that has been downloaded from a common repository for reuse and local integration, or could also be provided as a remote service, e.g. from a Test Bed or Test Agent specialized in providing various messaging protocols. Additional Testing Capabilities or services may be added to validate the conformance to a specified message choreography and business rules. For each of such capability, a common interface will be defined so that any test service provider can implement a test component or service specific to a certain standard and can plug-in the Testing Capability to the GITB Test Bed. In GITB phase 3, further capability types other than messaging and document validation may be identified and the architecture may be extended accordingly by the same approach.

This architecture promotes the reusability of Testing Resources and Capabilities among different domains and different standards. As shown in Figure 4-5**Figure 4-5**, a Test Designer developing a Test Case for a certain eBusiness profile or standard may need a test service (e.g. profile may state that in a certain

transaction the communication should be performed via ebXML messaging, so we need ebXML communications with the SUT) which may already be developed and published by other Test Designers and test service providers working for another domain or standard. In this way, the GITB Testing Framework leverages the existing, distributed test services related to eBusiness testing and allows users to discover them and access them via the Test Registry and Repository.

Part II: Core Test Bed Implementation Specifications and Proof-of-Concept

GITB Phase 3 complements and refines the GITB Testing Framework and the Test Bed Architecture defined in the previous phases by

- defining the **implementation specifications for GITB service interfaces and selected artifacts** to achieve interoperability between testing facilities developed for different domains, specifications, or regions.

- designing a **reference Test Bed Architecture** and underlying **Test Description Language** based on GITB principles for stakeholders in different domains for their future testing facilities (domains that do not have structured conformance and interoperability test frameworks)

- developping the open source **Proof-of-Concept Implementation of a Test Bed** based on GITB specifications and architecture.

Part II of this report summarizes GITB Phase 3 outcomes related to the Core Test Bed. It is relevant for **testing experts and architects** that are interested in the detailed Test Bed Architecture and Specifications.

5 Overview of Core Test Bed Implementation Specifications

5.1 Relevant Core Test Bed Service Specifications and Artifacts

The **GITB Service Specifications** are a group of specifications for testing facilities, Test Beds, content validation tools, simulators, messaging handlers to achieve reusability of testing functionalities among them. The following are brief descriptions for each of them:

- The **GITB Content Validation Service Specification** defines a service where any content validation tool can implement to wrap its functionalities and serve them as a content validation service to other stakeholders. In some domains, there are already such services used by testing systems to delegate the content validation job to remote services. The Gazelle External Validation Service (EVS) in eHealth domain and PEPPOL Document Validation Service in eProcurement domain are some examples in this respect.

- In conformance and interoperability testing, testbeds need mechanisms to communicate with SUTs based on the protocol specified in the target specification or in other words simulate a specific actor to handle these communications. Communication protocols are used among different domains and reuse of these simulation facilities among the testing frameworks of different domain will be very useful. The **GITB Messaging (Simulator) Service Specification** defines a service to achieve this interoperability between testbeds and simulators. For example, an AS4 protocol simulator can be used by different domains in their testing frameworks to establish AS4 communication with SUTs. Then these domains will only concentrate on their specific testing requirements based on their extensions or profiling approach over the AS4 protocol.

- In addition to the reusability of more granular testing facilities, accessing testbeds' facilities in a common way will also be very useful for conformance and interoperability testing. The **GITB Test Bed Service Specification** will define this common service definition to drive a testbed remotely for the execution of a complete conformance or interoperability testing scenario. As specifications referring other specifications for conformance (profiles, customizations) are becoming more and more common in many domains, a testbed using another testbed's facilities is also becoming a basic requirement. With this specification, it is also possible to implement individual test monitoring interfaces driving multiple testbeds for test scenario executions.

All these services require a common model for a number of test artifacts:

- All these services require a common model to report the results of the performed tests so that the client side can understand the results and render them to its users. The **GITB Report Format Specification** defines a model for representing test reports. It is a wrapper format to describe the brief summary of the results. Based on the validation methodology any report format (ex:

Schematron Validation Report Language for schematron validations, a proprietary format for XML schema validations) can be used within this model.

- In order to realize GITB Test Bed Service, a common model is needed to describe a test scenario between the testbeds. As different testbeds use different models or languages to represent executable test scenario descriptions, it is not possible to find a common executable model. In fact, it is not necessary. The model only needs to define the basics of the execution flow by describing it in terms of granular test steps with a simple categorization. The **GITB Test Presentation Language Specification** provides this model to represent a conformance or interoperability test scenario.

In addition to the service specifications, the GITB reference Test Bed Architecture is designed based on GITB principles in this phase. An important part of this architecture is the **GITB Test Description Language (TDL)** that defines the high level executable scripting language for the Test Bed. Stakeholders that need but do not have such conformance and interoperability test frameworks can use this architecture and the TDL as reference to build one for their specific needs. As the architecture is designed with GITB principles, the resulting testing frameworks will facilitate reusing of testing capabilities among different stakeholders and domains.

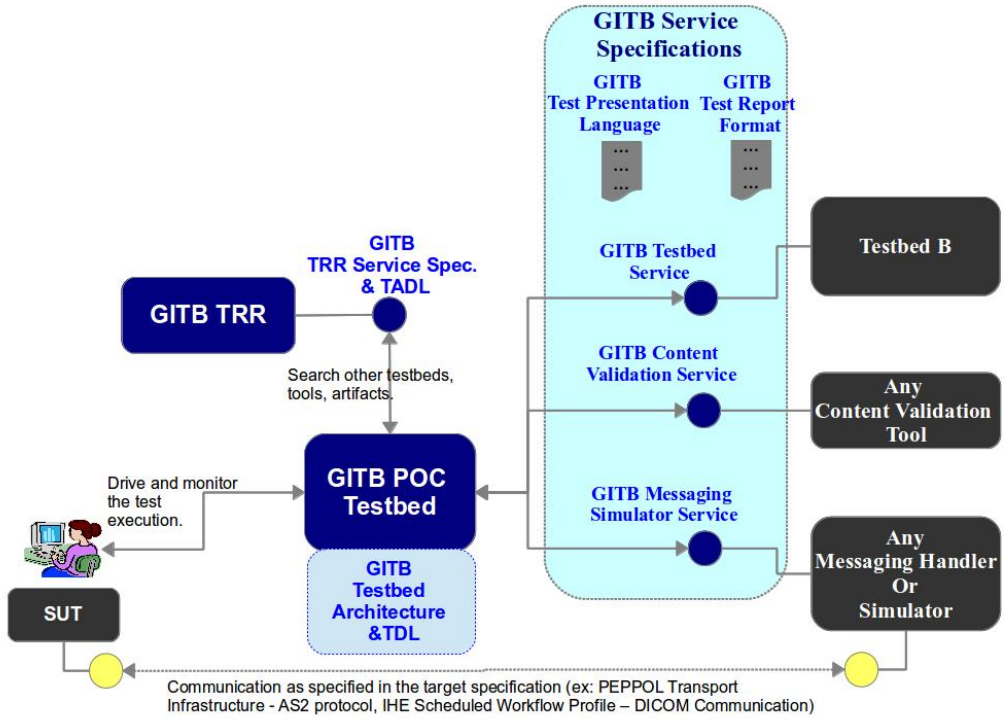

Figure 5-1: GITB Implementation Specifications

5.2 GITB Namespaces and Common Element Definitions

In this section, we describe the common element definitions used by all Test Bed Implementation Specifications. We recommend using this section as a reference while reading other parts.

Table 5-1: GITB Namespaces

Prefix	XML Namespace	Comments
gitb:	http://www.gitb.com/core/v1/	The core schema defining the common elements for other models.
vs:	http://www.gitb.com/vs/v1/	GITB Validation Service namespace

ms:	http://www.gitb.com/ms/v1/	GITB Messaging Service namespace
tbs:	http://www.gitb.com/tbs/v1/	GITB Testbed Service namespace
tpl:	http://www.gitb.com/tpl/v1/	GITB Test Presentation Language namespace
tr:	http://www.gitb.com/tr/v1/	GITB Test Reporting Model namespace
tdl:	http://www.gitb.com/tdl/v1/	GITB Test Description Language namespace

The **<gitb:Metadata>**is a common element to describe the metadata of the container element (ex: Testcase, TestModule, TestSuite).

- **title –** Name of the container. Should be descriptive for users.
- **type (0..1) –** Only used for testcases and indicates the type of the test case (CONFORMANCE or INTEROPERABILITY).
- **description (0..1) –** Long description of the container.
- **version** – Version of the container description
- **authors (0..1)** - List of authors who compose the container artifact
- **issued (0..1)** – Publication date for the container artifact
- **modified (0..1)** – Last modification date for the container artifact

The **<gitb:TestRole>** declares the actor in a Test Case definition that will take part in the test scenario;

- **id** – The unique identifier for the actor. It should be recommended to use URN format to uniquely identify the actor for the related test bed.
- **name –** Short name given to the actor (for referencing actor within the test case definition).
- **role** – The role of the actor within the test scenario. Value should be used from the enumeration (SUT, SIMULATED, MONITOR). If the test case aims to test (either conformance or interoperability testing) an actor, the SUT role should be given. If the role of a given actor is played by the test engine or some simulator within the test scenario, the SIMULATED should be used. The MONITOR value is used for further scenarios representing users that do not involve in the target business process but involve in the testing process for monitoring the test execution or perform manual validations.

The **<gitb:AnyContent>** class is used to embed some content (ex: the message or document content) in a container element related with a messaging, validation or user interaction operation in a generic way and while transferring data between GITB modules/services. It describes the way to reach the content, abstract type of the content (in the type system of Test Bed) and how it is serialized to the receiver module so that it can parse the content accordingly.

- **item (1..*)** – **AnyContent –** If the content carries list of contents then this element recursively represents the carried content. For simple contents only <gitb:value> element should be used. For container types (list or map) each item represents the content of the container items.
- **value (0..1) –** The actual content itself (either in string or base64 encoded representation) or the URL to access to the actual content.
- **name –** Name for the content item.
- **embeddingMethod (0..1) –** This attribute states the method that describes how the content is embedded in the value part. The value should be from the enumeration **<gitb: ValueEmbeddingEnumeration>** (BASE64, STRING, URI). The BASE64 indicates that the content is embedded in the format of base64 encoded string within the value. The STRING indicates that the xs:string representation of content is embedded into the value. Finally, URI indicates that an URL is given in the value from which the actual content is accessible over the Internet. The default is BASE64. **type (0..1) –** GITB enables implementers to extend the abstract type system of GITB when implementing GITB compliant services and testbeds. This attribute indicates the type of the content according to the type system of the target GITB compliant testbed or service. (ex: DICOM object, EDI content, etc). See GITB Type system.
- **encoding (0..1) –** If the type is given this attribute provides the serialization format of the content for the given abstract type (ex: XML serialization, JSON serialization, etc).

The **<gitb:Parameter>** defines a configuration parameter for any GITB module or service.

- **name –** Name of the parameter
- **use (0..1)** – Specifies whether parameter is required or optional for the operation. (R: required, O:optional). Default is “R”.
- **kind (0..1)** – Configuration value can be simple string or a binary content read from a file and this attribute indicates the kind of configuration (SIMPLE,BINARY). Default is SIMPLE.
- **desc (0..1) –**Describes the functionality of the configuration parameter within the related process.
- **value –** Default value of the configuration parameter if not provided within the operation.

The **<gitb:TypedParameter>** extends <gitb:Parameter> class to represent a typed value for input and output definitions of modules, constructs, and services defined in GITB

- **type –** Identifier for abstract parameter type (Based on the type system of the target GITB Compliant Service or Test Bed)
- **encoding (0..1) –** Identifier for the serialization format for the type (Based on the type system of the testbed). The default encoding of the type should be assumed when this attribute is not supplied.

The **<gitb:Configuration>**element is used to provide the value of a configuration parameter for the container construct.

- **name –** Name of the parameter
- **value** – Value of the parameter

The **<gitb:ActorConfiguration>** is used to provide configurations for each SUT in the process between two testing facility.

- **actor** – Identifier of the actor that the configurations are supplied for.
- **endpoint (0-1) –** Identifier for the endpoint that the configurations are supplied for. If actor has only one endpoint there is no need to supply it .
- **config (1..*): <gitb:Configuration>** – List of configurations for the given system (playing the given actor)

The **<gitb:Actor>** element defines an actor in a testbed and declares the endpoints of the actor and the required configuration parameters for those endpoints.

- **name –** Unique identifier of the actor (URN) within the testbed.
- **desc (0..1) –**The textual description of the actor
- **endpoint (1..*) – <gitb:Endpoint> –** The list of endpoint definitions.
  - **name –** Name of the endpoint (should be unique within the Actor definition)
  - **config (0..*) – <gitb:Parameter> –** Configuration parameters for the actor. When a SUT claims conformance to this actor, before the execution of a related test scenario, the configurations stated here should be collected from the SUT administrator.

5.2.1 XML Schema for Common Elements

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xsd:schema            xmlns="http://www.gitb.com/core/v1/"           xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.gitb.com/core/v1/" elementFormDefault="qualified" version="1.0">
     <xsd:element name="module" type="TestModule"/>
     <!--General Metadata element to describe the metadata of artifacts-->
     <xsd:simpleType name="ID">
          <xsd:restriction base="xsd:string">
               <xsd:pattern value="[a-zA-Z0-9_]"/>
          </xsd:restriction>
     </xsd:simpleType>
     <xsd:complexType name="Metadata">
          <xsd:sequence>
               <xsd:element name="title" type="xsd:string"/>
             <xsd:element name="type" type="TestCaseType" default="CONFORMANCE" minOccurs="0"/>
               <xsd:element name="version" type="xsd:string"/>
               <xsd:element name="authors" type="xsd:string" minOccurs="0"/>
```

```
            <xsd:element name="description" type="xsd:string" minOccurs="0"/>
            <xsd:element name="issued" type="xsd:string" minOccurs="0"/>
            <xsd:element name="modified" type="xsd:string" minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="ActorConfiguration">
        <xsd:sequence>
            <xsd:element name="config" type="Configuration" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="actor" type="xsd:string" use="required"/>
        <xsd:attribute name="endpoint" type="xsd:string" use="optional"/>
    </xsd:complexType>
    <!-- List of Actor Definitions-->
    <xsd:complexType name="Actors">
        <xsd:sequence>
            <xsd:element name="actor" type="Actor" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <!-- Complete Definition of an Actor-->
    <xsd:complexType name="Actor">
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="desc" type="xsd:string" minOccurs="0"/>
            <xsd:element name="endpoint" type="Endpoint" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="ID" use="required"/>
    </xsd:complexType>
    <!--Endpoint definition-->
    <xsd:complexType name="Endpoint">
        <xsd:sequence>
            <xsd:element name="config" type="Parameter" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="ID" use="required"/>
        <xsd:attribute name="desc" type="xsd:string" use="optional"/>
    </xsd:complexType>
    <!-- List of Testcase Role Definitions-->
    <xsd:complexType name="Roles">
        <xsd:sequence>
            <xsd:element name="actor" type="TestRole" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <!--List of Testcase Role definition-->
    <xsd:complexType name="TestRole">
        <xsd:sequence>
            <xsd:element name="endpoint" type="Endpoint" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID" use="required"/>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
        <xsd:attribute name="role" type="TestRoleEnumeration" use="required"/>
    </xsd:complexType>
    <!-- Configuration name-value pair -->
    <xsd:complexType name="Configuration">
        <xsd:simpleContent>
            <xsd:extension base="xsd:string">
                <xsd:attribute name="name" type="xsd:string" use="required"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>
    <!-- Abstract TestModule definition -->
    <xsd:complexType name="TestModule">
        <xsd:sequence>
            <xsd:element name="metadata" type="Metadata"/>
            <xsd:element name="inputs" type="TypedParameters" minOccurs="0"/>
            <xsd:element name="outputs" type="TypedParameters" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:string"/>
        <xsd:attribute name="uri" type="xsd:string"/>
        <xsd:attribute name="isRemote" type="xsd:boolean" default="true"/>
        <xsd:attribute name="serviceLocation" type="xsd:string" use="optional"/>
    </xsd:complexType>
    <!-- Validation Module Definition-->
    <xsd:complexType name="ValidationModule">
        <xsd:complexContent>
            <xsd:extension base="TestModule">
                <xsd:sequence>
                    <xsd:element name="configs" type="ConfigurationParameters" minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="operation" type="xsd:string" use="optional"/>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <!--Messaging Module Definition-->
    <xsd:complexType name="MessagingModule">
        <xsd:complexContent>
            <xsd:extension base="TestModule">
                <xsd:sequence>
                    <xsd:element name="actorConfigs" type="ConfigurationParameters"/>
                    <xsd:element name="transactionConfigs" type="ConfigurationParameters" minOccurs="0"/>
                    <xsd:element name="listenConfigs" type="ConfigurationParameters" minOccurs="0"/>
                    <xsd:element name="receiveConfigs" type="ConfigurationParameters" minOccurs="0"/>
                    <xsd:element name="sendConfigs" type="ConfigurationParameters" minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="isProxy" type="xsd:boolean" use="optional" default="true"/>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <!--List of configuration parameters-->
    <xsd:complexType name="ConfigurationParameters">
        <xsd:sequence>
```

```
            <xsd:element name="param" type="Parameter" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <!--List of typed paremeters-->
    <xsd:complexType name="TypedParameters">
        <xsd:sequence>
            <xsd:element name="param" type="TypedParameter" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <!--Parameter Definition-->
    <xsd:complexType name="Parameter">
        <xsd:simpleContent>
            <xsd:extension base="xsd:string">
                <xsd:attribute name="name" type="xsd:string" use="required"/>
                <xsd:attribute name="use" type="UsageEnumeration" use="optional" default="R"/>
                <xsd:attribute name="kind" type="ConfigurationType" use="optional" default="SIMPLE"/>
                <xsd:attribute name="desc" type="xsd:string" use="optional"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>
    <!--Typed parameter definition-->
    <xsd:complexType name="TypedParameter">
        <xsd:simpleContent>
            <xsd:extension base="Parameter">
                <xsd:attribute name="type" type="xsd:string" use="required"/>
                <xsd:attribute name="encoding" type="xsd:string" use="optional"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>
   <!-- Enumeration for test case types-->
   <xsd:simpleType name="TestCaseType">
       <xsd:restriction base="xsd:NMTOKEN">
          <xsd:enumeration value="CONFORMANCE"/>
          <xsd:enumeration value="INTEROPERABILITY"/>
       </xsd:restriction>
   </xsd:simpleType>
    <!-- Enumeration for usage indicator-->
    <xsd:simpleType name="UsageEnumeration">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="R"/>
            <xsd:enumeration value="O"/>
        </xsd:restriction>
    </xsd:simpleType>
    <!-- Enumeration for usage indicator-->
    <xsd:simpleType name="ConfigurationType">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="SIMPLE"/>
            <xsd:enumeration value="BINARY"/>
            <!-- simple type, i.e. string -->
            <!-- binary type -->
        </xsd:restriction>
    </xsd:simpleType>
    <!--Representation (serialization) of a GITB value -->
    <xsd:complexType name="AnyContent">
        <xsd:choice>
            <xsd:element name="item" type="AnyContent" minOccurs="1" maxOccurs="unbounded"/>
            <xsd:element name="value" type="StringUriOrBase64Type" minOccurs="1" maxOccurs="1" />
        </xsd:choice>
        <xsd:attribute name="name" type="xsd:string" use="optional"/>
        <xsd:attribute name="embeddingMethod" type="ValueEmbeddingEnumeration" use="optional" default="BASE64"/>
        <xsd:attribute name="type" type="xsd:string" use="optional"/>
        <xsd:attribute name="encoding" type="xsd:string" use="optional"/>
    </xsd:complexType>
    <!-- Enumeration for embedding method for the value -->
    <xsd:simpleType name="ValueEmbeddingEnumeration">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="STRING"/>
            <xsd:enumeration value="BASE64"/>
            <xsd:enumeration value="URI"/>
        </xsd:restriction>
    </xsd:simpleType>
    <!-- Type definition for the actual content-->
    <xsd:simpleType name="StringUriOrBase64Type">
        <xsd:union memberTypes="xsd:string xsd:base64Binary xsd:anyURI"/>
    </xsd:simpleType>
    <!-- Enumeration for representing the format of the given content-->
    <xsd:simpleType name="TestRoleEnumeration">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="SUT"/>
            <xsd:enumeration value="SIMULATED"/>
            <xsd:enumeration value="MONITOR"/>
        </xsd:restriction>
    </xsd:simpleType>
    <!-- Enumeration indicating the status of a test step execution-->
    <xsd:simpleType name="StepStatus">
        <xsd:restriction base="xsd:NMTOKEN">
            <xsd:enumeration value="PROCESSING"/>
            <xsd:enumeration value="SKIPPED"/>
            <xsd:enumeration value="WAITING"/>
            <xsd:enumeration value="ERROR"/>
            <xsd:enumeration value="COMPLETED"/>
        </xsd:restriction>
    </xsd:simpleType>
   <xsd:simpleType name="ErrorCode">
       <xsd:restriction base="xsd:string">
          <xsd:enumeration value="ACTOR_DEFINITION_NOT_FOUND"/>
          <xsd:enumeration value="ARTIFACT_NOT_FOUND"/>
          <xsd:enumeration value="CANCELLATION" />
          <xsd:enumeration value="DATATYPE_ERROR" />
```

```
            <xsd:enumeration value="INTERNAL_ERROR"/>
            <xsd:enumeration value="INVALID_SESSION" />
            <xsd:enumeration value="INVALID_TEST_CASE" />
            <xsd:enumeration value="MISSING_CONFIGURATION" />
            <xsd:enumeration value="INVALID_CONFIGURATION" />
            <xsd:enumeration value="TEST_CASE_DEFINITION_NOT_FOUND"/>
        </xsd:restriction>
    </xsd:simpleType>
    <xsd:complexType name="ErrorInfo">
        <xsd:sequence>
            <xsd:element name="errorCode" type="ErrorCode" />
            <xsd:element name="description" type="xsd:string" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

6 Test Presentation Language (TPL)

Every Test Bed or testing tool uses some model to define the test execution flows for the automated processing of test scenarios. Most of them use some type of scripting languages (ex: TTCN3, OASIS TAML) to represent the execution model. Some of them do not have such a concrete representation, but still have some abstract model behind which is implemented either within a software or database. Generally, the details of these models are strongly dependent on the underlying testbed architecture and technology. However, it is observed that these models and approaches show strong similarities when we focus on the unit testing actions and their main functionalities. For example, all test execution models have constructs to define a messaging step between a SUT and the simulator or to define a verification step to validate message content. From the user (SUT administrator) point of view, the important point is to be able to understand the basics of the testing process (the test flow, what is realized in each step in general, and what is expected from him and the SUT). The situation is similar for the scenarios, where a Test Bed drives another Test Bed or testing tool to perform specific set of actions and tests. A common abstract test scenario definition model will help us to establish this agreement among the components (testbeds, tools, test monitoring environments) and users (software vendors, SDOs, test developers) of our vision of global interoperability testing network.

The **GITB Test Presentation Language (TPL)** will provide the specification for this common model or language to represent a conformance or interoperability test scenario. The resulting language is neither a scripting language, nor used for automated test execution. Rather, its purpose is to present the flow and the test steps in a granular way to users and other testing software that want to interoperate with the testbed providing the test execution service for the scenario. Any testbed can easily map their internal test execution models, or test scripting languages to this abstract common model to describe its test scenarios to the outside world.

6.1 Abstract Model

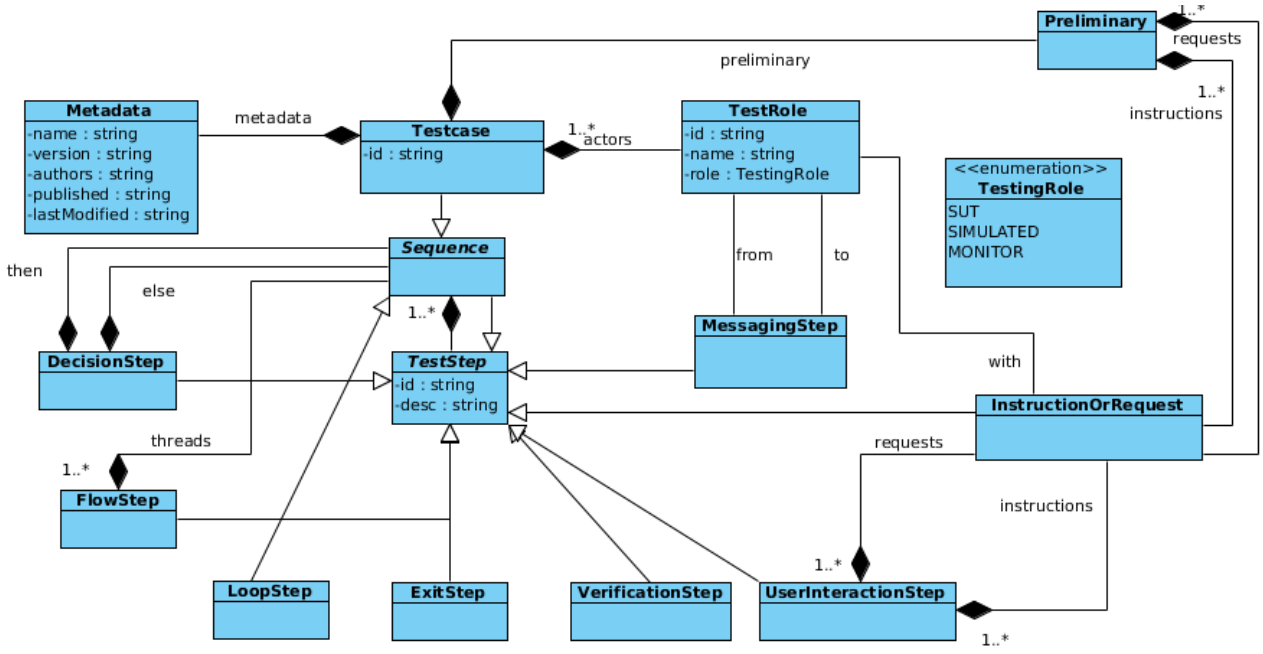

Figure 6-1: Test Presentation Language Model

Figure 6-16-1: illustrates the abstract model of the TPL. The root element representing the test scenario is the **<tpl:*Testcase*>**. Its attributes and elements are as follows;

- **id** – Attribute defines the unique identifier for the test case. It is recommended to use a URN for the value of this attribute. (ex: urn:gitb:ihe:xds-document-source-conformace-test, urn:gitb:peppol:lime-protocol-conformance-test)
- **metadata:<gitb:Metadata>** – Describes the metadata attributes (name, description, author, version, etc) of the test case.
- **actors (1..*): <gitb:TestRole>** – Describes the actors in the business process defined by the test scenario's target specification (ex: Supplier in PEPPOL profiles, Document Consumer in IHE profiles) and the role assignments regarding the testing process.
- **preliminary (0..1): <tpl:Preliminary>** – Describes the preliminary requirements that should be shown to the SUT administrators before starting the test execution.
- **steps: <tpl:Sequence> -** List of test step descriptions that describes the flow and each test step.

The **<tpl:Preliminary>** element is a container for the preliminary steps in the test case;

- **instruct (0..*): <tpl:Instruction> –** Preliminary instructions for the SUT administrators that describes some requirement regarding the test scenario.
- **request (0..*): <tpl:UserRequest> –** Preliminary requests from the SUT administrators related with the test scenario. The SUT administrators are expected to provide the requested information as an input to the test case definition where these inputs will be used later in the test execution process. Inputs will be related with the test scenario requirements.

The **<tpl:Sequence>** element is a container for test steps that will be processed in the given order.

- **steps (1..*) <tpl:TestStep> –** <tpl:TestStep> is an abstract class that describes the granular unit step of a test case. The Sequence class is a list of these test steps which will be executed in linear order. Test steps are categorized in seven categories; VerificationStep, MessagingStep, DecisionStep, FlowStep, LoopStep, ExitStep and UserInteractionStep and each extends the TestStep definition.

The **<tpl:TestStep>** is the abstract class that represents a test step in the definition.

- **id –** The unique identifier for the step within the test case definition. This identifier will be used to bind test step reports to test steps. Since a test execution can include decision steps, concurrent executions and loops, a special identification scheme is recommended for test step identification (See Section 6.2).
- **desc (0..1)** – Textual description of the test step which can be shown to the user to describe what this test step is doing and what is expected from the user.

The **<tpl:VerificationStep>** describes the actual validation or verification step (ex: XML schema validation of message/document content, Schematron validation of message/document content, XPATH expression validation of a value within the message/document content, or custom validation of a non-XML content).

The **<tpl:MessagingStep>** describes a messaging step between a SUT and a simulator or between two SUTs (in interoperability tests). This step indicates that a message communication is expected between the given actors at that time. The related SUT administrators should behave accordingly (drive the SUT) to initiate the messaging if necessary (some messages are initiated by other messages without any manual intervention with the user). Each <tpl:MessagingStep> represents only one way of communication, in other words, for request-response type communication two <tpl:MessagingStep> should exist.

- **from** – Refers to the actor (Actor.name) which is expected to send the message.
- **to** – Refers to the actor which is expected to receive the message.

Test execution flows generally include decision points where the execution is continued on certain branch based on a condition. The **<tpl:DecisionStep>** element defines such supplementary test steps. The desc attribute should describe the condition in textual form in order to enable users understand the behaviour and test flow.

- **then (0..1):<tpl:Sequence>** – Gives the sequence of test steps that test execution will follow when the condition holds.
- **else (0..1):<tpl:Sequence>** – Gives the sequence of test steps that test execution will follow when the condition does not hold.

The **<tpl:FlowStep>** element indicates the concurrent execution of the child sequences. The step is completed when all branches are completed.

- **thread (1..*): <tpl:Sequence> –** The child sequences that are executed concurrently.

The **<tpl:LoopStep>** extending the <tpl:Sequence> indicates that child steps will be executed a number of times in loop based on some condition. As the main aim of the TPL is showing the flow of the scenario and describing it textually to the users, the looping condition should be described in the “desc” attribute textually.

The **<tpl:ExitStep>** indicates that the test execution will be stopped with this step.

The **<tpl:UserInteractionStep>** indicates that testbed will interact with the specified users in this step in order to instruct them or get some input regarding test execution.

- **with (0..1):** Refer to the actor that this user interaction is performed with.
- **instruct (0..*): <tpl:Instruction>** – Indicates that an instruction will be shown to thespecified user
- **request (0..*): <tpl:UserRequest> –** Indicates that some input is requested from the specified user

The <tpl:Instruction> and <tpl:UserRequest> types extend **<tpl:InstructionOrRequest>** and they represent an interaction step either an instruction for a SUT administrator (former) or an input request from a SUT administrator (later).

- **with –** Refer to the actor (<gitb:TestRole>.name) that this instruction will be shown.

6.2 Test Step Identification

The basic requirement for the identification of test steps is that each step should have unique ids within the same test case definition. However, we recommend a methodology to assign identifiers to test steps in order to make them more readable and understandable. The test case definition with the TPL is in fact an execution flow tree where each node is a test step. Traversal of this tree represents the execution order of the test steps. The identification methodology is in fact an id scheme to represent this traversal. The following rules are to follow while assigning ids to test steps;

- Test steps within the main sequence are assigned successive numbers starting from “1”. Therefore, first test step in the sequence will be identified as “1”, second as “2”, and so on.
- If a step is <tpl:DecisionStep>, the child sequence “then” will be identified as the concatenation of the id of <tpl:DecisionStep> with “[T]”. For example, if <tpl:DecisionStep> is the third step in the main sequence, then sequence will be “3[T]”. Similarly, the “else” sequence will be identified as “[F]”.
- A step within a sequence is identified as the concatenation of the id given to the sequence, the dot (“.”) and the successive numbering given to the step. For example, if we have a sequence identified as 3[T], the first step will be “3[T].1”, and the second will be “3[T].2”.
- If a step is <tpl:FlowStep>, the child sequences will be identified with the concatenation of the id of the <tpl:FlowStep> with the successive numbers for the child sequences within brackets. For example, is <tpl:FlowStep> has id “3[T].2”, the first child sequence of <tpl:FlowStep> will be “3[T].2[1]”, and the second child sequence will be “3[T].2[2]”.
- For a <tpl:LoopStep>, as the <tpl:LoopStep> is also a sequence the rule for child numbering of a seuqnece is applicable. For example, if <tpl:LoopStep> is numbered as “5”, the first child will be “5.1”.
- <tpl:UserInteractionStep> should be viewed as the <tpl:Sequence> and same numbering scheme should be applied.

6.3 XML Schema for TPL

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xsd:schema            xmlns="http://www.gitb.com/tpl/v1/"            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:gitb="http://www.gitb.com/core/v1/"                          targetNamespace="http://www.gitb.com/tpl/v1/"
elementFormDefault="qualified" version="1.0">
     <xsd:import namespace="http://www.gitb.com/core/v1/" schemaLocation="gitb_core.xsd"/>
     <xsd:element name="testcase" type="TestCase"/>
     <xsd:complexType name="TestCase">
          <xsd:sequence>
               <xsd:element name="metadata" type="gitb:Metadata"/>
               <xsd:element name="actors" type="gitb:Roles"/>
               <xsd:element name="preliminary" type="Preliminary" minOccurs="0"/>
               <xsd:element name="steps" type="Sequence"/>
          </xsd:sequence>
          <xsd:attribute name="id" type="xsd:string" use="required"/>
     </xsd:complexType>
     <xsd:complexType name="Sequence">
          <xsd:complexContent>
               <xsd:extension base="TestStep">
                    <xsd:sequence>
                         <xsd:choice maxOccurs="unbounded">
                              <xsd:element name="msg" type="MessagingStep"/>
                              <xsd:element name="decision" type="DecisionStep"/>
                              <xsd:element name="loop" type="Sequence"/>
                              <xsd:element name="flow" type="FlowStep"/>
                              <xsd:element name="verify" type="TestStep"/>
                              <xsd:element name="exit" type="ExitStep"/>
                              <xsd:element name="interact" type="UserInteractionStep"/>
                         </xsd:choice>
                    </xsd:sequence>
               </xsd:extension>
          </xsd:complexContent>
     </xsd:complexType>
     <xsd:complexType name="TestStep">
          <xsd:sequence>
               <xsd:element name="desc" type="xsd:string" minOccurs="0"/>
          </xsd:sequence>
          <xsd:attribute name="id" type="xsd:string" use="required"/>
     </xsd:complexType>
     <xsd:complexType name="DecisionStep">
          <xsd:complexContent>
               <xsd:extension base="TestStep">
                    <xsd:sequence>
                         <xsd:element name="then" type="Sequence"/>
                         <xsd:element name="else" type="Sequence" minOccurs="0"/>
                    </xsd:sequence>
               </xsd:extension>
          </xsd:complexContent>
     </xsd:complexType>
     <xsd:complexType name="ExitStep">
          <xsd:complexContent>
               <xsd:extension base="TestStep">
                    <xsd:sequence/>
               </xsd:extension>
          </xsd:complexContent>
     </xsd:complexType>
     <xsd:complexType name="FlowStep">
          <xsd:complexContent>
               <xsd:extension base="TestStep">
                    <xsd:sequence>
                         <xsd:element name="thread" type="Sequence" maxOccurs="unbounded"/>
                    </xsd:sequence>
               </xsd:extension>
          </xsd:complexContent>
     </xsd:complexType>
     <xsd:complexType name="UserInteractionStep">
          <xsd:complexContent>
               <xsd:extension base="TestStep">
                    <xsd:sequence>
                         <xsd:choice maxOccurs="unbounded">
                              <xsd:element name="instruct" type="Instruction"/>
                              <xsd:element name="request" type="UserRequest"/>
                         </xsd:choice>
                    </xsd:sequence>
                    <xsd:attribute name="with" type="xsd:string"/>
               </xsd:extension>
          </xsd:complexContent>
     </xsd:complexType>
     <xsd:complexType name="Preliminary">
          <xsd:sequence>
               <xsd:choice maxOccurs="unbounded">
                    <xsd:element name="instruct" type="Instruction"/>
                    <xsd:element name="request" type="UserRequest"/>
               </xsd:choice>
          </xsd:sequence>
     </xsd:complexType>
     <xsd:complexType name="Instruction">
          <xsd:complexContent>
               <xsd:extension base="InstructionOrRequest"/>
          </xsd:complexContent>
     </xsd:complexType>
     <xsd:complexType name="UserRequest">
          <xsd:complexContent>
               <xsd:extension base="InstructionOrRequest"/>
          </xsd:complexContent>
     </xsd:complexType>
```

```
    <xsd:complexType name="InstructionOrRequest">
        <xsd:complexContent>
            <xsd:extension base="TestStep">
                <xsd:attribute name="with" type="xsd:string" use="optional"/>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="MessagingStep">
        <xsd:complexContent>
            <xsd:extension base="TestStep">
                <xsd:sequence>
                    <xsd:element name="from" type="xsd:string"/>
                    <xsd:element name="to" type="xsd:string"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
</xsd:schema>
```

7 Test Reporting Format

Test Reports document the result of verifying the behavior or output of one or more SUT(s), or verifying Test Items such as Business Documents. The following sections define a model for representing test reports so that the client side can understand the results and render them to its users.

7.1 Abstract Model

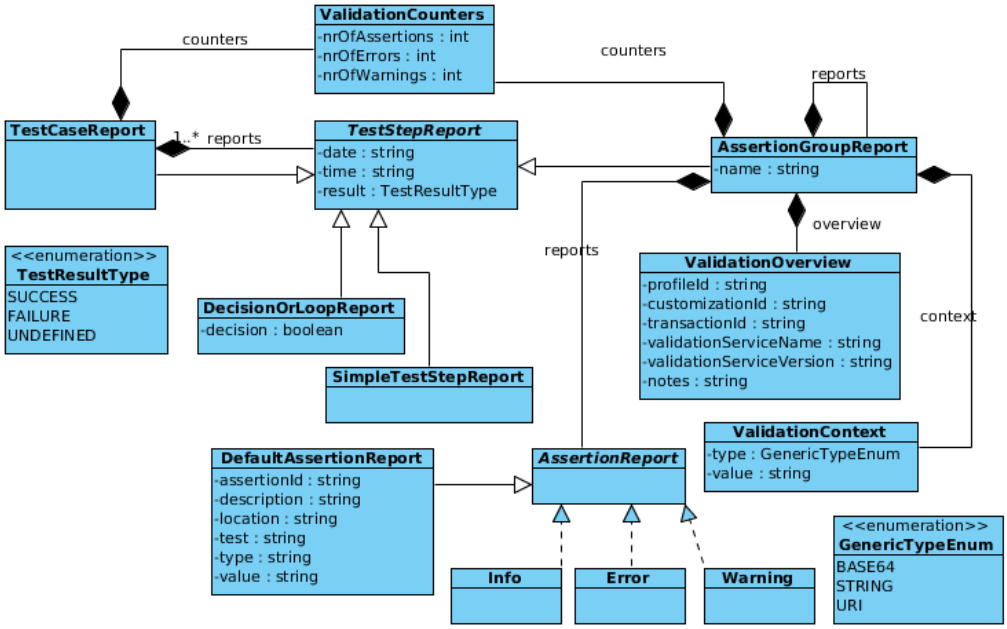

Figure 7-1: GITB Test Reporting Model

Figure 7-17-1: illustrates the abstract model for test reporting. The <tr:TestCaseReport> represents the report for a test case execution and composed of <tr:TestStepReport> elements each represent the report for the defined test step in the TPL.

The **<tr:TestCaseReport>** contains the following elements;

- **id –** The identifier of the test execution instance (test instance id) that this report is related.
- **date** – Date and time that the test case is executed
- **result** – the result of the test execution, can take values from the <tr:TestResultType> enumeration (SUCCESS, FAILURE, UNDEFINED). SUCCESS indicates that case is successfully executed and FAILURE indicates that there are some errors (non-conformant parts). UNDEFINED represents other situations where test case is not executed completely or existence of some errors (internal system errors) not related with conformity or testing process.
- **counters (0..1):<tr:ValidationCounters>** – Provides the number of assertions, errors and warnings based on the tests done within the test case.
- **reports (0..*): <tr:TestStepReport>** – The list of test step reports for each step executed within the test case execution.

The **<tr:TestStepReport>** is the base class for representing any test step report.

- **id –** Identifier of the test step that this report is related
- **date –** Date and time that this test step is executed.
- **result -** The result of the test execution, can take values from the <tr:TestResultType> enumeration (SUCCESS, FAILURE, UNDEFINED). SUCCESS indicates that step is successfully executed and FAILURE indicates that there are some errors (non-conformant parts) related with the step.

UNDEFINED represents other situations where step is not executed completely or existence of some errors (internal system errors) not related with conformity or testing process.

The **<tr:DR>** element (DecisionOrLoopReport) represents reports for decision steps and loop steps that changes the execution flow based on some condition.

- **decision** – Provides the resulting Boolean value for the condition. This value indicates how test flow will continue in the next steps.

The **<tr:SR> (SimpleTestStepReport)** represents the reports for all other step types (<tpl:ExitStep>, <tpl:UserInteractionStep>, and <tpl:FlowStep>). It is basicly the implementation of the abstract <tpl:TestStepReport> class.

The**<tr:TAR> (TestAssertionReport)** is used to represent reports for messaging and verification steps.

- **name –** Descriptive name for the test assertion group (ex: XML Schema Validation, Business Rule Validations, etc).
- **overview (0..1): <tr:ValidationOverview> –** Provides information about the validation tool/service used for the validation process and target specification for conformance checks.
- **counters (0..1): <tr:ValidationCounters> –** Provides the number of assertions, errors and warnings based on the tests done within this assertion group
- **context (0..1): <gitb:AnyContent> –** For verification steps, this element provides the content that validation is done on. For example, the XML message where schematron validation is performed. For messaging steps, similarly it provides the related part of the message.
- **reports (0..*): <tr:TestAssertionGroupReport> –** Different testbed infrasturctures, validation procedures may have different methodology to group validations/assertions. A validation step may correspond to a simple assertion (ex: XPath expression validation) or a set of validation procedures to perform a complete conformance test of a message content (ex: XML schema validation + Schematron Validation). In order to provide the flexibility to testbeds for the grouping of assertions, TestAssertionGroup element is designed to include further groups recursively.

The **<tr:TestAssertionGroupReport>** is used to represent the reports for a set of test assertions or test assertion groups recursively. The element either includes assertion group reports recursively (reports element) or reports for each assertion (info,warning or error elements) in the group

- **reports (1..*) : <tr:TAR> –** If the reports are organized into a further grouping, each of these elements provides the reports for child groups.
- **[info] OR [warning] OR [error]**:**<tr:TestAssertionReport> –** Represents the leaf reports for the smallest unit of validation process (ex: report from a schematron assertion). <tr:**TestAssertionReport>** is an abstract class and one of the realizations; Info, Error,Warning elements will be used for reports. **<tr:Info>** represents the assertions that are successful. **<tr:Error>**represents the assertions with erroneous result and **<tr:Warning>**represents the successful results but with some warnings. <tr: **TestAssertionReport** > abstract class provides a wrapper for different report formats. Testbeds can extend it to define their own report formats for different validation procedures. One implementation of it, the <tr:BAR> is given in this section to be an example assertion report format for basic validation procedures.

The **<tr:ValidationOverview>**provides some further information regarding the validation procedure and the target specification.

- **profileId (0..1)** – An identifier for the target specification that the validation step is related with. SDOs generally assigns identifiers to their specifications or part of the specifications (scenarios, use cases), and these can be used for this attribute.
- **customizationId (0..1)** – If the target specification is customized to a specific region/country or some specific purpose, an identifier for this customization can be given for this attribute.
- **transactionId (0..1)** – An identifier for the transaction/mesage or document type that the validation is performed on.
- **validationServiceName (0..1)** – Name of the validation service or tool that performs the validation

- **validationServiceVersion (0..1) –** Version of the validation service or tool that performs the validation
- **note (0..1)** – Any textual note regarding the validation

The **<tr:ValidationCounters>**is the container for the validation statistics.

- **nrOfAssertions (0..1)** – Total number of assertions evaluated in this assertion group
- **nrOfErrors (0..1)** –Total number of errors from those assertions within the assertion group
- **nrOfWarnings (0..1)** – Total number of warnings from those assertions within the assertion group

The **<tr:BAR> (BasicAssertionReport)** provides a default <tr:TestAssertionReport> realization that can be used for many of the existing validation methodologies.

- **assertionId (0..1)** – Optional attribute to give an identifier to the assertion. It can be used to vind the assertion to a constraint, rule, or similar concepts defined within the specification.
- **description** - Textual description for the assertion result.
- **location (0..1)** – The expression that indicates the location of the error or warning within the content. For example, an XPATH expression can be used for Schematron reports to indicate the error location.
- **test (0..1)** This attribute gives the expression itself that performs the validation if it is possible to give such an expression.
- **type (0..1)** – This attribute describes the type of assertion (ex: cardinality check, usage control) if testbed provides such categorization
- **value (0..1)** – Some assertions checks the value of an element or attribute within a message/document content (ex: check if it is equal to some value, or in some specific format). This attribute gives the actual value within the content to enable the user to understand the assertion semantics and error better.

7.1.1 XML Schema for Test Reporting Format

```
 <?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.gitb.com/tr/v1/"
           xmlns="http://www.gitb.com/tr/v1/"
           xmlns:gitb="http://www.gitb.com/core/v1/"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified">

    <xsd:import namespace="http://www.gitb.com/core/v1/" schemaLocation="gitb_core.xsd"/>

    <!-- Root elements for TestCaseReport and TestStepReport-->
    <xsd:element name="TestCaseReport" type="TestCaseReportType"/>
    <xsd:element name="TestStepReport" type="TestStepReportType"/>
    <!-- Represents the Testcase Report-->
    <xsd:complexType name="TestCaseReportType">
        <xsd:complexContent>
            <xsd:extension base="TestStepReportType">
                <xsd:sequence>
                    <xsd:element name="counters" type="ValidationCounters" minOccurs="0"/>
                    <xsd:element name="reports" type="TestStepReportType" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <!-- Abstract type for Test Step Report-->
    <xsd:complexType name="TestStepReportType" abstract="true">
        <xsd:sequence>
            <xsd:element name="date" type="xsd:dateTime"/>
            <xsd:element name="result" type="TestResultType"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:string" use="optional"/>
    </xsd:complexType>
    <!--Report format for Decision and Loop steps-->
    <xsd:complexType name="DR">
        <xsd:complexContent>
            <xsd:extension base="TestStepReportType">
                <xsd:sequence>
                    <xsd:element name="decision" type="xsd:boolean"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <!-- Report format for simple steps (exit, interact, flow)-->
    <xsd:complexType name="SR">
        <xsd:complexContent>
```

```
            <xsd:extension base="TestStepReportType"/>
        </xsd:complexContent>
    </xsd:complexType>
    <!--Report format for Messaging and Validation Steps-->
    <xsd:complexType name="TAR">
        <xsd:complexContent>
            <xsd:extension base="TestStepReportType">
                <xsd:sequence>
                    <xsd:element name="overview" type="ValidationOverview" minOccurs="0"/>
                    <xsd:element name="counters" type="ValidationCounters" minOccurs="0"/>
                    <xsd:element name="context" type="gitb:AnyContent" minOccurs="0"/>
                    <xsd:element name="reports" type="TestAssertionGroupReportsType" minOccurs="0"/>
                </xsd:sequence>
                <xsd:attribute name="name" type="xsd:string" use="optional"/>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <!-- The Overview of Validation procedure-->
    <xsd:complexType name="ValidationOverview">
        <xsd:sequence>
            <xsd:element name="profileID" type="xsd:string" minOccurs="0"/>
            <xsd:element name="customizationID" type="xsd:string" minOccurs="0"/>
            <xsd:element name="transactionID" type="xsd:string" minOccurs="0"/>
            <xsd:element name="validationServiceName" type="xsd:string" minOccurs="0"/>
            <xsd:element name="validationServiceVersion" type="xsd:string" minOccurs="0"/>
            <xsd:element name="note" type="xsd:string" minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>
    <!-- Statistics of the Validation -->
    <xsd:complexType name="ValidationCounters">
        <xsd:sequence>
            <xsd:element name="nrOfAssertions" type="xsd:integer" minOccurs="0"/>
            <xsd:element name="nrOfErrors" type="xsd:integer" minOccurs="0"/>
            <xsd:element name="nrOfWarnings" type="xsd:integer" minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>
    <!--Supplementary class to include either AssertionReports or AssertionGroups -->
    <xsd:complexType name="TestAssertionGroupReportsType">
        <xsd:choice>
            <xsd:element name="reports" type="TAR" maxOccurs="unbounded"/>
            <xsd:choice maxOccurs="unbounded">
                <xsd:element name="info" type="TestAssertionReportType"/>
                <xsd:element name="warning" type="TestAssertionReportType"/>
                <xsd:element name="error" type="TestAssertionReportType"/>
            </xsd:choice>
        </xsd:choice>
    </xsd:complexType>
    <!-- Abstract Test Assertion Report class -->
    <xsd:complexType name="TestAssertionReportType" abstract="true"/>
    <!-- Base assertion report format for GITB-->
    <xsd:complexType name="BAR">
        <xsd:complexContent>
            <xsd:extension base="TestAssertionReportType">
                <xsd:all>
                    <xsd:element name="assertionID" type="xsd:string" minOccurs="0"/>
                    <xsd:element name="description" type="xsd:string"/>
                    <xsd:element name="location" type="xsd:string" minOccurs="0"/>
                    <xsd:element name="test" type="xsd:string" minOccurs="0"/>
                    <xsd:element name="type" type="xsd:string" minOccurs="0"/>
                    <xsd:element name="value" type="xsd:string" minOccurs="0"/>
                </xsd:all>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <!-- Enumeration for representing the test result-->
    <xsd:simpleType name="TestResultType">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="SUCCESS"/>
            <xsd:enumeration value="FAILURE"/>
            <xsd:enumeration value="UNDEFINED"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:schema>
```

8 GITB Test Service Specifications

The aim of the GITB Test Service specifications is to provide common service specifications for the existing conformance and interoperability testing facilities so that they can be used remotely by others, either from within the same domain or from a different domain, for their own testing requirements. Based on the GITB Testing Framework, three main services are identified as modular services that can be used between different testing setups when executing conformance and interoperability tests. In this section, we will describe these services by providing abstract service specification and Web Service binding (WSDL description) for each of the services:

- Content Validation Service
- Messaging (Simulation) Service
- TestBed Service

8.1 Content Validation Service

8.1.1 Service Overview

Figure 8-1 illustrates the remote content validation scenario between the *ValidationService* and *ValidationClient* actors. The content validation tools that want to implement the Content Validation Service interface should play the ValidationService role in the scenario. The ValidationClient role can be played by i) testbeds that wants to use the remote testing capability for specific content validations, and ii) any other systems or organizations (vendors, etc) that want to use the system for their internal testing procedures.

1. **vs:getModuleDefinition:** The first step in the interaction is to retrieve the definition of the validation module. Module definition provides the details regarding the validation operations that module supports and inputs that the module takes if validation operation is supported.
2. **vs:validate:** This is the actual validation operation. ValidationClient should prepare the inputs based on the module definition and supply them properly. Any content validation tool can be wrapped as Content Validation Service with this operation. All the validation operations returns the test report (the <tr:TAR> in the TPL) providing the overall result and description of performed assertions, found errors and warnings.

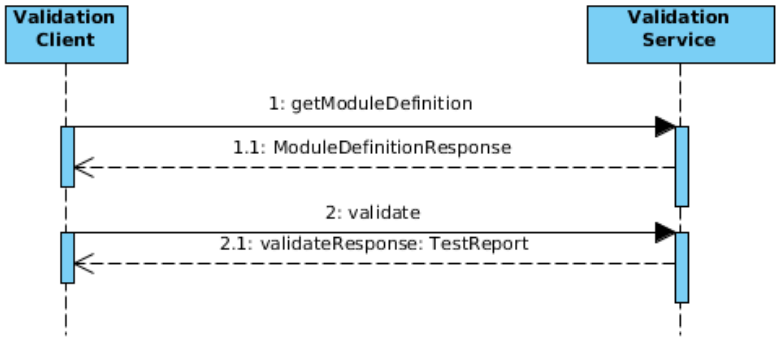

Figure 8-1: Sequence Diagram for Content Validation Service

8.1.2 Abstract Service Description

8.1.2.1 ValidationClient Requests Module Definition

The **<vs:GetModuleDefinitionRequest>** does not take any parameter and is used to request the service's description object. In response, namely **<vs:GetModuleDefinitionResponse>**, the service should return the **<gitb:ValidationModule>** for which the model is described below;

- **id** – A unique identifier for the validation service itself (can be used by client sides to distinguish different validation services).
- **uri –** The URL of the service endpoint

- **operation(0..1)**– Operation supported by the service. Should take value form enumeration (VC: validateByContentType, VS: validateBySchema, V:validate). The default is “V”.
- **metadata: <gitb:Metadata> –** Metadata regarding the service (name, description, authors, version, etc).
- **config (0..*): <gitb:Parameter> -** Configuration parameters for the module to change the behavior in the validation process
- **input (0..*): <gitb:TypedParameter> -** Describes the input parameters for the validation operation if it is supported.

8.1.2.2 Validation

The ValidationClient should send **<vs:ValidateRequest>** message to the service with the following details;

- **sessionId -** An identifier for the session between the client and the service.
- **config(0..*): <gitb:Configuration> -** Supplied configuration parameter values for the validation process.
- **input(1..*): <gitb:AnyContent> –** The supplied input parameters for the validation. The parameters should be in the same order with the parameters as defined in the module definition and should be matched for the type and encoding.

The <vs:ValidationResponse> should be returned.

8.1.3 Web Service Description (WSDL)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!-- edited with XMLSpy v2008 sp1 (http://www.altova.com) by SRDC (EMBRACE) -->
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http://www.gitb.com/vs/v1/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" name="ValidationService"
targetNamespace="http://www.gitb.com/vs/v1/">
    <types>
        <xsd:schema>
            <xsd:import namespace="http://www.gitb.com/vs/v1/" schemaLocation="gitb_vs.xsd"/>
        </xsd:schema>
    </types>
    <message name="getModuleDefinition">
        <part name="parameters" element="tns:GetModuleDefinitionRequest"/>
    </message>
    <message name="getModuleDefinitionResponse">
        <part name="parameters" element="tns:GetModuleDefinitionResponse"/>
    </message>
    <message name="validate">
        <part name="parameters" element="tns:ValidateRequest"/>
    </message>
    <message name="validateResponse">
        <part name="parameters" element="tns:ValidationResponse"/>
    </message>
    <portType name="ValidationService">
        <operation name="getModuleDefinition">
            <input message="tns:getModuleDefinition"/>
            <output message="tns:getModuleDefinitionResponse"/>
        </operation>
        <operation name="validate">
            <input message="tns:validate"/>
            <output message="tns:validateResponse"/>
        </operation>
    </portType>
    <binding name="ValidationServicePortBinding" type="tns:ValidationService">
        <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="getModuleDefinition">
            <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
        <operation name="validate">
            <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>
    <service name="ValidationService">
        <port name="ValidationServicePort" binding="tns:ValidationServicePortBinding">
            <soap:address location="/service/ValidationService"/>
```

```
        </port>
    </service>
    <!-- to generate sources in given package -->
    <jaxws:bindings xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
        <jaxws:package name="com.gitb.vs">
      </jaxws:package>
    </jaxws:bindings>
</definitions>
```

8.1.4 XML Schema for Request/Response Messages

```
 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xsd:schema version="1.0" targetNamespace="http://www.gitb.com/vs/v1/"
    xmlns="http://www.gitb.com/vs/v1/"
     xmlns:tns="http://www.gitb.com/vs/v1/"
    xmlns:tr="http://www.gitb.com/tr/v1/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:gitb="http://www.gitb.com/core/v1/">

    <xsd:import namespace="http://www.gitb.com/tr/v1/" schemaLocation="gitb_tr.xsd"/>
    <xsd:import namespace="http://www.gitb.com/core/v1/" schemaLocation="gitb_core.xsd"/>

     <xsd:element name="GetModuleDefinitionRequest" type="tns:Void" />
     <xsd:element name="GetModuleDefinitionResponse" type="tns:GetModuleDefinitionResponse" />
    <xsd:element name="ValidateRequest" type="tns:ValidateRequest" />
    <xsd:element name="ValidationResponse" type="tns:ValidationResponse" />

    <xsd:complexType name="Void">
          <xsd:sequence/>
     </xsd:complexType>

     <xsd:complexType name="GetModuleDefinitionResponse">
          <xsd:sequence>
               <xsd:element name="module" type="gitb:ValidationModule" />
          </xsd:sequence>
     </xsd:complexType>

    <xsd:complexType name="ValidateRequest">
        <xsd:sequence>
            <xsd:element name="sessionId" type="xsd:string" />
            <xsd:element name="config" type="gitb:Configuration" minOccurs="0" maxOccurs="unbounded" />
            <xsd:element name="input" type="gitb:AnyContent" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="ValidationResponse">
        <xsd:sequence>
            <xsd:element name="report" type="tr:TAR" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

8.2 Messaging (Simulation) Service

8.2.1 Service Overview

Figure 8-28-2 illustrates a scenario where a client (a testbed) drives a Messaging Service to handle the message communication in a test scenario for a specific communication protocol.

1. **ms:getModuleDefinition:** Client can use this operation to get the module definition, which is in fact the definition of the details related with inputs and outputs of the actual messaging commands.
2. **ms:initiate:** Before starting the use the messaging service for messaging simulations, a session should be established between the client and the service to prepare the service for the operations. The client provides the required configurations (ex: network configurations) of the SUTs that will communicate with the simulator. The service should perform all the initializations and preparations for the messaging operations in this phase. In response, a session id and the related configurations of the simulator (ex: network configurations, etc) should be returned.
3. **ms:beginTransaction:** Different domains requires different communication protocols which differs in message exchanging patterns. Most of the protocols (ex: Web Service communication) are based on request-response pattern over a single network connection. However, there are more complex patterns (ex: DICOM Communication) that requires multiple message exchanges, even not in pairs of request-response, over a single network connection. In order to be generic while handling the connections and message exchanges, separate commands are designed to notify the service accordingly. This command notifies the service that a new communication will start with a SUT within the next messaging command and makes the service to be ready for this.

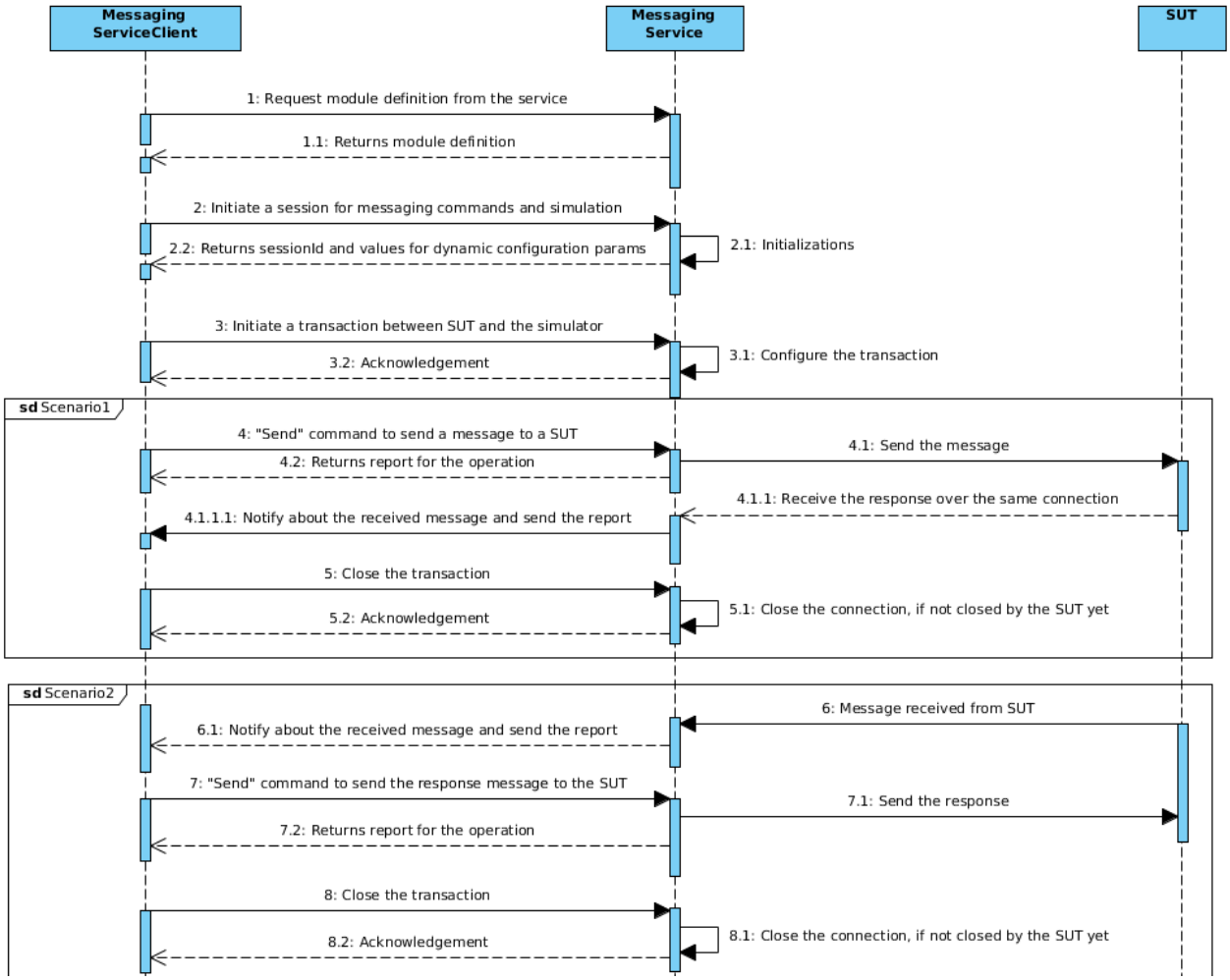

Figure 8-2: Messaging Service Scenario

Scenario I: We will describe the messaging commands by different scenarios to describe the expected behaviours better. In the first scenario, assume that according to our test case we need to send a message to the SUT and then it will return a response to it.

4. **ms:send:** The “send” command is used to drive the service to send a message to a SUT. Necessary configurations and message contents should be supplied within the command as described in the module definition. The service should prepare the whole message and send it to the SUT. When message is sent, the client will be notified. According to our assumption regarding the protocol, SUT immediately returns the response. The Messaging Service is expected to notify the client with this reponse and its validation report.
5. **ms:endTransaction:** This commands notify the service that this transaction is completed. The service can release the resources (connections, etc) related with the transaction.
6. **ms:finalize:** Finalize the session between messaging service and the client.

Scenario II: This time the SUT is expected to initiate the communication by sending a message to our simulator service.

7. **ms:NotifyForMessage:** The service should be ready to receive the message from the SUT any time after beginTransaction command. When the message is received, service notifies the client with the received message and its report. The client then use the “send” command to send the response to this message. The service sends the given message to the SUT and returns the report. GITB Messaging Service does not differentiate between acknowledgements and application level responses. It is the responsibility of test designers and module implementers to design the service and arrange the commands according to the scenario and setup. For example, using a reference implementation or messaging software as a Messaging Service can automatically return acknowldgements. In that case, test designer does not need to put another send command for the acknowledgement.

8.2.2 Abstract Service Description

8.2.2.1 Requesting Module Definition (GetModuleDefinition)

The client use this operation to retrieve the module definition from the service to understand the input and output parameters for messaging commands. The **<ms:GetModuleDefinitionRequest>** does not take any parameter and is used to request the service's description object. In response, namely **<ms:GetModuleDefinitionResponse>**, the service should return the **<gitb:MessagingModule>** element.

8.2.2.2 Initiating the Session (Initiate)

The client uses this operation to establish a session with MessaginService and provides the configuration parameters of the SUTs that will involve in messaging. The details of the **<ms:InitiateRequest>**are as follows;

- **actorConfiguration (1..*): <gitb:ActorConfiguration>** – Configurations for each SUT that will communicate with this simulator in the process
  - **name** – An identifier for the actor that the configurations are supplied for.
  - **config (1..*): <gitb:Configuration>** – List of configurations for the given system (playing the given actor)

In response, the service should return the **<ms:InitiateResponse>**;

- **sessionId –** An unique identifier for the session.
- **actorConfiguration (1..*): <gitb: ActorConfiguration>** – List of configurations for the simulator.

8.2.2.3 Initiating a Transaction (BeginTransaction)

This command is used to notify the service that communication is expected between the SUT and itself after this point of time. The details of **<ms:BeginTransactionRequest>** are as follows;

- **sessionId** – The session identifier related with this transaction
- **config (0..*): <gitb:Configuration> -** Further configurations related with the transaction
- **from (0..1)** – The name of the actor (refers the name of the actor given in the <gitb:ActorConfiguration> in Initiate operation) that will initiate the transaction
- **to (0..1)** – The name of the actor that will be on the other side

8.2.2.4 Commanding Messaging Service to Send a Message (Send)

This command is used to make the service to send the given message to a SUT. The details of **<ms:SendRequest>** are as follows;

- **sessionId** - The session identifier related with this operation
- **to** - The name of the actor that the message will be send
- **input (1..*): <gitb:AnyContent> –** The inputs (message parts) supplied to the service. The service will use this inputs to construct the actual message.

In response the SendResponse should be returned;

- **report: <tr:AssertionGroupReport>** – The validation report generated for the operation.

8.2.2.5 Notification of the Client for Received or Proxied Messages (NotifyForMessage callback)

When the service received a message from the SUT as expected according to the scenario, it should use the callback and send the **<ms:NotifyForMessageRequest>**to the client;

- **sessionId –** The session identifier related with the message
- **from (0..1)** – The name of the actor that message is received from
- **to (0..1) –** The name of the actor that message is sent to (only used for Listen/proxy operations)

- **report: <tr:AssertionGroupReport>** – The received message parts and validation report generated for the operation.

8.2.2.6 Closing the Transaction (EndTransaction)

When the communication between two actors is completed according to the scenario, the client can use this operation to notify the service about this so that it can release the related resources. The **<ms:EndTransactionRequest>** is as follows;

- **sessionId** - The session identifier related with this transaction

8.2.2.7 Closing the Session (Finalize)

When the messaging is finished, this command can be used to finalize the session. The **<ms:FinalizeRequest>** is used in this operation;

- **sessionId** - The session identifier

8.2.3 Web Service Description (WSDL) for Messaging Service (Service Provider)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.gitb.com/ms/v1/"
             name="MessagingService"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:tns="http://www.gitb.com/ms/v1/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
             xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl">
    <types>
        <xsd:schema>
            <xsd:import namespace="http://www.gitb.com/ms/v1/" schemaLocation="gitb_ms.xsd" />
        </xsd:schema>
    </types>
    <message name="getModuleDefinition">
        <part name="parameters" element="tns:GetModuleDefinitionRequest" />
    </message>
    <message name="getModuleDefinitionResponse">
        <part name="parameters" element="tns:GetModuleDefinitionResponse" />
    </message>
    <message name="initiate">
        <part name="parameters" element="tns:InitiateRequest" />
    </message>
    <message name="initiateResponse">
        <part name="parameters" element="tns:InitiateResponse" />
    </message>
    <message name="beginTransaction">
        <part name="parameters" element="tns:BeginTransactionRequest" />
    </message>
    <message name="beginTransactionResponse">
        <part name="parameters" element="tns:BeginTransactionResponse" />
    </message>
    <message name="send">
        <part name="parameters" element="tns:SendRequest" />
    </message>
    <message name="sendResponse">
        <part name="parameters" element="tns:SendResponse" />
    </message>
    <message name="endTransaction">
        <part name="parameters" element="tns:EndTransactionRequest" />
    </message>
    <message name="endTransactionResponse">
        <part name="parameters" element="tns:EndTransactionResponse" />
    </message>
    <message name="finalize">
        <part name="parameters" element="tns:FinalizeRequest" />
    </message>
    <message name="finalizeResponse">
        <part name="parameters" element="tns:FinalizeResponse" />
    </message>
    <portType name="MessagingService">
        <operation name="getModuleDefinition">
            <input
                    wsam:Action="http://gitb.com/MessagingService/getModuleDefinition"
                    message="tns:getModuleDefinition" />
            <output
                    wsam:Action="http://gitb.com/MessagingService/getModuleDefinitionResponse"
                    message="tns:getModuleDefinitionResponse" />
        </operation>
        <operation name="initiate">
            <input
                    wsam:Action="http://gitb.com/MessagingService/initiate"
                    message="tns:initiate" />
            <output
                    wsam:Action="http://gitb.com/MessagingService/initiateResponse"
```

```
                    message="tns:initiateResponse" />
        </operation>
        <operation name="send">
            <input
                    wsam:Action="http://gitb.com/MessagingService/send"
                    message="tns:send" />
            <output
                    wsam:Action="http://gitb.com/MessagingService/sendResponse"
                    message="tns:sendResponse" />
        </operation>
        <operation name="beginTransaction">
            <input
                    wsam:Action="http://gitb.com/MessagingService/beginTransaction"
                    message="tns:beginTransaction" />
            <output
                    wsam:Action="http://gitb.com/MessagingService/beginTransactionResponse"
                    message="tns:beginTransactionResponse" />
        </operation>
        <operation name="endTransaction">
            <input
                    wsam:Action="http://gitb.com/MessagingService/endTransaction"
                    message="tns:endTransaction" />
            <output
                    wsam:Action="http://gitb.com/MessagingService/endTransactionResponse"
                    message="tns:endTransactionResponse" />
        </operation>
        <operation name="finalize">
            <input
                    wsam:Action="http://gitb.com/MessagingService/finalize"
                    message="tns:finalize" />
            <output
                    wsam:Action="http://gitb.com/MessagingService/finalizeResponse"
                    message="tns:finalizeResponse" />
        </operation>
    </portType>
    <binding name="MessagingServicePortBinding" type="tns:MessagingService">
        <wsaw:UsingAddressing required="true"/>
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                      style="document" />
        <operation name="getModuleDefinition">
            <soap:operation soapAction="" />
            <input>
                <soap:body use="literal" />
            </input>
            <output>
                <soap:body use="literal" />
            </output>
        </operation>
        <operation name="initiate">
            <soap:operation soapAction="" />
            <input>
                <soap:body use="literal" />
            </input>
            <output>
                <soap:body use="literal" />
            </output>
        </operation>
        <operation name="send">
            <soap:operation soapAction="" />
            <input>
                <soap:body use="literal" />
            </input>
            <output>
                <soap:body use="literal" />
            </output>
        </operation>
        <operation name="beginTransaction">
            <soap:operation soapAction="" />
            <input>
                <soap:body use="literal" />
            </input>
            <output>
                <soap:body use="literal" />
            </output>
        </operation>
        <operation name="endTransaction">
            <soap:operation soapAction="" />
            <input>
                <soap:body use="literal" />
            </input>
            <output>
                <soap:body use="literal" />
            </output>
        </operation>
        <operation name="finalize">
            <soap:operation soapAction="" />
            <input>
                <soap:body use="literal" />
            </input>
            <output>
                <soap:body use="literal" />
            </output>
        </operation>
    </binding>
    <service name="MessagingServiceService">
        <port name="MessagingServicePort" binding="tns:MessagingServicePortBinding">
            <soap:address location="REPLACE_WITH_ACTUAL_URL" />
        </port>
    </service>
```

```
    <!-- to generate sources in given package -->
    <jaxws:bindings xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
        <jaxws:package name="com.gitb.ms">
        </jaxws:package>
    </jaxws:bindings>
</definitions>
```

8.2.4 Web Service Description (WSDL) for Messaging Service Client (Service Consumer)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.gitb.com/ms/v1/"
     name="MessagingServiceConsumer"
     xmlns="http://schemas.xmlsoap.org/wsdl/"
     xmlns:tns="http://www.gitb.com/ms/v1/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
     xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl">
     <types>
          <xsd:schema>
               <xsd:import namespace="http://www.gitb.com/ms/v1/" schemaLocation="gitb_ms.xsd" />
          </xsd:schema>
     </types>
     <message name="notifyForMessage">
          <part name="parameters" element="tns:NotifyForMessageRequest" />
     </message>
     <message name="notifyForMessageResponse">
          <part name="parameters" element="tns:NotifyForMessageResponse" />
     </message>
     <portType name="MessagingClient">
          <operation name="notifyForMessage">
               <input
                    wsam:Action="http://gitb.com/MessagingClient/notifyForMessage"
                    message="tns:notifyForMessage" />
               <output
                    wsam:Action="http://gitb.com/MessagingClient/notifyForMessageResponse"
                    message="tns:notifyForMessageResponse" />
          </operation>
     </portType>
     <binding name="MessagingClientPortBinding" type="tns:MessagingClient">
          <wsaw:UsingAddressing required="true"/>
          <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
               style="document" />
          <operation name="notifyForMessage">
               <soap:operation soapAction="" />
               <input>
                    <soap:body use="literal" />
               </input>
               <output>
                    <soap:body use="literal" />
               </output>
          </operation>
     </binding>
     <service name="MessagingClientService">
          <port name="MessagingClientPort" binding="tns:MessagingClientPortBinding">
               <soap:address location="REPLACE_WITH_ACTUAL_URL" />
          </port>
     </service>
     <!-- to generate sources in given package -->
     <jaxws:bindings xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
          <jaxws:package name="com.gitb.ms">
          </jaxws:package>
     </jaxws:bindings>
</definitions>
```

8.2.5 XML Schema for Request/Response Messages

```
 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xsd:schema version="1.0" targetNamespace="http://www.gitb.com/ms/v1/"
            xmlns="http://www.gitb.com/ms/v1/"
            xmlns:tns="http://www.gitb.com/ms/v1/"
            xmlns:tr="http://www.gitb.com/tr/v1/"
            xmlns:gitb="http://www.gitb.com/core/v1/"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:import namespace="http://www.gitb.com/tr/v1/" schemaLocation="gitb_tr.xsd"/>
    <xsd:import namespace="http://www.gitb.com/core/v1/" schemaLocation="gitb_core.xsd"/>

    <xsd:element name="GetModuleDefinitionRequest" type="tns:Void" />
    <xsd:element name="GetModuleDefinitionResponse" type="tns:GetModuleDefinitionResponse" />
    <xsd:element name="InitiateRequest" type="tns:InitiateRequest" />
    <xsd:element name="InitiateResponse" type="tns:InitiateResponse" />
    <xsd:element name="SendRequest" type="tns:SendRequest" />
    <xsd:element name="SendResponse" type="tns:SendResponse" />
    <xsd:element name="BeginTransactionRequest" type="tns:BeginTransactionRequest" />
    <xsd:element name="BeginTransactionResponse" type="tns:Void" />
```

```
    <xsd:element name="NotifyForMessageRequest" type="tns:NotifyForMessageRequest" />
    <xsd:element name="NotifyForMessageResponse" type="tns:Void" />
    <xsd:element name="EndTransactionRequest" type="tns:BasicRequest" />
    <xsd:element name="EndTransactionResponse" type="tns:Void" />
    <xsd:element name="FinalizeRequest" type="tns:FinalizeRequest" />
    <xsd:element name="FinalizeResponse" type="tns:Void" />

    <xsd:complexType name="Void">
        <xsd:sequence/>
    </xsd:complexType>

    <xsd:complexType name="GetModuleDefinitionResponse">
        <xsd:sequence>
            <xsd:element name="module" type="gitb:MessagingModule" />
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="InitiateRequest">
        <xsd:sequence>
            <xsd:element       name="actorConfiguration"       type="gitb:ActorConfiguration"       minOccurs="1"
maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="InitiateResponse">
        <xsd:sequence>
            <xsd:element name="sessionId" type="xsd:string" />
            <xsd:element       name="actorConfiguration"       type="gitb:ActorConfiguration"       minOccurs="1"
maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="BasicRequest">
        <xsd:sequence>
            <xsd:element name="sessionId" type="xsd:string" />
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="BeginTransactionRequest">
        <xsd:complexContent>
            <xsd:extension base="BasicRequest">
                <xsd:sequence>
                    <xsd:element name="config" type="gitb:Configuration" minOccurs="0" maxOccurs="unbounded" />
                    <xsd:element name="from" type="xsd:string" minOccurs="0" />
                    <xsd:element name="to" type="xsd:string" minOccurs="0" />
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="SendRequest">
        <xsd:complexContent>
            <xsd:extension base="BasicRequest">
                <xsd:sequence>
                    <xsd:element name="to" type="xsd:string"/>
                    <xsd:element name="input" type="gitb:AnyContent" maxOccurs="unbounded" minOccurs="0"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="SendResponse">
        <xsd:sequence>
            <xsd:element name="report" type="tr:TAR" />
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="NotifyForMessageRequest">
        <xsd:complexContent>
            <xsd:extension base="BasicRequest">
                <xsd:sequence>
                    <xsd:element name="from" type="xsd:string" minOccurs="0" />
                    <xsd:element name="to" type="xsd:string" minOccurs="0" />
                    <xsd:element name="report" type="tr:TAR" />
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="FinalizeRequest">
        <xsd:sequence>
            <xsd:element name="sessionId" type="xsd:string" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

8.3 Test Bed Service

8.3.1 Service Overview

Figure 8-3 illustrates a scenario where a client system uses the remote testbed to execute a test scenario while it provides monitoring capability for its user (the SUT administrator). TestbedClient and TestbedService role represents the client and service sides respectively.

1. **tbs:getTestCaseDefinition:** The scenario starts with user selecting the test case to execute, but it is out of scope for the service specification. Then TestbedClient calls this operation to retrieve the test case definition which will be presented in the TPL format as described in Section 0). TestbedClient should render the description and present it to the user in some way.
2. **tbs:getActorDefinition:** In the test case definition, only the identifier and role of the actor in the test scenario is returned. TestbedClient needs to know the required configuration parameters for actors in order to get the configurations from the SUT administrators. This operation is used to retrieve these actor definitions.
3. **tbs:initiate:** TestbedClient have to use the initiate operation by suppliying the test case identifier to initiate the test execution. In response, the TestbedService should return an unique identifier for the testcase execution session.
4. **tbs:configure:** After this phase, TestbedService is expecting TestbedClient to send the configurations related with the SUT (or for all SUTs in case of interoperability tests). In response, TestbedService also compiles the configurations for the simulated actors and returns them. TestbedClient will show all these configurations to the user so that he can configure the SUT accordingly (ex: providing network parameters of the corresponding actor).
5. **tbs:initiatePreliminary:** After configuration phase, if test case description has some preliminary phase, TestbedClient should use the initiatePreliminary to start the preliminary phase. In response, TestbedService returns all instructions and input requests for the user. TestbedClient will show these instructions and requests to the user.
6. **tbs:provideInput:** When the user supply the requested information, TestbedClient use this operation to send these inputs to the TestbedService.
7. **tbs:start:** When the preliminary phase end by collecting all the inputs, user can start the testing phase at any time. When he does, TestbedClient use “start” command to initiate the execution. After processing the test steps defined in the test case definition, TestbedService calls the **tbs:updateStatus** callback to notify TestbedClient about the latest status of the execution. If an user interaction step exists within the flow, the **tbs:interactWithUsers** callback will be called to initiate the interaction. TestbedClient will use the **tbs:provideInput** operation to supply the inputs if the interaction includes input requests. After completion of these, execution continues from the next steps. When execution finished, the overall report for test case will be sent to the TestbedClient.
8. **tbs:stop:** This operations is not shown in the figure, but it can be used to stop the execution any time.
9. **tbs:restart:** This is also not shown in the figure. When the execution is stopped (either finished normally, stopped by user, or by exit step), this command can be used to restart the execution with the same configurations and preliminary requirements.

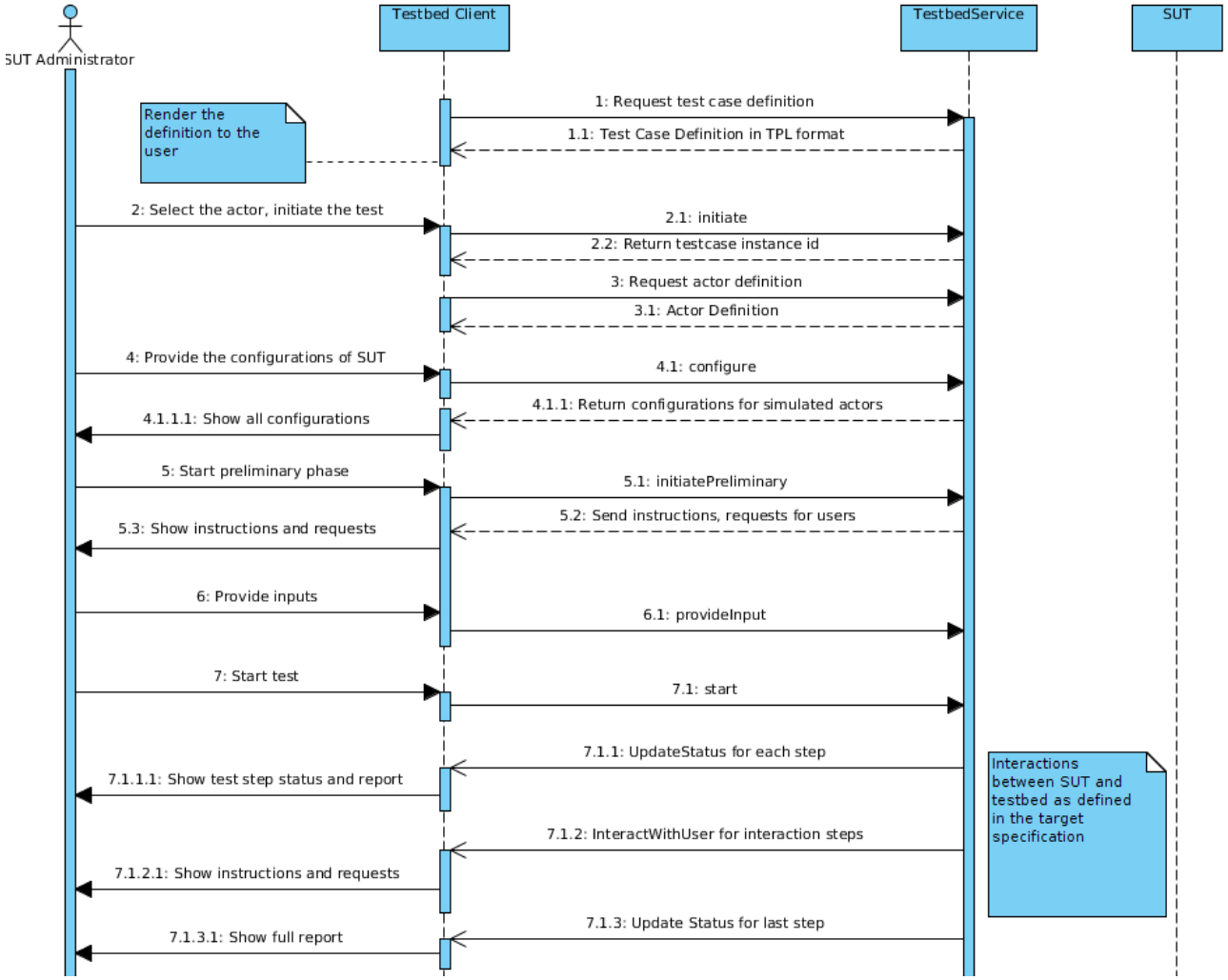

Figure 8-3: Testbed Service Scenario

8.3.2 Abstract Service Description

8.3.2.1 Requesting Test Case Definition (GetTestcaseDefinition)

This operation is used by TestbedClient to retrieve the test case definition in TPL format. The **<tbs:GetTestcaseDefinitionRequest>** is sent for the operation;

- **tcId** - The identifier for the test case

The **<tbs:GetTestcaseDefinitionResponse>** is returned in response;

- **testcase: <tpl:TestCase> –** Definition of test case

8.3.2.2 Initiating Test Process (Initiate)

This is operation is used to initiate the execution for a test scenario. TestbedService is expected to generate an unique identifier for the execution. The **<tbs:InitiateRequest>**is sent for the operation;

- **tcId -** The identifier for the test case. Used if the execution is not initiated yet.

The **<tbs:InitiateResponse>**is returned as a response;

- **tcInstanceId** – The identifier for the execution session

8.3.2.3 Requesting Actor Definition (GetActorDefinition)

This operation is used to get the full definition of actors together with the required configuration parameters for SUTs that wants to play the actor. The **<tbs:GetActorDefinitionRequest>**is sent for the operation;

- **tcId** – The identifier for the test case
- **actorId** – The identifier for the actor

The **<tbs:GetActorDefinitionResponse>**is returned as a response;

- **actor: <gitb:Actor>** – Definition of actor

8.3.2.4 Configure Test Execution (Configure)

This operation is used to supply the configurations of the SUTs that will participate to the testing process. As each SUT will play an actor defined in the test case definition, configurations will be mapped with the actor name. The **<tbs:ConfigureRequest>**element is used for request;

- **tcInstanceId –** The identifier for the execution session
- **actorConfiguration (1..*): <gitb:ActorConfiguration>** – Configurations for each SUT in the process

The **<tbs:ConfigureResponse>**is returned as a response;

- **actorConfiguration (1..*): <gitb:ActorConfiguration>** – Configurations for the simulated actors

8.3.2.5 Initiate Preliminary Phase (InitiatePreliminary)

This operation is used to initiate preliminary phase when TestbedClient is ready after configurations are done. TestbedService should execute the preliminary phase and return all the resulting instructions and input requests. The **<tbs:InitiatePreliminaryRequest>**is sent for the operation;

- **tcInstanceId** – The identifier for the execution session

The **<tbs:InitiatePreliminaryResponse>**is returned as a response;

- **preliminary**: **<tpl:Preliminary>** – Instructions and input requests for the SUT administrators

8.3.2.6 Providing User Input for Execution (ProvideInput)

This operation is used both in preliminary phase or execution phase to supply the requested inputs from the SUT administrators to the TestbedService. The response is just the acknowledgement of the operation. The **<tbs:ProvideInputRequest>**is sent as the request;

- **tcInstanceId -** The identifier for the execution session
- **input (1..*): <tbs:UserInput> -** Inputs supplied by the users. Extends the <gitb:AnyContent>
    - **stepId -** Associated step id for the request ()

8.3.2.7 Starting the Execution Phase (Start)

When preliminary phase is completed by providing all requested inputs, TestbedClient can start the execution at any time. TestbedService starts to execute the test steps as defined in the test case description and send status updates by calling the **tbs:updateStatus** callback. The reponse to this request is just an acknowledgement. The **<tbs:StartRequest>**is send for the operation;

- **tcInstanceId -** The identifier for the execution session

8.3.2.8 Status Updates for Testcase Execution (UpdateStatus callback)

This is the callback to notify TestbedClient about the execution of each test step defined in the definition. TestbedService should send a notification for each test step once when it starts to processing it and once when it completed processing. For the completed steps, it also provides the report for the test step. The **<tbs:UpdateStatusRequest>**is sent for the callback;

- **tcInstanceId -** The identifier for the execution session
- **stepId –** The identifier of the test step (the identifier of the corresponding step in the test case definition)
- **status** – Status of the processing for that step. Values can be;
  - "PROCESSING": Used when the testbed starts processing the step.
  - "SKIPPED": Used when a step is skipped (One branch of decision step).

we"WAITING": Used for messaging steps or interaction steps when some input is expected either from SUTs or SUT administrators. (Replace PROCESSING for such steps)

  - "COMPLETED": Used when processing is completed for the step.
- **report (0..1): <tr:TestStepReport> –** When a step is completed, this element is used to provide the report.

8.3.2.9 User Interaction During Execution (InteractWithUsers callback)

This is the callback to notify the TestbedClient that interaction is required with certain users (SUT administrators) at this step to show them some instructions or request some input from them. As in the InitiatePreliminaryResponse, TestbedService will supply all the instructions and input requests in this callback. TestbedClient should interact with the given users, collect the input and call the ProvideInput operation to supply the inputs to the TestbedService back. The **<tbs:InteractWithUsersRequest>** is used for the callback;

- **tcInstanceId** - The identifier for the execution session
- **interaction: <tpl:UserInteraction> –** Include instructions and input requests regarding the expected user interaction

8.3.2.10 Stopping the Execution (Stop)

This operation can be used at any time to stop the test execution. The **<tbs:StopRequest>** is used;

- **tcInstanceId -** The identifier for the execution session

8.3.2.11 Restarting the Execution Phase (Restart)

If the test execution is completed normally or stopped by some reason during the execution, this operation can be used the restart the execution phase. In this way, the configuration and preliminary phases do not have to be repeated. However, the TestbedService should initiate a new execution session. The **<tbs:RestartRequest>** is used;

- **tcInstanceId -** The identifier for the execution session

The **<tbs:RestartResponse>** is returned as response;

- **tcInstanceId -** The identifier for the new execution session

8.3.3 Web Service Description (WSDL) for Testbed Service (Service Provider)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.gitb.com/tbs/v1/"
             name="TestbedService"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:tns="http://www.gitb.com/tbs/v1/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
             xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl">
```

```
    <types>
        <xsd:schema>
            <xsd:import namespace="http://www.gitb.com/tbs/v1/" schemaLocation="gitb_tbs.xsd"/>
        </xsd:schema>
    </types>
    <message name="getTestCaseDefinitionRequest">
        <part name="parameters" element="tns:GetTestCaseDefinitionRequest"/>
    </message>
    <message name="getTestCaseDefinitionResponse">
        <part name="parameters" element="tns:GetTestCaseDefinitionResponse"/>
    </message>
    <message name="getActorDefinitionRequest">
        <part name="parameters" element="tns:GetActorDefinitionRequest"/>
    </message>
    <message name="getActorDefinitionResponse">
        <part name="parameters" element="tns:GetActorDefinitionResponse"/>
    </message>
    <message name="initiateRequest">
        <part name="parameters" element="tns:InitiateRequest"/>
    </message>
    <message name="initiateResponse">
        <part name="parameters" element="tns:InitiateResponse"/>
    </message>
    <message name="configureRequest">
        <part name="parameters" element="tns:ConfigureRequest"/>
    </message>
    <message name="configureResponse">
        <part name="parameters" element="tns:ConfigureResponse"/>
    </message>
    <message name="provideInputRequest">
        <part name="parameters" element="tns:ProvideInputRequest"/>
    </message>
    <message name="provideInputResponse">
        <part name="parameters" element="tns:Void"/>
    </message>
    <message name="initiatePreliminaryRequest">
        <part name="parameters" element="tns:InitiatePreliminaryRequest"/>
    </message>
    <message name="initiatePreliminaryResponse">
        <part name="parameters" element="tns:InitiatePreliminaryResponse"/>
    </message>
    <message name="startRequest">
        <part name="parameters" element="tns:StartRequest"/>
    </message>
    <message name="startResponse">
        <part name="parameters" element="tns:Void"/>
    </message>
    <message name="stopRequest">
        <part name="parameters" element="tns:StopRequest"/>
    </message>
    <message name="stopResponse">
        <part name="parameters" element="tns:Void"/>
    </message>
    <message name="restartRequest">
        <part name="parameters" element="tns:RestartRequest"/>
    </message>
    <message name="restartResponse">
        <part name="parameters" element="tns:Void"/>
    </message>
    <portType name="TestbedService">
        <operation name="getTestCaseDefinition">
            <input
                    wsam:Action="http://gitb.com/tbs/getTestCaseDefinition"
                    message="tns:getTestCaseDefinitionRequest"/>
            <output
                    wsam:Action="http://gitb.com/tbs/getTestCaseDefinitionResponse"
                    message="tns:getTestCaseDefinitionResponse"/>
        </operation>
        <operation name="getActorDefinition">
            <input
                    wsam:Action="http://gitb.com/tbs/getActorDefinition"
                    message="tns:getActorDefinitionRequest"/>
            <output
                    wsam:Action="http://gitb.com/tbs/getActorDefinitionResponse"
                    message="tns:getActorDefinitionResponse"/>
        </operation>
        <operation name="initiate">
            <input wsam:Action="http://gitb.com/tbs/initiate"
                   message="tns:initiateRequest"/>
            <output wsam:Action="http://gitb.com/tbs/initiateResponse"
                    message="tns:initiateResponse"/>
        </operation>
        <operation name="provideInput">
            <input wsam:Action="http://gitb.com/tbs/provideInput"
                   message="tns:provideInputRequest"/>
            <output wsam:Action="http://gitb.com/tbs/provideInputResponse"
                    message="tns:provideInputResponse"/>
        </operation>
        <operation name="configure">
            <input wsam:Action="http://gitb.com/tbs/configure"
                   message="tns:configureRequest"/>
            <output wsam:Action="http://gitb.com/tbs/configureResponse"
                    message="tns:configureResponse"/>
        </operation>
        <operation name="initiatePreliminary">
            <input wsam:Action="http://gitb.com/tbs/initiatePreliminary"
                   message="tns:initiatePreliminaryRequest"/>
            <output wsam:Action="http://gitb.com/tbs/initiatePreliminaryResponse"
                    message="tns:initiatePreliminaryResponse"/>
```

```
        </operation>
        <operation name="start">
            <input wsam:Action="http://gitb.com/tbs/start"
                   message="tns:startRequest"/>
            <output wsam:Action="http://gitb.com/tbs/startResponse"
                    message="tns:startResponse"/>
        </operation>
        <operation name="stop">
            <input wsam:Action="http://gitb.com/tbs/stop"
                   message="tns:stopRequest"/>
            <output wsam:Action="http://gitb.com/tbs/stopResponse"
                    message="tns:stopResponse"/>
        </operation>
        <operation name="restart">
            <input wsam:Action="http://gitb.com/tbs/restart"
                   message="tns:restartRequest"/>
            <output wsam:Action="http://gitb.com/tbs/restartResponse"
                    message="tns:restartResponse"/>
        </operation>
    </portType>
    <binding name="TestbedServicePortBinding" type="tns:TestbedService">
        <wsaw:UsingAddressing required="true"/>
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                      style="document"/>
        <operation name="getTestCaseDefinition">
            <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
        <operation name="getActorDefinition">
            <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
        <operation name="initiate">
            <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
        <operation name="configure">
            <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
        <operation name="provideInput">
            <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
        <operation name="initiatePreliminary">
            <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
        <operation name="start">
            <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
        <operation name="stop">
            <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
        <operation name="restart">
            <soap:operation soapAction=""/>
            <input>
```

```
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>
    <service name="TestbedServiceService">
        <port name="TestbedServicePort" binding="tns:TestbedServicePortBinding">
            <soap:address location="/service/TestbedService"/>
        </port>
    </service>
    <!-- to generate sources in given package -->
    <jaxws:bindings xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
        <jaxws:package name="com.gitb.tbs">
        </jaxws:package>
    </jaxws:bindings>
8.3.3.1.1.1.1.1    </definitions>
```

8.3.4 Web Service Description (WSDL) for Testbed Service Client (Service Consumer)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.gitb.com/tbs/v1/"
     name="TestbedServiceConsumer"
     xmlns="http://schemas.xmlsoap.org/wsdl/"
     xmlns:tns="http://www.gitb.com/tbs/v1/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
     xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl">
     <types>
          <xsd:schema>
               <xsd:import namespace="http://www.gitb.com/tbs/v1/" schemaLocation="gitb_tbs.xsd"/>
          </xsd:schema>
     </types>
     <!--request/responses for Callbacks-->
     <message name="updateStatusRequest">
          <part name="parameters" element="tns:UpdateStatusRequest"/>
     </message>
     <message name="updateStatusResponse">
          <part name="parameters" element="tns:Void"/>
     </message>
     <message name="interactWithUsersRequest">
          <part name="parameters" element="tns:InteractWithUsersRequest"/>
     </message>
     <message name="interactWithUsersResponse">
          <part name="parameters" element="tns:Void"/>
     </message>
     <portType name="TestbedClient">
          <operation name="updateStatus">
               <input
                    wsam:Action="http://gitb.com/tbs/updateStatus"
                    message="tns:updateStatusRequest"/>
               <output
                    wsam:Action="http://gitb.com/tbs/updateStatusResponse"
                    message="tns:updateStatusResponse"/>
          </operation>
          <operation name="interactWithUsers">
               <input
                    wsam:Action="http://gitb.com/tbs/interactWithUsers"
                    message="tns:interactWithUsersRequest"/>
               <output
                    wsam:Action="http://gitb.com/tbs/interactWithUsersResponse"
                    message="tns:interactWithUsersResponse"/>
          </operation>
     </portType>
     <binding name="TestbedClientPortBinding" type="tns:TestbedClient">
          <wsaw:UsingAddressing required="true"/>
          <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
               style="document"/>
          <operation name="updateStatus">
               <soap:operation soapAction=""/>
               <input>
                    <soap:body use="literal"/>
               </input>
               <output>
                    <soap:body use="literal"/>
               </output>
          </operation>
          <operation name="interactWithUsers">
               <soap:operation soapAction=""/>
               <input>
                    <soap:body use="literal"/>
               </input>
               <output>
                    <soap:body use="literal"/>
               </output>
          </operation>
     </binding>
     <service name="TestbedClientService">
          <port name="TestbedClientPort" binding="tns:TestbedClientPortBinding">
               <soap:address location="/service/TestbedClient"/>
          </port>
     </service>
     <!-- to generate sources in given package -->
```

```
     <jaxws:bindings xmlns:jaxws="http://java.sun.com/xml/ns/jaxws">
          <jaxws:package name="com.gitb.tbs">
          </jaxws:package>
     </jaxws:bindings>
</definitions>
```

8.3.5 XML Schema for Request/Response Messages

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xsd:schema version="1.0" targetNamespace="http://www.gitb.com/tbs/v1/"
            xmlns="http://www.gitb.com/tbs/v1/"
            xmlns:tns="http://www.gitb.com/tbs/v1/"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:tpl="http://www.gitb.com/tpl/v1/"
            xmlns:tr="http://www.gitb.com/tr/v1/"
            xmlns:gitb="http://www.gitb.com/core/v1/"
            elementFormDefault="qualified">

    <!-- Go up to top module folder first, than go down to schema folder through "target" -->
    <xsd:import namespace="http://www.gitb.com/tpl/v1/" schemaLocation="gitb_tpl.xsd"/>
    <xsd:import namespace="http://www.gitb.com/core/v1/" schemaLocation="gitb_core.xsd"/>
    <xsd:import namespace="http://www.gitb.com/tr/v1/" schemaLocation="gitb_tr.xsd"/>
    <xsd:import namespace="http://www.gitb.com/tdl/v1/" schemaLocation="gitb_tdl.xsd"/>

    <!-- TestbedService request/responses-->
    <xsd:element name="GetTestCaseDefinitionRequest" type="tns:BasicRequest"/>
    <xsd:element name="GetTestCaseDefinitionResponse" type="tns:GetTestCaseDefinitionResponse"/>
    <xsd:element name="GetActorDefinitionRequest" type="tns:GetActorDefinitionRequest"/>
    <xsd:element name="GetActorDefinitionResponse" type="tns:GetActorDefinitionResponse"/>
    <xsd:element name="InitiateRequest" type="tns:BasicRequest"/>
    <xsd:element name="InitiateResponse" type="tns:InitiateResponse"/>
    <xsd:element name="ConfigureRequest" type="tns:ConfigureRequest"/>
    <xsd:element name="ConfigureResponse" type="tns:ConfigureResponse"/>
    <xsd:element name="ProvideInputRequest" type="tns:ProvideInputRequest"/>
    <xsd:element name="InitiatePreliminaryRequest" type="tns:BasicCommand"/>
    <xsd:element name="InitiatePreliminaryResponse" type="tns:UserInteractionRequest"/>
    <xsd:element name="StartRequest" type="tns:BasicCommand"/>
    <xsd:element name="StopRequest" type="tns:BasicCommand"/>
    <xsd:element name="RestartRequest" type="tns:BasicCommand"/>
    <!--Callback request/responses-->
    <xsd:element name="UpdateStatusRequest" type="tns:TestStepStatus"/>
    <xsd:element name="InteractWithUsersRequest" type="tns:InteractWithUsersRequest"/>

    <xsd:element name="Void" type="tns:Void"/>

    <xsd:complexType name="BasicRequest">
        <xsd:sequence>
            <xsd:element name="tcId" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="BasicCommand">
        <xsd:sequence>
            <xsd:element name="tcInstanceId" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="GetTestCaseDefinitionResponse">
        <xsd:sequence>
            <xsd:element name="testcase" type="tpl:TestCase"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="GetActorDefinitionRequest">
        <xsd:complexContent>
        <xsd:extension base="BasicRequest">
            <xsd:sequence>
                <xsd:element name="actorId" type="xsd:string"/>
            </xsd:sequence>
        </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="GetActorDefinitionResponse">
        <xsd:sequence>
            <xsd:element name="actor" type="gitb:Actor"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="InitiateResponse">
        <xsd:sequence>
            <xsd:element name="tcInstanceId" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="ConfigureRequest">
        <xsd:complexContent>
            <xsd:extension base="BasicCommand">
                <xsd:sequence>
                    <xsd:element name="configs" type="gitb:ActorConfiguration" maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
```

```
    <xsd:complexType name="ConfigureResponse">
        <xsd:sequence>
            <xsd:element name="configs" type="gitb:ActorConfiguration" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="ProvideInputRequest">
        <xsd:complexContent>
            <xsd:extension base="BasicCommand">
                <xsd:sequence>
                    <xsd:element name="stepId" type="xsd:string"/>
                    <xsd:element name="input" type="UserInput" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="UserInput">
        <xsd:complexContent>
            <xsd:extension base="gitb:AnyContent">
                <xsd:attribute name="id" type="xsd:string"/>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="UserInteractionRequest">
        <xsd:sequence>
            <xsd:choice maxOccurs="unbounded">
                <xsd:element name="instruction" type="Instruction"/>
                <xsd:element name="request" type="InputRequest"/>
            </xsd:choice>
        </xsd:sequence>
        <xsd:attribute name="with" type="xsd:string" use="optional"/>
    </xsd:complexType>

    <xsd:complexType name="Instruction">
        <xsd:complexContent>
            <xsd:extension base="gitb:AnyContent">
                <xsd:attribute name="id" type="xsd:string"/>
                <xsd:attribute name="desc" type="xsd:string"/>
                <xsd:attribute name="with" type="xsd:string"/>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="InputRequest">
        <xsd:attribute name="id" type="xsd:string"/>
        <xsd:attribute name="desc" type="xsd:string"/>
        <xsd:attribute name="with" type="xsd:string"/>
        <xsd:attribute name="type" type="xsd:string" use="optional"/>
        <xsd:attribute name="encoding" type="xsd:string" use="optional"/>
    </xsd:complexType>

    <xsd:complexType name="TestStepStatus">
        <xsd:sequence>
            <xsd:element name="tcInstanceId" type="xsd:string"/>
            <xsd:element name="stepId" type="xsd:string"/>
            <xsd:element name="status" type="gitb:StepStatus"/>
            <xsd:element name="report" type="tr:TestStepReportType" minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="InteractWithUsersRequest">
        <xsd:sequence>
            <xsd:element name="tcInstanceid" type="xsd:string"/>
            <xsd:element name="interaction" type="UserInteractionRequest"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="Void">
        <xsd:sequence/>
    </xsd:complexType>
</xsd:schema>
```

9 GITB Test Description Language (TDL)

An important part of the GITB Architecture is the **GITB Test Description Language (TDL)** that defines the high level executable scripting language for the Test Bed. Using the GITB TDL facilitates reusing of testing capabilities among different stakeholders and domains.

9.1 GITB Test Bed Concepts and Interfaces

Before presenting the GITB Test Description Language (TDL), we need to describe the main concepts and assumptions that it is based on to setup a global interoperability testbed.

9.1.1 Basic Concepts

The TDL defines the model and the format to describe a conformance or interoperability test scenario in a way that, when executed, the Test Bed realizes the business process as defined in the target eBusiness specification between the SUTs and the simulated actors and performs the intended testing procedures. The definition of such a scenario with TDL is called **Test Case** definition. In order to check conformance or interoperability of a system for a target specification, in general, multiple test scenarios may be needed to test different aspects of the system in alternative scenarios. Therefore, a logical grouping among the Test Case definitions is required. The concept of **Test Suite** represents such grouping for a specific objective of the test designer. Furthermore, a Test Suite includes some common definitions for all included Test Cases. With these definitions, the testing process can be defined as a business process between the testbed, SUTs, and SUT administrators which is managed by the testbed itself by getting the Test Case definition and its attached Test Suite definition as input.

The GITB testing process and model is based on the **actor concept**. All eBusiness specifications define some type of actor (party) in their business choreography. These abstract definitions represent systems and implementations in the real world and software implementers use these concepts to claim conformance to the target specification. The actors that will participate in the testing process are defined in a Test Case definition. Furthermore, their **roles** in the testing process are specified. The actors that are tested should be indicated as SUT and actors that are simulated by the testbed should be indicated as simulated. Systems claiming conformance to the corresponding actor, indicated as SUT in the test case definition, can initiate (or join for interoperability test scenarios) the testing process by playing the role.

9.1.2 Type System and Expressions

Test Beds deal with messages, documents and intermediate results computed from them during test execution. A requirement for all Test Beds is to have mechanisms to temporarily store the intermediate results, pass them to other modules as input, navigate on the content to reach more granular values or parts, and compute further results from those. Like any programming or scripting language, these are handled by defining a type system and expression language working on that type system.

Although many of the current eBusiness specifications use XML as the format for the content, there are some domains or specifications that have different data models and message or document formats (e.g. DICOM for digital images in medical domain, JSON for many lightweight specifications and EDI). In order to provide a generic testing platform and support all of them, we need an extendible type system for GITB. Furthermore, it should not be complex to ease the test definition process.

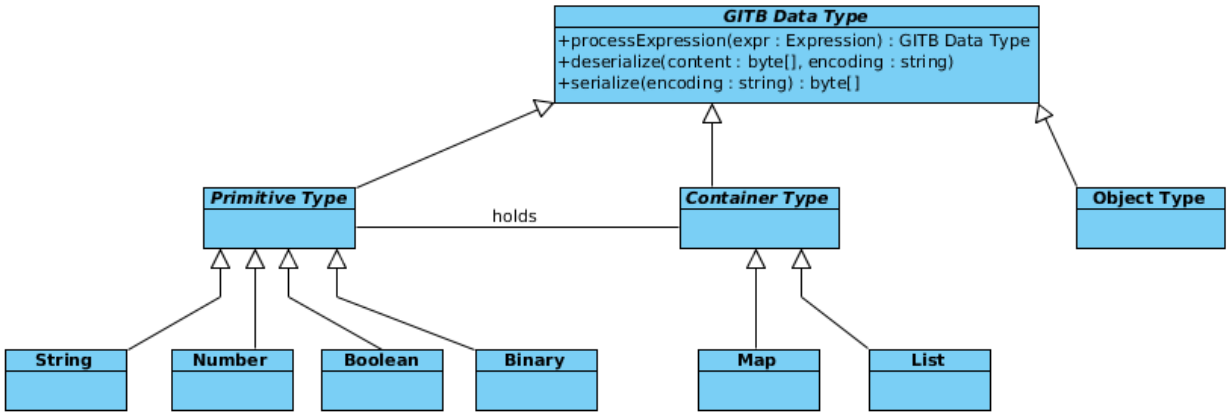

Figure 9-1: GITB Type System

Figure 9-19-1 illustrates the GITB type system. The **GITB Data Type** represents the root abstract type where all other types will be inherited. The **Primitive Types** represent simple values. The two **Container Types** are used to hold multiple values. The **List Type** is used to represent a list of values in a specific GITB Data Type (homogen internally). The **Map Type** is used to hold (key, value) pairs where value is any GITB Data Type (not homogen within the map). The **Object Type** is the root type for complex structures. It is the extension point for registering new types to the Test Bed for supporting different requirements within a specific domain or specific to a standard. In GITB, the type system concept is rather abstract. For example, we can register a type to represent EDI content (or XML, DICOM, etc.) in order to handle the related operations in our test scenarios. A pluggable type handling mechanism is assumed for the GITB Test Bed where a new type handler is plugged-in for a new registered type. In other words, a complex type is viewed as a black box system (type handler) implementing the following operations defined in the abstract GITB Data Type class.

- **deserialize –** A type handler should have the capability to construct the instance of the target type from a byte stream. Abstract type may have different serialization formats. For example, originally DICOM has a non-XML binary data format used in the actual processes. However, a special XML serialization and string serialization are defined by some tools for testing and monitoring purposes as the original format is not human readable. In order to benefit from these formats in our testing scenarios (for example, to supply a message template to a messaging step for sending messages to SUTs), our type registered for DICOM can support these serialization formats. The parameter **encoding** represents the format that the byte stream is encoded (in our example XML, original DICOM encoding, or special string serialization). This operation is used to construct the instance from a message received from SUT, or from a file (sample message to send).
- **serialize –** Similar to deserialization, a type handler should have the capability to serialize an instance of the type to a byte stream in one of the supported serialization formats. The parameter **encoding** indicates the format for serialization.
- **processExpression –** In addition to the type system, an expression language is required to navigate on the content to reach granular content parts, elements, attributes, etc. As XML is the most frequently used format for eBussiness specifications, an XPath 2.0 based expression language is selected as default for GITB. Furthermore, many non-XML content specifications have already XML serializations (DOM representations) and support XPath for their models. A type handler should implement a mechanism to evaluate the given XPath expression on its data model. This mechanism can just be applying XPath to the special XML serialization of the content.

9.1.3 Modularity for Specific Functionalities

As described in the GITB Testing Framework (CWA 16408:2012), the main testing functionalities like messaging and validation can be handled by pluggable modules within a modular architecture. GITB Messaging Service and GITB Validation Service are services that enables this modularity remotely between different testing facilities. In order to handle this modularity within a single Test Bed architecture, the GITB POC Test Bed is also designed to be modular in term of its messaging and validation capabilities. The modules that will handle the communication with SUTs within the business process are called **Messaging Adapters** (for example, an adapter to handle AS4 messaging)**.** On the other side, the modules that will

perform specific validation procedures are called **Validation Adapters** (for example, adapter for schematron validations). These modules can be internal pluggable modules only available within the Test Bed itself or implemented as GITB Validation Service or GITB Messaging Service to open the functionality to outside world as a reusable remote service.

While plugging these modules to the Test Bed, each module should provide a definition describing its configuration, input and output parameters. The **<gitb:TestModule>** abstract class provides the details for this definition:

- **id** – A unique identifier for the module itself within the Test Bed (Test Case definitions will use this identifier to refer the module).
- **uri –** The path (address) used to access to the module. It will be a URL if the module is a service.
- **metadata: <gitb:Metadata> –** Metadata regarding the module (name, description, authors, version, etc).
- **inputs (0..*): <gitb:TypedParameter> –** Describes the input parameters for the module.
- **outputs (0..*): <gitb:TypedParameter> –** Describes the outputs of the module (not used for validation adapters).
- **isRemote –** Indicates if this module is a remote service, such as a GITB compliant Validation or Messaging Service.

<gitb:ValidationModule> extends <gitb:TestModule> for validation adapters with the following elements:

- **configs (0..*): <gitb:ConfigurationParameters> –** Configuration parameters for the module to change the behavior in the procedure (validation process or messaging process)

<gitb:MessagingModule> extends <gitb:TestModule> for messaging adapters with the following elements:

- **actorConfigs: <gitb:ConfigurationParameters> –** Generic configuration parameters for the systems that will communicate.
- **transactionConfigs: <gitb:ConfigurationParameters> –** Configuration parameters specific to a transaction.
- **listenConfigs: <gitb:ConfigurationParameters> –** Configuration parameters specific to listen operations.
- **receiveConfigs: <gitb:ConfigurationParameters> –** Configuration parameters specific to receive operations.
- **sendConfigs: <gitb:ConfigurationParameters> –** Configuration parameters specific to send operations.

Test designers while writing the related part of the test scenario can use these definitions like an API to supply the required configuration and input parameters to the module or bind the outputs to internal variables within TDL. Furthermore, the Test Bed itself will use these definitions to behave accordingly while communicating with these modules. Figure 9-9-22 illustrates the abstract model of the GITB TDL.

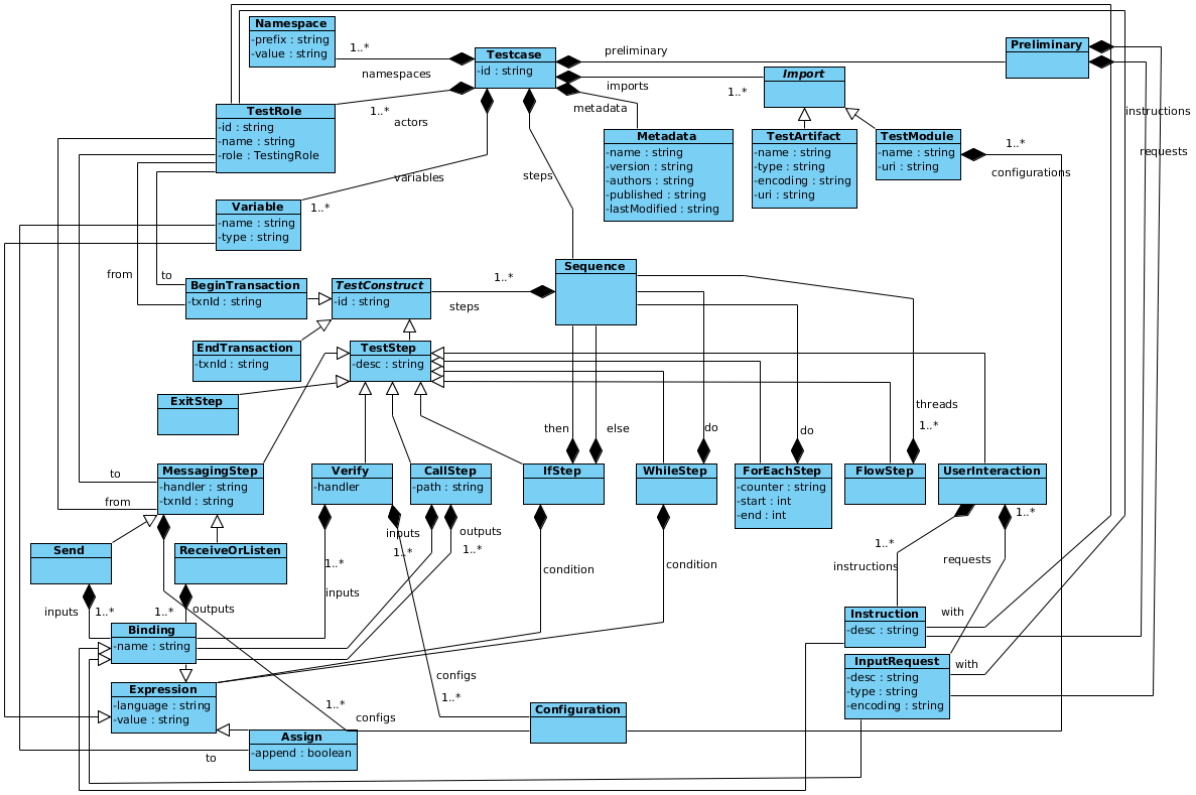

Figure 9-2: GITB Test Description Language Model

9.2 Test Suite Definition

The **<tdl:TestSuite>** element represents a package (logical grouping) of executable test scenarios to check adherence of implementations to one or more normative statements in a specification. The methodology to form these logical groups is not within the scope of TDL. Test designers may choose different strategies in this respect. For example, a <tdl:TestSuite> may represent the package of all conformance test scenarios to check conformance against a whole specification (for example, a conformance test suite for IHE XDS Profile, conformance test suite for PEPPOL Busdox). Or, it may represent a package of conformance test scenarios for a specific actor/party defined in a specification (for example, a conformance Test Suite for IHE XDS Document Registry, or a conformance Test Suite for CENBII Tender Notification Customer Role). Or it may correspond to a more granular set (such as a specific business scenario, etc).

The **<tdl:TestSuite>** definition provides some basic information required for the execution of child Test Case definitions:

- **metadata: <gitb:Metadata> –** Metadata related to the Test Suite definition (name, description, author, creation time, etc).
- **actor (1..*) :<gitb:Actor> –** Definition of actors that takes part in the business processes in the target specification which are related to the test scenarios in this Test Suite.
- **testcases (1..*): <tdl:TestcaseEntry>** – List of Test Cases in this Test Suite.

The **<tdl:TestcaseEntry>** indicates the identifier of the Test Case and its prequisite Test Cases:

- **id** – The identifier for the test case
- **prequisite (0..*)** - Identifiers of prequisite test cases for this test case

9.3 Test Case Definition

The **<tdl:Testcase>** element represents an executable conformance or interoperability test scenario that evaluates the adherence of implementations to one or more normative statements in a specification. The following attributes make up the Test Case definition:

- **id** – Defines the unique identifier for the Test Case. It is recommended to use a URN for the value of this attribute. (ex: urn:gitb:ihe:xds-document-source-conformace-test, urn:gitb:peppol:lime-protocol-conformance-test)
- **metadata: <gitb:Metadata> –** Describes the metadata attributes (name, description, author, version, etc) of the test case.
- **namespace (0..*): <tdl:Namespace> –** The list of namespace declarations and their prefix bindings that will be used in the expressions used in the test case.
- **import (0..*): <tdl:Import> –** The list of import statements to declare the external test modules or test artifacts required for the execution.
- **actor (1..*) : <gitb:TestRole>** – Describes the actors in the business process defined by the target specification of the test scenario and the role assignments regarding the testing process. (ex: Supplier in PEPPOL profiles, Document Consumer in IHE profiles)
- **variable (0..*): <tdl:Variable> –** The global variable definitions for the Test Case execution. Variables are used to temporarily store message/document parts or specific values during the execution.
- **preliminary (0..1): <tdl:UserInteraction> –** Container for describing preliminary requirements of the Test Case that should be shown to the SUT administrators before starting the test execution.
- **steps: <tdl:Sequence> –** The root container for the definition of test steps and their flows.
- **scriplet (0..*):<tld:Scriplet> –** A subsequence of test steps, or in other words a sub test flow, which can be used within the test case definition more than once. Similar to the concept of function definition in a programming language.

9.3.1 Namespace Declarations

The GITB TDL has an abstract expression language concept for processing/selecting elements from message/document contents, or compute further values from such content. Although, a default expression format (XPath based) will be proposed in this document for the POC Testbed implementation, any expression language designed for these purposes can be used as an TDL expression. Namespaces are important in expressions while refering the element or attribute names in a document or message model. This **<tld:Namespace>** element is used to declare the namespace and the prefix bounded to the namespace within the execution scope. These prefixes can then be used in expressions to refer the elements defined in the corresponding namespace.

- **prefix –** The prefix binding for the namespace.
- **value** – The string representing the namespace.

9.3.2 Importing External Test Modules and Artifacts

The GITB architecture allows a Test Bed to use remote testing facilities and existing Test Artifacts (for example, schematrons, schemas, sample messages) within the test execution. The import statements provide the details for the test engine. They describe how to import those modules or artifacts so that the test engine can remotely access them and use them during the test execution.

The **<tdl:TestModule>** element is used to import external test modules. The following are the details of the element:

- **name** – The name of the imported module. This name will be used to refer this module within the test case definition.
- **uri –** The URI to access to the module.
- **config (0..*): <gitb:Configuration>** – Configuration parameters for the module. The test engine should configure the module with the supplied parameters before the execution.

The **<tdl:TestArtifact>** element is used to import external test artifacts required for test execution. The following are the details of the element:

- **name** – A name assigned to the artifact. This name will be used to refer this artifact within the test case definition.
- **uri** – The URI to access to the artifact.
- **type –** Indicates the type of the content. Should refer one of the default GITB types or plugged-in types for the testbed. (ex: gitb-types:DOM for schematron or XML schema artifacts)
- **encoding (0..1) –** Indicates the serialization format of the artifact content (ex: XML for schematron or XML schema artifacts). If not supplied the default format for the given type is assumed to be used for the artifact.

9.3.3 Defining the Actors and Roles in the Test Case

As mentioned earlier, the GITB testing process and model is based on the actor concept, which is very common for eBusiness specifications. A Test Case definition should declare the actors who are participating in the target testing scenario (The details of the actors are defined within the Test Suite definition). Furthermore, based on the objective of the test scenario, the role of the actor in terms of testing should also be declared. For a conformance test scenario, the actor for which the conformity will be checked should take the System-Under-Test (SUT) role. The other actors will be simulated by the test engine. For an interoperability test scenario, the actors that will be tested should be indicated as SUT. The **<gitb:TestRole>** element is used to define an actor along with the following details:

- **id** – The unique identifier of the actor definition within the GITB Test Bed.
- **name** – A short name assigned to the actor. This name is used in the Test Case definition to refer this actor in the related constructs.
- **role –** The role of the actor for testing process. The value should be from TestRoleEnumeration;
  - SUT – Indicates that this actor will be tested in this test scenario. It means that the systems that want to be tested can only participate to the test with this actor.
  - SIMULATED **–** Indicates that this actor will be simulated by the Test Bed itself in the test scenario.

9.3.4 Defining the Variables

Like any programming language, test scripting languages need variables to store intermediate results (message/document parts, computed values, etc) during test execution. The **<tdl:Variable>** element defines a variable with its type and initial value if supplied. In GITB TDL, variable declarations are either done in a Test Case or in a Scriptlet and the scope of the variable is the enclosed container. The variables defined in the Test Case are global variables for test execution. Variables in Scriptlets are local variables for that scriptlet.

- **name –** Name of the variable. Should be unique within the scope (Test Case or Scriptlet). The expressions refer this name to access the value (see expression handling).
- **type –** Indicates the type of the variable (see GITB TDL type system).
- **value (0..*): <tdl:Binding> –** Provides the initial value assigned to the variable before the execution of test steps. The <tdl:Binding> element is a named expression and the evaluated value of this expression is assigned for the value. For composite types (map and list), more than one “value” element can be supplied. For “list” type, a list will be composed from the evaluated value of all supplied “value” elements. For “map” type, each “value”element will provide the key (the name attribute of <tdl:Binding> element), value (evaluated expression value) pair.

9.3.5 Preliminary Phase for the Execution

The **<tdl:UserInteraction>** construct is the container for the steps in the preliminary phase of the test scenario, but also used for user interactions during execution. This is the phase where SUT administrators are notified with preliminary requirements of the test scenario. These preliminary requirements can be instructions for SUT admins to do something on their implementations related to the scenario before the execution begins. This instruction will probably be related with a message/document exchanged between the SUT and test engine (or another SUT). For example, it may request from SUT admin to create a user profile in the system with the given values (id, name, address, etc) and check if the message part related with the

created user profile complies with the given requirements. This type of preliminary interactions are represented by the **<tdl:InstructionOrRequest>** element and its attributes are as follows;

- **desc –** The textual instruction to be shown to the SUT administrator.
- **with (0..1) –** Refers the actor (name attribute of TestRole element) that this instruction will be shown. If not supplied, it is assumed that this instruction is shown to all SUT actors defined in the test.
- **type (0..1) –** If a value (computed at run time) is supplied together with the instruction, the type of the value should be specified in this attribute. An example instruction can be "Please use the following value for the User identifier for the scenario".
- **encoding (0..1) –** If a value is supplied together with the textual instruction, this attribute indicates the representation format of the value for the given type.
- **expr (0..1) –** <tdl:InstructionOrRequest> extends <tdl:Expression> and the evaluated value of expression will be supplied as the value to be shown in the instruction.

Sometimes rather than enforcing requirements (specific values in the scenario), it can be more convenient to give the freedom to the SUT administrators to set certain values in a testing scenario. For the same example, the instruction can be changed to "Please create a user profile in your system and copy the user identifier that your system assigns to the user to the following space". Such instructions are also represented with **<tdl:InstructionOrRequest>** element. The semantics of the attributes are as follows:

- **desc –** The textual instruction to be shown to the SUT administrator.
- **with –** Refers the actor (name attribute of TestRole element) that this instruction will be shown and input will be requested.
- **expr (0..1) –** <tdl:InstructionOrRequest> extends <tdl:Expression> and if an expression is supplied, it means that input taken from the user will be assigned to the mentioned variable in the expression. In other words, supplied expression should be a variable expression (left value). If this element exists, type and name attributes are not necessary and test engine should not process them. The type of the expected input is deducted from the type of the variable.
- **type(0..1)** – If the requested input is not assigned to a variable by using the **expr,** this attribute should be used to indicate the type of the requested input according to type system of the testbed.
- **name (0..1) –** If the requested input is not assigned to a variable by using the **expr,** this attribute will specify the name of the input unique within the container (Preliminary or UserInteraction). Then this name will be used to access the value within the later expressions.
- **encoding (0..1) –** Indicates the serialization format of the requested input. If not supplied the default format of the given type is assumed.

9.3.6 Test Steps and Commands

The **<tdl:Sequence>** is used to represent a sequence of test commands for the test engine to execute sequentially. The root **<tdl:Sequence>** element in the **<tdl:Testcase>** definition is the entry point for the main execution phase. A Sequence may include the following constructs;

- btxn: **<tdl:BeginTransaction>**
- etxn: **<tdl:EndTransaction>**
- send: **<tdl:Send>**
- receive: **<tdl:Receive>**
- listen: **<tdl:Listen>**
- if: **<tdl:IfStep>**
- while: **<tdl:WhileStep>**
- forEach: **<tdl:ForEachStep>**
- flow: **<tdl:FlowStep>**
- exit: **<tdl:ExitStep>**
- assign: **<tdl:Assign>**
- group: **<tdl:Group>**

- verify: **<tdl:Verify>**
- call: **<tdl:CallStep>**
- interact: **<tdl:UserInteraction>**

Some of these constructs (btxn, etxn, assign, call) are supplementary constructs and are not designated as test steps. Others are actual test steps which are presented to the users and should extend **<tdl:TestStep>** class. The common attributes for test steps are;

- **desc** – The textual description of the test step. It should be written as an instruction to the SUT administrators when some action is expected.

9.3.7 Messaging Steps

Handling communication among SUTs and the simulated actors (i.e. the Test Bed itself) based on the target protocol (according to the rules and requirements stated by the target specification) is one of the major part for automated test processing. Communication can be between a SUT and a simulated actor, or between two SUTs and the required TDL commands and mechanisms are required to handle and drive the communication. The TDL has three messaging commands to represent these operations:

- **send –** This is used when the Test Bed (over a simulated actor) needs to send a message to a SUT based on the target specification for that step.
- **receive –** This is used when a SUT is expected to send a message to the Test Bed (over a simulated actor) for that step based on the target specification.
- **listen –** This is used when a SUT is expected to send a message to another SUT and the Test Bed is expected to listen (like a proxy) to this message.

Each of these test steps represents only one side of the communication between actors. The communication protocols (ex: SOAP Web Services, RESTFul Services, AS2, AS3, AS4, ebMS, etc) generally are based on request-response scheme at the application layer. Therefore, in order to handle the full communication two complimentary messaging steps should be used. For example, assume that we are testing a Web service client and out Test Bed is simulating the Web service. For this communication, we need a **receive** command as a first step to get the Web service request. Then after doing some validations and processing, we can use the **send** command to send the response message. However, some domain specific protocols (ex: DICOM communication protocol in eHealth) define more complex messaging schemes (multiple requests, responses in specific orders) at the application layer. In order to support all of these protocols, TDL defines the **Transaction concept** which is used to relate the messaging steps that simulate a complete communication between two actors at the application layer. The **<tdl:BeginTransaction>** command is used to notify the test engine that a Transaction will start with the given in in the next messaging steps. The details of the element are as follows:

- **txnId** – An identifier assigned to the transaction. This identifier is used to relate the messaging steps with this transaction. Therefore, it should be unique among other transactions in the test case definition.
- **from** – The actor that will participate in the communication related to this transaction. The **name** attribute of the actor stated in the corresponding TestRole element should be used for the value for referral. As notation, the actor that will start the communication should be refered from attribute. Generally, specifications define single endpoints for their actor definitions. However, some may define more than one endpoint for an actor supporting different protocols. If an actor definition has more than one endpoint definition, then the value for this attribute should be in the following format “<actor-name>.<endpoint-name>”.
- **to** – The actor that will participate in the communication related with this transaction.

The **<tdl:EndTransaction>** class is used to notify test engine that a transaction is finalized and no following messaging steps will refer this transaction any more.

- **txnId –** The id of the transaction.

A common base class, **<tdl:MessagingStep>** is designed for the three messaging steps.

- **txnId** – The id of the transaction that this messaging step belongs to.

- **handler –** The unique identifier for the handler (messaging module) within the Test Bed that will handle the communication stated with this messaging step.
- **from** – The actor that will send the message. The **name** attribute of the actor stated in the corresponding TestRole element should be used for the value for referral. If an actor definition has more than one endpoint definition, then the value for this attribute should be in the following format "<actor-name>.<endpoint-name>".
- **to –** Refers the actor that will receive the message (see **from** - Same rules apply).
- **config (0..*): <gitb:Configuration> –** List of configuration parameters to configure the messaging module for the communication.

In a GITB Test Bed, every registered messaging module has a definition file that defines its configuration, as well as the input and output parameters required for the operation. Supply of the configuration parameters and inputs and binding of the output parameters in the TDL are perfomed based on these definitions.

The **<tdl:Send>** command extends the <tdl:MessagingStep> with the following extra elements;

- **input (0..*): <tdl:Binding>** – The list of input elements that will be supplied to the messaging module (most of the time inputs will be the parts of the message that will be send to the SUT). The Binding class is an Expression with a **name** attribute. The expression is evaluated and the value is given as the input parameter. The binding of the supplied input elements to the input parameters of the module can be done in two different ways; either by name binding, or by the order of parameters. By name binding, the **name** attribute in each input element should refer the parameter name defined in module definition. In this way, the optional parameters may not be supplied. For the other way, supplied input elements should be in the same order defined in the module definition and for optional parameters; if the test designer does not want to supply the parameter he should use an empty input element.

The **<tdl:Receive>** and **<tdl:Listen>** commands extends the <tdl:MessagingStep> with the following extra elements:

- **output (0..*): <tdl:Binding> –** Messaging modules returns a set of outputs (message parts) as a result of receive or listen operations. These elements are used to bind the outputs to some variables in the Test Case definition. The binding of returned results to these elements can be done in two ways either by name binding, or by the order of parameters as in the case of input elements in the Send command. The expression given in the output element should be a variable expression. The value given in the corresponding output is then assigned to this variable.
- **id (0 .1) –** TDL provides a syntactic sugar for test designers to use the results (received messages) of Receive or Listen commands. Rather than binding the outputs to some variable, test designer can use the id attribute for the command without using any output elements. In this way, a map type variable will be created with this supplied id (name). The outputs of the step will be stored in this map where the keys are the output names defined in the module definition.

9.3.8 Validation Step

The **<tdl:Verify>** is used to represent validation steps in the test scenario where a specific validation methodology is applied on a given content.

- **handler –** The identifier (URN) for the validation module that will perform the actual validation.
- **config (0..*): <gitb:Configuration> –** The list of configuration parameters supplied to the validation module for the validation process.
- **input (1..*): <tdl:Binding> –** The list of inputs (content, schemas, etc) supplied to the validation module. For each input element the expression is evaluated and the value will be supplied as input parameter for the module. The same methodology described in messaging steps is used to bind the values to the parameters (binding by order and binding by names).
- **id (0..1) –** If test designer needs a decision step or loop step that depends on the result of a validation step, the id attribute can be used to give a name to the validation result. In this case, a Boolean variable named with the given id will be created and this variable can be accessed by further expressions in other steps.

9.3.9 User Interaction During Execution

The **<tdl:UserInteraction>** is used for steps to interact with SUT administrators during test execution. The Test Bed is expected to interact with users, show the child instructions and get the requested inputs for this step. When the interaction is finalized, the step is assumed to be completed and execution continues with the next step.

- **with –** Interaction can be with a specific SUT administrator for this step. In that case this attribute should refer (TestRole.name) the corresponding actor. If this attribute is not supplied, the "with" attribute should be supplied for each included Instruction or InputRequest element.
- **instruction (0..*): <tdl:InstructionOrRequest> –** The list of instructions for this interaction group. The details of the Instruction element are described in preliminary phase section.
- **request (0..*): <tdl:InstructionOrRequest> –** The list of input requests for this interaction group. The details of the Instruction element are described in preliminary phase section.

9.3.10 Interim Computations

Test designers may need some interim computations on the received content (messages, documents, inputs) between test steps to use them in later steps. The **<tdl:Assign>** is the supplementary test construct designed for this purpose. It extends Expression and the computation/processing performed with the expression is stored to a variable as a result of the construct.

- **to –** A variable expression indicating the variable to store the resulting value.
- **append (0..1)** – This attribute is only used for the list type variables to indicate whether the value calculated by the expression is a list type so a normal assignment is performed or it is not a container type and the value is appended to the list as a result.

9.3.11 Test Flow Steps

Sometimes a test scenario can include decision points where execution continue with a specific branch based on a decision. The **<tdl:IfStep>** is used to indicate such decision points.

- **cond: <tdl:Expression>** – A Boolean expression representing the condition for the decision point
- **then (0..1): <tdl:Sequence> –** The branch of steps that should be executed when the condition is evaluated true.
- **else (0..1): <tdl:Sequence> –** The branch of steps that should be executed when the condition is evaluated false.

Another construct required by any computational language is the loop construct to execute a part of the script in a loop based on some condition. The **<tdl:WhileStep>** is one of them in TDL for generic loops.

- **cond: <tdl:Expression> –** A Boolean expression representing the condition that decides to continue to the loop or not.
- **do: <tdl:Sequence> –** The sequence of steps to loop on.

The **<tdl:RepeatUntilStep>** is another loop construct which executes the child steps at least once and then decides to loop over based on the given condition.

- **do: <tdl:Sequence> –** The sequence of steps to loop on.
- **cond: <tdl:Expression> –** A Boolean expression representing the condition that decides to continue to the loop or not.

The **<tdl:ForEachStep>** is another loop construct for executing steps for a given number of times (iteration over a list type variable).

- **counter (0..1) –** Name of the iteration variable (number type). The default value is "i" if not supplied. The scope of the counter variable is the child steps given in the **do** sequence. It can be used as index for iteration over lists. For each iteration, the value of the variable is incremented.

- **start (0..1) –** The starting value for the counter variable. The default value is 0.
- **end –** The end value for the counter variable. If the value of the counter variable becomes larger than this value, the loop ends.

Some test scenarios need concurrent branches of steps that should be executed concurrently. The **<tdl:FlowStep>**is used for this purpose.

- **thread (1..*):<tdl:Sequence>** – Each thread represents a branch that should be executed concurrently.

The **<tdl:ExitStep>** is used to exit from the Test Case execution from any branch.

The **<tdl:GroupStep>,** extending Sequence class, is used to form a logical group of sequence of steps in order to better present the test scenario to SUT administrators.

9.3.12 Modular Test Scripting

Like developers writing code in any programming language, test designers need to group a set of steps, a sub execution flow, and reuse them in their Test Case descriptions more than once. The **<tdl:Scriptlet>** is used to define such function**–**like partial test scripts. Scriptlets can be defined within a Test Case definition or globally within the Test Suite packages. The following are the details of the element:

- **id –** The identifier used to identify the definition within the Test Case definition or a Test Suite package
- **metadata (0..1): <gitb:Metadata> –** The metadata of this partial test script definition.
- **namespace (0..*): <tdl:Namespace> –** The list of namespace declarations and their prefix bindings that will be used in the expressions used in the Scriptlet definition.
- **import (0..*): <tdl:Import> –** The list of import statements to declare the external test modules or test artifacts required for the execution of this partial definition.
- **param (0..*): <tdl:Variable> –** The definition of input parameters of this Scriptlet (like function parameters).
- **var (0..*): <tdl:Variable> –** Definition of local variables where the scope is this Scriptlet definition.
- **steps: <tdl:Sequence> –** The sequence of test steps for this partial test execution flow.
- **output (0..*): <tdl:TypedBinding> –** Definition of return values for this partial test script. The callee can access these values from its internal scope when the execution of the scriptlet is finalized. The TypeBinding extends Expression and the evaluated value of expression is returned as a result. As an extension to the Binding class, TypeBinding indicates the type of the returned result with the **type** attribute.

The **<tdl:CallStep>** is used to call a <tdl:Sciptlet> within a test case or another scriptlet definition.

- **path** – The identifier of the <tdl:Scriptlet> to call.
- **input (0..*): <tdl:Binding>** – Input parameters supplied to the scriptlet. Binding is performed similar to bindings in other constructs.
- **output (0..*): <tdl:Binding> –** Defines the bindings of the Scriptlet outputs to some variable in the context. Binding is performed similar to the bindings in other constructs.
- **id (0..1) –** Can be used to directly access the results of the scriptlet from a map type variable initialized by this name. In that case, the test designer does not need to use the output elements.

9.3.13 Expressions and Bindings

The **<tdl:Expression>** element is used to represent TDL expressions and used in all other TDL constructs as described in the above sections.

- **lang (0..1) –** Any expression language (both in terms of syntax and semantics) can be used as the TDL expressions if the Test Bed supports it. This attribute provides the unique identifier (URN) for

the language for this expression. TDL provides a default XPath-based expression language and if this attribute is not supplied this scheme should be assumed.

- **source (0..1) –** The input source for the expression. In other words the expression will be evaluated based on this source content. Should be a variable expression (left value).
- **expr (0..1)** – The string representing the expression. If not supplied, the result is Null value.

The Default TDL Expression scheme extends the XPath 2.0 with the following simple extensions;

- The Variable References in the XPath expressions are extended to access the values of GITB container typed (list and map) variables. The following rules apply:
  - $<variable-name> is used as usual for the value of a variable (ex: $x to access to the x's value).
  - $<variable-name>{<key-name>} is used to access an entry (with the given key) in a map typed variable (ex: $x{name} to access the entry in the x with key “name”).
  - $<variable-name>{<numeric-index>} is used to access an entry of the list type attribute at the given index (ex: $x{0} access to the first element in the x list).

The same rules applied to the TDL Variable Expressions (reference to the variables) to refer the variables.

Some test constructs in TDL (messaging steps, interaction steps, validation steps, calling scriptlets) get inputs and return outputs within the test execution. In order to bind values to these input and output parameters, the **<tdl:Binding>** element is used. Binding extends Expression with the the following attribute:

- **name (0..1) –** Name of the input parameter that the evaluated expression value will be bind while supplying input. Similarly, it can be the name of the output parameter that its return value will be bind to the given variable reference. As described in the related constructs, if this attribute is not used, the parameters will be bound in the same order.

9.4 XML Schema for TDL

```
 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xsd:schema            xmlns="http://www.gitb.com/tdl/v1/"            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:gitb="http://www.gitb.com/core/v1/"                          targetNamespace="http://www.gitb.com/tdl/v1/"
elementFormDefault="qualified" version="1.0">
     <xsd:import namespace="http://www.gitb.com/core/v1/" schemaLocation="gitb_core.xsd"/>
     <xsd:element name="testcase" type="TestCase"/>
     <xsd:element name="testsuite" type="TestSuite"/>
     <xsd:complexType name="TestSuite">
          <xsd:sequence>
               <xsd:element name="metadata" type="gitb:Metadata"/>
               <xsd:element name="actors" type="gitb:Actors"/>
               <xsd:element name="testcase" type="TestCaseEntry" maxOccurs="unbounded"/>
          </xsd:sequence>
          <xsd:attribute name="id" type="xsd:string" use="required"/>
     </xsd:complexType>
     <xsd:complexType name="TestCaseEntry">
          <xsd:sequence>
               <xsd:element name="prequisite" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element name="option" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
          </xsd:sequence>
          <xsd:attribute name="id" type="xsd:string" use="required"/>
     </xsd:complexType>
     <xsd:complexType name="TestCase">
          <xsd:sequence>
               <xsd:element name="metadata" type="gitb:Metadata"/>
               <xsd:element name="namespaces" type="Namespaces" minOccurs="0"/>
               <xsd:element name="imports" type="Imports" minOccurs="0"/>
               <xsd:element name="preliminary" type="UserInteraction" minOccurs="0"/>
               <xsd:element name="variables" type="Variables" minOccurs="0"/>
               <xsd:element name="actors" type="gitb:Roles"/>
               <xsd:element name="steps" type="Sequence"/>
               <xsd:element name="scriptlets" type="Scriptlets" minOccurs="0"/>
          </xsd:sequence>
          <xsd:attribute name="id" type="xsd:string" use="required"/>
     </xsd:complexType>
     <xsd:complexType name="Namespaces">
          <xsd:sequence>
               <xsd:element name="ns" type="Namespace" maxOccurs="unbounded"/>
          </xsd:sequence>
     </xsd:complexType>
     <xsd:complexType name="Namespace">
          <xsd:simpleContent>
               <xsd:extension base="xsd:string">
                    <xsd:attribute name="prefix" type="xsd:string" use="required"/>
               </xsd:extension>
          </xsd:simpleContent>
```

```
    </xsd:complexType>
    <xsd:complexType name="Imports">
        <xsd:sequence>
            <xsd:choice maxOccurs="unbounded">
                <xsd:element name="artifact" type="TestArtifact"/>
                <xsd:element name="module" type="TestModule"/>
            </xsd:choice>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="TestArtifact">
        <xsd:simpleContent>
            <xsd:extension base="xsd:string">
                <xsd:attribute name="name" type="xsd:ID" use="required"/>
                <xsd:attribute name="type" type="xsd:string" use="required"/>
                <xsd:attribute name="encoding" type="xsd:string" use="optional"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>
    <xsd:complexType name="TestModule">
        <xsd:sequence>
            <xsd:element name="config" type="gitb:Configuration" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
        <xsd:attribute name="uri" type="xsd:string" use="required"/>
    </xsd:complexType>
    <xsd:complexType name="Variables">
        <xsd:sequence>
            <xsd:element name="var" type="Variable" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="Variable">
        <xsd:sequence>
            <xsd:element name="value" type="TypedBinding" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
        <xsd:attribute name="type" type="xsd:string" use="required"/>
    </xsd:complexType>
    <xsd:complexType name="Scriptlets">
        <xsd:sequence>
            <xsd:element name="scriptlet" type="Scriptlet" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="Scriptlet">
        <xsd:sequence>
            <xsd:element name="metadata" type="gitb:Metadata" minOccurs="0"/>
            <xsd:element name="namespaces" type="Namespaces" minOccurs="0"/>
            <xsd:element name="imports" type="Imports" minOccurs="0"/>
            <xsd:element name="params" type="Variables" minOccurs="0"/>
            <xsd:element name="variables" type="Variables" minOccurs="0"/>
            <xsd:element name="steps" type="Sequence"/>
            <xsd:element name="output" type="Binding" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:string" use="required"/>
    </xsd:complexType>
    <xsd:complexType name="Sequence">
        <xsd:sequence>
            <xsd:choice maxOccurs="unbounded">
                <!-- Messaging Test Steps Constructs -->
                <xsd:element name="send" type="Send"/>
                <xsd:element name="receive" type="Receive"/>
                <xsd:element name="listen" type="Listen"/>
                <xsd:element name="btxn" type="BeginTransaction"/>
                <xsd:element name="etxn" type="EndTransaction"/>
                <!-- Flow constructs -->
                <xsd:element name="if" type="IfStep"/>
                <xsd:element name="while" type="WhileStep"/>
                <xsd:element name="repuntil" type="RepeatUntilStep"/>
                <xsd:element name="foreach" type="ForEachStep"/>
                <xsd:element name="flow" type="FlowStep"/>
                <xsd:element name="exit" type="ExitStep"/>
                <!--Testing & Supplemantary constructs -->
                <xsd:element name="assign" type="Assign"/>
                <xsd:element name="group" type="Group"/>
                <xsd:element name="verify" type="Verify"/>
                <xsd:element name="call" type="CallStep"/>
                <xsd:element name="interact" type="UserInteraction"/>
            </xsd:choice>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="TestConstruct">
        <xsd:attribute name="id" type="xsd:string" use="optional"/>
    </xsd:complexType>
    <xsd:complexType name="TestStep">
        <xsd:complexContent>
            <xsd:extension base="TestConstruct">
                <xsd:attribute name="desc" type="xsd:string" use="required"/>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="MessagingStep">
        <xsd:complexContent>
            <xsd:extension base="TestStep">
                <xsd:sequence>
                    <xsd:element name="config" type="gitb:Configuration" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
                <xsd:attribute name="from" type="xsd:string" use="required"/>
                <xsd:attribute name="to" type="xsd:string" use="required"/>
                <xsd:attribute name="txnId" type="xsd:string" use="required"/>
            </xsd:extension>
```

```
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="Send">
        <xsd:complexContent>
            <xsd:extension base="MessagingStep">
                <xsd:sequence>
                    <xsd:element name="input" type="Binding" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="ReceiveOrListen">
        <xsd:complexContent>
            <xsd:extension base="MessagingStep">
                <xsd:sequence>
                    <xsd:element name="output" type="Binding" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="Receive">
        <xsd:complexContent>
            <xsd:extension base="ReceiveOrListen"/>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="Listen">
        <xsd:complexContent>
            <xsd:extension base="ReceiveOrListen"/>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="BeginTransaction">
        <xsd:complexContent>
            <xsd:extension base="TestConstruct">
                <xsd:sequence>
                    <xsd:element name="config" type="gitb:Configuration" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
                <xsd:attribute name="from" type="xsd:string" use="required"/>
                <xsd:attribute name="to" type="xsd:string" use="required"/>
                <xsd:attribute name="txnId" type="xsd:string" use="required"/>
                <xsd:attribute name="handler" type="xsd:string" use="required"/>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="EndTransaction">
        <xsd:complexContent>
            <xsd:extension base="TestConstruct">
                <xsd:attribute name="txnId" type="xsd:string" use="required"/>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="IfStep">
        <xsd:complexContent>
            <xsd:extension base="TestStep">
                <xsd:sequence>
                    <xsd:element name="cond" type="Expression"/>
                    <xsd:element name="then" type="Sequence"/>
                    <xsd:element name="else" type="Sequence"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="WhileStep">
        <xsd:complexContent>
            <xsd:extension base="TestStep">
                <xsd:sequence>
                    <xsd:element name="cond" type="Expression"/>
                    <xsd:element name="do" type="Sequence"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="RepeatUntilStep">
        <xsd:complexContent>
            <xsd:extension base="TestStep">
                <xsd:sequence>
                    <xsd:element name="do" type="Sequence"/>
                    <xsd:element name="cond" type="Expression"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="ForEachStep">
        <xsd:complexContent>
            <xsd:extension base="TestStep">
                <xsd:sequence>
                    <xsd:element name="do" type="Sequence"/>
                </xsd:sequence>
                <xsd:attribute name="counter" type="xsd:string" use="optional" default="i"/>
                <xsd:attribute name="start" type="xsd:integer" use="optional" default="0"/>
                <xsd:attribute name="end" type="xsd:integer" use="required"/>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="ExitStep">
        <xsd:complexContent>
            <xsd:extension base="TestStep"/>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="FlowStep">
```

```
          <xsd:complexContent>
               <xsd:extension base="TestStep">
                    <xsd:sequence>
                         <xsd:element name="thread" type="Sequence" maxOccurs="unbounded"/>
                    </xsd:sequence>
               </xsd:extension>
          </xsd:complexContent>
     </xsd:complexType>
     <xsd:complexType name="Assign">
          <xsd:complexContent>
               <xsd:extension base="Expression">
                    <xsd:attribute name="to" type="xsd:string" use="required"/>
                    <xsd:attribute name="append" type="xsd:boolean" use="optional" default="false"/>
                    <xsd:attribute name="type" type="xsd:string" use="optional" />
               </xsd:extension>
          </xsd:complexContent>
     </xsd:complexType>
     <xsd:complexType name="Group">
          <xsd:complexContent>
               <xsd:extension base="Sequence">
                    <xsd:attribute name="id" type="xsd:string" use="optional"/>
                    <xsd:attribute name="desc" type="xsd:string" use="required"/>
               </xsd:extension>
          </xsd:complexContent>
     </xsd:complexType>
     <xsd:complexType name="Verify">
          <xsd:complexContent>
               <xsd:extension base="TestStep">
                    <xsd:sequence>
                         <xsd:element name="config" type="gitb:Configuration" minOccurs="0" maxOccurs="unbounded"/>
                         <xsd:element name="input" type="Binding" maxOccurs="unbounded"/>
                    </xsd:sequence>
                    <xsd:attribute name="handler" type="xsd:string" use="required"/>
               </xsd:extension>
          </xsd:complexContent>
     </xsd:complexType>
     <xsd:complexType name="CallStep">
          <xsd:complexContent>
               <xsd:extension base="TestConstruct">
                    <xsd:sequence>
                         <xsd:element name="input" type="Binding" minOccurs="0" maxOccurs="unbounded"/>
                         <xsd:element name="output" type="Binding" minOccurs="0" maxOccurs="unbounded"/>
                    </xsd:sequence>
                    <xsd:attribute name="path" type="xsd:string" use="required"/>
               </xsd:extension>
          </xsd:complexContent>
     </xsd:complexType>
     <xsd:complexType name="UserInteraction">
          <xsd:complexContent>
               <xsd:extension base="TestStep">
                    <xsd:sequence>
                         <xsd:choice maxOccurs="unbounded">
                              <xsd:element name="instruct" type="Instruction"/>
                              <xsd:element name="request" type="UserRequest"/>
                         </xsd:choice>
                    </xsd:sequence>
                    <xsd:attribute name="with" type="xsd:string" use="optional"/>
               </xsd:extension>
          </xsd:complexContent>
     </xsd:complexType>
     <xsd:complexType name="Instruction">
          <xsd:complexContent>
               <xsd:extension base="InstructionOrRequest"/>
          </xsd:complexContent>
     </xsd:complexType>
     <xsd:complexType name="UserRequest">
          <xsd:complexContent>
               <xsd:extension base="InstructionOrRequest"/>
          </xsd:complexContent>
     </xsd:complexType>
     <xsd:complexType name="InstructionOrRequest" abstract="true">
          <xsd:complexContent>
               <xsd:extension base="TypedBinding">
                    <xsd:attribute name="desc" type="xsd:string" use="required"/>
                    <xsd:attribute name="with" type="xsd:string" use="required"/>
                  <xsd:attribute name="contentType" type="gitb:ValueEmbeddingEnumeration" use="optional"/>
                    <xsd:attribute name="encoding" type="xsd:string" use="optional"/>
               </xsd:extension>
          </xsd:complexContent>
     </xsd:complexType>
     <xsd:complexType name="Expression">
          <xsd:simpleContent>
               <xsd:extension base="xsd:string">
                    <xsd:attribute name="lang" type="xsd:string" use="optional"/>
                    <xsd:attribute name="source" type="xsd:string" use="optional"/>
               </xsd:extension>
          </xsd:simpleContent>
     </xsd:complexType>
     <xsd:complexType name="Binding">
          <xsd:complexContent>
               <xsd:extension base="Expression">
                    <xsd:attribute name="name" type="xsd:string" use="optional"/>
               </xsd:extension>
          </xsd:complexContent>
     </xsd:complexType>
     <xsd:complexType name="TypedBinding">
          <xsd:complexContent>
               <xsd:extension base="Binding">
                    <xsd:attribute name="type" type="xsd:string" use="optional"/>
```

```
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:simpleType name="TestModuleTypes">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="MESSAGING"/>
            <xsd:enumeration value="VALIDATION"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:schema>
```

10 GITB Proof of Concept (PoC) Test Bed Implementation

This section presents an overview of the GITB Test Bed implementation, which has been developed in GITB Phase 3 as Proof-of-Concept (PoC) for the GITB architecture and specifications.

The GITB PoC Test Bed is an open source project and its source code can be found at GitHub Repository[7]. For code contribution, Git[8], which is a distributed revision control and source code management (SCM) system, is used. Git enables distributed development and provides strong support for non-linear (branching and merging) development, which is important for the GITB PoC Test Bed implementation.

The collaboration model that is followed in GITB PoC Test Bed development adopts a feature based workflow that suggests creation of a new branch for each new feature. When a new feature is decided to be developed, a new development branch, which denotes a slightly different direction in which the development is proceeding, is created. After the feature is implemented, if the feature is complete, it is merged into the master development branch. Therefore, a development branch never affects a stable release.

10.1 Software Architecture

The software architecture behind the GITB PoC Test Bed consists of two main components – the GITB Testbed component and the GITB Execution Interface.

- **GITB Testbed** is responsible for execution of conformance and interoperability tests through a set of services.
- **GITB Execution Interface** provides a Graphical User Interface (GUI) and a REST API to manage a number of user activities (account and SUT registration, conformance statement definition, etc.) and testing operations by utilizing the services exposed by GITB Testbed.

Figure 10-110-1the interactions between the two main components and their modules.

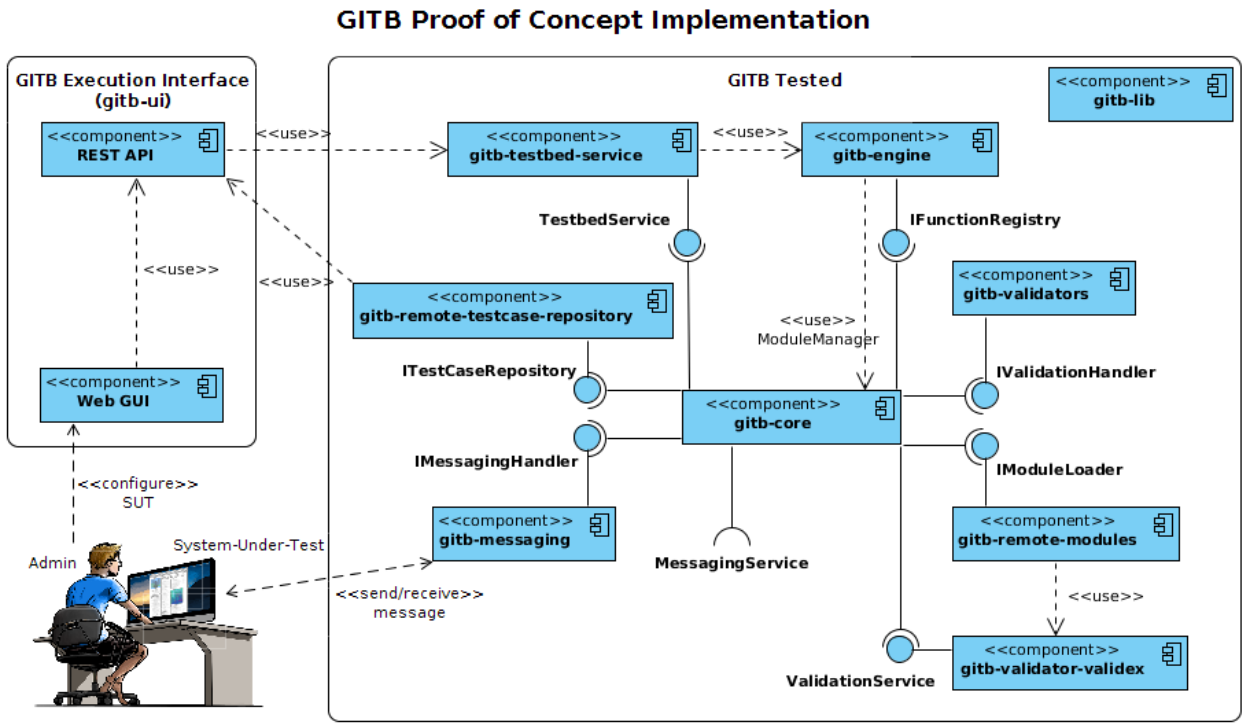

Figure 10-1: GITB PoC Implementation Components

[7] https://github.com/srdc/gitb

[8] http://git-scm.com/

10.1.1 GITB Testbed

In order to be able to support and test a wide range of messaging protocols, business document formats and document exchange choreographies, a modular approach needs to be embraced by GITB Testbed component. This is achieved by adopting an interface-based architecture. The latter enables modularity and adaptability, thus, increases maintainability and facilitates development of additional auxiliary modules. In this way, the GITB Testbed component is built as a collection of modules, and API calls among its modules can only be established through the defined interfaces. At software level, this structure is realized by utilizing Apache Maven[9], which is a software management and comprehension tool. With the help of interfaces developed and Maven's powerful module management facilities, new modules can be developed and existing modules can be integrated without requiring much effort. In this way, the GITB Testbed's capabilities can be further extended without hindering the Test Bed execution. Maven is also used for dependency management, build automation, and for a broad range of plugins.

Each module within the GITB Testbed component has an XML file representation kept in a file named **pom.xml.** This representation is called Project Object Model (POM) and is the central construct of the Maven's build management philosophy. The POM file describes the intended software project being developed, its dependencies on other modules or external software libraries, the build order, managed resources and needed plug-ins. It comes with pre-defined targets for managing the life-cycle phases such as compilation, packaging and deployment.

A multi-module software project like the GITB Testbed is defined by a parent POM (or top-level POM) referencing its modules. As a result of doing so, modules are grouped together. When a Maven command is executed against the parent POM, the same command will be executed at child modules, as well. For instance, building the parent will eventually build all modules, without the need of building each module separately. The top-level POM of GITB Testbed components can be seen below.

Parent POM also defines a set of Maven coordinates: groupId is *com.gitb*, the artifactId is *GITB* and the version is *1.0-SNAPSHOT*. Furthermore, some global properties such as *compiler.version, gitb.version* are defined in this POM, so that they are available in all child modules. The parent project does not create a JAR or a WAR like other modules; instead, it is simply a POM that refers to its child modules. Additionally, every child module has to specify who their parent POM is, in their POM files, too.

[9] http://maven.apache.org/

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <properties>
    <gitb.version>1.0-SNAPSHOT</gitb.version>
    <compiler.version>1.7</compiler.version>
  </properties>

  <modelVersion>4.0.0</modelVersion>
  <name>GITB</name>
  <groupId>com.gitb</groupId>
  <artifactId>GITB</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>

  <build>
     <plugins>
       <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-compiler-plugin</artifactId>
         <version>3.0</version>
         <configuration>
           <source>${compiler.version}</source>
           <target>${compiler.version}</target>
         </configuration>
       </plugin>
       <plugin>
         <groupId>org.eclipse.jetty</groupId>
         <artifactId>jetty-maven-plugin</artifactId>
         <version>9.2.2.v20140723</version>
         <configuration>
           <jvmArgs>-Dorg.eclipse.jetty.annotations.maxWait=180</jvmArgs>
           <contextXml>gitb-testbed-service/src/main/webapp/WEB-INF/jetty-context.xml</contextXml>
         </configuration>
       </plugin>
     </plugins>
  </build>

  <modules>
     <module>gitb-core</module>
     <module>gitb-engine</module>
     <module>gitb-lib</module>
    <module>gitb-messaging</module>
    <module>gitb-remote-testcase-repository</module>
    <module>gitb-remote-modules</module>
    <module>gitb-testbed-service</module>
    <module>gitb-validator-validex</module>
    <module>gitb-validators</module>
  </modules>
</project>
```

10.1.2 GITB Testbed Modules

10.1.2.1 The Central Part of the GITB Testbed: gitb-core

gitb-core module is the central part of the modular architecture of the GITB Testbed component. It provides a data model for the GITB type system explained in section 9.1.2, exceptions that may be thrown during the execution of the Test Bed and classes used for messaging operations, as well as an internal API through a set of interfaces for extendable functionalities. These functionalities encompass development of additional messaging and validation modules as well as integration of external messaging and validation services,

external test case repositories and additional function registries. The current GITB Test Bed implementation provides a number of already-developed functionalities which will be explained in the upcoming sections.

The internal APIs provided by gitb-core module are the following:

- **IMessagingHandler:** Provides abstract methods for implementing a new messaging service instead of integrating an existing one. These methods have similar signatures and functionalities as the ones defined in the Messaging (Simulation) Service Specifications. They are briefly explained below:
  - **getModuleDefinition:** Returns the module details including messaging configurations and inputs that the module requires and the outputs that the module provides.
  - **initiate:** Creates a session between the messaging service client (SUT) and GITB Testbed before any transaction is realized.
  - **beginTransaction:** Creates a communication between SUT and GITB Testbed with the given session ID and configurations.
  - **sendMessage:** Allows GITB Testbed to send given messages to a SUT according to given configurations over an existing transaction.
  - **receiveMessage:** Allows GITB Testbed to receive messages from a SUT according to given configurations over an existing transaction.
  - **listenMessage:** Allows GITB Testbed to listen messages between two SUTs according to given configurations over an existing transaction.
  - **endTransaction:** Destroys the given transaction in a session.
  - **endSession:** Destroys the session.

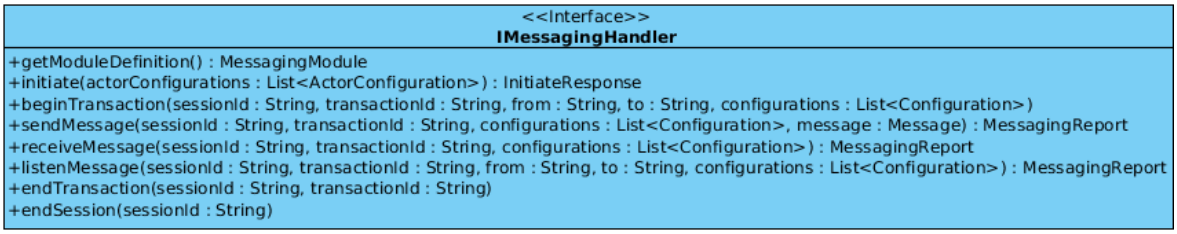

Figure 10-2: Class Diagram of IMessagingHandler Interface

- **IModuleLoader:** Provides abstract methods for retrieving proxies of external Content Validation Services and Messaging (Simulation) Services implementing IValidationHandler and IMessagingHandler interfaces.
  - **loadValidationHandlers:** Returns all integrated Content Validation Service proxies implementing IValidationHandler interface.
  - **loadMessagingHandlers:** Returns all integrated Messaging (Simulation) Service proxies implementing IMessagingHandler interface.

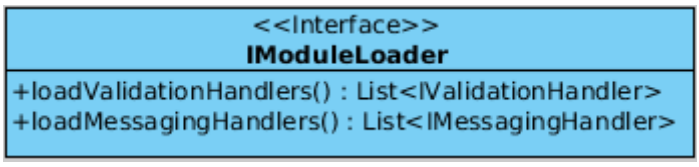

Figure 10-3: Class Diagram of IModuleLoader Interface

- **IFunctionRegistry:** An interface for writing user-defined, reflexive extension functions to offer additional features beyond those specified in the XPath Specifications. In other words, extension functions enable invocation of JAVA methods, as if calling XPath functions, during Test Case processing since GITB Testbed uses an XPath 2.0 based expression language, by default.

An extension function is written in JAVA and invoked by using the pattern, *prefix:localname()* within a Test Case document. The prefix must be the prefix associated with a namespace declaration in the Test Case within **tdl:Namespace** element.

- **getName:** Returns the unique name of the function registry to identify it among the other function registries.
- **isFunctionAvailable:** Returns a Boolean value indicating the existence of a function, which is referenced from a test case, within a function registry.
- **callFunction:** Invokes a function defined in a function registry with given name and arguments.

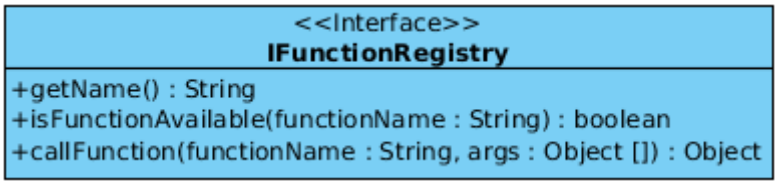

Figure 10-4: Class Diagram of IFunctionRegistry Interface

- **ITestCaseRepository:** This interface defines generic methods for retrieval of various Testing resources (Test Cases, Test Suites, scriptlets or various Test Artifacts) on demand. Modules implementing this interface do not have to contain the Testing Resources locally within the module; instead, they can provide access to external repositories. Therefore, any remote Test Resources repository serving on a specific protocol (local or remote file system, TCP, HTTP, etc) can be integrated with GITB Testbed by implementing the ITestCaseRepository methods according to the protocol requirements.
  - **getName:** Returns the unique name of the Test Case repository to identify it among the other Test Case repositories.
  - **isTestCaseAvailable:** Returns a Boolean value indicating the existence of a Test Case with given ID in a repository.
  - **getTestCase:** Retrieves a tdl:TestCase object with given ID
  - **isTestSuiteAvailable:** Returns a Boolean value indicating the existence of a test suit with given ID in a repository.
  - **getTestSuite:** Retrieves a tdl:TestSuite object with given ID
  - **isScriptletAvailable:** Returns a Boolean value indicating the existence of a scriptlet with given ID in a repository
  - **getScriptlet:** Retrieves a tdl:Scriptlet object with given ID
  - **isTestArtifactAvailable:** Returns a Boolean value indicating the existence of a Testing Resource with given path in a repository
  - **getTestArtifact:** Returns a JAVA InputStream for the Testing Resource from given path.

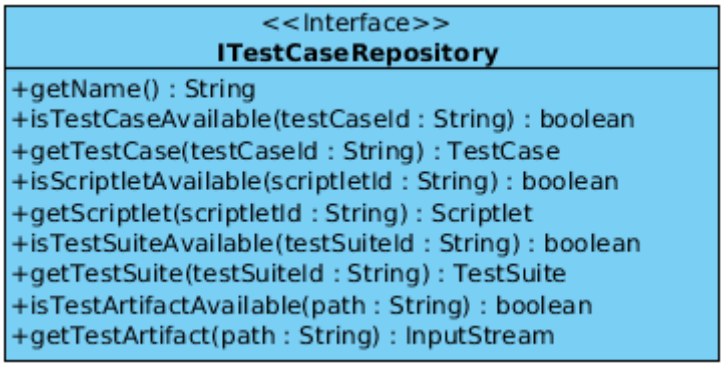

Figure 10-5: Class Diagram of ITestCaseRepository Interface

- **IValidationHandler:** Provides abstract methods for implementing a new validation service instead of integrating an existing one. These methods have similar signatures and functionalities as the methods defined in Content Validation Service Specifications and they are briefly explained below:
    - **getModuleDefinition:** Returns the module details including validation configurations and inputs that the module requires.
    - **validate:** Validates the content with given inputs and configurations and returns a report about the validation operation, providing the overall result and description of performed assertions, found errors and warnings.

Figure 10-6 Class Diagram of IValidationHandler Interface

- **ValidationService:** Provides Web service methods of GITB Content Validation Service.
- **MessagingService:** Provides Web service methods of GITB Messaging (Simulation) Service.
- **TestbedService:** Provides Web service methods of GITB Testbed Service.

As mentioned before, the GITB Testbed component consists of several modules and gitb-core is the center of this modular architecture. That is because gitb-core enables access to all these functionalities from a single point, which is, in fact, a singleton class called ModuleManager. In order to achieve this behavior, each implementor class, which denotes a concrete implementation of either the abovementioned interfaces or abstract GITB Data Type class, registers itself as a service provider through the JAVA Service Provider mechanism. This registration process is managed via one line of code by utilizing an annotation-driven META-INF/services auto-generator library[10]. By annotating the implementor classes with **@MetaInfServices(<interface_name>.class)** annotation provided by this library, a service provider configuration file, whose name is the fully-qualified binary name of the interface (e.g. com.gitb.messaging.IMessagingHandler) or abstract GITB Data Type class (com.gitb.types.DataType), is generated automatically and this file contains the names of implementor classes, one per line. Then, when GITB Testbed starts, the ModuleManager class aggregates all the implementations of the interfaces or GITB Data Types by calling the *load* method, which scans all the service provider configuration files in the run-time environment and returns them, of **java.util.ServiceLoader** class and serves them to the other modules. Identification of validation and messaging modules is carried out with the identifiers defined in their module definitions whereas each Test Case repository, function registry or GITB Data Type must have unique names within a running instance. As it can be seen in the class diagram below, ModuleManager is a singleton class. Concrete implementors are kept inside of private java.util.Map objects and accessed by public accessors.

[10] http://metainf-services.kohsuke.org/

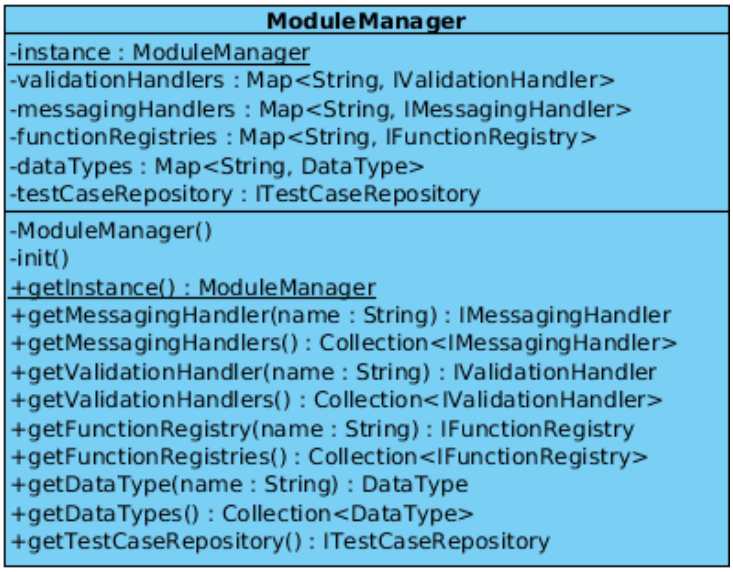

Figure 10-7: Class Diagram of ModuleManager Class

Another important characteristic of gitb-core module is that it also provides the object models for XML Schemas for TDL, TPL and Test Report format and Web Service stubs for TestbedService, MessagingService and ValidationService. However, these object models are not provided directly. Instead, gitb-core converts XML Schemas and WSDL files given in previous sections into JAVA objects by utilizing Maven JAXB plug-in[11] during *generate-sources* lifecycle phase of Maven. Therefore, every GITB Testbed module requires gitb-core module as a dependency in order to coordinate testing activities.

10.1.2.1.1 Utility Classes: gitb-lib

gitb-lib module provides useful utility classes that define a set of methods performing common and reusable functions to be used by other modules, without encapsulating any state information. These classes are grouped under *com.gitb.utils* package and their job is summarized below:

- **ActorUtils:** Provides utilities for extracting information such as, actor IDs, actor configurations, endpoint IDs, endpoint name from gitb:ActorConfiguration objects.
- **BindingUtils:** Provides a method determining whether all elements in a list of tdl:Binding have the optional “name” attribute.
- **ConfigurationUtils:** Provides methods for manipulating gitb:Configuration objects.
- **DataTypeUtils:** Provides methods for conversion between GITB types and gitb:AnyContent.
- **EncodingUtils:** GITB Testbed is able to process messages in different types of encoding schemes. This class provides utilities for conversion among those encodings.
- **JarUtils:** Provides methods for retrieving JAR files of external messaging or validation proxy modules to access their module definitions.
- **TimeUtils:** Provides methods for performing formatting and conversion operations on date & time information represented by JAVA String and Date classes.
- **XMLDateTimeUtils:** Provides methods for manipulating date & time information of type XMLGregorianCalendar, which is the representation for W3C XML Schema 1.0 date/time datatypes.
- **XMLUtils:** Provides many methods for XML processing, conversion, transformation and etc.

10.1.2.1.2 Access to Test Case Artifacts: gitb-remote-testcase-repository

gitb-remote-testcase-repository module provides access to Testing Resources served by GITB Execution Interface API, over HTTP protocol, so that Test Cases artifacts can be retrieved and processed for test execution. There are only 2 classes within this module:

[11] https://jax-ws-commons.java.net/jaxws-maven-plugin/

- **RemoteTestCaseRepository:** Implements ITestCaseRepository interface to retrieve Testing Resources from GITB Execution Interface. Also, utilizes a LRU (Least Recent Used) Cache mechanism in order to avoid repetitive HTTP calls for same resources.
- **TestCaseRepositoryConfiguration:** Provides configuration parameters such as URLs of GITB Execution Interface repository services to retrieve Test Cases and Artifacts. Configurations are retrieved from **remote-testcase-repository.properties** located in the resources folder. The content of remote-testcase-repository.properties file can be seen below:

```
#name of the placeholder string in remote.testcase.repository.url configuration
remote.testcase.test-id.parameter = test_id

#name of the placeholder string in remote.testresource.repository.url configuration
remote.testcase.resource-id.parameter = resource_id

#URL of the test case provider service
remote.testcase.repository.url = http://localhost:9000/repository/tests/:test_id/definition

#URL of the test resource provider service
remote.testresource.repository.url = http://localhost:9000/repository/suites/:resource_id
```

10.1.2.1.3 Validators: gitb-validators

This module provides a built-in validation architecture as well as a simple reporting mechanism for generating validation results with basic validators. The latter are, actually, concrete implementations of **IValidationHandler** interface of the internal API provided by gitb-core module. This architecture and reporting mechanism is responsible for checking the validity and syntactical correctness of the business messages/documents retrieved by messaging adapters and for reporting the overall result of validation operation with performed assertions, found errors and warnings.

Currently, gitb-validators module provides 3 validators with their module definition information. Each module converts the corresponding XML file containing the module definition into a JAVA object with the help of XMLUtils class provided by gitb-lib module. An example module definition can be seen below:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<module xmlns="http://www.gitb.com/core/v1/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    id="XSDValidator" uri="urn:com:gitb:validation:XSDValidator" xsi:type="ValidationModule">
  <!--Describes the metadata of the validator (e.g. its name, version, description, etc) -->
  <metadata>
    <name>XSD Validator</name>
    <version>1.0</version>
  </metadata>
  <!--Inputs to this validator. Generally, a validator takes two inputs: validator schema and content to be validated -
->
  <inputs>
    <param type="schema" use="R" name="xsddocument" desc="XSD Document" />
    <param type="object" use="R" name="xmldocument" desc="XML Document to be validated" />
  </inputs>
</module>
```

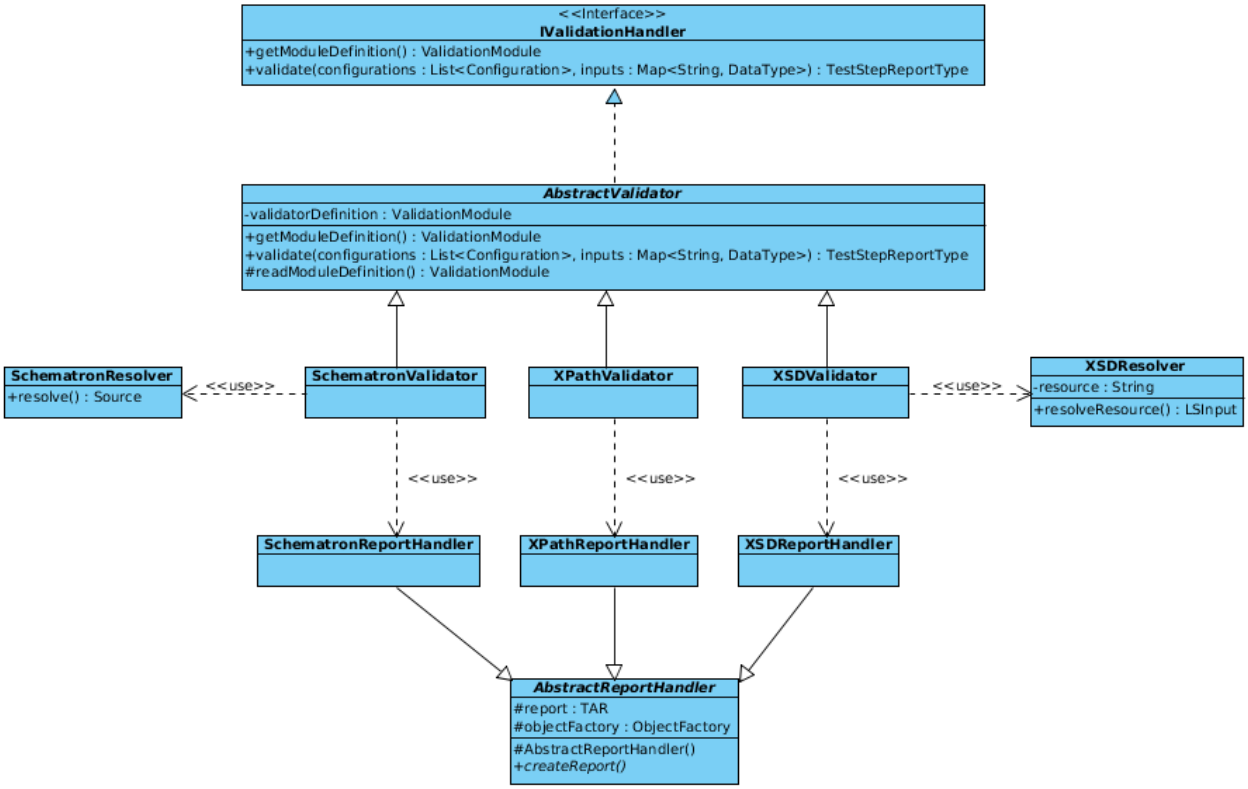

Figure 10-8: Class Diagram of GITB Validators

- **SchematronValidator:** Validates XML content against a Schematron, which is a rule-based validation language for making assertions about the presence or absence of XML elements[12], by using ph-schematron[13] library. Schematron files may import external Schematron's and resources and they must be resolved during document validation. Therefore, SchematronValidator, utilizes a helper resource resolver class, SchematronResolver.
- **XPathValidator:** Evaluates an XPath expression on a given XML content. One requirement of XPath validation is that the evaluation result of a provided XPath expression must yield a Boolean value.
- **XSDValidator:** Validates XML content against an XSD Schema, by using JAVA XML Validation API. XSD files may import external XSD files and they must be resolved during document validation. Therefore, XSDValidator, utilizes a helper resource resolver class, XSDResolver.

10.1.2.1.4 Messaging Adapters: gitb-messaging

This module provides a comprehensive messaging architecture with basic messaging adapters which are concrete implementations of **IMessagingHandler** interface of the internal API provided by gitb-core module. This architecture enables the communication with SUTs over specified transport and communication protocols and provides a simple validation mechanism to check if retrieved messages conform to transport level specifications. To enable communication over different protocols at the same time, messaging architecture is designed to consist of 3 layers.

1. **Messaging API**
   Messaging API enables 3 main types of communication: **receive**, **send** and **listen.** As mentioned before, GITB Testbed may **receive** messages from SUTs, **send** messages to SUTs and **listen** messages between two SUT actors. **listen** operation is actually a combination of **receive** and **send** operations where the testbed receives message from sender SUT actor and sends the received message to

[12] http://www.schematron.com/

[13] https://github.com/phax/ph-schematron

receiver SUT actor, therefore “listens” the communication between them. In order to realize the specified communication type in a generic and isolated way, 3 generic interfaces are designed: **IReceiver**, **ISender** and **IListener**.

IReceiver interface defines the following methods:

- **receive:** Performs the actual receive operation according to transport level specifications.
- **onError:** Called when an error occurs during receive operation.
- **onEnd:** Called when receive operation completes.

ISender interface defines the following methods:

- **send:** Performs the actual send operation according to transport level specifications.
- **onEnd:** Called when send operation.

IListener interface defines the following operations.

- **listen:** Performs the listen operation by utilizing receive method of IReceiver, then send method of ISender.
- **tranformMessage:** There may be differences between the structure of the input messages for send operation and the structure of the output messages retrieved from receive operation. Considering that, listen message is a combination of receive and send operations, message received from receive operations must be transformed into the appropriate message format for send operation. Therefore listeners must implement this method to perform such a transformation.
- **transformConfigurations:** Like in the case of transformMessage method, this method must be implemented to transform receive operation configurations into send operation configurations.

Currently, gitb-messaging module supports two core protocols of the Internet protocol suite: UDP and TCP. Thereby, two additional interfaces are provided for each of communication types. **ITransactionReceiver** and **IDatagramReceiver** interfaces extend IReceiver and realize receive operation over TCP and UDP protocols, respectively. **ITransactionSender** and **IDatagramSender** interfaces extend ISender interface and enable send operation over TCP and UDP protocols, respectively. Finally, **ITransactionListener** and **IDatagramListener** extend IListener interface and realize listen operation over TCP and UDP protocols, respectively. These six interfaces are implemented by corresponding abstract classes to construct a basis for the internal messaging API in which the actual communication is established with network sockets. **AbstractTransaction*** implementers utilizes java.net.Socket classes and perform low-level operations such as creation of sockets and blocking listener threads until a message received for TCP communication while **AbstractDatagram*** implementers benefit from java.net.DatagramSocket and java.net.DatagramPacket classes and perform similar low-level operations over UDP protocol. When developing receivers, senders or listeners for a specific protocol, appropriate abstract classes described above should be extended and implemented so that, implementation of receive, send and listen operations for TCP and UDP protocols can be separated from each other. Finally, lifecycle of receivers, senders and listeners are handled by corresponding messaging handlers which form the highest level of the layered architecture of gitb-messaging module and their details will be explained later.

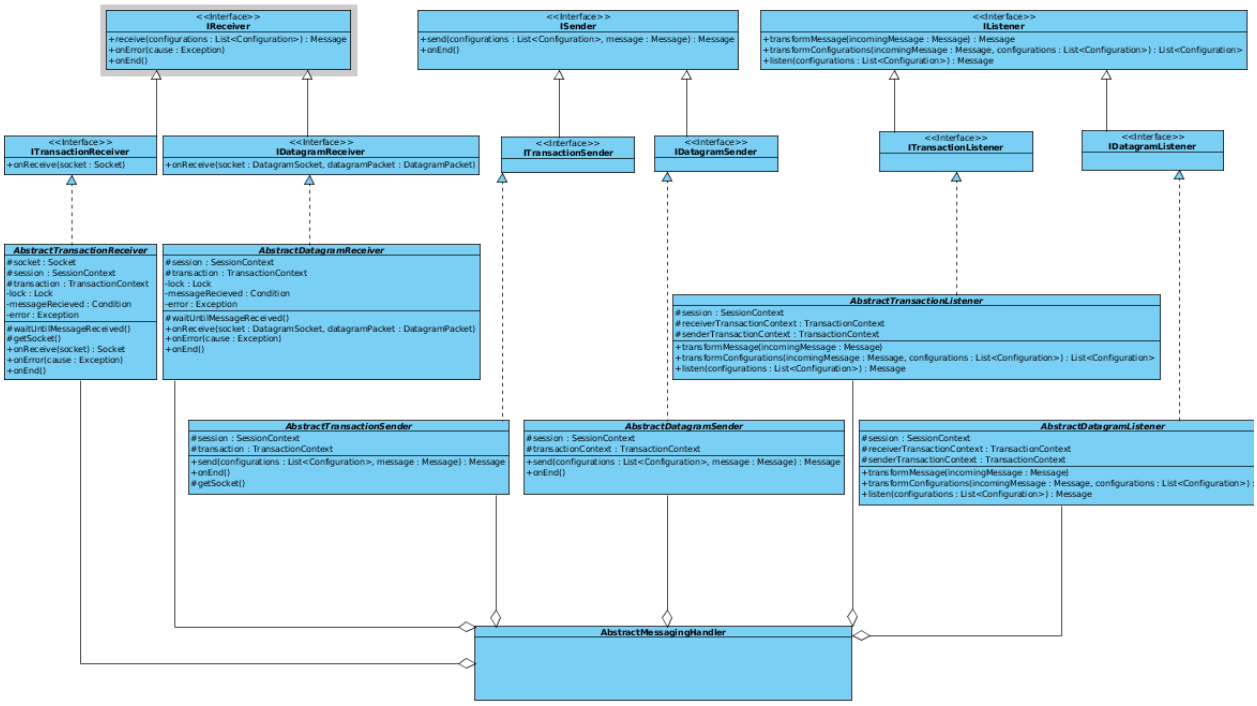

Figure 10-9: Class Diagram of Messaging API Layer

2. Messaging Servers

In order to be able to listen for connections from SUTs, gitb-messaging module utilizes an internal server architecture based on network sockets. There are two main interfaces to support different types of protocols:

- **IMessagingServerWorker:** Provides the following methods to initiate/complete listening for connections at network socket level.
  - **start:** Starts a listener JAVA Thread.
  - **stop:** Closes the listener JAVA Thread.
  - **isActive:** Checks if the listener JAVA Thread is active.
  - **getPort:** Returns the port number listened.
  - **getNetworkSessionManager:** Returns the NetworkSessionManager instance. The NetworkSessionManager class maps IP addresses of SUTs to testing sessions and provides information on these mappings.
- **IMessagingServer:** This interface provide methods for creating messaging servers for different types of protocols and administration of IMessagingServerWorker instances which internally manages connections with JAVA Threads (workers). This interface is implemented by AbstractMessagingServerWorker abstract class.
  - **getActiveWorkers:** Returns all the active workers.
  - **listenNextAvailablePort:** Creates a worker to listen next available port from a range of port numbers.
  - **close:** Closes a connection for specified protocol.

There are two concrete implementations of AbstractMessagingServerWorker class. They are **TCPMessagingServerWorker** and **UDPMessagingServerWorker** for accepting connections by utilizing JAVA Threads over TCP and UDP protocols, respectively. In addition, two concrete implementations of IMessagingServer exist. They are **TCPMessagingServer** and **UDPMessagingServer** for managing lifecycle TCP and UDP workers, respectively. As in the case of messaging API, lifecycle of servers are managed by messaging handlers.

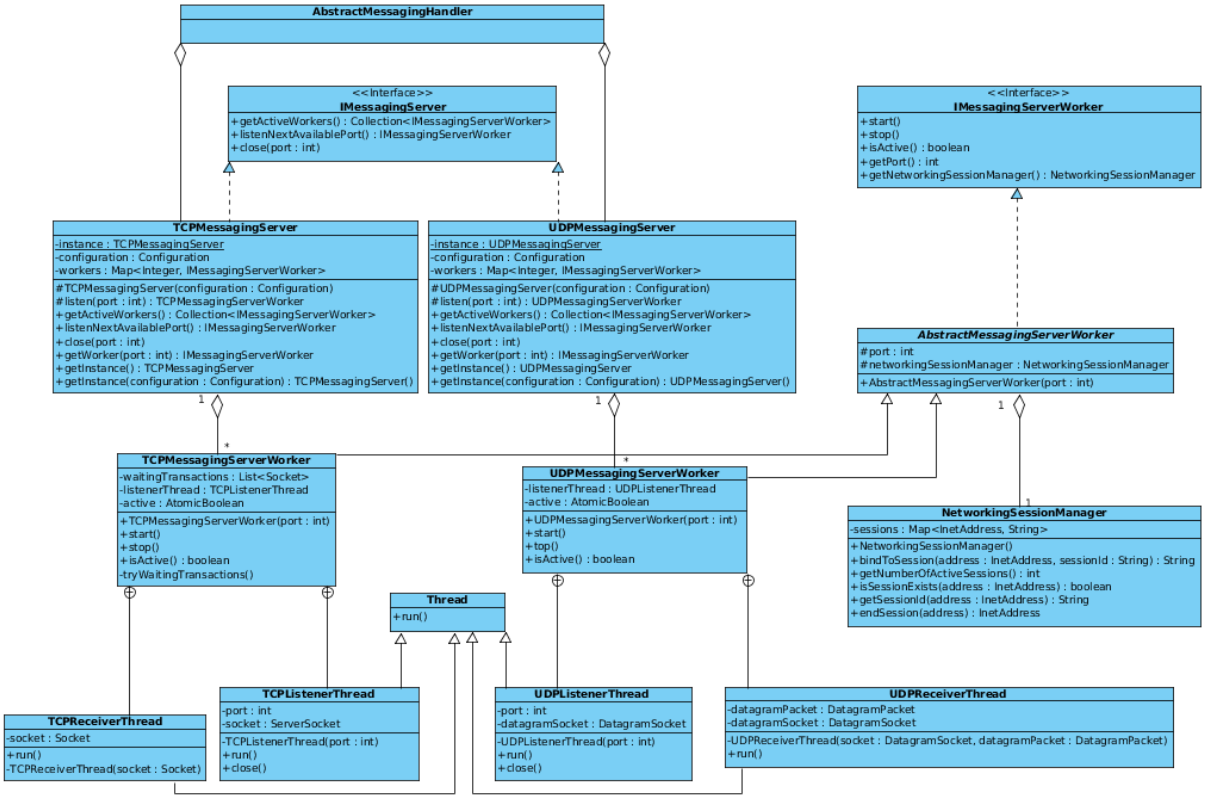

Figure 10-10-10: Class Diagram of Messaging Server Layer

3. Messaging Handlers

Messaging handlers fulfill important tasks including management of the communication sessions and transactions between Testbed and external SUTs by utilizing internal messaging API and servers; administration of receiver, sender, listener and server lifecycle; validation of input/output messages to/from receivers/senders/listeners as well as their configuration parameters according to transport level specifications; and reporting of erroneous and successful communication. **AbstractMessagingHandler** abstract class provides base implementations of the IMessagingHandler interface methods and other utilities. For this purpose, it exposes the aforementioned messaging mechanism to be utilized by other modules. At the moment, the gitb-messaging module provides the following messaging handlers. The hierarchy between them can be seen in the class diagram below.

- **TCPMessagingHandler:** Manages network sockets to enable communications over TCP which is at network layer of the Internet protocol suite. At application level, the following messaging handlers have been implemented:
    - **HttpMessagingHandler:** Handles HTTP communication by utilizing Apache HttpComponents[14] library with the network sockets created at transport layer. The following messaging handlers are derived from HttpMessagingHandler:
        - **HttpsMessagingHandler:** Provides secure HTTP communication on top of SSL/TLS protocols by utilizing X.509 certificates.
            - **PeppolAS2MessagingHandler:** Derived from HttpsMessagingHandler, PeppolAS2MessagingHandler manages communications according to PEPPOL[15] Transport Level Specifications[16] over AS2 protocol which is based on HTTP and S/MIME.

[14] https://hc.apache.org/

[15] http://www.peppol.eu/

- **SoapMessagingHandler:** Allows communications over XML based SOAP protocol.
- **SMPMessagingHandler:** Simulates Service Metadata Publisher architecture[17] specified in PEPPOL project.

- **UDPMessagingHandler:** Manages datagram sockets and packets to enable communications over UDP which is in network layer of the Internet protocol suite. At application level, the following messaging handlers have been implemented:
  - **DNSMessagingHandler:** Handles sending/receiving DNS queries.
    - **SMLMessagingHandler:** Simulates Service Metadata Locator architecture[18] specified in PEPPOL project.

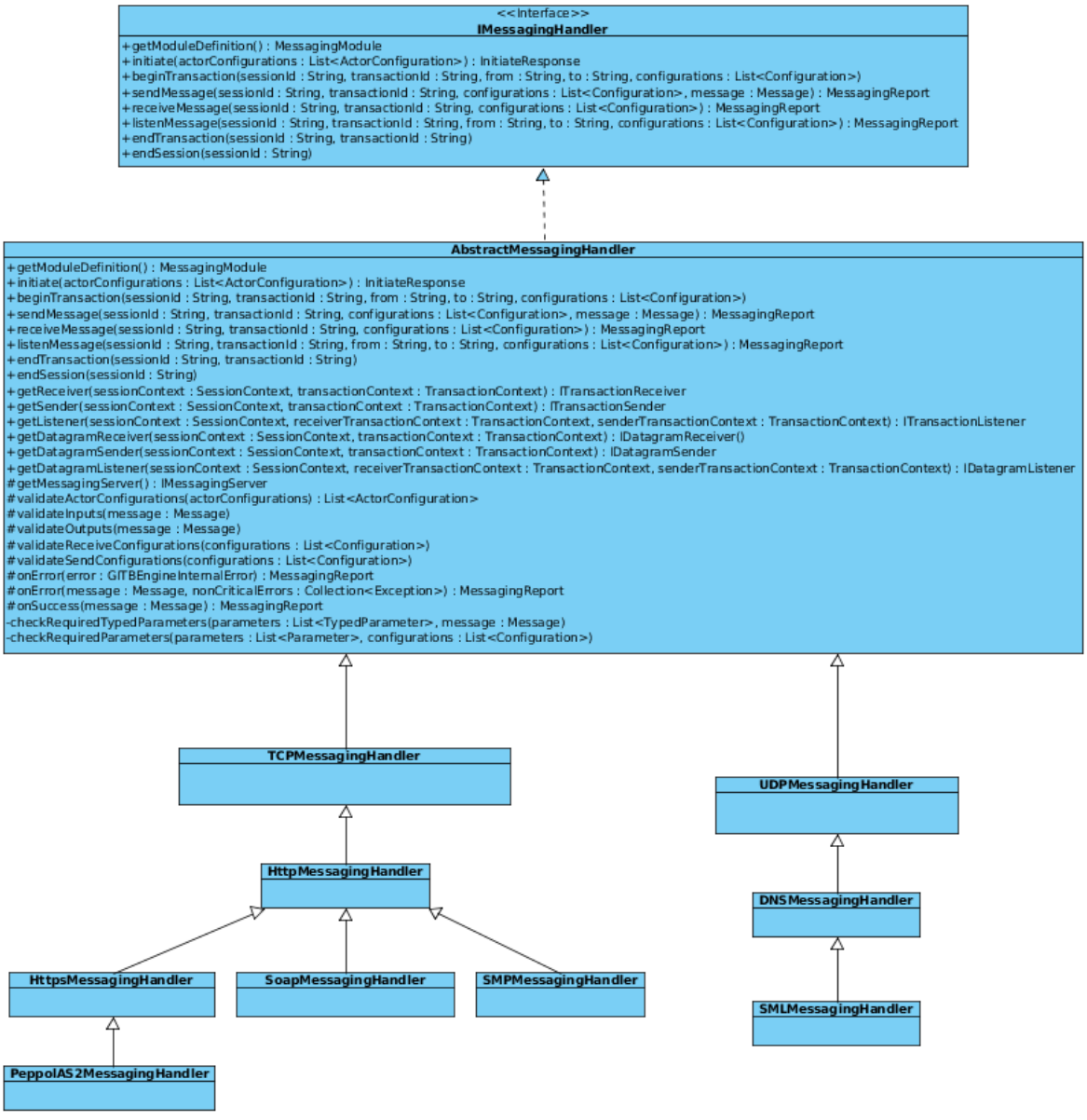

Figure 10-11: Class Diagram of Messaging Handler Layer

[16] https://joinup.ec.europa.eu/svn/peppol/TransportInfrastructure/ICT-Transport-AS2_Service_Specification-2014-01-15.pdf

[17] https://joinup.ec.europa.eu/svn/peppol/PEPPOL_EIA/1-ICT_Architecture/1-ICT-Transport_Infrastructure/13-ICT-Models/ICT-Transport-SMP_Service_Specification-110.pdf

[18] https://joinup.ec.europa.eu/svn/peppol/PEPPOL_EIA/1-ICT_Architecture/1-ICT-Transport_Infrastructure/13-ICT-Models/ICT-Transport-SML_Service_Specification-101.pdf

When a communication is about to be established by messaging handlers, a session between the SUT and Testbed is created by utilizing methods of a singleton, independent SessionManager class. All transactions are realized within a communication session. The SessionManager class also provides a number of methods for messaging handlers to manipulate sessions. The session concept is realized by the SessionContext class which keeps session-related information such as session ID, IP address and port of the SUT, workers handling the communication threads and a list of TransactionContext objects. As its name implies, the transaction concept is realized by TransactionContext classes. They keep transaction-related information such as transaction ID, configuration parameters that messaging handlers demand and, most importantly, objects (messages, intermediate results) that are created during the transaction.

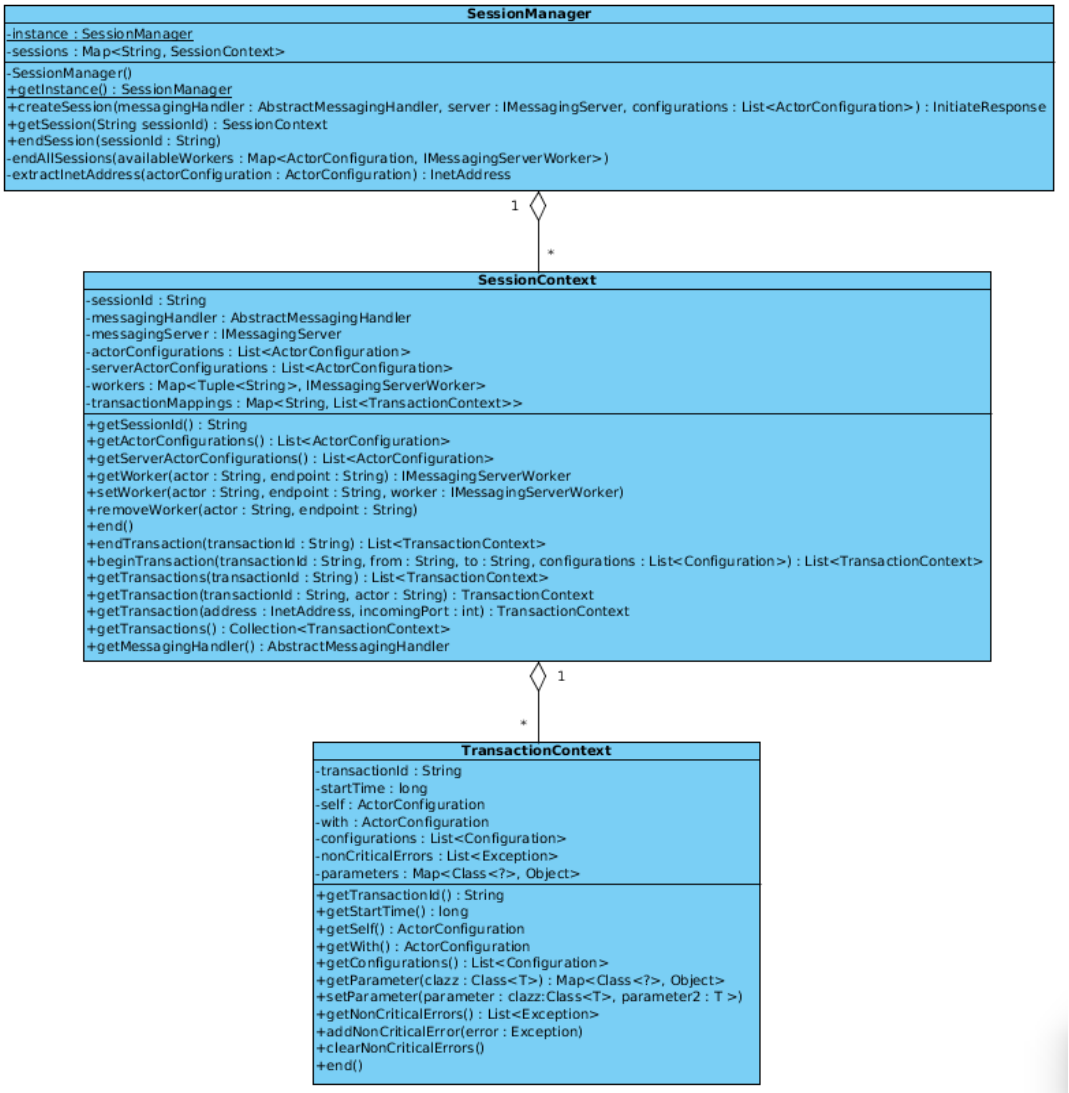

Figure 10-12: Class Diagram for Session Management

As mentioned before, gitb-messaging module currently provides 9 messaging adapters with their module definition information. Each module converts the corresponding XML file containing the module definition into a JAVA object with the help of the XMLUtils class provided by the gitb-lib module. An example module definition can be seen below:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<module xmlns="http://www.gitb.com/core/v1/"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    id="HttpMessaging" uri="urn:com:gitb:messaging:HttpMessaging" xsi:type="MessagingModule">
  <!--Describes the metadata of the validator (e.g. its name, version, description, etc) -->
  <metadata>
    <name>HTTP Messaging</name>
    <version>1.0</version>
  </metadata>
  <!-- Input message fragments to be sent to SUTs for senders -->
  <inputs>
    <param name="http_version" type="string" use="O"/>
    <param name="http_headers" type="map" use="O"/>
    <param name="http_body" type="binary" use="O"/>
  </inputs>
  <!-- Output messages fragments after receiving messages from SUTs for receivers -->
  <outputs>
    <param name="http_method" type="string"/>
    <param name="http_version" type="string"/>
    <param name="http_path" type="string" use="O"/>
    <param name="http_headers" type="map" use="O"/>
    <param name="http_body" type="binary" use="O"/>
  </outputs>
  <!-- Configurations that are expected from SUTs -->
  <actorConfigs>
    <param name="network.host" desc="Hostname/IP address for the actor"/>
    <param name="network.port" desc="Port address for the actor"/>
    <param name="http.uri" use="O" desc="Request URI for Http message">/</param>
  </actorConfigs>
  <!-- Configurations for receive operations for receivers -->
  <receiveConfigs>
    <param name="status.code" use="O" desc="Status code for responses"/>
  </receiveConfigs>
  <!-- Configurations for send operations for senders -->
  <sendConfigs>
    <param name="http.method" desc="Http Method to use"/>
    <param name="http.uri" use="O" desc="Request URI for Http message">/</param>
    <param name="http.uri.extension" use="O" desc="Http URI extension for the address"/>
    <param name="status.code" use="O" desc="Status code for responses"/>
  </sendConfigs>
</module>
```

10.1.2.1.5 Central Processing Module: gitb-engine

gitb-engine is the central processing module of the GITB Testbed component and responsible for Test Case execution. It utilizes the internal API provided by gitb-core and its concrete implementations realized in abovementioned modules. In addition, it processes the requests received by TestbedService, performs the required actions according to the Test Bed Service Specifications explained in section 8.3 and returns the results.

When a request is received by TestbedService, it is delivered to an appropriate manager class to be processed. The TestEngine singleton class is responsible for test execution, whereas the TestCaseManager provides utilities for various Test Case operations. Moreover, when Test Bed users request to execute a Test Case, a session needs to be created for each execution. The SessionManager singleton class manages user sessions and contains all test execution related data within instances of the TestCaseContext class. TestCaseContext stores all intermediate results, SUT configurations, messaging context, etc. during a test session.

Intermediate results must be treated specially, since these results are stored in variables and may be referenced from other test steps. For this purpose, intermediate results or variables, in short, are stored in a class called TestCaseScope and they are accessed by their variable names. Variables in TestCaseScope are globally accessible from each step of the test execution. However, variables created in a Scriptlet are local variables and cannot be reached outside of it.

Variables defined in a Test Case can be referenced from XPath expressions in the form of $variable_name. In that case, its value must be resolved in order to evaluate the XPath expression and proceed to the next steps. For this purpose, gitb-engine module provides a class **VariableResolver** which implements XPathVariableResolver JAVA XPath API interface whose **resolveVariable** method is called automatically when a variable is referenced. The VariableResolver class has access to TestCaseScope and can retrieve the value of a variable by its name. If the referenced variable is not found in TestCaseScope, then a null value is returned.

The same mechanism applies to resolve user defined, extensive functions referenced from an XPath expression. In order to resolve user defined functions, gitb-engine module provides a class **FunctionResolver** which implements XPathFunctionResovler JAVA XPath API interface whose **evaluate** method is called automatically when a function that is not a default XPath function, is referenced. Then, the FunctionResovler scans all the IFunctionRegistry implementations through the ModuleManager, invokes the first available function having the same signature as the referenced function and returns the value returned from the function. If no function is found among all IFunctionRegistry implementers, a null value is returned.

As mentioned before, an extensive function is invoked by using *prefix:localname()* pattern. For a successful user defined function invocation, the prefix must be resolved, as well, to a valid namespace URI defined in tdl:Namespace element. gitb-engine module provides a class **NamespaceContext** which implements javax.xml.namespace.NamespaceContext interface, for the resolution of prefixes. The NamespaceContext class has access to TestCaseContext class through TestCaseScope and can retrieve corresponding namespace URI's for each prefix and vice versa. If the prefix is valid, then FunctionResolver class tries to resolve the function and invoke it.

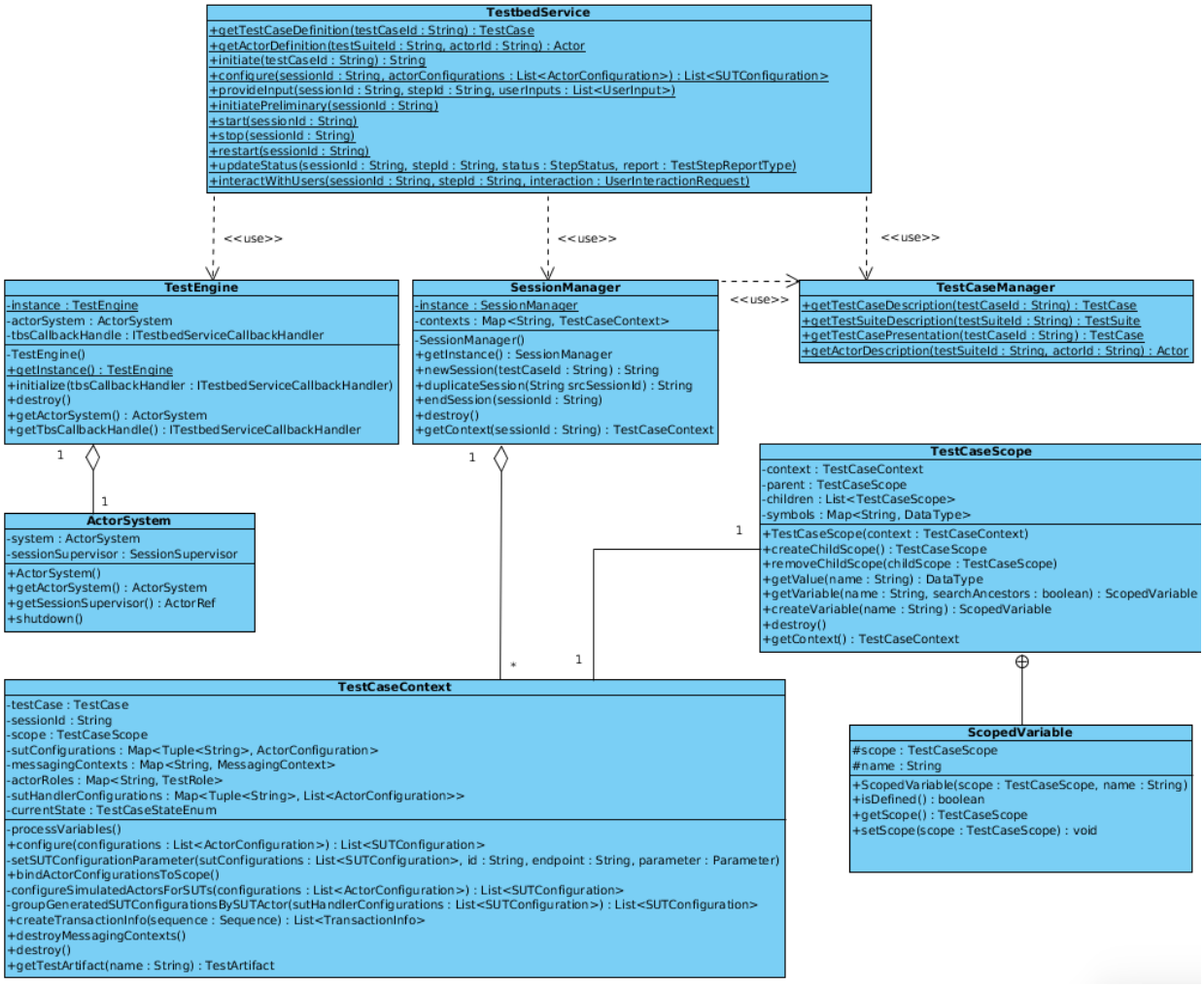

Figure 10-13: Class Diagram of Test Engine

In order to enable concurrent and asynchronous execution of several test cases at the same time in parallel, the Akka framework[19] is used. Akka is a framework that enables development of distributed and concurrent applications. The philosophy underlying Akka embraces the actor model which is a mathematical model of concurrent computation that treats actors as the universal primitives of concurrent computation. Akka concurrency is asynchronous and achieved by actors sending messages to each other. In other words, Akka actors are lightweight concurrent entities and process messages in an asynchronous way by using an event-driven receive loop. Actors are created and bound within an ActorSystem and each of them is a part of an actor-hierarchy.

TestEngine singleton class has an ActorSystem object which is responsible for creating actors. There are two types of actors used within Testbed execution: Session and Processing Actors. When a user demands to execute a Test Case, a SessionActor is created to manage the test execution with commands such as, start, stop, configure etc. After gitb-engine retrieves the Test Case with one of the ITestCaseRepository implementers and receives start command, it creates a TestCaseProcessActor which takes over the test execution. TestCaseProcessActor then parses the Test Case and employs appropriate Test Step Processor Actors to execute each test step: AssignStepProcessorActor to process Assign test step, ReceiveStepProcessorActor to process Receive step and so on. In other words, there is a corresponding processor actor for each test step identified in section 9.3.6. After executing each of the test steps, TestCaseProcessorActors notifies the corresponding SessionActor. The latter notifies the TestbedService that the test execution is completed.

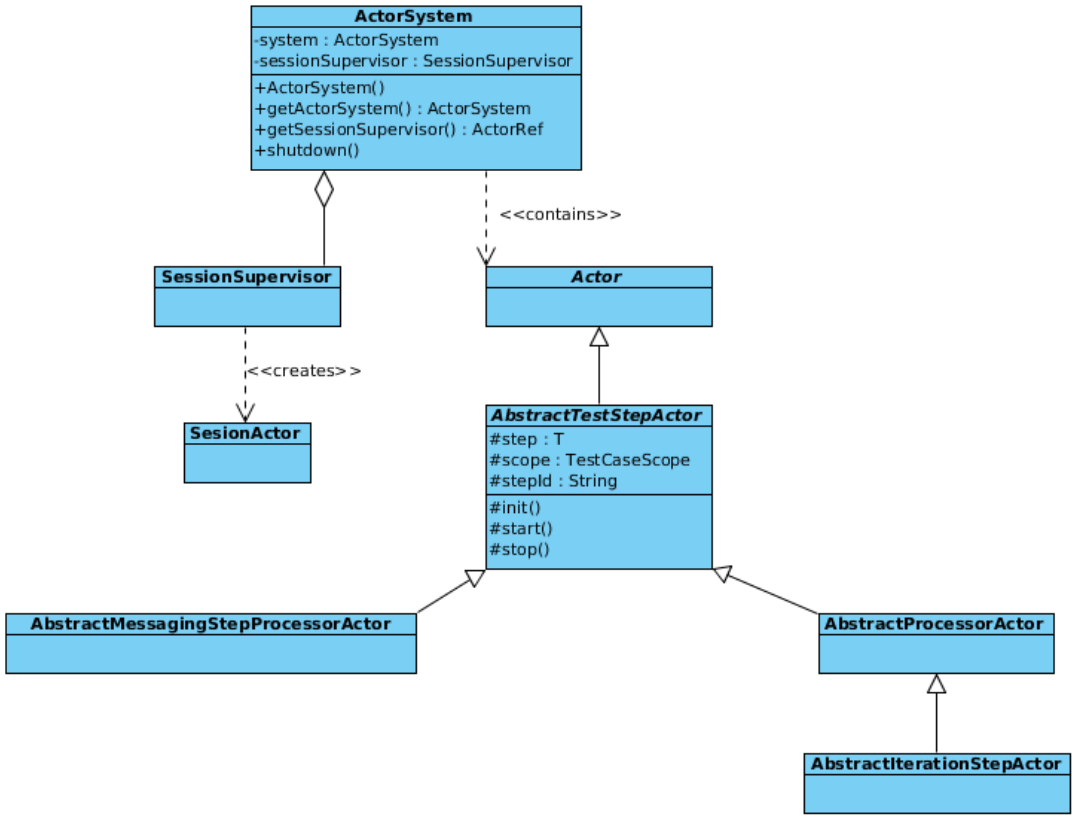

Figure 10-14: Class Diagram of the Actor System Based on Akka

During a test execution, it is natural that run-time exceptions can be thrown as it is very likely that gitb-engine may have to process erroneous input from SUTs. For instance, a validator may come across an invalid

[19] http://akka.io

document or a messaging handler may receive a message that does not conform to transport level specifications. These exceptions are caught during test execution and sent to gitb-execution-interface as a test step report indicating the cause of failure. SUT administrators can review the report and fix problematic parts of their systems.

In order to handle exceptions in each module, a common exception model is created and provided by gitb-core module. The abstract **GITBEngineRuntimeException** class is the base of the common exception model. Its concrete implementation, **GITBEngineInternalError**, provides the necessary information regarding the cause of the exception thrown during test execution. To implement such behaviour, when an exception (of any run-time or compile-time exception type) is caught, it is wrapped within GITBEngineInternalError and rethrown so that it is caught within gitb-engine module. After gitb-engine catches the exception of the type GITBEngineInternalError, it creates a report from it and sends the report to gitb-execution-interface, as mentioned before.

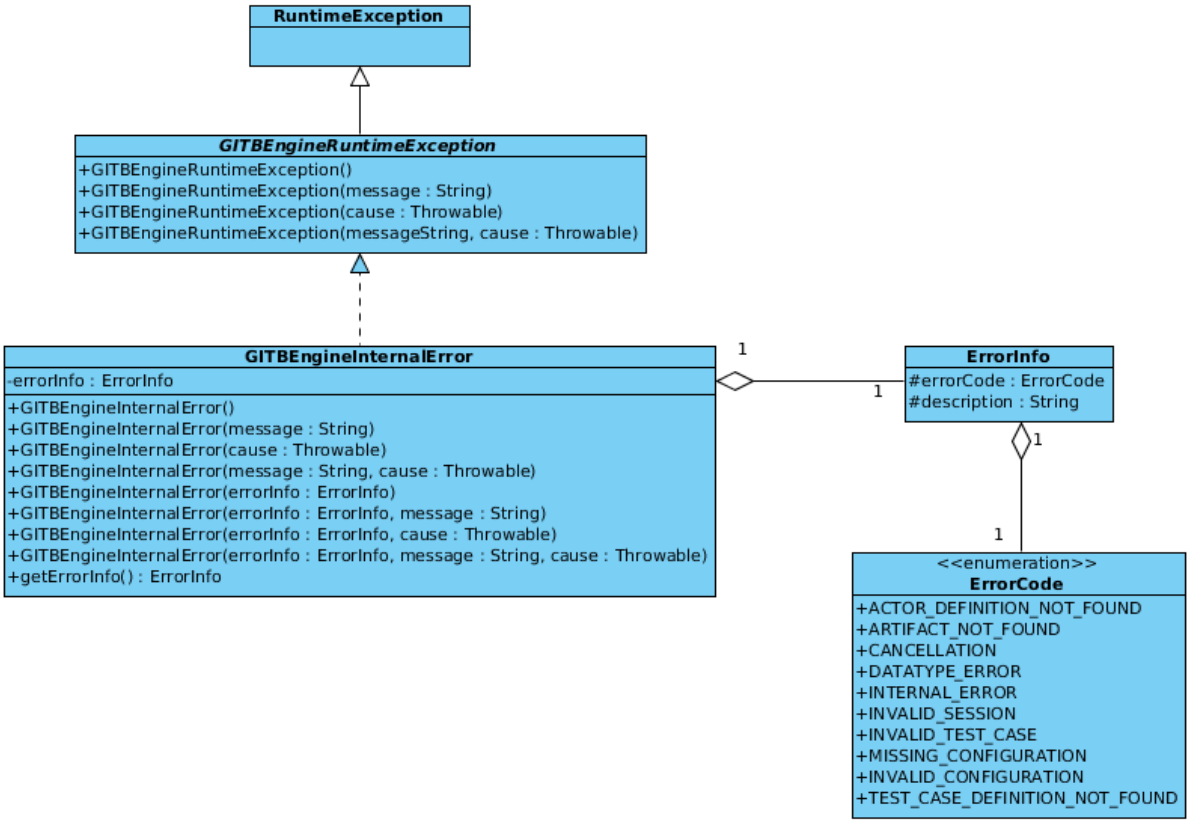

Figure 10-15: GITB Testbed Exception Model

10.1.2.1.6 GITB Test Bed Service: gitb-testbed-service

The gitb-testbed-service module provides the implementation (TestbedServiceImpl class) for Testbed Service Specifications. It packages every JAR file for the GITB Testbed component module into a WAR (Web Application Archive) which can then be deployed to an application server. It also contains the necessary files (deployment description - web.xml, JAX-WS RI deployment descriptor - sun-jaxws.xml, Jetty deployment description - jetty-context.xml) to deploy the WAR file as a web application. The deployment can be either manually by copying WAR file into appropriate application server folders or automatically by using Maven plugins. The current GITB Testbed implementation uses maven-jetty-plugin to deploy the WAR file onto an Jetty Application server[20].

[20] http://www.eclipse.org/jetty/

10.1.2.1.7 Integration of Remote Modules: gitb-remote-modules

gitb-remote-modules module enables integration of external validation and messaging services and implements the IModuleLoader interface methods. To integrate a remote module, a module definition XML file must be created and put into the resources folder. Additionally, a separate Maven module with Validation Service or Messaging Service and client implementations for the external service must be developed. In this way, at the start of GITB Testbed component, the integrated modules are loaded by ModuleManager class in gitb-core module and will be ready to be used. Currently, Validex[21] which is an online XML message validation service has been integrated with GITB Testbed.

10.1.3 GITB Execution Interface

The GITB Execution Interface is designed to be the management interface of the GITB Testbed component to be exposed to outer world. Users of the GITB Execution Interface can perform their requests through a Web GUI to a REST server. The latter then delivers these requests to GITB Testbed component through TestbedService. Thus, the GITB Execution Interface is designed to manage the test executions within the GITB Testbed component on a graphical user interface through REST Services. In this section, the design and implementation details of the GITB Execution Interface are presented.

At software level for the client side implementation, there are numerous JavaScript frameworks providing different functionalities at different level. For the GITB Execution Interface, the Play Framework is selected. The latter is an open source Web application framework written in Scala and Java and follows the MVC (Model View Controller) architecture. The MVC architecture provides rich API for models and views, and integrates with the REST services provided by the framework. Because of the model-view separation as well as JSON based REST support, the Play Framework fits perfect for the requirements of GITB Execution Interface.

10.1.3.1 How to Use the GITB POC Interface

Figure 10-1610-16 presents the main screen of the GITB Execution Interface Web GUI. On the upper right corner, information about the authenticated user is shown along with a dropdown menu to manage settings and invalidate the session information. The **Tutorial** button navigates users to a page with a viewlet[22] on how to use GITB Execution Interface. Moreover, **Systems** returns the users to the main screen.

Figure 10-16: GITB Execution Interface Main Screen

[21] https://validex.net/

[22] Available on https://www.youtube.com/watch?v=4K7eEvUS1UA

When users login to the system, the main screen retrieves all the available SUTs and displays them. All users have to register their systems as SUTs in order to be able execute Test Cases for them. This is achieved by clicking on the **New System** button. A popup window is then displayed asking some information regarding the SUT as illustrated in Figure 10-1710-17.

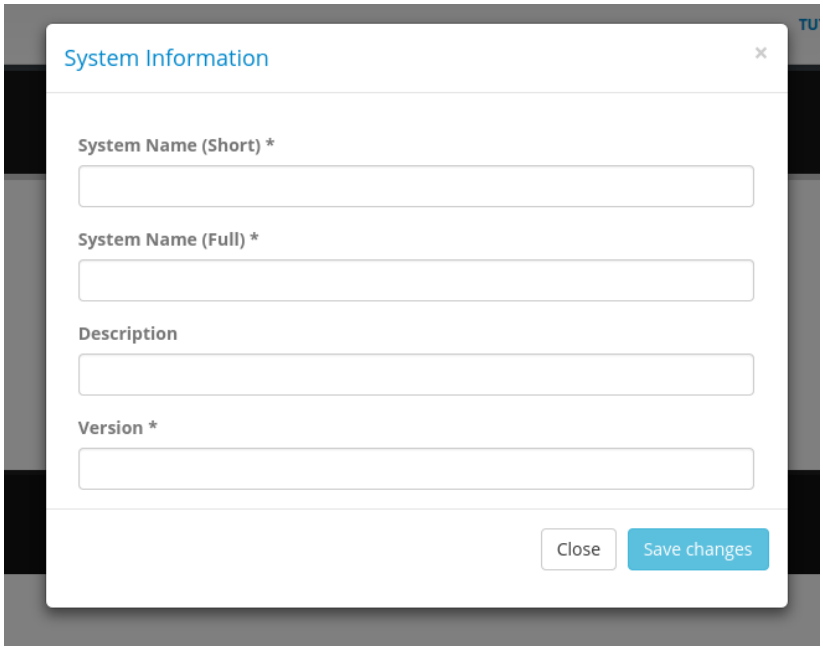

Figure 10-17: Screen for Adding a New System

After entering the required information, SUT is registered, but not ready to be tested yet. In order to be able initiate a test for a SUT, a conformance statement must be defined, after clicking on SUT name. By defining a conformance statement, users tell the GITB Execution Interface that the SUT conforms to a specific eBusiness standard or specification. They get a number of Test Cases to prove their conformance.

Creating a conformance statement is a four-step process. In the first step, the appropriate eBusiness domain selected. All available specifications of this domain are then retrieved and the user is asked to select the relevant specification in the second step. In the third step, the actors implemented by the SUT are selected. Finally, if the selected actors define any optional parameters, they are selected in the fourth step.

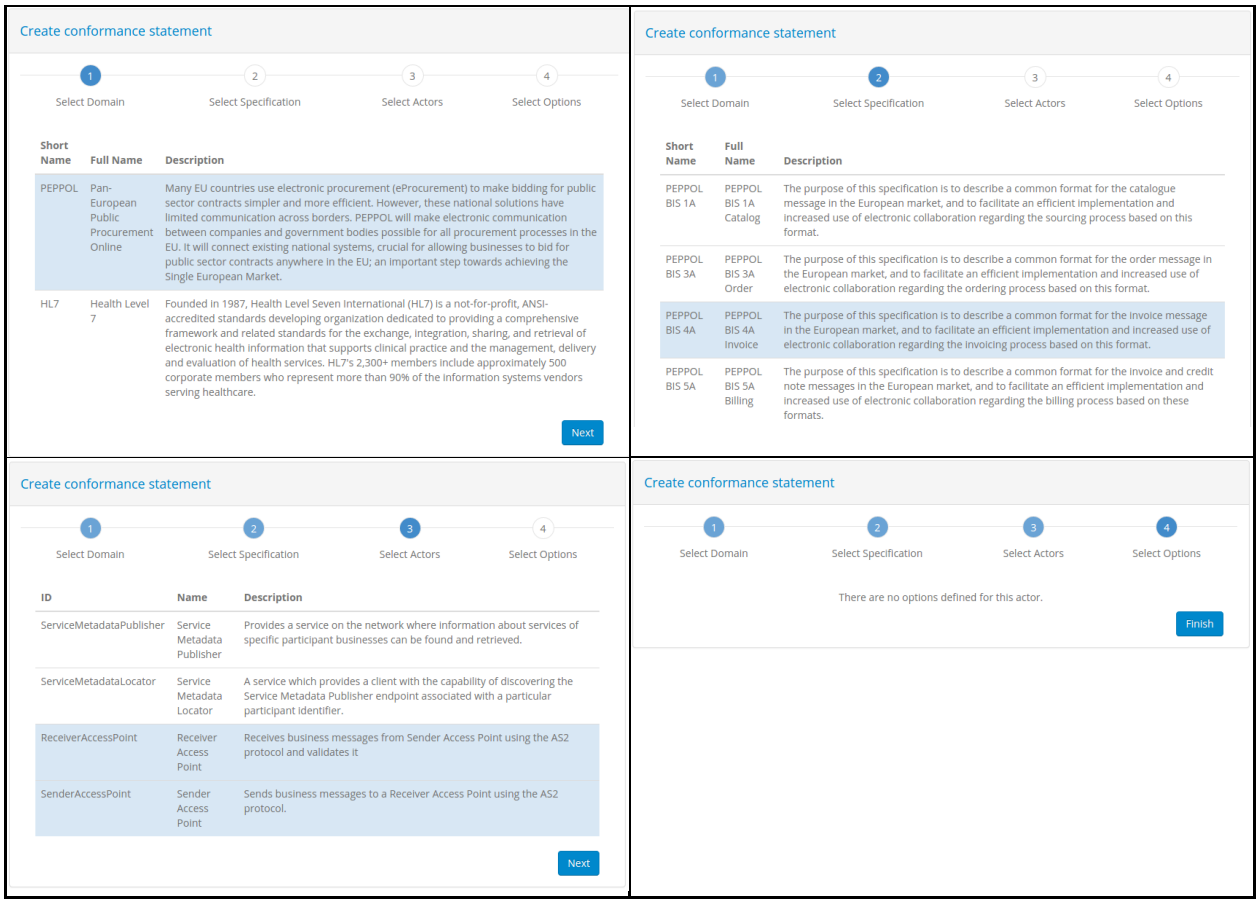

Figure 10-18: Steps for Defining a Conformance Statement

After a conformance statement is defined, it should be selected to continue with the Conformance Statement Detail page. Each SUT actor defines one or more endpoints to be reached over the network. For each endpoint, a number of configuration parameters related to the SUT itself or its endpoint have to be provided to GITB Testbed, so that it can use these parameters during test execution. These configuration parameters can be simple string values (i.e IP address, port number, etc) as well as files (public keys, processable text files, etc). Finally, all Test Cases that are related to the selected actors are listed at the bottom of this page. In order to prove the claim of conformance, all these Test Cases have to be executed and passed.

Conformance Statement Details

Delete

Actor Details

ID: SenderAccessPoint

Name: Sender Access Point

Domain Details

Name: Pan-European Public Procurement Online

Description: Many EU countries use electronic procurement (eProcurement) to make bidding for public sector contracts simpler and more efficient. However, these national solutions have limited communication across borders. PEPPOL will make electronic communication between companies and government bodies possible for all procurement processes in the EU. It will connect existing national systems, crucial for allowing businesses to bid for public sector contracts anywhere in the EU; an important step towards achieving the Single European Market.

Endpoints

Name: as2
Description:

Endpoint parameters

Name	Usage	Type	Configured
network.host	R	SIMPLE	✔
network.port	R	SIMPLE	✔
public.key	R	BINARY	✔
participant.identifier	R	SIMPLE	✔

Conformance Tests

Short Name	Full Name	Description	Last Result
PEPPOL-ReceiverAccessPoint-Invoice	PEPPOL-ReceiverAccessPoint-Invoice	The objective of this Test Scenario is to ensure the Receiver Access Point (the System Under Test) can receive a conformant PEPPOL BIS 4A electronic invoice from a Sender Access Point using the AS2 protocol.	SUCCESS
PEPPOL-SenderAccessPoint-Invoice-BusDox	PEPPOL-SenderAccessPoint-Invoice-BusDox	The objective of this Test Scenario is to ensure the Sender Access Point (the System Under Test) is capable of querying both SML and SMP as well as submitting a conformant PEPPOL BIS 4A electronic invoice to a Receiver Access Point using the AS2 protocol. Then submitted document is validated by UBL 2.1 schema and PEPPOL Schematron rules.	SUCCESS
PEPPOL-SenderAccessPoint-Invoice-Validation	PEPPOL-SenderAccessPoint-Invoice-Validation	The objective of this Test Scenario is to ensure the Sender Access Point (the System Under Test) can submit a conformant PEPPOL BIS 4A electronic invoice to a Receiver Access Point using the AS2 protocol. Then submitted document is validated by UBL 2.1 schema and PEPPOL Schematron rules.	UNDEFINED
PEPPOL-SenderAccessPoint-Invoice-Validex	PEPPOL-SenderAccessPoint-Invoice-Validex	The objective of this Test Scenario is to ensure the Sender Access Point (the System Under Test) can submit a conformant PEPPOL BIS 4A electronic invoice to a Receiver Access Point using the AS2 protocol. Then submitted document is validated by Validex.	SUCCESS
PEPPOL-SenderAccessPoint-Invoice-BusDox-Validex	PEPPOL-SenderAccessPoint-Invoice-BusDox-Validex	The objective of this Test Scenario is to ensure the Sender Access Point (the System Under Test) is capable of querying both SML and SMP as well as submitting a conformant PEPPOL BIS 4A electronic invoice to a Receiver Access Point using the AS2 protocol. Then submitted document is validated by Validex.	SUCCESS

Figure 10-19: Conformance Statement Details Page

After selecting a Test Case, the user is navigated to the test execution page. Test execution is a three-step process. In the first step, all the SUT configurations are checked against missing inputs. If there are any missing configurations, the Web GUI does not let the user proceed to the next steps. In the second step, a testing session within GITB Testbed is created for the user. After that, configuration parameters of each actor that is simulated by the GITB Testbed component are listed. At this point, SUT admins have to configure their SUTs according to these configurations. Finally, the third step is where the actual test execution takes place. Here all the actors, the messaging choreography between them, the validation steps etc. are displayed by means of sequence diagram elements. The actor role which the SUT plays is indicated with (SUT). Actors, that are simulated by the GITB Testbed component and the GITB Testbed itself are indicated with (SIMULATED) and Test Engine, respectively. Test execution starts by clicking the **Start** button.

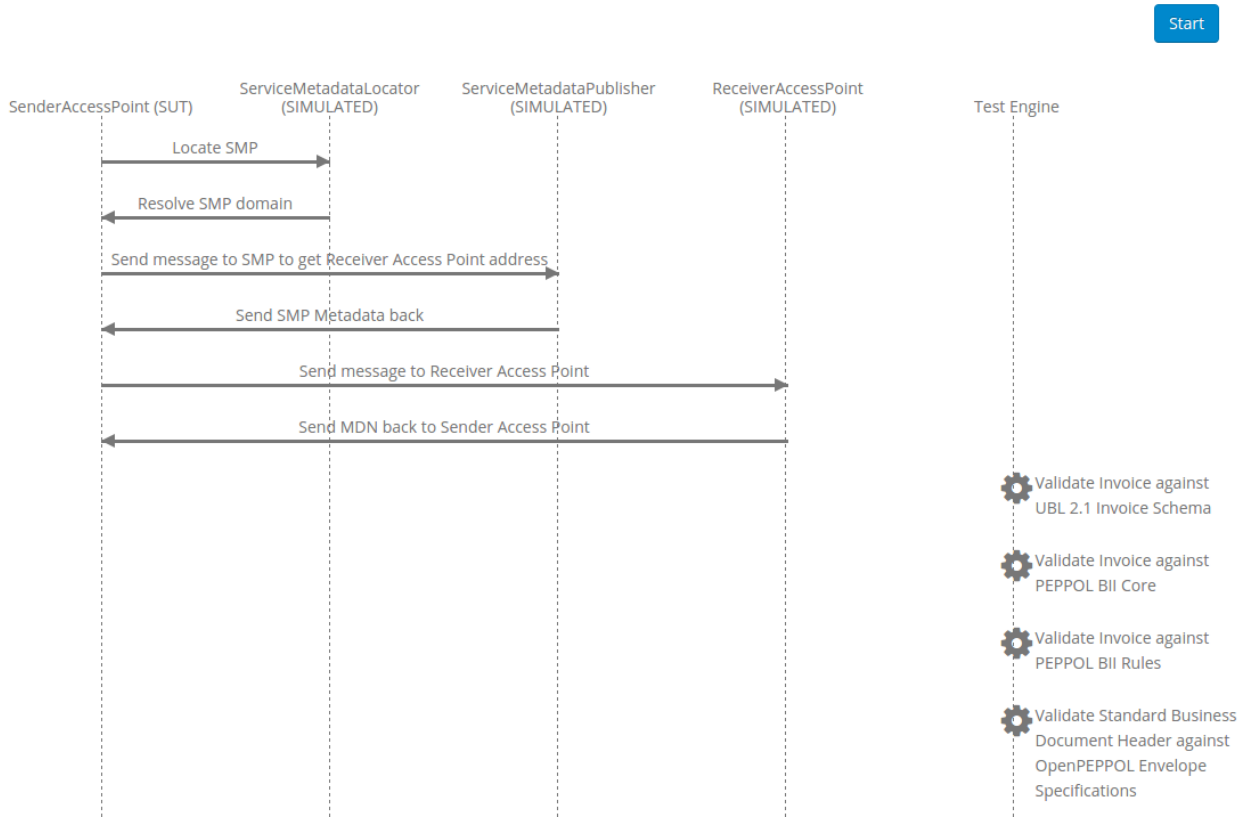

Figure 10-20: Test Execution Interface

After a test case is executed, the Web GUI indicates success or failure of each step with green and red arrows, respectively. Moreover, GITB Testbed creates reports summarizing the results of each test step as well as their execution time, message content and headers (if it is a messaging step), validated and validator document content (if it is a validation step) and so on.

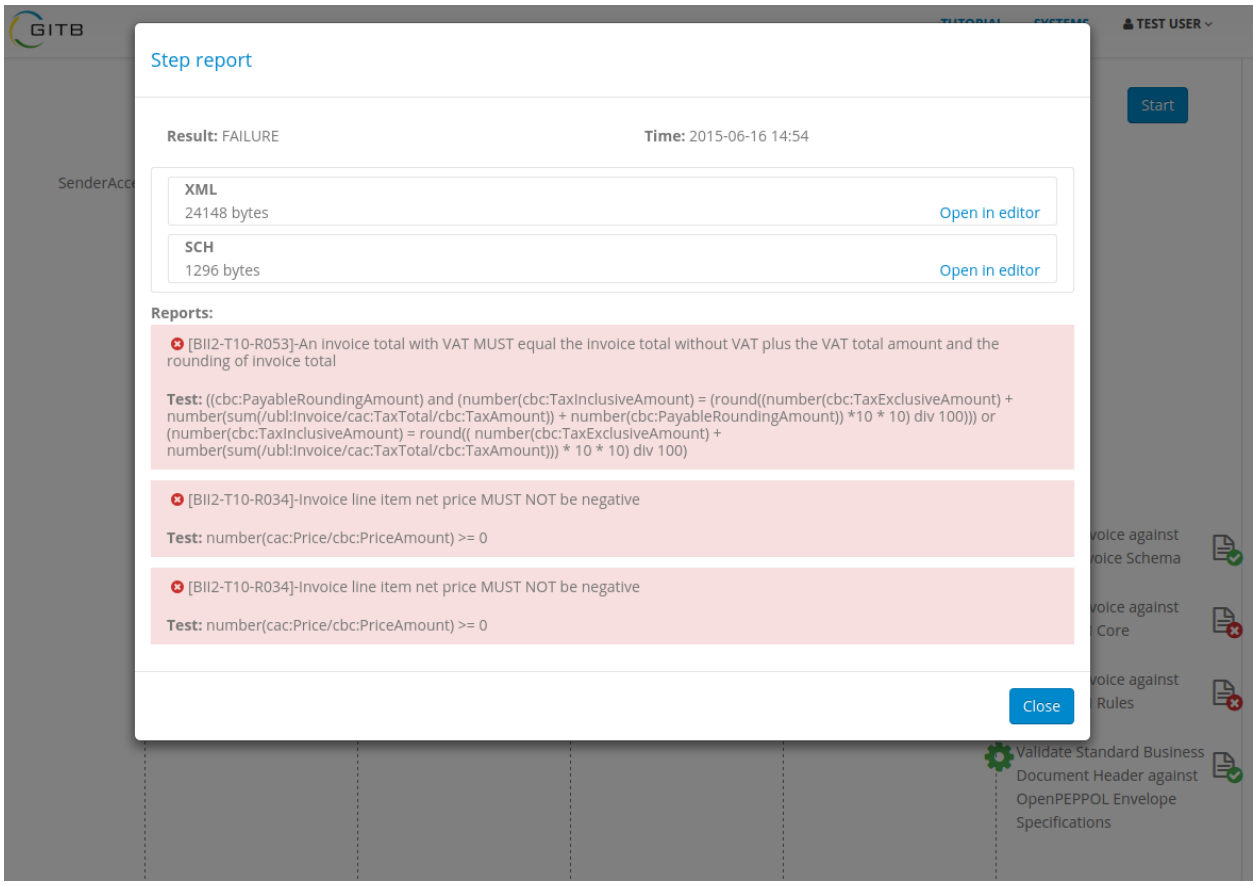

Figure 10-21: Test Report Screen Indicating the Errors

The GITB Execution Interface also provides an Admin Panel for system administrators. Through this panel, system admins add new domains, specifications and deploy Test Suites.

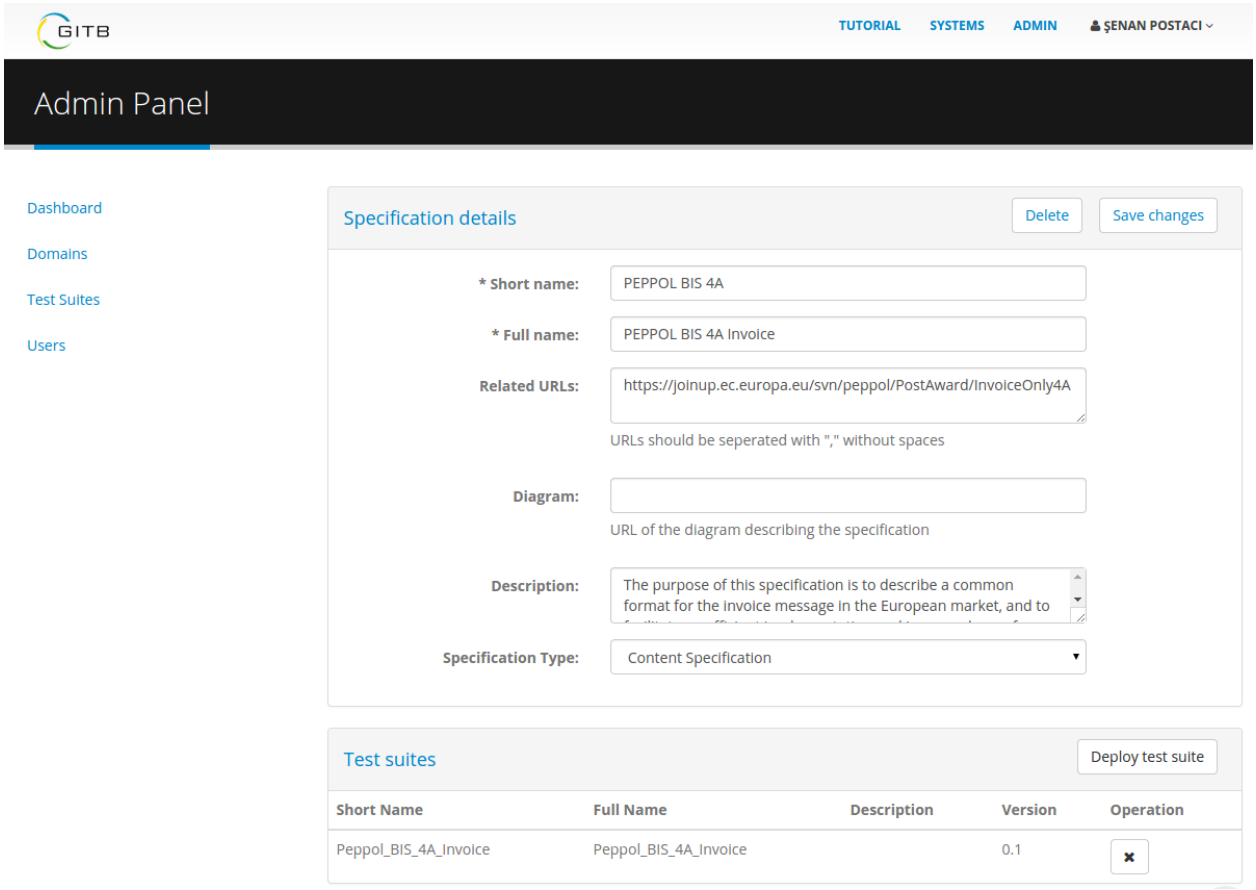

Figure 10-22: GITB Execution Interface Admin Panel

10.1.3.2 REST API

The GITB Execution Interface's REST API is responsible for managing the functionalities explained in previous section and exposing them to the outer-world as well as invoking the TestbedService methods of GITB Testbed. Furthermore, the API enables persistence of user and Test Case information within a MySQL database and provides access to them for authorized users. The GITB Execution Interface's REST API provides following services:

- **AccountService:** Provides methods for user and vendor related operations such as registration, user/vendor profile retrieval, etc.
- **AuthenticationService:** Provides methods for user sessions by providing them access tokens. Secure services can only be invoked by access tokens.
- **ConformanceService:** Provides methods invoked when defining conformance statement for SUTs
- **ReportService:** Enables persistence of test results and reports retrieved from GITB Testbed and provides them to users on demand.
- **RepositoryService:** Provides access to Test Cases and resources on demand. RemoteTestCaseRepository retrieves test artifacts from this service.
- **SystemService:** Provides methods for creating and managing SUTs.
- **TestService:** Provides methods to enable access to GITB Testbed through TestbedService.
- **TestSuiteService:** Provides methods for deployment/undeployment of test suites
- **WebSocketService:** Delivers results of execution of test steps to Web GUI through WebSockets.

10.2 Case Studies with POC Test Bed

10.2.1 UBL Use Case - Conformance Tests for PEPPOL BIS4A Invoice Only Specification

The GITB PoC Test Bed implements a Test Scenario for the PEPPOL BIS4A Invoice Only Profile, which is a real-world eBusiness specification in the public procurement area (see section 15). A Test Suite with a set of Test Cases have been developed for this purpose. The Test Suite can be found in test-suites folder which comes along with the source code. More details regarding Test Suite and Test Cases can be found in the following sections.

10.2.1.1 Test Suite Definition

Basically, a Test Suite defines a number of Test Cases and the actors of the specification that take part in a Test Case. Furthermore, actors may define endpoints which denote the network protocol that the actor is accessible from, with a set of configuration parameters (e.g. network host, port, etc). As an example in the Test Suite below, all the Test Cases and actors of this suite are listed. It should be noted that, the Test Suite does not specify any relationships among the actors, it rather defines the endpoints and their configurations.

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite xmlns="http://www.gitb.com/tdl/v1/" xmlns:gitb="http://www.gitb.com/core/v1/">
  <!--Describes the metadata of the test suite (e.g. its name, version, author(s), description, etc) -->
 <metadata>
   <gitb:name>Peppol_BIS_4A_Invoice</gitb:name>
   <gitb:version>0.1</gitb:version>
   </metadata>

  <!-- All the actors that takes part in the business processes in the specification which are related with the test
scenarios in this test suite  -->
 <actors>
    <gitb:actor id="SenderAccessPoint">
      <gitb:name>Sender Access Point</gitb:name>
      <gitb:desc>Sends business messages to a Receiver Access Point using the AS2 protocol.</gitb:desc>

      <!-- List of configurations for enabling communication with this endpoint -->
      <gitb:endpoint name="as2">
        <gitb:config name="network.host" />
        <gitb:config name="network.port" />
        <gitb:config name="public.key" kind="BINARY"/>
       <gitb:config name="participant.identifier"/>
      </gitb:endpoint>
    </gitb:actor>
    <gitb:actor id="ReceiverAccessPoint">
      <gitb:name>Receiver Access Point</gitb:name>
      <gitb:desc>Receives business messages from Sender Access Point using the AS2 protocol and validates
it</gitb:desc>
      <gitb:endpoint name="as2">
        <gitb:config name="network.host" />
        <gitb:config name="network.port" />
        <gitb:config name="http.uri" use="O"/>
        <gitb:config name="public.key" kind="BINARY"/>
       <gitb:config name="participant.identifier"/>
      </gitb:endpoint>
    </gitb:actor>
    <gitb:actor id="ServiceMetadataLocator">
      <gitb:name>Service Metadata Locator</gitb:name>
      <gitb:desc>A service which provides a client with the capability of discovering the Service Metadata Publisher
endpoint associated with a particular participant identifier.</gitb:desc>
      <gitb:endpoint name="http">
        <gitb:config name="network.host" />
        <gitb:config name="network.port" />
      </gitb:endpoint>
    </gitb:actor>
```

```
    <gitb:actor id="ServiceMetadataPublisher">
      <gitb:name>Service Metadata Publisher</gitb:name>
      <gitb:desc>Provides a service on the network where information about services of specific participant
businesses can be found and retrieved.</gitb:desc>
      <gitb:endpoint name="http">
        <gitb:config name="network.host" />
        <gitb:config name="network.port" />
      </gitb:endpoint>
    </gitb:actor>
  </actors>

  <!-- References to test cases of this test suite-->
  <testcase id="PEPPOL-Interoperability-Invoice" />
  <testcase id="PEPPOL-ReceiverAccessPoint-Invoice" />
  <testcase id="PEPPOL-SenderAccessPoint-Invoice-BusDox" />
  <testcase id="PEPPOL-SenderAccessPoint-Invoice-BusDox-Validex"/>
  <testcase id="PEPPOL-SenderAccessPoint-Invoice-Validation" />
  <testcase id="PEPPOL-SenderAccessPoint-Invoice-Validex" />
</testsuite>
```

10.2.1.2 Development of the Necessary Messaging Handlers

The gitb-messaging module provides a set of widely used network protocols (e.g. TCP, HTTP(S), SOAP). The messaging architecture behind the gitb-messaging module allows extending these protocols in order to develop more complex ones. For instance, three additional messaging adapters have been developed to implement this case study. The AS2 adapter is built by extending the HTTPS adapter, Service Metadata Publisher (SMP) is developed on the HTTP adapter and Service Metadata Locator (SML) is implemented on top of the DNS adapter. Each new messaging adapter inherits the capabilities of the base adapter; therefore developing new adapters is basically adding new rules or abilities on already existing protocol implementations.

10.2.1.3 Definition of Test Artifacts

Test Cases are very likely to require one or more external resources such as validation schemas or message templates, during test execution. Therefore, required Test Artifacts must be included within a Test Case and be referenced with their relative paths to the Test Case. After that, their content is retrieved by the configured Test Case repository adapter and utilized during the test execution.

A Test Case defines all the Testing Resources imports, namespace and variable declarations, target actors of specification with their roles and test steps to execute. Target actors must be selected from the Test Suite and their role must be specified as either **SUT** (if the role will be played by the real SUT) or **SIMULATED** (if the actor will be simulated by GITB Testbed). If there will be any communication between actors, the messaging choreography between them must be defined by a number of transactions (see the example below). More information can be found in the Test Case below.

```
<?xml version="1.0" encoding="UTF-8"?>
<testcase id="PEPPOL-SenderAccessPoint-Invoice-Definition of The Test Case”>

BusDox" xmlns="http://www.gitb.com/tdl/v1/" xmlns:gitb="http://www.gitb.com/core/v1/">
  <!--Describes the metadata of the test case (e.g. its name, version, authors, description, etc) -->
  <metadata>
    <gitb:name>PEPPOL-SenderAccessPoint-Invoice-BusDox</gitb:name>
    <gitb:type>CONFORMANCE</gitb:type>
    <gitb:version>0.1</gitb:version>
    <gitb:description>The objective of this Test Scenario is to ensure the Sender Access Point (the System Under
Test) is capable of querying both SML and SMP as well as submitting a conformant PEPPOL BIS 4A electronic invoice
to a Receiver Access Point using the AS2 protocol. Then submitted document is validated by UBL 2.1 schema and
PEPPOL Schematron rules.
    </gitb:description>
```

```
  </metadata>

  <!-- Namespace declarations used for expressions when referring the element or attribute names in a document or message model  -->
  <namespaces>
  </namespaces>

  <!-- Provides the details about the test artifacts for test engine so that it can remotely access and use them during the test execution. -->
  <imports>
    <artifact type="schema" encoding="UTF-8" name="UBL_Invoice_Schema_File">Peppol_BIS_4A_Invoice/artifacts/UBL/maindoc/UBL-Invoice-2.1.xsd</artifact>
    <artifact type="schema" encoding="UTF-8" name="PEPPOL_BII_CORE_Invoice_Schematron_File">Peppol_BIS_4A_Invoice/artifacts/PEPPOL/BII CORE/BIICORE-UBL-T10-V1.0.sch</artifact>
    <artifact type="schema" encoding="UTF-8" name="PEPPOL_BII_RULES_Invoice_Schematron_File">Peppol_BIS_4A_Invoice/artifacts/PEPPOL/BII RULES/BIIRULES-UBL-T10.sch</artifact>
    <artifact type="schema" encoding="UTF-8" name="SBDH_Schematron_File">Peppol_BIS_4A_Invoice/artifacts/PEPPOL/SBDH.sch</artifact>
    <artifact type="object" encoding="UTF-8" name="SMP_Metadata_Template">Peppol_BIS_4A_Invoice/artifacts/PEPPOL/peppol-smp-metadata-template.xml</artifact>
  </imports>

  <!-- Declare the actors who are participating in this test case. The System-Under-Test role is SUT whereas roles of actors simulated by GITB Tested are indicated as SIMULATED -->
  <actors>
    <gitb:actor id="SenderAccessPoint" name="SenderAccessPoint" role="SUT" />
    <gitb:actor id="ReceiverAccessPoint" name="ReceiverAccessPoint" role="SIMULATED">
      <gitb:endpoint name="as2">
        <gitb:config name="participant.identifier">0088:12345test</gitb:config>
      </gitb:endpoint>
    </gitb:actor>
    <gitb:actor id="ServiceMetadataLocator" name="ServiceMetadataLocator" role="SIMULATED" />
    <gitb:actor id="ServiceMetadataPublisher" name="ServiceMetadataPublisher" role="SIMULATED" />
  </actors>

  <!-- Declares the variables to store intermediate results (message/document parts, computed values, etc) during test execution -->
  <variables>
    <var name="as2_address" type="string" />

    <!-- Participant Identifier of Sender Access Point (System Under Test). Must be retrieved
         from SUT representative -->
    <var name="sender_participant_identifier" type="string" />
    <!-- Participant Identifier of Receiver Access Point (Simulated) -->
    <var name="receiver_participant_identifier" type="string" />
    <!-- Represents the type of document that the recipient is able to handle -->
    <var name="document_identifier" type="string">
      <value>urn:oasis:names:specification:ubl:schema:xsd:Invoice-2::Invoice##urn:www.cenbii.eu:transaction:biitrns010:ver2.0:extended:urn:www.peppol.eu:bis:peppol4a:ver2.0::2.1</value>
    </var>
    <!-- Root namespace of the document identifier  -->
    <var name="document_identifier_ns" type="string">
      <value>urn:oasis:names:specification:ubl:schema:xsd:Invoice-2</value>
    </var>
    <!-- The identifier of the process -->
    <var name="process_identifier" type="string">
      <value>urn:www.cenbii.eu:profile:bii04:ver2.0</value>
    </var>
    <!-- XML local element name of the root-element in the business message -->
    <var name="business_message_type" type="string">
      <value>Invoice</value>
```

```
    </var>

  </variables>

  <!-- Steps to be executed by GITB Testbed -->
  <steps>
    <!-- Assign step assigns the result of an interim computation into a variable -->
    <assign to="$as2_address">concat("https://", $SenderAccessPoint{ReceiverAccessPoint}{network.host}, ":",
$SenderAccessPoint{ReceiverAccessPoint}{network.port})</assign>
    <assign to="$receiver_participant_identifier"
source="$SenderAccessPoint{ReceiverAccessPoint}{participant.identifier}" />
    <assign to="$sender_participant_identifier"   source="$SenderAccessPoint{participant.identifier}" />

    <!-- BeginTransaction step notifies the GITB Testbed that a transaction will start for the given messaging adapter
(handler) in the next messaging steps -->
    <btxn from="SenderAccessPoint" to="ServiceMetadataLocator" txnId="t3" handler="SMLMessaging"/>
    <!-- Receive step receives messages from the SUTs →
    <!-- Here ServiceMetadataLocator actor simulated by test engine will receive a message from SenderAccessPoint
actor which represents the real SUT-->
    <receive id="sml_output" desc="Locate SMP" from="SenderAccessPoint" to="ServiceMetadataLocator"
txnId="t3">
      <config name="dns.domain">B-351cd3bce374194b60c770852a53d0e6.iso6523-actorid-
upis.localhost.</config>
    </receive>
    <!-- Receive step sends messages to the SUTs →
    <!-- Here ServiceMetadataLocator actor simulated by test engine will send a message to SenderAccessPoint actor
which represents the real SUT-->
    <send desc="Resolve SMP domain" from="ServiceMetadataLocator" to="SenderAccessPoint" txnId="t3">
      <input name="dns.address" source="$SenderAccessPoint{ServiceMetadataPublisher}{network.host}"/>
    </send>
    <!-- EndTransaction notifies test engine that a transaction is finalized and no following messaging steps will refer
this transaction any more -->
    <etxn txnId="t3"/>

    <btxn from="SenderAccessPoint" to="ServiceMetadataPublisher" txnId="t2" handler="SMPMessaging"/>
    <receive id="smp_output" desc="Send message to SMP to get Receiver Access Point address"
from="SenderAccessPoint" to="ServiceMetadataPublisher" txnId="t2" />
    <send id="smp" desc="Send SMP Metadata back"  from="ServiceMetadataPublisher"
to="SenderAccessPoint" txnId="t2">
      <input name="smp_metadata" source="$SMP_Metadata_Template"/>
    </send>
    <etxn txnId="t2"/>

    <btxn from="SenderAccessPoint" to="ReceiverAccessPoint" txnId="t1" handler="PeppolAS2Messaging"/>
    <receive id="as2_output" desc="Send message to Receiver Access Point" from="SenderAccessPoint"
to="ReceiverAccessPoint" txnId="t1" >
      <config name="document.identifier">urn:oasis:names:specification:ubl:schema:xsd:Invoice-
2::Invoice##urn:www.cenbii.eu:transaction:biitrns010:ver2.0:extended:urn:www.peppol.eu:bis:peppol4a:ver2.0::2.1</co
nfig>
      <config name="process.identifier">urn:www.cenbii.eu:profile:bii04:ver2.0</config>
    </receive>
    <send id="mdn" desc="Send MDN back to Sender Access Point"  from="ReceiverAccessPoint"
to="SenderAccessPoint" txnId="t1">
      <input name="http_headers" source="$as2_output{http_headers}" />
    </send>
    <etxn txnId="t1"/>

    <!--Verify step applies a specific validation methodology on a given content.  -->
    <verify handler="XSDValidator" desc="Validate Invoice against UBL 2.1 Invoice Schema">
      <input name="xmldocument">$as2_output{business_message}</input>
      <input name="xsddocument" source="$UBL_Invoice_Schema_File"/>
    </verify>
    <verify handler="SchematronValidator" desc="Validate Invoice against PEPPOL BII Core">
      <input name="xmldocument">$as2_output{business_message}</input>
```

```
      <input name="schematron" source="$PEPPOL_BII_CORE_Invoice_Schematron_File"/>
    </verify>
    <verify handler="SchematronValidator" desc="Validate Invoice against PEPPOL BII Rules">
      <input name="xmldocument">$as2_output{business_message}</input>
      <input name="schematron" source="$PEPPOL_BII_RULES_Invoice_Schematron_File"/>
    </verify>
    <verify handler="SchematronValidator" desc="Validate Standard Business Document Header against
OpenPEPPOL Envelope Specifications">
      <input name="xmldocument">$as2_output{business_header}</input>
      <input name="schematron" source="$SBDH_Schematron_File"/>
    </verify>
  </steps>
</testcase>
```

10.2.2 Using a GITB Compliant Validation Service Within A Test Case (here: Validex)

GITB PoC implementation integrates an online validation tool, called Validex, to demonstrate its capabilities of integrating external testing services according to GITB Service Specifications.

10.2.2.1 How to Integrate

In order to integrate an external content validation tool with GITB Testbed, it must implement the GITB Content Validation Service Specifications to wrap its functionalities. To realize this integration, a new module, **gitb-validator-validex** has been created with the **ValidationServiceImpl** class that implements the **ValidationService** web service interface. With this implementation, Validex becomes accessible through the GITB Testbed Validation Service. Requests to this service are delivered to Validex and responses are wrapped with internal test reporting format and returned to tester.

The module definition of Validex Validator can be seen below. One difference from the module definitions provided in gitb-validators module is an additional **serviceLocation** attribute which denotes the endpoint of the wrapper validation service. When the test engine utilizes this validator wrapping the functionalities of Validex, it calls this service which delivers the request to Valdiex, as mentioned before.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<module xmlns="http://www.gitb.com/core/v1/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    id="ValidexValidator" uri="urn:com:gitb:validation:ValidexValidator"
    xsi:type="ValidationModule" isRemote="true"
serviceLocation="http://localhost:9091/service/ValidationService">
  <metadata>
    <name>Validex Validator</name>
      <version>1.0</version>
      <description>Validex wrapper validation service</description>
  </metadata>
  <inputs>
    <param type="string" use="R" name="name" desc="Name of the document to be validated" />
    <param type="object" use="R" name="document" desc="XML document to be validated" />
  </inputs>
  <outputs>
    <param name="string" type="name" desc="Name of the document to be validated"/>
    <param name="string" type="document" desc="XML document to be validated"/>
    <param name="string" type="reportId" desc="Report id given by the Validex service"/>
    <param name="string" type="reportLink" desc="Report link to the Validex reportin interface"/>
  </outputs>
</module>
```

10.2.2.2 Definition of the Test Case

In order to benefit from the external validation tools within a Test Case, it is enough to set the id of the validator in Verify step. For instance, as it can be seen from the module definition of Validex, the id attribute, **ValidexValidator** is set as the validation handler in the below Verify step.

```
<?xml version="1.0" encoding="UTF-8"?>
<testcase id="PEPPOL-SenderAccessPoint-Invoice-Validex" xmlns="http://www.gitb.com/tdl/v1/"
xmlns:gitb="http://www.gitb.com/core/v1/">
  <metadata>
    <gitb:name>PEPPOL-SenderAccessPoint-Invoice-Validex</gitb:name>
    <gitb:type>CONFORMANCE</gitb:type>
    <gitb:version>0.1</gitb:version>
    <gitb:description>The objective of this Test Scenario is to ensure the Sender Access Point (the System
Under
      Test) can submit a conformant PEPPOL BIS 4A electronic invoice to a Receiver Access Point using the
AS2
      protocol. Then submitted document is validated by Validex.
    </gitb:description>
  </metadata>
  <namespaces>
  </namespaces>
  <imports>
  </imports>
  <actors>
    <gitb:actor id="SenderAccessPoint" name="SenderAccessPoint" role="SUT"/>
    <gitb:actor id="ReceiverAccessPoint" name="ReceiverAccessPoint" role="SIMULATED">
      <gitb:endpoint name="as2">
        <gitb:config name="participant.identifier">0088:12345test</gitb:config>
      </gitb:endpoint>
    </gitb:actor>
  </actors>
  <variables>
  </variables>
  <steps>
    <btxn from="SenderAccessPoint" to="ReceiverAccessPoint" txnId="t1" handler="PeppolAS2Messaging"/>
    <receive id="as2_output" desc="Send message to Receiver Access Point" from="SenderAccessPoint"
        to="ReceiverAccessPoint" txnId="t1">
      <config name="document.identifier">urn:oasis:names:specification:ubl:schema:xsd:Invoice-
2::Invoice##urn:www.cenbii.eu:transaction:biitrns010:ver2.0:extended:urn:www.peppol.eu:bis:peppol4a:ver2.0
::2.1</config>
      <config name="process.identifier">urn:www.cenbii.eu:profile:bii04:ver2.0</config>
    </receive>
    <send id="mdn" desc="Send MDN back to Sender Access Point" from="ReceiverAccessPoint"
to="SenderAccessPoint"
 txnId="t1">
   <input name="http_headers" source="$as2_output{http_headers}"/>
    </send>
    <etxn txnId="t1"/>

    <verify handler="ValidexValidator" desc="Validate Invoice document using the Validex validation service">
      <input name="document">$as2_output{business_message}</input>
      <input name="name">"Invoice document"</input>
    </verify>
  </steps>
</testcase>
```

11 GITB Compliance

11.1 GITB Compliance Levels

Test Beds can achieve different levels of compliance with the GITB recommendations:

- **GITB Framework Compliance** implies that a Test Bed provides the specific functional capabilities required for eBusiness testing (see GITB Phase 1, CWA 16093:2010, Chapter 5.7). It thereby fulfils certain quality criteria in eBusiness testing and, in other words, we can state that it is "GITB Friendly".

- **GITB Service Compliance** implies that a test service implements one of the GITB Test Service specifications outlined in this report (see GITB Phase 3, Chapters 8). Based on the functionality of the test service, Testing Resources can be reused or shared with other "GITB Service Compliant" service. The service claiming the compliance can either be i) a service consumer reusing the testing resource or ii) a service provider; sharing the testing resource. When claiming GITB Service Compliance, the service name and the role should be stated as either of the followings;
    - **GITB Compliant Content Validation Service Consumer (or Provider**)
    - **GITB Compliant Messaging/Simulation Service Consumer (or Provider**)
    - **GITB Compliant Test bed Service Consumer (or Provider**)

- **GITB TDL Compliance** ensures that test cases written in the GITB Test Description Language (TDL) can be reused and shared. It can be claimed in the following contexts;
    - **GITB Compliant TDL Processor** – a Test bed has the ability to process the test cases written in GITB TDL format and execute the corresponding test scenarios accordingly
    - **GITB Compliant TDL Producer** – any Test Bed or software component that produces test cases in TDL

All levels of compliance are complementary and fulfil different purposes (see Table 11-1) – functional capabilities for GITB Framework Compliance, service-oriented interoperability for GITB Service Compliance, and reusing test scenario definitions for GITB TDL Compliance. The criteria to achieve GITB Compliance are described in the following sections.

Table 11-1: GITB Compliance Levels

Level of compliance	Description	Criteria	Added Value for the Test Bed/Service
GITB Framework Compliance	Comply with the functional requirements stated in the GITB Testing Framework	Identified in GITB Phase 1 - CWA 16093:2010 (Chapter 5.7): Provide functional capabilities (test execution & test case model)	Demonstrate the specific functional capabilities required for eBusiness testing.
GITB Service Compliance	Share the testing resources with other systems in a standard way (Service Provider), OR Reuse GITB compliant testing services within your system (Service Consumer).	Identified in GITB Phase 3 (see Chapters 5 to 8): Either provide one of the service interface (Chapter 8) as a provider for sharing Testing Resources or use the service interface as a consumer of the provided Testing Resource. In relation with the service specification either produce (for service providers), or consume (for service consumers) Test Report artifacts complying with the GITB Test Report format (Chapter 7).	Share or reuse Testing Resources with other GITB Service Compliant systems. Become part of the GITB network of Test Services.

GITB TDL Compliance	Test bed's ability to process the test description language defined in GITB OR Ability of producing test cases in the test description language defined in GITB.	Identified in GITB Phase 3 (see Chapter 9): Be able to process GITB TDL scripts and execute the testing scenario accordingly (configurations, messaging, content validations, etc). or Be able to export or produce test cases written in TDL to share them with others.	Share or reuse test case definitions among different testbeds as a result different domains, initiatives, projects.

11.2 GITB Framework Compliance

The criteria for achieving GITB Framework Compliance have been elaborated in GITB Phase 1 as engineering-level requirements (CWA 16093:2010, Chapter 5.7) and have been reviewed and complemented in the current phase.

A Test Bed can claim to be **GITB Framework Compliant** if it provides the functional capabilities related to

a. the test case model (FUC-TCM), i.e. it provides the capabilities to represent the test-related configuration information, procedural information, verification information as well as the test suites (containing test cases) and test data. – criteria specified in Table 11-211-2,
b. the test execution (FUC-TCE), i.e. it provides capabilities of test preparation and setup, controlling test steps, message exchange, message pre-/post-processing, validation and recovery, reporting and B2B system emulation – criteria specified in Table 11-311-3.

Table 11-2: Criteria for GITB Framework Compliance – Test Case Model (FUC-TCM)

FUC-TCM/R01 Capability of representing test configuration information			
1)	*Capability of representing declaration of messaging protocol to be used*	CWA 16093:2010	For each message exchanged in a test case, the protocol to be used need to be clearly identified
2)	*Capability of representing for each tested actor the type of configuration parameters that are needed*	GITB3 - Addition	For each tested actor, the list of configuration parameters that are needed from the SUT should be identified
FUC-TCM/R02 Capability of representing test procedural information			
1)	*Capability of representing message to be sent*	CWA 16093:2010	
2)	*Capability of representing message choreography*	CWA 16093:2010	
3)	*Capability of representing conditional expression (test step) for test case*	CWA 16093:2010	
4)	*Capability of representing iterative expression*	CWA 16093:2010	
5)	*Capability of representing manual steps*	CWA 16093:2010	
FUC-TCM/R03 Capability of representing test verification information			
1)	*Capability of using external document for verification*	CWA 16093:2010	The tool offers the capability of using external reference material (document, services, schema....) for verification of exchanged messages/content
FUC-TCM/R05 Capability of representing test suite (containing test cases)			
1)	Capability of representing precedence relationships between	CWA 16093:2010	

	test cases		
2)	Capability of grouping test cases into a test suite	GITB3 - Addition	
FUC-TCM/R05 Capability of representing test data			
1)	*Capability of representation of user's defined values*	CWA 16093:2010	The tool offers the capability of referencing data value set for the context of the test
2)	*Capability of representation of automatically generated values*	CWA 16093:2010	The tool is capable of generating data for testing purpose

Table 11-3: Criteria for GITB Framework Compliance – Test Execution Model (FUC-TCE)

FUC-TCE/R01 Capability of test preparation and setup			
1)	*Capability of providing the setup information to SUTs*	CWA 16093:2010	Capability of the tools to provide the SUT operator with the test configuration paramaters
2)	*Capability of requesting SUTs parameters and information*	CWA 16093:2010	Capability of the tools to request from the SUT operator the test configuration paramaters
3)	*Capability of test case customisation*	CWA 16093:2010	Before the execution of the test !
FUC-TCE/R02 Capability of controlling test steps			
1)	*Capability of display of test flow and test progress*	CWA 16093:2010	
2)	*Capability of requesting/storing user's information*	CWA 16093:2010	The user can upload evidences in the tool or input data
3)	*Capability of binding user' information into test*	CWA 16093:2010	The use can provide pointers (URL, URI...) to information external to the test bed
4)	*Capability of manual execution of test steps*	CWA 16093:2010	
FUC-TCE/R03 Capability of message exchange			
1)	*Capability of sending/receiving message*	CWA 16093:2010	The tool is able to send and receive messages. Removed payload from description
2)	*Capability of uploaded/downloading message*	CWA 16093:2010	Manual upload/download of exchanged messages
3)	*Capability of capturing message*	CWA 16093:2010	Automatic capture of exchanged messages by the test bed
FUC-TCE/R04 Capability of message pre-/post-processing			
1)	*Capability of decomposing messages*	CWA 16093:2010	The test bed is capable of processing the messages exchange in the supported protocols. It is capable of using message content for subsequent steps
2)	*Capability of retrieving a value from message*	CWA 16093:2010	
3)	*Capability of generation message template from schema*	CWA 16093:2010	
4)	*Capability of generation of test data for a specific message template*	CWA 16093:2010	
5)	*Capability of message transformation*	CWA 16093:2010	Add here : for better readability (Binary to Text, HL7 ER7 to Tree or XML)
FUC-TCE/R05 Capability of validation and recovery			
1)	*Capability of detecting unknown problems*	CWA 16093:2010	Remove unknown. The test execution can detect errors and report them
2)	*Capability of employing the existing validation engines messages*	CWA 16093:2010	Capibility of using external validation tools and display/process their report
3)	*Capability of recovery from errors*	CWA 16093:2010	Whenever this is possibile the validation tool shall continue processing messages analysis although an error has been identified. Test

			case execution should not abort up on error discovery when this does not impact the test case
FUC-TCE/R06 Capability of reporting			
1)	*Capability of display of error location*	CWA 16093:2010	This section is about the reporting of logged evidence to the testers
2)	*Capability of display of test log information*	CWA 16093:2010	
3)	*Capability of display of detail test result*	CWA 16093:2010	
FUC-TCE/R07 Capability of B2B system emulation			
1)	*Capability of emulation of an arbitrary business unit*	CWA 16093:2010	The test bed offers the capability of emulate a business unit relevant to the tested context. Simulation tool

11.3 GITB Service Compliance

The added value of GITB Service Compliance is that it provides a standard way to share and reuse some common Testing Resources (functionalities) with other GITB Service Compliant systems.

According to the functionality of the Testing Resource, the following compliance statements are possible;

1. **GITB Compliant Content Validation Service Provider or Consumer:**

   A document validation software tool or service can claim to be a **GITB Compliant Content Validation Service Provider** if

   a. it conforms to the GITB Document Validation Service Specification for providing the service and performs the intended content validations as a result of the validation operation calls.
   b. it produces test reports conformant with GITB Test Report Format as a result of the validation operations defined in the service specification.

   A software system (Test bed or any software) can claim to be a **GITB Compliant Content Validation Service Consumer** if

   a. it is able to call any **GITB Compliant Content Validation Service** by conforming the specification.
   b. it consumes the test reports as a result of the validation operation.

2. GITB Compliant Messaging (Simulator) Service Provider or Consumer:

   A software system or service can claim to be a **GITB Compliant Messaging (Simulator) Service Provider** if

   a. it conforms to the GITB Messaging (Simulator) Service Specification and perform the intended communications with or between SUTs as a result of the operations defined in the specification.
   b. it produces the test reports conformant with GITB Test Report Format as a result of those messaging operations.

   A software system (Test bed or any software) can claim to be a **GITB Compliant Messaging (Simulator) Service Consumer** if

   a. it implements the callback operations defined in GITB Messaging (Simulator) Service Specification and is able to call the service operations by conforming the specification.
   b. it consumes the test reports resulted of the message exchange and related validation operations.

3. GITB Compliant Test Bed Service Provider or Consumer:

A **Test Bed service** can claim to be a **GITB Compliant Test Bed Service Provider** if

a. it conforms to the GITB Test Bed Service Specification and performs the intended testing process remotely based on the operation calls on the service.
b. it can map internal testing process definition to the GITB Test Presentation Language (TPL) format and return it as a result of corresponding operation defined in the specification.
c. it produces the test reports conformant with GITB Test Report Format as a result of the execution of a test scenario.

A software system (Test bed or any software) can claim to be **GITB Compliant Test Bed Service Consumer** if

a. It is able to interact with the GITB Compliant Test bed services and make the necessary operation calls to make the remote service preform the intended testing process.
b. It can render the testing process definition in GITB Test Presentation Language (TPL) format and visualize it for system users (SUT administrators)
c. It implements the callback operations defined in GITB Test bed Service Specification to receive the test step reports and status updates.
d. It consumes the resulting test reports conformant with GITB Test Report Format and visualize the results to its users.

Criteria to become **GITB Service Compliant** are summarized in Table 11-411-4:

Table 11-4: Criteria for GITB Service Compliance

Element	Role	Methodology	Related GITB Specifications
GITB Content Validation Service Compliance	Service Provider	Adapt your existing implementation to expose your own Test Service logic through the GITB Content Validation Service	GITB Content Validation Service Specification (see section 9.1) GITB Test Report Format (see section 8)
	Service Consumer	Implement a client that can make calls for the service to make it validate a given content remotely and receive the reports.	
GITB Messaging/Simulator Service Compliance	Service Provider	Adapt your existing implementation to expose your own simulation logic through the Messaging/Simulator service.	GITB Messaging/Simulator Service Specification (see section 9.2) GITB Test Report Format (see section 8)
	Service Consumer	Implement a client that implements the callback function as defined in the specification and can make the necessary operation calls on the service to drive it for intended messaging operations with SUTs.	
GITB Test Bed Service Compliance	Service Provider	Adapt your existing implementation to expose your own Test Bed logic through the Testbed Service to enable remote programmatic execution of test scenarios.	GITB Test Bed Service Specification (see section 9.3) GITB Test Report Format (see section 8) GITB Test Presentation Language (see section 7)
	Service Consumer	Implement a client that implements the callback function as defined in the specification and can make the necessary operation calls on the service to drive it to execute the test scenarios programmatically.	

The implementers of GITB service specifications are free to use their own approach, software architecture, or technology to implement their tools. The only requirement is to conform to the web service specifications to serve their functionalities to others in a common way.

The main benefit of being GITB service compliant is that Test Beds can cooperate in a network and share their Testing Capabilities and Testing Resources with other GITB-service compliant Test Beds. Their own Testing Capabilities will then be usable (as add-ons) by other GITB-service compliant Test Beds, as they make use of standardized plug-in interfaces and exhibit the expected level of standardization for their Test Artifacts.

11.4 GITB TDL Compliance

The following compliance statements are possible:

A Test Bed can claim to be **GITB Compliant TDL Processor** if it is able to process the test scenario definitions written in TDL and execute the test scenario by performing the intented operations accordingly as described in detail in GITB TDL Specification. We assume that the complete package of the required test artifacts are available to the Test Bed. Furthermore, we assume that GITB Compliant services referenced in the test case description are available and in operation.

A Test Bed or any software (e.x. a TDL editor) can claim to be **GITB Compliant TDL Producer** if it can export or produce test cases in TDL format.

Even the **partial compliance** to TDL is possible, that is the partial support of TDL commands (like looping, parallel executions, etc) for this CWS it is stated as out of scope to decrease the complexity.

Part III: GITB Test Registry and Repository (TRR) Specifications and Prototype

The GITB Architecture foresees a Test Registry and Repository (TRR) as access point for any published Testing Resource. The Testing Resource could include not only Test Artifacts like Test Cases, script rules, and Test Suites, but also test components with their APIs and interfaces.

Part III of this report summarizes GITB Phase 3 outcomes related to the TRR.

- TRR specifications in the form of an ADMS profile (Section 12).
- The prototype implementation based on Joinup (Section 13).

This part is relevant for **testing experts and architects** that are interested in registries and repositories for managing, archiving and sharing Testing Resources.

12 GITB Test Registry and Repository (TRR) Specifications

12.1 Role of TRR in the GITB Architecture

In the GITB Architecture, the Test Registry and Repository (TRR) is aimed at supporting the Test Bed for managing, archiving and sharing distributed Testing Resources in a central location, accessible by other parties. The TRR can also be used as a long-term archival platform for Testing Resources. The TRR is part of the GITB testing infrastructure, however, it is not considered as part of a Test Bed, as it is a software component that is independently deployed, managed and accessed. The TRR is accessible through a graphical user interface called the TRR Client or through a Web service interface.

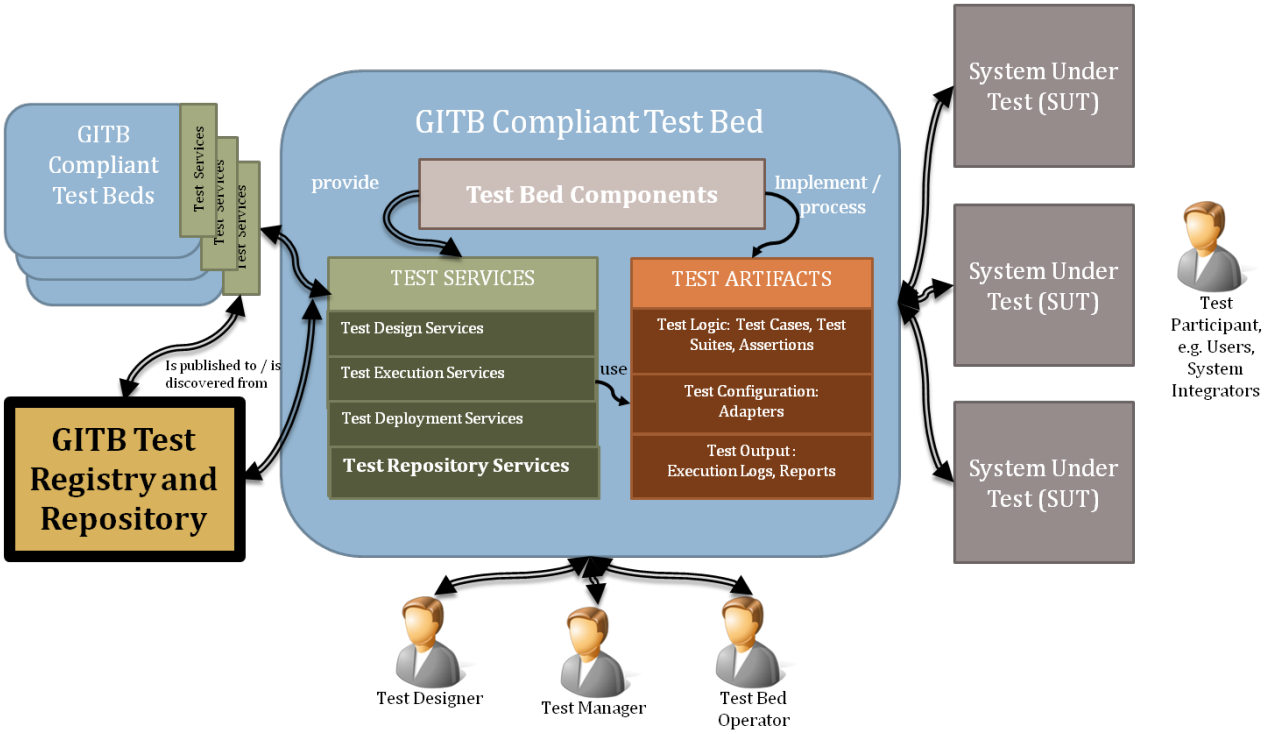

Figure 12-1: TRR in the GITB Architecture

When it comes to managing and sharing resources in a distributed environment, a distinction is typically made between the concept of a **registry**, that lists items with pointers to find the items, and the concept of a **repository**, that stores the actual items. The TRR can fulfil both purposes.

The main capability offered by the TRR is the search functionality that allow users to find information about and locate existing Testing Resources and Test Beds. The TRR features are described in details in section 12.5.

12.2 User Classes and Roles

The following table summarizes the user and user classes of Test Beds, by referring to the roles defined in the GITB Testing Framework (see section 4.3).

Table 12-1: List of the Test Bed Actors

User Classes	Roles in the GITB Testing Framework (see 4.3.)	Short Description
Test Experts: Provider of test beds or testing services	**Test Designer**	Creator and editor of Testing Resources
	Test Manager	Executor or execution facilitator of Test Suites
	Test Bed Provider	Operator of a Test Bed
Test Participants: Owner or operator of a SUT	**End User**	All organizations – from private and public sectors – which implement eBusiness scenarios
	Industry Consortia and Formal SDOs	Communities of end-users, public authorities and other interested parties
	Software Vendors	Vendors of enterprise applications that intend to be compliant with existing eBusiness Specifications
	Testing Laboratories	Laboratories specialized in increasing efficiency and reliability of interoperable implementation of standards

As the TRR is part of the GITB testing infrastructure, it is expected that the Test Bed users would also be users of the TRR. However, as the TRR is an independent system, new user roles or actors shall be introduced. The suggested TRR user roles are listed in the following table. Test Bed Users and business users as previously defined could have any of the roles specific to the GITB Compliant TRR platform.

Table 12-2: List of the TRR Actors

TRR User Roles	Short Description
TRR Administrator	Administrator of the TRR platform
Workspace Administrator	Workspace User who has the administrator rights on a workspace, a private place for a set of users, where users can administrate their folders and Testing Resources
Workspace User	Authenticated User invited to participate to a workspace by a workspace administrator
Authenticated User	Anonymous User who has created an user account on the platform and has logged in
Anonymous User	User who is not logged in the platform

12.3 Basic Concepts

12.3.1 Testing Resources Managed by the TRR

Testing Resource is a generic term introduced in GITB that designates any part of a Test Bed (Test Artifact, Test Service interface, core or plug-in Test Bed Component), or a combination of these.

The Testing Resources managed by the TRR are the following resources:

- Test Artifacts like Test Cases, script rules, and Test Suites,
- Test Components (also known as Testing Capability Components) with their APIs and interfaces.

The following table lists the Testing Resources defined in the GITB Testing Framework (see section 4) which are managed by the TRR.

Table 12-3: Different Type of Testing Resources managed by the TRR

Primary concepts		**Type**	**Short description**
Test Artifact	Test Logic Artifact	**Document Assertion**	a package of artifacts used to validate a Business Document, typically including one or more of the following: a schema (XML), consistency rules, codelists, etc. These artifacts are generally machine-processable
		Test Case	an executable unit of verification and/or of interaction with an SUT, corresponding to a particular testing requirement, as identified in an eBusiness Specification
		Test Suite	defines a workflow of Test Case executions and/or Document Validator executions
Test Capability Component		**Messaging Adapter**	specialized for messaging protocol stacks such as ebXML Messaging, Web Services with SOAP or REST, AS2/AS4, and the underlying transports: SMTP, HTTP, etc.
		Document Validator	responsible for validating the content of the documents retrieved from the Messaging Adapters in terms of both structure and semantic such as EDI: ANSI, EDIFACT, XML

12.3.2 Metadata

As registry and repository, the TRR can contain Testing Resources that are either actual Test Artifacts, or references to a Test Artifact contained in another system (e.g. a repository, a Test Bed), or a reference to an actual Test Bed.

To facilitate the management, discovery and identification of the Testing Resource, the data stored or referenced within the TRR need to be associated with metadata.

The National Information Standards Organization[25] defines metadata as the “structured information that describes, explains, locates, or otherwise makes it easier to retrieve, use, or manage an information resource. Metadata is often called data about data or information about information”. Metadata provides data or information that enables to make sense of data, concepts and real-world entities. Metadata is a particular

[25] http://www.niso.org/publications/press/UnderstandingMetadata.pdf

kind of information, associated to Test Artifacts and to Test Beds. For example, title, author, creation date, name are default metadata associated to the Testing Resources.

For providing proper metadata to the Testing Resources, the good practice[26] is to reuse existing vocabularies developed by standards and specifications. For example, the following general purpose standards and specification can be reused: Dublin Core for published material (text, images), FOAF or ISA Core Vocabularies[27] for people and organisations, SKOS for concept collections. These standards and specifications are available as RDF datasets.

The use of RDF to structure, organize and model metadata is aligned with the actual ways to model and present data and follows the Linked Data[28] principles.

12.4 The Asset Description Metadata Schema application profile for TRR

The Asset Description Metadata Schema (ADMS)[29] was originally drafted to describe semantic interoperability assets. An ADMS application profile is a specification for data interchange that adds additional constraints to the original ADMS, so the scope of the ADMS is extended or restricted for specific purpose by modifying required terms, classes and properties, etc.

A specific metadata schema has been developed for identifying and describing the Testing Resources managed by the TRR by reusing and combining existing terms from different standards and specifications. The metadata schema for Testing Resources is an ADMS application profile called ADMS.TRR.

Table 12-4: The primary concepts introduced by ADMS

Primary concept	Definition	Concept of GITB
Asset Repository	A system or service that provides facilities for storage and maintenance of descriptions of Assets and Asset Distributions, and functionality that allows users to search and access these descriptions.	The concept of TRR
Asset	An abstract entity that reflects the intellectual content of the asset and represents those characteristics of the asset that are independent of its physical embodiment.	The concept of an abstract design or model of a Testing Resource
Asset Distribution	A particular physical embodiment of an Asset. A Distribution is typically a downloadable computer file (but in principle it could also be a paper document or API response) that implements the intellectual content of an Asset.	The concept of a concrete representation of a Testing Resource

In the following sections, the developed ADMS application profile is presented, and in particular we:

[26] This is what we see from the European initiatives about software reuse and promotion of the Linked Open Data: https://joinup.ec.europa.eu/community/ods/description

[27] For example: http://www.w3.org/TR/vocab-regorg/

[28] http://linkeddata.org/

[29] http://www.w3.org/TR/vocab-adms/

- introduce a namespace for Testing Resource (section12.4.2),
- list the classes to reuse and the ones to introduce (section 12.4.3),
- for each class, list its properties and their scope: mandatory, recommended, optional (section 0),
- for some properties, point to an existing vocabulary (e.g. Eurovoc domains category) or specify a new vocabulary (paragraph 12.4.5).

Controlled vocabularies, which are predefined lists of values to be used as values for a specific property in the metadata schema, are used in the metadata schema for Testing Resource. Common controlled vocabularies make metadata understandable across systems.

12.4.1 Logical view of the metadata

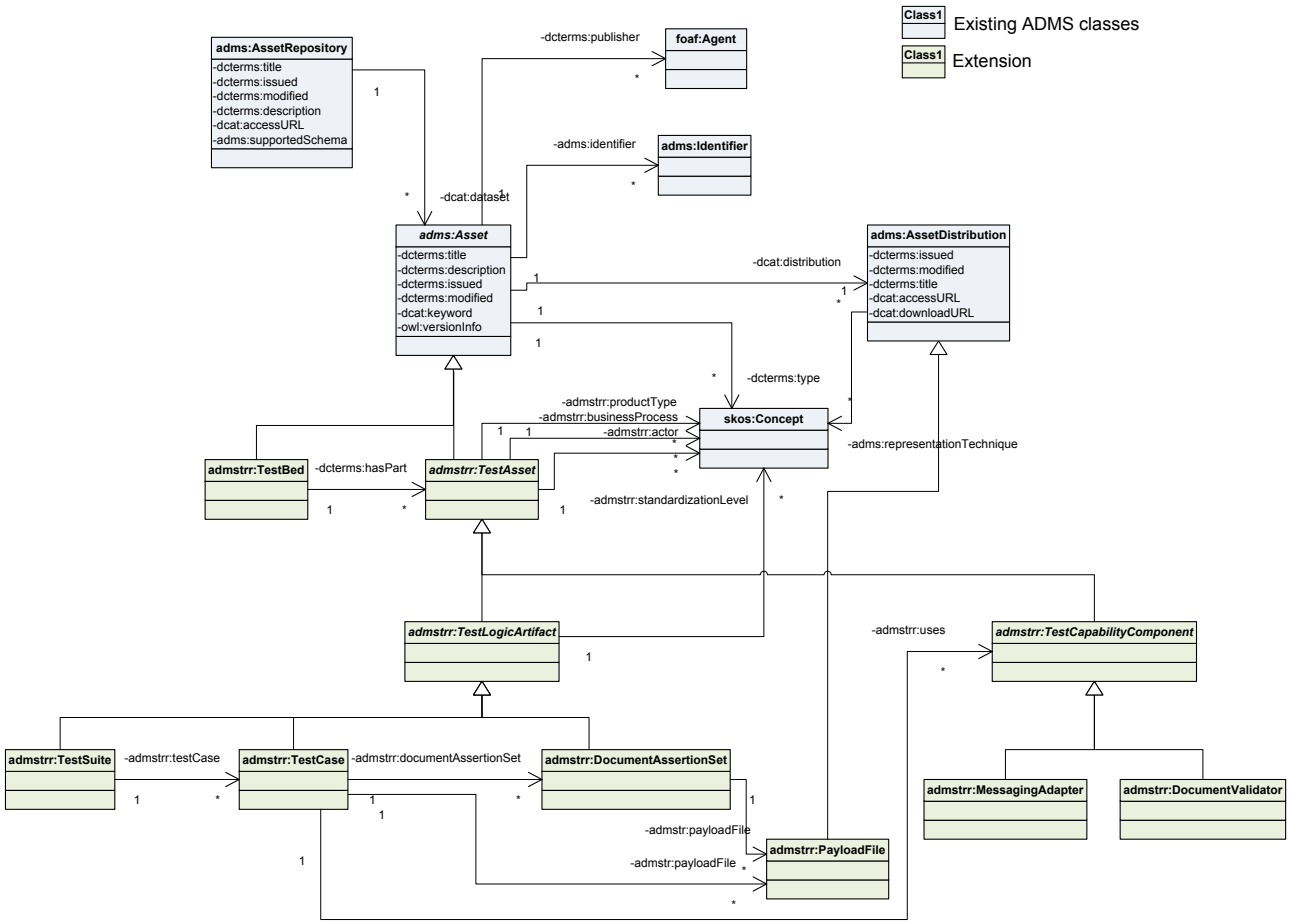

Figure 12-2: The TRR Metadata Schema

The ADMS.TRR application profile extends the existing ADMS specification[30]. Also, the extended ADMS specification[31] provided within the Joinup platform is the reference of this application profile. It means that some of the classes presented here have been reutilized as-is from the ADMS specification.

12.4.2 Namespaces

In the following sections, classes and properties are grouped under headings ‘mandatory’, ‘recommended’ and ‘optional’. These terms have the following meaning.

- **Mandatory class**: a receiver of data MUST be able to process information about instances of the class; a sender of data MUST provide information about instances of the class.

[30] http://www.w3.org/TR/vocab-adms/

[31] https://joinup.ec.europa.eu/catalogue/distribution/Extended_ADMS_Specification_v100zip

- **Recommended class**: a receiver MUST be able to process information about instances of the class; a sender SHOULD provide the information if it is available.
- **Optional class**: a receiver MUST be able to process information about instances of the class; a sender MAY provide the information but is not obliged to do so.
- **Mandatory property**: a receiver MUST be able to process the information for that property; a sender MUST provide the information for that property.
- **Recommended property**: a receiver MUST be able to process the information for that property; a sender SHOULD provide the information for that property if it is available.
- **Optional property**: a receiver must be able to process the information for that property; a sender MAY provide the information for that property but is not obliged to do so.

The table below lists the namespace prefixes that are used in the following sections with the corresponding namespaces URIs.

Table 12-5: Namespaces of the ADMS application profile

Namespace Prefix	Namespace URI
adms	http://www.w3.org/ns/adms#
admstrr	http://purl.org/adms/trr/
dcat	http://www.w3.org/ns/dcat#
dcterms	http://purl.org/dc/terms/
doap	http://usefulinc.com/ns/doap#
foaf	http://xmlns.com/foaf/0.1/
qb	http://purl.org/linked-data/cube#
rad	http://www.w3.org/ns/radion#
rdfs	http://www.w3.org/2000/01/rdf-schema#
schema	http://schema.org/
skos	http://www.w3.org/2004/02/skos/core#
spdx	http://spdx.org/rdf/terms#
swid	http://standards.iso.org/iso/19770/-2/2009/
trove	http://sourceforge.net/api/trove/index/rdf#
v	http://www.w3.org/2006/vcard/ns#
wdrs	http://www.w3.org/2007/05/powder-s#
xsd	http://www.w3.org/2001/XMLSchema#

12.4.3 Application Profile Classes

These classes include the Test Bed and Test Suite classes (at least one of them is mandatory depending of the scope) and all classes that appear as the range of mandatory properties in the description of instances of these two classes.

Table 12-6: Mandatory Classes

Class name	Usage note for the Application Profile	URI
Asset	Abstract entity that reflects the intellectual content of an Asset and represents those characteristics that are independent of its physical embodiment. This abstract entity combines the FRBR[32] entities work (a distinct intellectual or artistic creation) and expression (the intellectual or artistic realization of a work). The physical embodiment of an Asset is called an Asset Distribution. A particular Asset may have zero or more Distributions.	adms:Asset
Test Bed	An actual test execution environment for Test Suites or Test Services. Contains Testing Capabilities and various Test Suites or Document Assertions.	admstrr:TestBed
Test Suite	Defines a workflow of Test Case executions and/or Document Validator executions. The Test Suite class is a subclass of Asset class.	admstrr:TestSuite
Publisher	Organisation making information available. This can be an organisation that has customized a particular standard to answer its specific business needs.	foaf:Agent
Identifier	Identifier of an Asset.	adms:Identifier
Specification Type	The type of specification the Test Bed or the Testing Resource refers to, using a controlled vocabulary (see section 12.4.5).	skos:Concept

The following classes are classified as Recommended to allow the user to give additional details about the content of a Test Suite or a Test Bed.

Table 12-7: Recommended Classes

Class name	Usage note for the Application Profile	URI
Asset Distribution	Particular physical embodiment of an Asset, which is an example of the FRBR entity manifestation (the physical embodiment of an expression of a work). A Distribution is typically a downloadable computer file (but in principle it could also be a paper document or API response)	adms:AssetDistribution

[32] Cataloguing Section. Functional Requirements for Bibliographic Records, section 3. Entities.

http://archive.ifla.org/VII/s13/frbr/frbr_current3.htm

	that implements the intellectual content of an Asset. A particular Distribution is associated with one and only one Asset, while all Distributions of an Asset share the same intellectual content in different physical formats.	
Document AssertionSet	A package of artifacts used to validate a Business Document, typically including one or more of the following: a schema (XML), consistency rules, codelists, etc. These artifacts are generally machine-processable. The Document Assertion Set class is a subclass of Asset class.	admstrr:DocumentAssertionSet
Test Case	An executable unit of verification and/or of interaction with an SUT, corresponding to a particular testing requirement, as identified in an eBusiness Specification. The Test Case class is a subclass of Asset class.	admstrr:TestCase
Payload File	represents a concrete document or part of it The Payload File class is a subclass of Asset Distribution class.	admstrr:PayloadFile
Messaging Adapter	specialized for messaging protocol stacks such as ebXML Messaging, Web Services with SOAP or REST, AS2/AS4, and the underlying transports: SMTP, HTTP, etc. The Messaging Adapter class is a subclass of Asset class.	admstrr:MessagingAdapter
Document Validator	responsible for validating the content of the documents retrieved from the Messaging Adapters in terms of both structure and semantic such as EDI: ANSI, EDIFACT, XML. The Document Validator class is a subclass of Asset class.	admstrr:DocumentValidator
Standardization Level	Level of standardization of the Test Artifacts (e.g. Level 1, Level 2, Level 3) of an Asset, using a controlled vocabulary (see section 12.4.5).	skos:Concept

Table 12-8: Optional Classes

Class name	Usage note for the Application Profile	URI
Asset Repository	System or service that provides facilities for storage and maintenance of descriptions of Assets and Asset Distributions, and functionality that allows users to search and access these descriptions. An Asset Repository will typically contain descriptions of several Assets and related Asset Distributions.	adms:AssetRepository
Representation Technique	Machine-readable language in which a Distribution is expressed, using a controlled vocabulary (see section 12.4.5).	skos:Concept

12.4.4 Application Profile Properties per Class

12.4.4.1 Asset

Table 12-9: Mandatory Properties

Property	Range	Usage note	Card.	GITB Concept
adms:identifier	adms:Identifier	identifier for the Asset	0..n	artifactId
dcterms:title	rdfs:Literal	name of the Asset	1..n	artifactName
dcterms:type	skos:Concept	type of the Asset, using a controlled vocabulary (see section 12.4.5)	1..n	

Table 12-10: Recommended Properties

Property	Range	Usage note	Card.	GITB Concept
owl:versionInfo	rdfs:Literal	version number or other designation of the Asset	0..n	version
dcterms:publisher	foaf:Agent	organisation making the Asset available	1..n	authors
dcterms:description	rdfs:Literal	descriptive text for the Asset	1..n	description
dcterms:spatial	dct:Location	geographic region or jurisdiction to which the Asset applies, using a controlled vocabulary (see section 12.4.5)	0..n	
dcat:distribution	adms:AssetDistribution	implementation of the Asset in a particular format	0..n	

Table 11-12-11: Optional Properties

Property	Range	Usage note	Card.	GITB Concept
dcterms:issued	rdfs:Literal typed as xsd:dateTime	date of formal issuance of this version of the Asset	0..1	origDate
dcterms:modified	rdfs:Literal typed as xsd:dateTime	date of latest update of Asset	1..1	modifDate
dcat:keyword	rdfs:Literal	word of phrase to describe the Asset	0..n	keywords

12.4.4.2 Asset Distribution

Table 12-12: Mandatory Properties

Property	Range	Usage note	Card.	GITB Concept
dcat:accessURL	rdfs:Resource	URL of the Distribution	1..n	

Table 12-13: Recommended Properties

Property	Range	Usage note	Card.	GITB Concept
dcat:downloadURL	rdfs:Resource	direct link to a downloadable file in a given format	0..n	
dcat:mediaType	dct:FileFormat	media type of the Distribution as defined by IANA33, using a controlled vocabulary	0..1	
dcterms:license	dct:LicenseDocument	conditions or restrictions for (re-) use of the Distribution	0..1	
adms:representationTechnique	skos:Concept	language in which the Distribution is expressed, using a controlled vocabulary (see section 12.4.5) Note: this is different from the file format, e.g. a ZIP file (file format) could contain an XML schema (representation technique)	0..1	Eg: XSD, DICOM, EDI messages X12, EDIFACT, ODETTE, VDA Rule script file: Schematron, JESS, XPATH

Optional properties

See document about the extended ADMS specification for the exhaustive list of optional properties.

Asset Repository

See document about the extended ADMS specification for the exhaustive list of optional properties.

Test Asset

The Test Asset class is an abstract subclass of the Asset class and therefore inherits all the latter's properties and relationships.

Table 12-14: Recommended Properties

Property	Range	Usage note	Card.	GITB Concept

[33] IANA (Internet Assigned Numbers Authority). MIME Media Types. http://www.iana.org/assignments/media-types

Property	Range	Usage note	Card.	GITB Concept
admstrr:actor	skos:Concept	The actor or business process role associated to this Asset	0..n	
admstrr:businessProcess	skos:Concept	The business process associated to this Asset The type of process that provides a way to unambiguously identify the business activity to which the Asset is associated	0..n	
admstrr:productType	skos:Concept	The type of product associated with this Asset	0..n	

12.4.4.3 Test Bed

The Test Bed class is a subclass of the Asset class and therefore inherits all the latter's properties and relationships. The expected properties for Test Bed are: identifier, title, publisher and landingPage.

12.4.4.4 Test Capability Component

The Test Capability Component class is an abstract subclass of the Test Asset class and therefore inherits all the latter's properties and relationships.

The Messaging Adapter and Document Validator classes are subclasses of the Test Capability Component class.

12.4.4.5 Test Logic Artifact

The Test Logic Artifact class is an abstract subclass of the Test Asset class and therefore inherits all the latter's properties and relationships.

The Test Suite, Test Case and Document Assertion Set classes are subclasses of the Test Logic Artifact class.

Table 12-15: Recommended Properties

Property	Range	Usage note	Card.
admstrr:standardizationLevel	skos:Concept	The level of standardization of the Test Artifacts	0..1

12.4.4.6 Test Suite

The Test Suite class is a subclass of the Test Logic Artifact class and therefore inherits all the latter's properties and relationships.

Table 11-12-16: Recommended properties

Property	Range	Usage note	Card.

Property	Range	Usage note	Card.
admstrr:testCase	admstrr:TestCase	The associated Test Case	0..n

12.4.4.7 Test Case

The Test Case class is a subclass of the Test Logic Artifact class and therefore inherits all the latter's properties and relationships.

Table 11-12-17: Recommended Properties

Property	Range	Usage note	Card.
admstrr:documentAssertionSet	admstrr:DocumentAssertionSet	The associated Document Assertion Set	0..n
admstrr:payloadFile	admstrr:PayloadFile	The associated Payload File	0..n
admstrr:uses	admstrr:TestCapabilityComponent	The associated Test Capability Component	0..n

Optional Properties

12.4.4.8 Payload File

The Payload File class is a subclass of the Asset Distribution class and therefore inherits all the latter's properties and relationships.

12.4.4.9 Messaging Adapter

The Messaging Adapter class is a subclass of the Test Capability Component class and therefore inherits all the latter's properties and relationships.

12.4.4.10 Document Validator

The Document Validator class is a subclass of the Test Capability Component class and therefore inherits all the latter's properties and relationships.

12.4.4.11 Specification Type

12.4.4.12 Identifier

Table 12-18: Mandatory Properties

Property	Range	Usage note	Card.
skos:notation	rdfs:Literal with datatype reflecting the identifier scheme	character string for the identifier	1..1

12.4.4.13 Publisher

Table 12-19: Mandatory Properties

Property	Range	Usage note	Card.
dct:type	skos:Concept	type of the Publisher, using a controlled vocabulary (see section 12.4.5)	0..n

12.4.4.14 Standardization Level

Table 12-20: Mandatory Properties

Property	Range	Usage note	Card.
rdfs:label	rdfs:Literal	label for the Standardization Level	0..1

12.4.4.15 Representation Technique

Table 12-21: Recommended Properties

Property	Range	Usage note	Card.
skos:notation	rdfs:Literal	label for the Representation Technique	0..n

12.4.5 Controlled Vocabularies to be Used

Property URI	used for class	Vocabulary	Vocabulary URI
dcterms:type	Asset		http://purl.org/adms/trr/specificationtype/
adms:representation Technique	Asset Distribution	ADMS Representation Technique Vocabulary	http://purl.org/adms/trr/representationtype/
admstrr:businessProcess	Asset	ADMS.TRR Business Process Vocabulary Based on the UNCEFACT Catalog of Common Business Process	http://www.ebxml.org/specs/bpPROC.pdf - needs to be enriched for each domain not covered

Property URI	used for class	Vocabulary	Vocabulary URI
admstrr:businessProcessRole	Asset	ADMS.TRR Business Process Role Vocabulary Based on the UNCEFACT Catalog of Common Business Process	http://www.ebxml.org/specs/bpPROC.pdf - needs to be enriched for each domain not covered
admstrr:productType	Asset	Common Procurement Vocabulary (CPV)	http://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX:32008R0213
admstrr:standardizationLevel	Test Logic Artifact	ADMS.TRR Standardization Level Vocabulary	http://purl.org/adms/trr/standardizationLevel
dct:type	Publisher	ADMS Publisher Type vocabulary	http://purl.org/adms/publishertype/
dct:spatial	Asset, Asset Repository	MDR Countries Named Authority List[34], MDR Places Named Authority List[35]	http://publications.europa.eu/resource/authority/country, http://publications.europa.eu/resource/authority/place/

[34] Publications Office of the EU. Metadata Registry. Authorities. Countries. http://publications.europa.eu/mdr/authority/country/

[35] Publications Office of the EU. Metadata Registry. Authorities. Places. http://publications.europa.eu/mdr/authority/place/

12.4.5.1 Specification Type of Asset

Code	URI - Definition
HL7	URI: http://purl.org/adms/trr/HL7 Definition: see · Source: CWA 16408:2012 Related terms: IHE
WS-I-BP2.0	URI: http://purl.org/adms/trr/WS-I-BP2.0 Definition: see · Source: CWA 16408:2012 Related terms:
Autogration	
MOSS	
ePRIOR	
eSENS	
OpenPEPPOL	

12.4.5.2 Representation Type of Asset Distribution

Based on the Representation technique of ADMS (http://purl.org/adms/representationtechnique/).

Code	URI - Definition
Schematron	Schematron
JESS	JESS
XPATH	XPATH
DICOM	DICOM
X12	X12
EDIFACT	EDIFACT
ODETTE	ODETTE
VDA	VDA
HumanLanguage	Human Language
Diagram	Diagram
UML	Unified Modelling Language
XMLSchema	XML Schema
SKOS	Simple Knowledge Organization System
RDFSchema	Resource Description Framework Schema

Code	URI - Definition
Genericode	genericode
IDEF	Integration Definition
BPMN	Business Process Modeling Notation
Archimate	Archimate
SBVR	Semantics of Business Vocabulary and Rules
DTD	Document Type Definition
OWL	Web Ontology Language Full/DL/Lite
SPARQL	SPARQL Query Language for RDF
SPIN	SPARQL Inference Notation
WSDL	Web Service Description Language
WSMO	Web Service Modelling Ontology
KIF	Knowledge Interchange Format
Prolog	Prolog
Datalog	Datalog
RuleML	Rule Markup Langauge
RIF	Rule Interchange Format
SWRL	Semantic Web Rule Language
TopicMaps	Topic Maps
CommonLogic	Common logic
RelaxNG	Relax NG

12.4.5.3 Standardization Level of Test Logic Artifact

Extracted from CWA_16408.

Code	URI - Definition
Level 1	Standardization of a general wrapper or header to the artifact (meta-data standardization).
Level 2	Level 1 plus standardization of external references or interfaces to other artifacts.
Level 3	Level 1 plus Level 2 plus whole content standardization (e.g. detailed XML schema reflecting the entire structure of the artifact).

12.5 Features

12.5.1 Overview

The TRR features are similar to the features of a Registry and a Repository, where storage, retrieval and search are the main features. Figure 12-312-3 shows the interactions between users and the GITB compliant TRR. The features of the TRR are:

- the workspace and folders management,
- the Testing Resources management,
- the bulletin board,
- the Testing Resources search,
- the general administration.

Users need to be able to access features both through a graphical user interface and through a Web service interface. Section 12.7 gives more detail about how users interact with the TRR features.

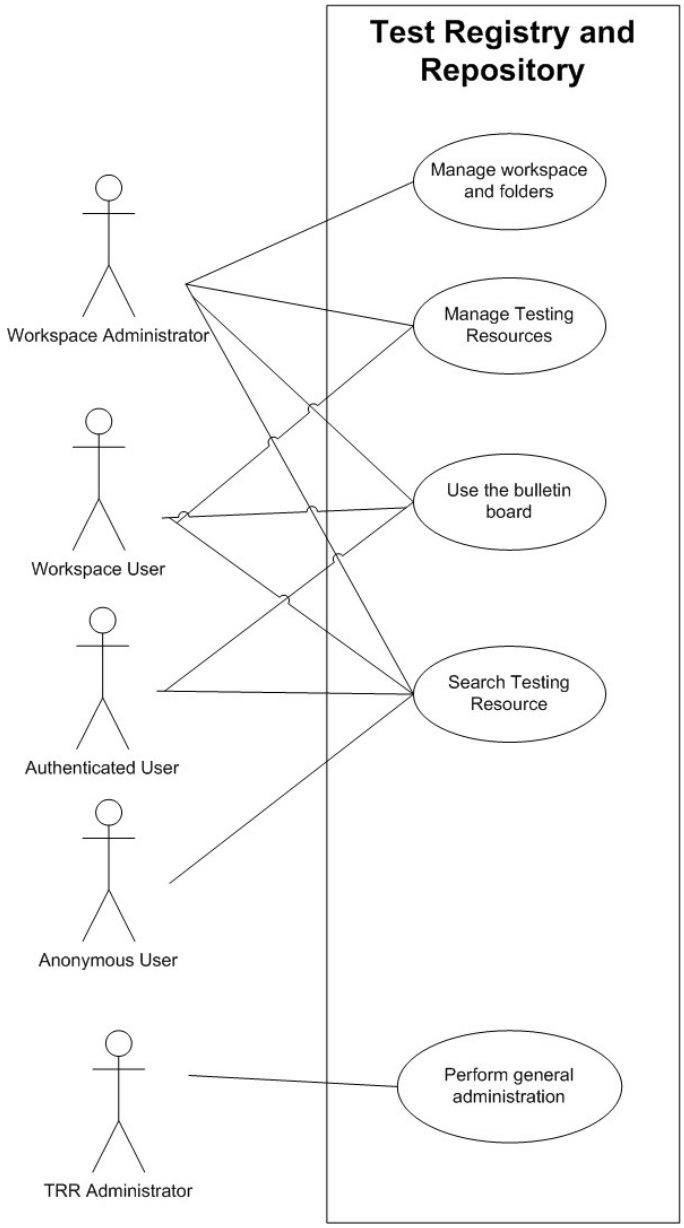

Figure 12-3: Use Case Diagrams of the TRR

12.5.2 Concepts

The following table introduces the concepts used to describe the features of the TRR.

Table 12-22: TRR Concepts

Concept	Description	Optional
Workspace	A workspace is the private place for a set of users, where users can administrate their folders and Testing Resources.	This is optional. Some Testing Resources and information about Test Beds require confidentiality, while others are publically available.
Folder	Testing Resources are organized in folders. It is possible to create a folder tree to organize and store Testing Resources. An archive is a top-level folder.	
Testing Resource	This is the content managed by the TRR introduced in section 12.3.1.	
Bulletin board	The bulletin board is a public place in the TRR to share announcements and comments on them.	

12.5.3 Search Testing Resources

This is about searching among the existing Testing Resources.

Actors	All user
Pre condition	None
Priority	High

Requirements

REQ-1. Perform a free text search: the user enters a plain text
REQ-2. Perform a search on meta-data fields: the user enters some values for specific meta-data fields to refine the scope of the search
REQ-3. Query the system: the user specifies a query in a formal language (e.g. SQL, SPARQL or others)
REQ-4. Download the Testing Resources: the user performs a search (free text, on meta-data or through a query) and can download Testing Resources associated with the results of the search
REQ-5. Access the Testing Resources or Test Bed: the user accesses a Test Bed or a Testing Resource stored in a remote system, and referenced within the TRR.

12.5.3.1 Typical searches

- Which testing resources are available for a specific context and a specific testing purpose?

    - Testing resources = as defined
    - Context = Industry A, Country B, eBusiness specification C, role D (an actor in the eBusiness specification)
    - Testing purpose = validation or simulation
- Which test beds / test providers have expertise in a particular context?
    - Context = Industry A, Country B, eBusiness specification C, role D (an actor in the eBusiness specification)
- Which testing resources are available for different layers of eBusiness specifications/standards?
    - messaging
    - business documents
    - message choreography
    - profile

12.5.3.2 Examples of search queries and their answer

This section gives examples of business queries and instantiations of the TRR metadata schema that answer these queries. The examples are expressed in RDF/Turtle.

The header of the following examples contains the required namespaces:

```
@prefix : <http://example.com/data#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix dct: <http://purl.org/dc/terms/> .
@prefix adms: <http://www.w3.org/ns/adms#> .
@prefix admstrr: <http://purl.org/adms/trr/> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix dcat: <http://www.w3.org/ns/dcat#> .
```

- What are the simulators that this Test Bed is providing?

Shows all instances of class MessagingAdapter linked to the specified Test Bed (dcterms:hasPart).

```
:hl7TestBed1 a admstrr:TestBed ;
  dct:title "HL7 Gazelle" ;
  dct:hasPart :hl7Simulator1 ;
  dct:hasPart :hl7Simulator2 ;
  dct:hasPart :hl7Validator1 ;
  dct:hasPart :hl7TestSuite1 .

:hl7Simulator1 a admstrr:MessagingAdapter ;
  dct:title "Patient Demographics Query simulator" ;
  admstrr:businessProcessRole <http://purl.org/adms/trr/businessprocessrole/PatientDemographicSupplier> ;
  admstrr:businessProcessRole <http://purl.org/adms/trr/businessprocessrole/PatientDemographicConsumer> .

:hl7Simulator2 a admstrr:MessagingAdapter ;
  dct:title "PIX Identity Cross-Reference Manager" ;
  admstrr:businessProcessRole <http://purl.org/adms/trr/businessprocessrole/PatientIdentitySource> .
```

- Which validation services are available for the particular eBusiness specification/standard?

Show all instances of class DocumentValidator linked to the specified eBusiness specification/standard (dcterms:type).

```
:hl7Validator1 a admstrr:DocumentValidator ;
  dct:title "Patient Demographics Query validator" ;
  admstrr:businessProcessRole <http://purl.org/adms/trr/businessprocessrole/PatientIdentitySource> ;
  admstrr:businessProcessRole <http://purl.org/adms/trr/businessprocessrole/PatientDemographicConsumer> ;
  dct:type <http://purl.org/adms/trr/specificationtype/HL7> .
```

```
:hl7Validator2 a admstrr:DocumentValidator ;
  dct:title "GazelleHL7v2Validator External Validation Service" ;
  dct:type <http://purl.org/adms/trr/specificationtype/HL7> .
```

- Which Test Beds support me in testing a particular eBusiness specification / standard?

List all the Test Beds that contain (dcterms:hasPart) Asset which specification type (dcterms:type) is equal to a particular standard (constrained through a controlled vocabulary detailed in paragraph 12.4.5.1).

```
:hl7TestBed1 a admstrr:TestBed ;
  dct:title "HL7 Gazelle" ;
  dct:hasPart :hl7Simulator1 ;
  dct:hasPart :hl7Simulator2 ;
  dct:hasPart :hl7Validator1 ;
  dct:hasPart :hl7Validator2 ;
  dct:hasPart :hl7TestSuite1 .

:hl7TestBed2 a admstrr:TestBed ;
  dct:title "HL7 SRDC TestBed" ;
  dct:hasPart :hl7TestSuite2 .

:hl7Simulator1 a admstrr:MessagingAdapter ;
  dct:title "Patient Demographics Query simulator" ;
  admstrr:businessProcessRole <http://purl.org/adms/trr/businessprocessrole/PatientDemographicSupplier> ;
  admstrr:businessProcessRole <http://purl.org/adms/trr/businessprocessrole/PatientDemographicConsumer> .

:hl7Simulator2 a admstrr:MessagingAdapter ;
  dct:title "PIX Identity Cross-Reference Manager" ;
  admstrr:businessProcessRole <http://purl.org/adms/trr/businessprocessrole/PatientIdentitySource> .


:hl7Validator1 a admstrr:DocumentValidator ;
  dct:title "Patient Demographics Query validator" ;
  admstrr:businessProcessRole <http://purl.org/adms/trr/businessprocessrole/PatientIdentitySource> ;
  admstrr:businessProcessRole <http://purl.org/adms/trr/businessprocessrole/PatientDemographicConsumer> ;
  dct:type <http://purl.org/adms/trr/specificationtype/HL7> .

:hl7Validator2 a admstrr:DocumentValidator ;
  dct:title "GazelleHL7v2Validator External Validation Service" ;
  dct:type <http://purl.org/adms/trr/specificationtype/HL7> .

:hl7TestSuite1 a admstrr:TestSuite ;
  adms:identifier :identifierTs1 ;
  dct:title "HL7V3-P1-TestSuite" ;
  admstrr:businessProcessRole <http://purl.org/adms/trr/businessprocessrole/PatientDemographicSupplier> ;
  admstrr:businessProcessRole <http://purl.org/adms/trr/businessprocessrole/PatientDemographicConsumer> ;
  owl:versionInfo 1.0;
  dct:publisher <http://example.com/data#publisher1>;
  dct:issued "2010-11-15T10:10:03-07:00"^^xsd:dateTime ;
  dct:modified "2011-11-22T10:10:03-07:00"^^xsd:dateTime ;
  dct:description "Test Suite for Profile 1 of HL7"@en ;
  dct:type <http://purl.org/adms/trr/specificationtype/HL7> ;
  dcat:keyword "WebServices"@en ;
  dcat:keyword "SOAP"@en ;
  dcat:keyword "HTTP"@en ;
  dcat:keyword "WSDL"@en ;
  dct:spatial <http://publications.europa.eu/resource/authority/country/FRA> ;
  admstrr:testCase :tc1 .

:hl7TestSuite2 a admstrr:TestSuite ;
  adms:identifier :identifierTs2 ;
  dct:title "HL7V3-TestSuite from SRDC" ;
  dct:type <http://purl.org/adms/trr/specificationtype/HL7> ;
  dcat:keyword "WebServices"@en ;
  dcat:keyword "WSDL"@en ;
  dct:spatial <http://publications.europa.eu/resource/authority/country/TUR> ;
  admstrr:testCase :tc2 .
```

- Given a particular actor or profile, what are the simulators, the validation suites and the Test Suites available?

Show all instances of class MessagingAdapter, DocumentValidator and TestSuite linked to the specified actor (admstrr:businessProcessRole).

- Given a particular validation formalism (e.g. schematron), which Test Bed supports it?

List all the Test Beds that contain (dcterms:hasPart) Asset linked to an AssetDistribution (dcat:distribution) or a PayloadFile when applicable (admstrr:payloadFile) which representation type (dcterms:type) is equal to a particular validation formalism (constrained through a controlled vocabulary detailed in paragraph 12.4.5.2).

```
:das_01_xml a admstrr:PayloadFile ;
  dct:description "XML encoding of das1." ;
  dcat:accessURL <http://www.engisis.com/das_01.xml> ;
  adms:representationTechnique <http://purl.org/adms/tr/representationtype/Schematron> ;
  dct:format <http://purl.org/NET/mediatypes/application/xml> .
```

12.5.4 Testing Resources management

This is a set of action to store and manipulate Testing Resources in the TRR.

Actors	If the workspace concept exists for TRR which require some confidentiality: Workspace User, Workspace Administrator Otherwise: Authenticated User
Pre condition	The user is logged in the TTR
Priority	High

Requirements

REQ-6. Add a Testing Resource: the user adds a new Testing Resource to a folder or adds the reference of a Test Bed or an existing Testing Resource stored in another system
REQ-7. Modify a Testing Resource: the user renames a Testing Resource
REQ-8. Delete a Testing Resource: the user deletes a Testing Resource
REQ-9. Share a Testing Resource: the user shares a Testing Resource with other users of the system, or makes it public (publish)
REQ-10. Subscribe to a Testing Resource: the user can subscribe to a Testing Resource to get notifications when the content of a Testing Resource changes
REQ-11. Change the version of Testing Resource: the user manually changes the version of a Testing Resource
REQ-12. Add meta-data to a Testing Resources: the users modifies the values of the meta-data associated with a Testing Resource
REQ-13. Add a comment to a Testing Resource: the user can write evaluations and comments associated to a Testing Resource

12.5.5 Secondary Features

12.5.5.1 Workspace and Folders Management

It is possible to create a folder tree to organize and store Testing Resources. An archive is a top-level folder.

Actors	Workspace Administrator
Pre condition	The user is logged in the TTR
Priority	Low

Requirements

REQ-14. Add an archive: the user adds a new top-level folder in its workspace
REQ-15. Add a folder: the user adds a new folder to an archive
REQ-16. Modify a folder: the user renames a folder
REQ-17. Delete a folder: the user deletes a folder
REQ-18. Share a folder: the user shares a folder with other users of the system, or makes it public (publish)
REQ-19. Subscribe to a folder: the user can subscribe to a folder to get notifications when the content of a folder changes

12.5.5.2 Bulletin board

Actors	Authenticated User, Workspace User, Workspace Administrator
Pre condition	The user is logged in the TTR
Priority	Low

Requirements

REQ-20. Post an announcement: the user posts an announcement on the bulletin board
REQ-21. Edit an announcement: the user edits a previously posted announcement
REQ-22. Delete an announcement: the user deletes an existing announcement
REQ-23. Comment on an announcement: the user comments on an existing announcement
REQ-24. Subscribe to an announcement: the user can subscribe to an announce to get notifications when the content of the announcement changes or when new comments are posted

12.5.5.3 General administration

A role is associated with a set of privileges. Depending on the user's roles given by the administrator, an user have access to functionalities.

Actors	TRR Administrator
Pre condition	The user is logged as administrator in the TTR
Priority	Medium (meta-data management is high)

Requirements

REQ-25. Manage roles: the user creates a new role, modifies and deletes existing roles
REQ-26. Manage users: the user creates a new user, modifies and deletes existing users
REQ-27. Manage roles with users: the user gives a role to an user, remove a role to an user
REQ-28. Manage workspaces: the user creates a new workspace, modifies and deletes existing workspaces
REQ-29. Add users to a workspace: the user adds existing users to an existing workspace
REQ-30. Manage meta-data: the user creates new meta-data fields to describe the Testing Resources

12.6 Process View

This part explains the TRR processes and how they communicate. It focuses on the runtime behaviour of the TRR due to the user interactions with the TRR.

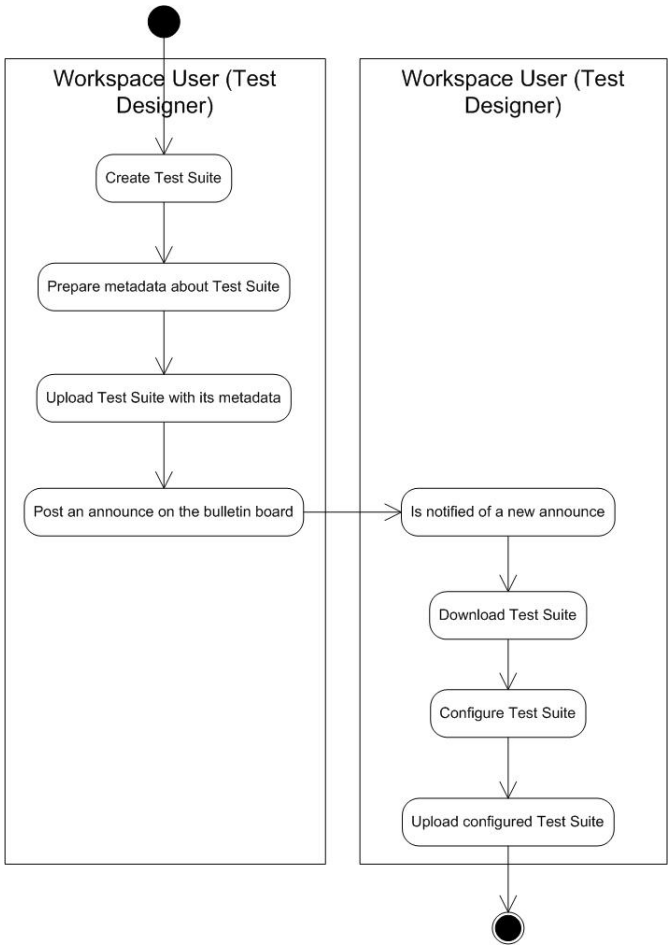

Figure 12-4: Publish new Test Suite for the Latest Version of an eBusiness Specification

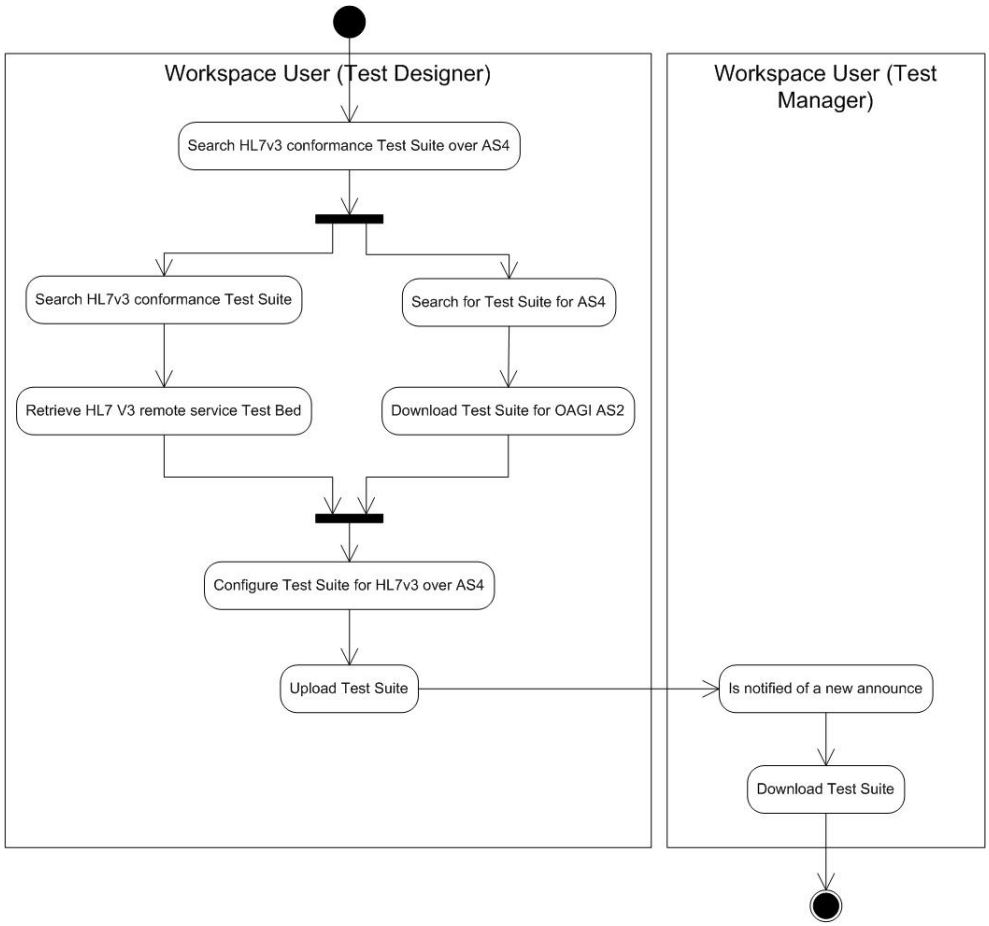

Figure 12-5: Find Available Testing Resources for an eBusiness Specification

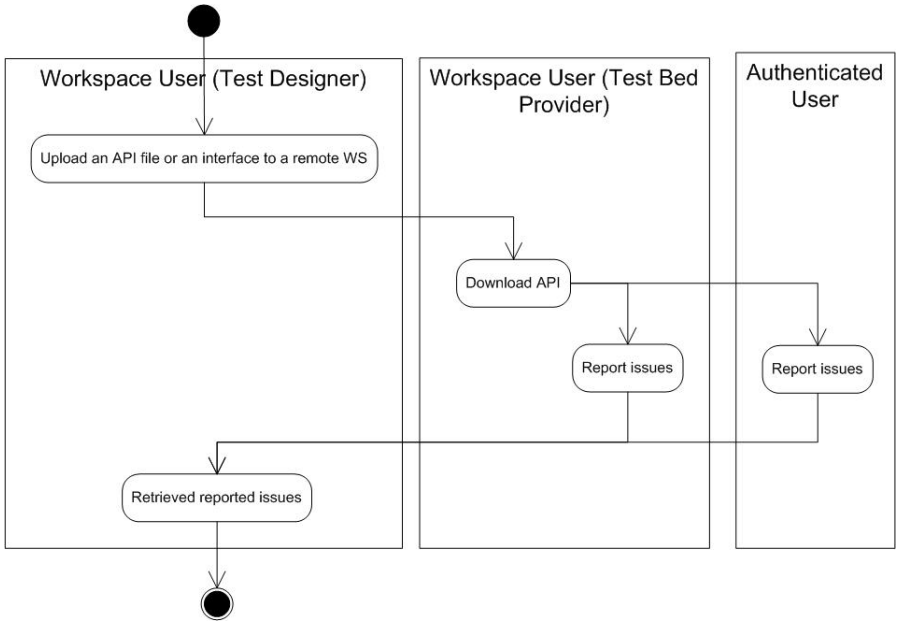

Figure 12-6: Publish a File or an Interface of new Testing Resources in TRR

12.7 External Interfaces

12.7.1 User Interfaces

The TRR provides a web-based user interface to publish, search and download the Testing Resources for the user who does not want to use the GITB Test Bed interface.

12.7.2 Software Interfaces

The TRR is connected with the GITB TestBed component (and particularly the Test Deployment Manager, part of the Test Bed) through a component called the GITB TRR Client.

12.7.3 Communications Interfaces

The TRR contains a messaging interface called the Test Repository Services, which makes the TRR core core functionality available. The messaging interface is based on the standard messaging protocols, such as ebXML, SOAP, REST. It is specified in CWA 16408, Chapter 17.4, page 148-151. The GITB TRR Client leverages this messaging interface.

Table 12-23: External Interfaces

General Search Functions	Get Test Artifacts Matching a Pattern
Administration functions	Create an Archive
	Duplicate an Archive
	Delete an Archive
	Set Access Rights for an Archive
Archival Functions	Store a Test Artifact
	Download a Test Artifact
	Select a Test Artifact or a Set of Artifacts
	Transfer a test Artifact or a Set of Artifacts

13 Test Registry and Repository (TRR) Prototype Implementation

As part of GITB 3, a prototype implementation for TRR has been performed with a reduced set of functionality based on the Joinup platform.

To facilitate the implementation of the TRR within Joinup, the set of features supported by the TRR and the ADMS.TRR have been simplified.

The following sections provide an overview of the prototype implementation. The TRR prototype is available in Joinup at https://joinup.ec.europa.eu/catalogue/repository/gitb-trr.

13.1 Joinup

Joinup is a collaborative platform created by the European Commission with the following capabilities:

- Sharing of information like news, case studies and events about a project,
- Cataloguing interoperability solutions software and searching on the catalogue.

Joinup is open source and uses ADMS extensively for content description.

The main reasons of using Joinup to host the GITB TRR are:

- the existing features of Joinup cover the GITB TRR required features,
- Joinup is released as an open source project which is actively maintained,
- the sustainability of the GITB TRR is assured after the GITB project ends,
- the mission of the ISA, the organization behind Joinup, is aligned with the mission of GITB and the TRR.

13.2 TRR in Joinup: Functional Specification

13.2.1 Use Case Diagram

The main features of the TRR are the following:

- management of Testing Resources (creation, view, update, deletion),
- search of Testing Resources.

The use case model describes the functional requirements for a specific workflow. More specifically, it shows the interactions between the actors and the system from a user's point of view. The following figure provides an overview of the different use cases foreseen for the simplified version of the TRR.

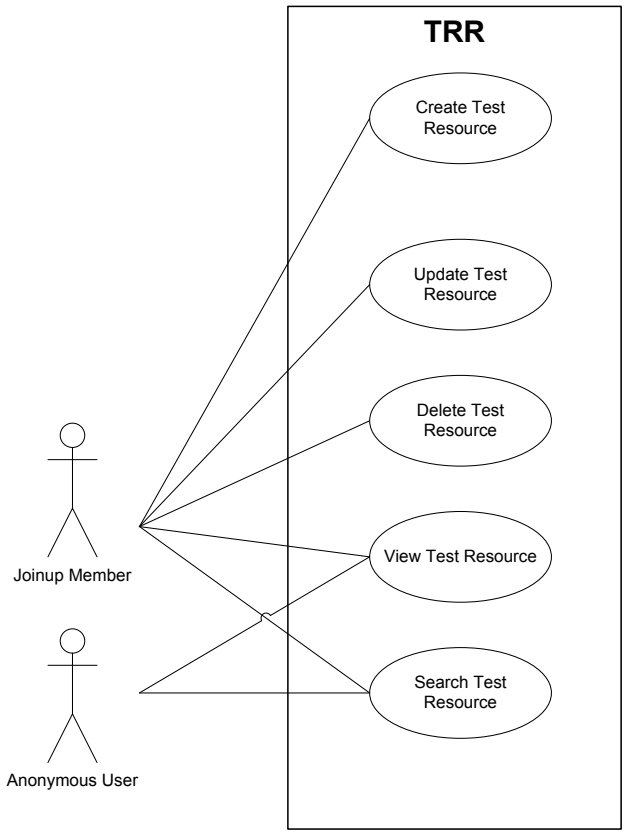

Figure 13-1: TRR Joinup Use Cases

13.2.2 Actors

13.2.2.1 Anonymous User

This actor represents anybody who has access or potential access to the Joinup platform, who can be logged in or not in the platform but does not belong to any Joinup project. This actor is allowed to search and view the Testing Resources.

13.2.2.2 Joinup Member

The Joinup Member actor represents all members of the Joinup platform that belong to at least one repository. The actor has the same authorisations as the anonymous user and can thus search and view Testing Resources.

When the user is a registered user, the user is able to create, update and delete Testing Resources within a Repository for Testing Resources.

The Joinup Member can also reference existing Testing Resources inside the Project page he belongs to. Once a Testing Resource has been created in a repository, it can be referenced in different interoperability solutions, project, repository, etc.

For example, a Testing Resource about eSens created in the TRR repository could be referenced in the eSens project, in the GITB project, etc.

13.2.3 Uses Cases

A detailed explanation of each Use Case is provided in this section.

13.2.3.1 Search Testing Resources within the Joinup Platform

Table 13-1: Search Testing Resources in Joinup

Actor	Anonymous User, Joinup Project Member
Trigger	Ad Hoc
Description	Whenever the actor wants to consult any of the Testing Resources they can access the Joinup platform and search for the relevant Testing Resources.
Preconditions	The Testing Resources are available on the platform.
Post conditions	Not applicable
Basic flow	1) The actor browses to the existing search section of the Joinup platform. 2) Search for the Testing Resource using the existing search capability of Joinup, filter on some fields introduced in paragraph **Erreur ! Source u renvoi introuvable.**: Business process, Standard / eBusiness specification, Actor, Type 3) View the result of the search.
Alternative flow	Not applicable
Exceptions	Not applicable
Includes	Not applicable
Priority	**High priority**
Frequency of use	Continuous
Business rules	Not applicable
Special requirements	Not applicable
Assumptions	Not applicable
Additional comments	No authentication is required to search the Testing Resources.

Example of typical search:

Which testing resources are available for a specific context and a specific testing purpose?

- Testing Resources = as defined
- Context = Industry A, Country B, eBusiness specification C, role D (an actor in the eBusiness specification)
- Testing purpose = validation or simulation (type of Testing Resource)

13.2.3.2 View Testing Resources

Table 13-2: View Testing Resources – Specification in Joinup

Actor	Anonymous User, Joinup Project Member
Trigger	Ad Hoc
Description	Whenever the actor wants to view any of the Testing Resources they can access the Joinup platform and select the relevant Testing Resource(s).
Preconditions	The Testing Resources are available on the platform.
Post conditions	Not applicable
Basic flow	1) The actor browses the Joinup platform. 2) The actor search Testing Resources using the search capability of Joinup. 3) Click on individual Testing Resources available for further details which brings the user to the details of the Testing Resource with its fields and allow to download the associated distribution.
Alternative flow	Not applicable
Exceptions	Not applicable
Includes	Not applicable
Priority	**High priority**
Frequency of use	Continuous
Business rules	Not applicable
Special requirements	Not applicable
Assumptions	Not applicable
Additional comments	No authentication is required to view the Testing Resources.

13.2.3.3 Create & Update Testing Resources

Table 13-3: Create & Update Testing Resources – Specification in Joinup

Actor	Joinup Project Member
Trigger	The member wants to create or update a Testing Resource.
Description	To create a Testing Resource, the actor accesses the Joinup platform, click on Propose your... and select "Testing Resource". To update a Testing Resource, the actor can click on edit on the view page.
Preconditions	The actor needs to be a registered member, i.e. the user needs to have the right to create or update content.
Post conditions	Not applicable
Basic flow	**1)** The actor browses the Joinup platform where he chooses to update a Testing Resource or create a new Testing Resource. **2)** The actor fills out the required fields of the Testing Resource. **3)** The actor saves and publishes the Testing Resource **4)** The Testing Resource is saved under a federated Repository
Alternative flow	**1)** The actor browses the Joinup platform where he chooses to update a Testing Resource or create a new Testing Resource. **2)** The actor fills out the required fields of the Testing Resource. **3)** The actor saves the Testing Resource as draft. **4)** The actor re-opens and continues the Testing Resource. **5)** The actor saves and publishes the Testing Resource.
Exceptions	Not applicable
Includes	Not applicable
Priority	**High priority**
Frequency of use	Continuous
Business rules	Not applicable
Special requirements	Not applicable
Assumptions	Not applicable
Additional comments	Not applicable

13.2.3.4 Delete Testing Resources

Actor	Joinup Project Member
Trigger	The member wants to delete a Testing Resource.
Description	To delete a Testing Resource, the actor clicks on delete on the view page.
Preconditions	The user needs to be a registered member, i.e. the user needs to have the right to create, update and delete content on Joinup.
Post conditions	Not applicable
Basic flow	**1)** The actor browses the Joinup platform where he chooses an existing Testing Resource. **2)** The actor deletes the Testing Resource
Alternative flow	
Exceptions	Not applicable
Includes	Not applicable
Priority	**High priority**
Frequency of use	Continuous
Business rules	Not applicable
Special requirements	Not applicable
Assumptions	Not applicable
Additional comments	Not applicable

13.2.4 Fields of Testing Resources

To limit the complexity of the technical development required to integrate the TRR in Joinup, it has been decided to reuse as much as possible the existing fields of the interoperability solution form (see section 13.2.4.1).

When possible, an existing field is slightly modified to match the vocabulary of the TRR (see section 13.2.4.2).

13.2.4.1 Reused Fields

Field name	Field type	Comments
ID		Existing field of Joinup
Name		Existing field of Joinup
Description		Existing field of Joinup
Distribution		Existing field of Joinup
Solution category		Do not add to the form is a specific page for Testing Resources is created, otherwise, add Testing Resource to the existing select list
Solution type		Shall be extended to contain the Testing Resource type, that's it: Test Bed, Test Suite, Test Case, Document Assertion Set, Messaging Adapter, Document Validator, Test Assertion
Keywords		Besides allowing the user to associate keywords with a Testing Resource, it will also be used to specify that a Testing Resource can work with the GITB reference implementation and if the Testing Resource is generic or not.
Geographic coverage		Existing field of Joinup
Status		Existing field of Joinup
Publisher		Existing field of Joinup
Licence		Existing field of Joinup
Homepage or Testing Resource Link		Used to reference a reference to a Testing Resource that is stored in a remote repository (covers the case when the Joinup TRR is used as a registry only)

13.2.4.2 Updated Fields

Field name	Field type	Comments
Business process	Select checkbox	Called *Themes* previously, based on an existing taxonomy (http://eurovoc.europa.eu/). The existing taxonomy can be used as it is for now. The type of process that provides a way to unambiguously identify the business activity to which the Asset is associated
Solution category	Select list	This is an existing list called Solution category to categorize the interoperability solutions. The list contains the following elements: Framework, Service, Tool. A Testing Resource is a particular interoperability solution.
Reference to another Testing	Asset	Called *Reference to another interoperability solution* previously

Resource		Allow the user to represent the following relationships: • hasPart (a TestBed references some Testing Resources like TestCase or DocumentValidator, etc.) • testCase (a TestSuite is composed by a set of TestCase) • documentAssertionSet (a TestCase is composed by a set of DocumentAssertionSet) • uses (a TestCase uses a MessagingAdapter or/and a DocumentValidator)
Relation Type	Select list	Default values are: Next Version, Previous Version, Translation, Included Asset, Related Asset, Sample Somehow we need to support versioning between Testing Resources so Next Version and Previous Version are important. Desired values are : • Next Version and Previous Version • either: Included Asset to have a generic way to link Testing Resource (e.g. a Test Suite contains several Test Cases) • or: contains (instead of hasPart), testCase, documentAssertionSet, uses
Standard / eBusiness specification	Asset	Called *Reference to another interoperability solution* previously Allow the user to represent the following relationships: • Link to an existing standard or specification
Actor	List of strings	Called *Keywords* previously The actor or business process role associated to this Asset. This is specific to a particular standard / eBusiness specification. Ideally, it would be an evolving text, i.e. once a user enters a new text that was not previously in the controlled vocabulary, it is added to the controlled vocabulary and becomes available for other users. Ex: PatientIdentitySource, PatientDemographicConsumer, etc.

Part IV: GITB Application and Validation based on Use Cases from Public Procurement, e-Health and Manufacturing Industries

In GITB, use cases are the basis for defining Testing Scenarios, instantiating the GITB Testing Architecture and developping Test Artifacts for the PoC implementation or other GITB-compliant Test Bes.

Part IV of this report describes testing scenarios for the previously selected business use cases from different industries. The general approach for applying GITB is first presented in section 14, before describing its application to the use cases:

- Public Procurement → OpenPEPPOL (Chapter 15), eSens (Chapter 16), CEF – Connecting Europe Facility (CEF) (Chapter 17), NHS (Chapter 18)
- In eHealth → Clinical Document Architecture (Chapter 19) and IHE XDS (Chapter 20)
- In the automotive and manufacturing industry → Electronic Invoicing based on EDIFACT and OFTP2 (Chapter 21), Cross-border Trade (Chapter 22), Test Bed Interoperability with Application for Truck Manufacturer (Chapter 23)

Part IV is targeted at eBusiness users, standard development organizations, industry consortia that are interested in applying the Test Bed Architecture to their eBusiness scenarios.

14 Applying GITB in Use Cases

14.1 Approach

Figure 14-114-1 outlines a step-wise approach for translating eBusiness scenarios into testing requirements and creating testing solutions based on the GITB Principles and Framework (see section 4).

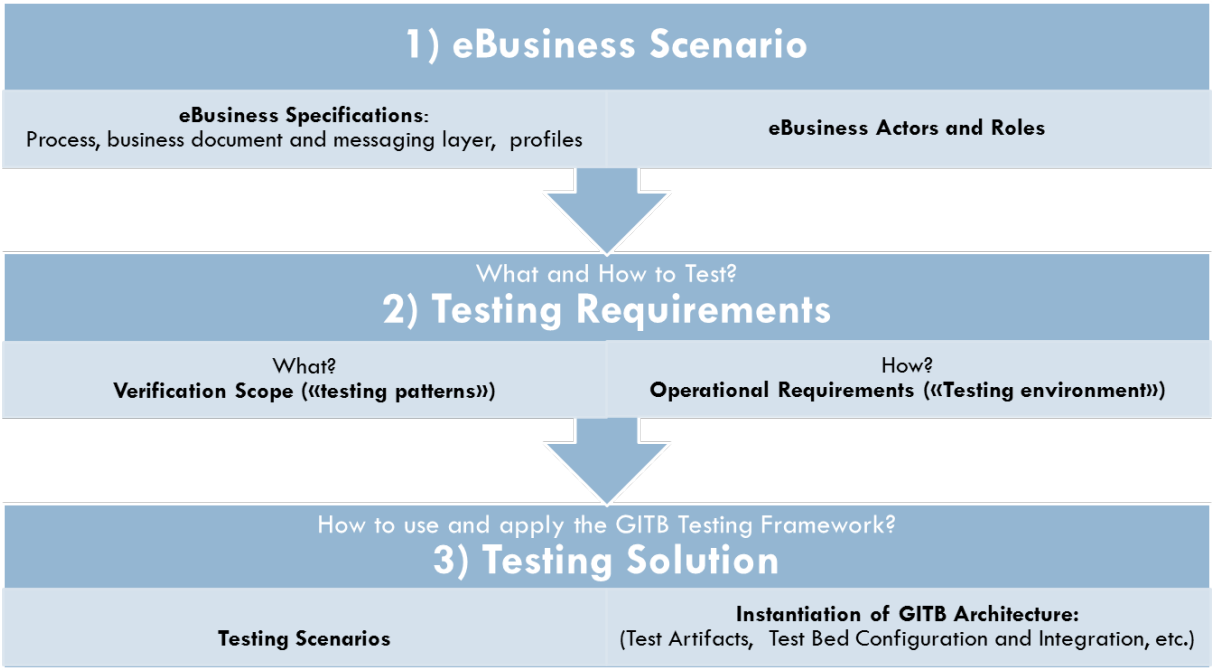

Figure 14-1: Applying GITB in Use Cases

The starting point is the business user's need for implementing and testing one or more eBusiness Scenarios. Business users define the relevant set of eBusiness Specifications as well as the actors involved in the eBusiness interactions with their roles. From the eBusiness Scenarios, the people testing the business scenarios, typically business users responsible for implementation or the integrators or software vendors working with the business users, elaborate on two types of testing requirements. On the one hand, they analyze "what to test" by deriving the exact Verification Scope from the eBusiness Specifications. On the

other hand, they determine “how to test” by specifying the testing environment with its operational requirements. From the testing requirements, the Test Designers and Test Managers can set up the appropriate Test Services and Test Artifacts supporting the Testing Scenarios.

14.2 Deriving Testing Requirements

Two types of testing requirements have to be taken into account prior to designing a testing solution: the Verification Scope (“What to test?”), which can be derived from the eBusiness Specifications, and the testing environment (“How to test”) that determines the operational testing requirements that have to be met by an appropriate testing solution.

14.2.1 Verification Scope (“What to Test?”)

When implementing eBusiness Scenarios, business users rely on one or more eBusiness Specifications referring to the different layers of eBusiness: Business Process, Business Document and messaging layer.

At the **Business Document Layer**, the Verification Scope may comprise structural and semantic validations as well as cross-layer validations with the Messaging Layer (see Table 14-1).

Table 14-1: Verification Scope (“Test Patterns”) for Business Document Layer Validation

Type of Validation	Verification Scope	Description
Structural validation	Document syntax and structure	Testing whether messages conform to the message definitions, e.g. as defined by EDIFACT or XML document schemas (xsd)
	Data types	UN/CEFACT Core Data Type Catalogue (CDT Catalogue)
	Document assembly	Testing whether messages conform to naming and design rules, e.g. as defined by OAGi or UN/CEFACT Core Components Business Document Assembly (CCBDA)
	Mandatory / optional fields	Testing whether all mandatory fields are correctly filled, e.g. as defined by content definition (e.g. xsd)
Semantic validation	Vocabulary and code list verification	Testing whether data fields comply with defined vocabulary, code lists (e.g. DUNS, ISO, UNECE, ...) or core components (e.g. UN/CEFACT CCL, ...)
	Business Document header definitions	Testing whether document headers are correct, e.g. as defined by UN/CEFACT Standardized Business Document Header (BDH) or OAGI BOD's application area
	Business rules	Testing of business rules, e.g. as specified by Schematron
QoS	Equivalent Business Document versions	Testing whether "equivalent" versions for the same document, could be used for the same transaction
	Equivalent syntax versions	Testing whether "equivalent" document syntax, could be used for the same transaction, e.g. different implementations of syntax neutral Business Document specifications

	Consistency of message header and Business Document content	Testing whether message header and Business Document content are aligned

At the **Messaging Layer**, the Verification Scope comprises structural validation, such as testing messaging protocols, validating message headers and testing the discovery of endpoints. It may also comprise validations for QoS and other validations (see Table 14-214-2).

Table 14-2: Verification Scope ("Test Patterns") for Messaging Layer Validation

Type of Validation	Verification Scope	Description
Structural validation	Messaging Protocol	Testing transport and communication level protocols, e.g. as defined by ebXML Messaging (ebMS), SOAP, EDIFACT X12, RosettaNet Implementation Framework (RNIF), Minimal Lower Layer Message Transport protocol (MLLP)
	Message header	Testing whether the message header is valid
	Addressing	Testing the discover of endpoints
Quality of service (QoS) validation	Security	Testing security protocols, e.g. as defined by WS-Security
	Other QoS	Testing QoS, e.g. as defined by WS-Policy
Others	Equivalent messaging styles / formats / versions	Testing "equivalent" messaging styles / formats / versions, that could be used for the same transaction

At the **Business Process Layer**, structural validation comprises testing message sequence and process choreography, the correct interpretation of roles as well as timing conditions. In addition, cross-layer validations (or profile validation) are performed with Business Document and Messaging Layer (see Table 14.3).

Table 14-3: Verification Scope ("Test Patterns") for Business Process Layer Validation

Type of Validation	Verification Scope	Description
Structural validation	Sequence of messages / choreography	Testing the correct sequence of messages, e.g. as defined by sequence diagrams; Testing process choreographies which are informally or formally defined
	Roles	Testing the different roles within a Business Processes, e.g. senders and receivers of messages
	Timing conditions	Testing the timing conditions in business transactions, e.g. as defined by triggering events or reaction times
Cross-layer validation / Profile validation	Data consistency across Business	Testing data relationships across different messages, e.g. as defined by a common information model

	Documents	
	Restrictions on the Business Document format and content	Testing syntactic and semantic restrictions on the Business Document format and content
	Restrictions on message header	Testing restrictions on message header and consistent use of conversation ID
	Restrictions on transport protocols	Testing restrictions on and correct use of transport protocols

14.2.2 Operational Requirements ("In Which Environment?")

The testing environment determines operational requirements that have to be met by an appropriate testing solution:

The testing context (cf. Section 3.4.2) denotes the situation when testing is performed. This can be

- during standard development for quality assurance of the developed eBusiness Specifications,
- when implementing new or upgrading existing eBusiness endpoints,
- when new partners are onboarding.

Testing integration in business environment: Several possibilities exist with regard to integrating in the business environment.

- **Testing system is the in-production system:** The user wants to do testing in the in-production system under exact business conditions (with same firewall setups, security setups, eBusiness gateway setup).
- **Testing system is a non-production system:** The user does not want to disturb currently deployed in-production system, but wants to test a system that is configured differently from the currently production system.
- **No integration in business environment:** In this case, testing is not integrated at all with the business environment, but is done manually.

Testing location:

- **On-premise testing:** In this case, end-users do not want to access a remote server to undergo testing of their own eBusiness endpoints. Instead, they download and install a test server, along with automated Test Suites. On-premise testing avoids external access to an in-production system and reconfiguration of the firewall. It provides the convenience of local control of the test environment. It requires that end-users have the IT expertise to do testing onsite.
- **Remote testing:** In this case, the end-user does not have to handle any test equipment locally, e.g. because of the IT overhead of doing so, or because it wants to test its SUT exactly in its production context (not in an off-production test harness). Testing may be controlled by the user (remotely) or operated by a third party.
- **Combination of remote and on-premise testing:** A combined approach is appropriate if end-users want to decouple test execution from test analysis. For example, test driving may be local on the user premises, whereas test analysis may rely on remote services.
- **Testing workshops:** In this case, a testing workshop is organized with different Test Participants.

Testing topology:

- Direct connection of systems (point-to-point)
- Mediation via business hub
- Mediation via testing hub

14.3 Deriving Test Scenarios and Solutions

The GITB Methodology for creating testing solutions for eBusiness Specifications relies on the step-wise approach presented in the previous section. Ideally, different Test Scenarios are performed sequentially, starting with standalone document validation (Test Scenario 1) and goes on to interactive Conformance Testing (Test Scenario 2) and Interoperability Testing (Test Scenario 3). The following table describes how the three test scenarios differ in terms of Verification Scope and integration in the business environment.

Verification Scope		**Manual testing / no interaction with SUT**	**Interaction with SUT**	**Interactions between SUTs**
Messaging layer	• Structural validation • Quality of service (QoS) validation • Address discovery • Others			Test scenario 3
Business Document layer	• Structural validation • Semantic validation • Others layer	Test scenario 1	Test scenario 2	
Business Process layer	• Structural validation • Cross-layer / profile validation			

Table 13-4: Testing Scenarios, Requirements and Integration in Business Environment

To setup the Testing Architecture and the Test Bed for realizing the test scenarios, Test Designers and Test Managers will search for existing Testing Resources and Artifacts using the TRR. If no existing resources are available, they will have to create the necessary Testing Capability components and artifacts. If a GITB-compliant Test Bed is available, it will provide the non-core components and be used as testing platform. The required Testing Capabilities can then be implemented as plug-in components.

Part IV. 1: Public Procurement

15 OpenPEPPOL

15.1 Background and Testing Requirements

With more than 70 Access Point service providers in Europe, the OpenPEPPOL[36] community is growing and gaining users in Europe. Some countries have mandated its use and are the tractor for private and public entities around the EU. Other countries are still looking at this open network infrastructure that enables the interconnection between public entities and private companies to drive electronic public procurement. Besides electronic public procurement, OpenPEPPOL is more and more being used in the private sector to exchange structured documents not only with public entities but also with other private companies.

In order to ensure interoperability, different service providers and Regional Authorities have implemented validation services. We have different examples:

- Norwegian DIFI has created a website for validation of document instances, for example http://vefa.difi.no/formatvalidering/invoice-validation-en.html.

- Private providers offer free validation services for PEPPOL instances, for example https://peppol.validex.net/.

Most of these existing validation services use Test Artifacts to ensure the electronic documents that have to be exchanged over the PEPPOL network are conformant to the OpenPEPPOL specifications. With so many Access Point service providers and users, the OpenPEPPOL community has the challenge to ensure that every document exchanged follows the OpenPEPPOL specifications; therefore providing a test service to validate electronic documents is key to promote interoperability.

Apart from conformance to the document specifications, OpenPEPPOL is currently facing another challenge: There has been a decision to move from the START transport protocol, created under the PEPPOL project, to a more common and widely adopted transport protocol called AS2. AS2 transport protocol has been used for several years now. It therefore offers more tools and is more stable than the new protocol developed under the PEPPOL Pilot project. However, moving a community of more than 70 Access Points from one transport protocol to another is not an easy task, and providing tools and services to test for conformance could be a major benefit.

The OpenPEPPOL community has created a **Validation and Quality Assurance project** intended for ensuring its growing community of companies and service providers implement their specifications properly.

The purpose of the Validation and Quality Assurance project is to further clarify and establish clear directions and rules in terms of responsibilities for quality assurance and validation in the OpenPEPPOL network. The project will also, if necessary, point to existing available resources and/or develop new resources (if necessary). Consequently, maintaining validation tools might be in scope for the project. The overarching objective is to allow parties exchanging information in the OpenPEPPOL network the capability to validate electronic documents based on the Business Interoperability Specifications (BIS) in a consistent manner. More complex tasks can be envisaged for the Validation and Quality Assurance project, providing test scenarios for conformance and interoperability testing.

The Test Scenario described in the following sections addresses a complex scenario, combining the exchange of a document instance using AS2 with the document validation. Its deployment into the Global Interoperability Test Bed (GITB) could be a first step to demonstrate how to develop additional Test Scenarios for the OpenPEPPOL community.

[36] http://www.peppol.eu/

15.2 Verification Scope – What Should Be Tested?

The business process that will be used as the basis for this Test Scenario is the submission of an electronic invoice through the OpenPEPPOL network using the AS2 protocol. The Test Scenario will focus on submitting an electronic invoice from a sending Access Point to a receiving Access Point.

15.2.1 Actors

The following actors assume a role in this business process:

- **Seller** – The original issuer of the electronic invoice. The submission of the electronic invoice to the sending Access Point is out of scope for this Test Scenario.

- **Buyer** – The original receiver of the electronic invoice. For the purpose of the test, the buyer will be always the one registered in the GITB SMP.

- **Sending Access Point** – The System Under Test.

- **Receiving Access Point** – Simulated by the GITB, receives electronic invoices in AS2 and validates them according to the BIS 4A rule set.

- **Service Metadata Locator** – Simulated by the GITB, receives a request and provides an URL to the Service Metadata Publisher. A service that provides a client with the capability of discovering the Service Metadata Publisher endpoint associated with a particular participant identifier. A client uses this service in order to find where information is held about services for a particular participant business.

- **Service Metadata Publisher** – Simulated by the GITB, receives a request and provides the AP endpoint. A service metadata publisher offers a service on the network where information about services of specific participant businesses can be found and retrieved. It is necessary for a client application to retrieve the metadata about the services for a target participant business before the client can use those services to send messages to the participant business.

15.2.2 Business Process

The business process has the following steps:

1. The seller **creates** an invoice based on the PEPPOL BIS 4A Business Interoperability Specification.

2. The seller **authenticates** with the sending Access Point and submits the electronic invoice. The authentication process of the seller to the sending Access Point is considered out of scope for this Test Scenario.

3. The sending Access Point **validates** the electronic invoice for conformance to BIS 4A.

4. The sending Access Point looks up the PEPPOL address of the endpoint of the buyer in the SML.

5. Using the SML address, the sending Access Point gets the SMP registry entry.

6. From the SMP entry, the sending Access Point gets the AS2 endpoint of the receiver.

7. The sending Access Point wraps the electronic invoice into a message envelope based on the SBDH specification.

8. The sending Access Point submits the electronic invoice using AS2 to the receiving Access Point.

9. The receiving Access Point **validates** the electronic invoice using the BIS 4A rule set.

15.2.3 Underlying eBusiness Specifications / Standards

The business process is described in the PEPPOL BIS 4A Invoice Only Specification and in the PEPPOL transport profiles and infrastructure specifications.

Table 15-1: OpenPEPPOL Test Scenario – Relevant eBusiness Specifications

	Relevant specifications / standards	References
Business Process	Business Process specification is defined in the PEPPOL BIS 4A. The BIS 4A is based in CEN BII2 Post Award CWA.	• PEPPOL BIS 4A • CWA 16562
Business Documents	UBL Invoice document customized following the CEN BII transaction data model. Attributes and code list defined using Genericode by CEN BII. Business rules defined in schematron by CEN BII2 and PEPPOL.	• UBL Invoice • CEN BII T10 Trdm
Transport and Communication (Messaging) Protocols	Messaging protocols for the PEPPOL network are based on OASIS Busdox Technical Specification. The transport protocols is AS2.	• Busdox • SML Service • SMP Service • RFC 4130R • AS2 PEPPOL • Policy for use of Identifiers • Policy for using envelopes (SBDH)
Profiles	PEPPOL BIS 4A defines the profile and provides test files for the electronic invoice.	• PEPPOL Use Case Test Files

15.3 Testing Environment – How Should Be Tested?

15.3.1 Testing Integration in Business Environment

SUT is a non-production system as these tests can be run in parallel to the development process.

15.3.2 Testing Location

It can be implemented as a web-based remote self-testing tool. The SUT operators can test their system whenever and wherever they want. The SUT Operators connect to the GITB Test Bed, execute the Test Suite/Cases and get the test results.

15.4 Test Scenario

15.4.1 Objectives and Success Criteria

This Test Scenario implements a conformance test of an Access Point to the PEPPOL specifications. The objective of this Test Scenario is to ensure the sending Access Point (the System Under Test) can submit a conformant PEPPOL BIS 4A electronic invoice to a receiving Access Point using the AS2 protocol.

The Access Point has to be able to discover the endpoint for the receiving Access Point based on the information on the electronic invoice header and has to submit the electronic document to this receiving endpoint using the AS2 protocol.

Success criteria:

- The Sending Access Point can obtain the endpoint address of the receiving Access Point
- The Sending Access Point can send the electronic invoice using the AS2 protocol
- The exchanged electronic invoice follows the BIS 4A specifications

15.4.2 Interaction Diagram/Choreography

15.4.2.1 Endpoint Lookup

The sending Access Point has to perform a lookup for the receiver's capabilities and technical endpoint information.

1. An electronic invoice is issued by a PEPPOL user and handed over to the sender Access Point for transportation to the receiving Access Point. The invoice is then finally delivered to the ultimate receiver. The method used to communicate between the user and the sender Access Point is out of scope of the test but the sender Access Point must assure the authenticity of the PEPPOL user and the validity of the electronic invoice message.

2. The message handed over by the user to the sending Access Point includes an envelope with required information such as:

    a. Recipient identifier and identifier type

    b. Sender identifier and identifier type

    c. Document identifier

    d. Process identifier

    These identifiers must follow the PEPPOL Policy on use of Identifiers.

3. The sender Access Point constructs an URL based on the business identifier of the receiver and queries the simulated SML.

4. The sender Access Point gets the address of the simulated SMP.

5. The sender Access Point requests service metadata to the receiver simuilated SMP creating a query with the document identifier and the receiver's identifier.

6. SMP replies with the metadata for the receiver's Access Point.

7. The sender Access Point validates that the metadata is signed using a PEPPOL certificate.

8. The sender Access Point gets the AS2 endpoint from the SMP reply.

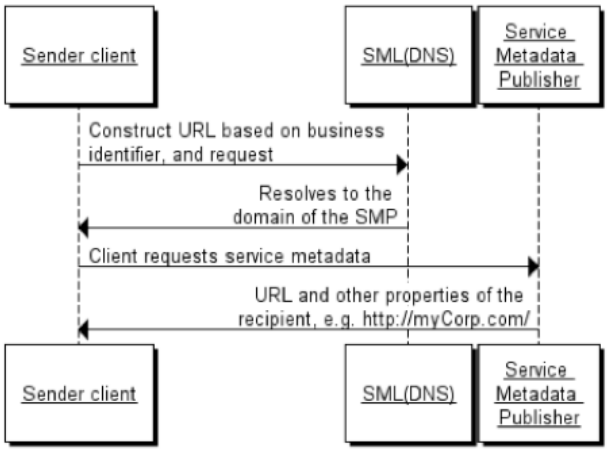

Figure 15-1: Endpoint Lookup

15.4.2.2 Document Exchange

OpenPEPPOL requires using AS2 protocol to exchange documents. The workflow between the sender Access Point and the receiving Access Point is as follows:

1. The sending Access Point gets the OpenPEPPOL issued Private Key X509 certificate for signing from its own certificate stores.

2. The sending Access Point MUST ensure that the message envelope carries the correct headers containing identifiers for recipient and sender, process type and document identifier.

3. The sending Access Point signs the message using the OpenPEPPOL AP Certificate Private Key.

4. The sending Access Point uses HTTPS to send message securely to the receiving simulated Access Point using the URL as retrieved from the SMP and in accordance with AS2 specification RFC 4130.

5. The receiving simulated Access Point responds synchronously with a signed proof-of-delivery message to the sending Access Point using the Message Delivery Notification (MDN) specification as specified in the AS2 specification RFC 4130.

6. Finally the sending Access Point archives the MDN as a signed proof-of-delivery of the message.

15.4.3 System Under Test (s)

The System Under Test (SUT) is the sending Access Point and belongs to a Service Provider. The GITB Test Bed simulates the systems of the OpenPEPPOL Service Metadata Locator (SML), and the Service Metadata Publisher (SMP) and the receiving Access Point.

15.4.4 Abstract Test Steps

This Test Scenario tests the conformance of a sending Access Point to the OpenPEPPOL specifications. It discovers the endpoint address of the buyer based on its recipient endpoint identifier, submits the electronic invoice using the AS2 protocol, and validates whether the electronic invoice is correct.

- The seller prepares a compliant PEPPOL BIS 4A electronic invoice.
- The seller authenticates with the sending Access Point and submits the electronic invoice.
- The sending Access Point validates the electronic invoice for conformance to BIS 4A.

- The sending Access Point looks up for the endpoint of the buyer in the simulated SML.
    - The SUT retrieves the buyer endpoint identifier from the electronic invoice and performs a DNS lookup into the GITB SML
        - **Conformance criteria 1 – Request is well formed**
        - How to test – The DNS lookup has to be done for a specific receiver.
- Using the SML address, the sending Access Point gets the SMP registry entry.
    - The SUT accesses the simulated SMP and retrieves the information about the protocol and endpoint where the electronic invoice has to be delivered.
        - **Conformance criteria 2 – Request is well formed**
        - How to test – Check URI request of the SMP record (e.g. http://smp.b2brouter.com/complete/iso6523-actorid-upis::9920:ESB63276174)
- From the SMP entry, the sending Access Point gets the AS2 endpoint of the receiver, wraps the electronic invoice with a SBDH envelope and submits the envelope using AS2 to the receiving Access Point.
    - The SUT creates the AS2 header and submits the electronic invoice using the AS2 protocol to the receiving Access Point
        - **Conformance criteria 3 – Header well formed**
        - How to test – The SBDH envelope has the correct format and the proper From and To fields
        - **Conformance criteria 4 – Valid electronic invoice document format**
        - How to test – Use the UBL XSD to check the syntax of the electronic invoice
        - **Conformance criteria 5 – The contents of the electronic invoice is valid**
        - How to test – The test bed receives the electronic invoice and performs the validation according to the PEPPOL BIS 4A validation artifacts.

15.5 Related Existing Test Artifacts/Tools/Services to Reuse in the Domain

15.5.1 Test Artifacts

The electronic invoice can be tested using the following Test Artifacts:

- UBL XSD Invoice shema:
    - UBL-Invoice-2.1.xsd
- CEN BII Transaction 10 schematron validation :
    - http://www.invinet.org/BII2conformance/BII2-resources/xslt/BIIRULES-UBL-T10.xsl
    - http://www.invinet.org/BII2conformance/BII2-resources/xslt/BIICORE-UBL-T10-V1.0.xsl
- PEPPOL BIS4a schematron validation

15.5.2 Test Tools and Services

There are several test services implementing the test artifacts described in the section above, but we have not found services or tools that implement testing for the transport of the documents using AS2.

15.6 Related Stakeholders

Standard Development Organizations (SDOs), industry consortia, companies, public authorities that may be interested to use the tests are the following:

- Industry consortia
    - OpenPEPPOL AISBL
    - EESPA
- Private companies
    - Service Providers
    - ERP Vendors

16 eSENS

16.1 Background and Testing Requirements

The aim of the e-SENS large-scale project is to develop the idea of the European Digital Market through innovative ICT solutions and consolidates, improves and extends experiences in previous large-scale pilots with the objective of facilitating cross-border processes.

The former large-scale projects are:

- SPOCS (Simple Procedures Online for Cross Border Services)[37]
- e-CODEX (e-Justice Communication via Online Data Exchange)[38]
- epSOS (European patient Smart Open Services)[39]
- PEPPOL (Pan European Public Procurement Online)[40]
- STORK (Secure idenTity acrOss boRders linKed)[41]

The e-SENS large-scale pilot has been organized into six core work packages:

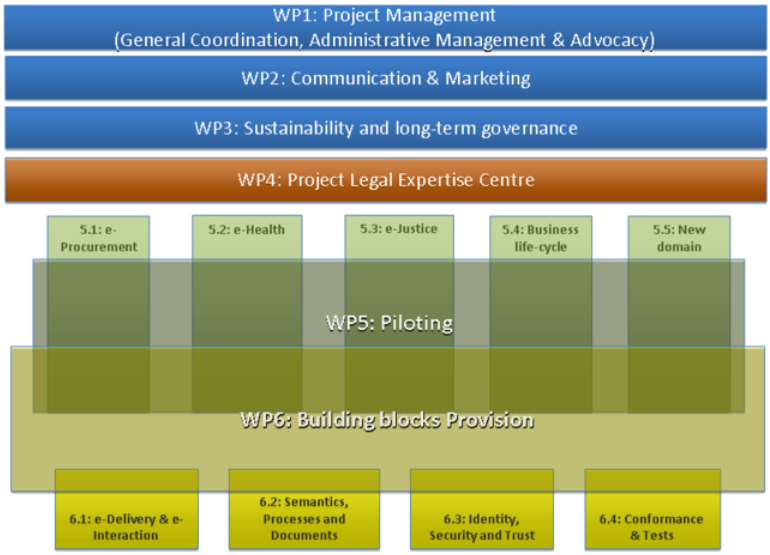

Figure 16-1: e-SENS Work Packages

There are four non-technical (general coordination and communication) and two technical-oriented work packages.

Work package 5 objectives are to demonstrate how to deploy real-life ICT services within European countries, and work package 6 shall create the technical building blocks for these pilots to be deployed.

37 www.eu-spocs.eu

38 www.e-codex.eu

39 www.epsos.eu

40 www.peppol.eu

41 www.eid-stork.eu and www.eid-stork2.eu

There are several domains for piloting projects (e-Procurement, e-Health, e-Justice and Business Lifecycle) in e-SENS. This testing scenario will be focused on the e-Procurement domain.

Within the e-Procurement domain, e-SENS stakeholders have suggested several pilots. The Test Scenario to define and deploy in the Global Interoperability Test Bed (GITB) is related with the pre-award area for public procurement. The Test Scenario will be focused on the subscription process, where an economic operator discovers a business opportunity and subscribes his interest for the contracting authority to send him the tender documents electronically.

e-SENS work package 6 has an specific requirement to create a conformance and test building block. Their aim is to provide an extensible, highly available and web based testing infrastructure in order to ensure interoperability conformance of the applications and organizations participating in e-SENS.

This Test Scenario and its deployment into the GITB framework does not compete with the e-SENS work package 6. On the contrary, it can be used as a template or initial work to develop additional Test Scenarios for other pilot projects in the e-Procurement or other domains within the e-SENS Large Scale Pilot. The use of an existing global interoperability Test Bed like the one being developed in the CEN WS GITB can be encouraged reusing the Test Scenario templates. This could potentially simplify the tasks for the different e-SENS domains when creating Test Scenarios and could also provide a common and interoperable set of artifacts to allow the deployment of such Test Scenarios in different Test Beds.

16.2 Verification Scope – What Should Be Tested?

As described below, the business process pilot that will be used to create this Test Scenario is the subscription of interest in a call for tender from the economic operator to the contracting authority.

16.2.1 Actors and Roles

The following actors participate in this business process:

- **Customer:** The customer is the legal person or organization who is in demand of a product or service. Examples of customer roles: buyer, consignee, debtor and contracting authority.

- **Supplier:** The supplier is the legal person or organization that provides a product or service. Examples of supplier roles: seller, consignor, creditor, and economic operator.

These actors play the following roles in this business process.

- **Contracting authority (CA):** 'Contracting authorities' means the state, regional or local authorities, bodies governed by public law, associations formed by one or several of such authorities or one or more such bodies governed by public law.

- **Economic operator (EO):** The terms 'contractor', 'supplier' and 'service provider' mean any natural or legal person or public entity or group of such persons and/or bodies which offers on the market, respectively, the execution of works and/or a work, products or services. The term 'economic operator' shall cover equally the concepts of contractor, supplier and service provider.

16.2.2 Business Process

The business process is described in the e-SENS work package D5.1 deliverable and in the CEN BII3 Profile 46. The objective of the pilot is to demonstrate how an economic operator can subscribe interest to a tender published by a contracting authority in a foreign country. The business process has the following steps:

10. CA[42] **sends** a notice to the Publisher

11. Publisher **receives** the notice and sends an acknowledgement back to the CA

[42] Contracting Authority, the public entity that is willing to purchase products, services or works.

12. EO[43] **starts a search** in the Publisher's site
13. Publisher **finds notices** meeting EO's criteria and sends the results back to the EO
14. EO **expresses his interest** to one procurement submitting a subscription request to the CA
15. CA receives the subscription request to the procurement by the EO
16. CA **subscribes** the interested EO and sends him a subscription response as an acknowledgement
17. EO **receives** the subscription response

16.2.3 Underlying eBusiness Specifications / Standards

Relevant specifications comprise the e-SENS work package D5.1 deliverable and the CEN BII3 Profile 46.

Table 16-1: e-SENS Test Scenario – Relevant eBusiness specifications

	Relevant specifications / standards	**References**
Business Process	• Business Process specified in the Profile 46 – Subscribe to procedure in CEN BII3	• Draft CEN BII3 Profile to be published as a formal standard in CEN BII.
Business Documents	• e-SENS Specification: WP5.1 Deliverable • Current work in CEN Business Interoperability Interfaces 3 • XVergabe. The documents that will be used are the ones from the XVergabe initiative, according to the CEN BII T81 and T82 information requirement models	• D5.1 Information Requirements eTendering • XVergabe Messages o Subscription request o Subscription response • CEN BII information models o T81 Expression of interest o T82 Business Opportunity subscription confirmation
Transport and Communication (Messaging) Protocols	This test scenario does not test the communication between the parties	
Profiles	CEN BII3 - Profile 46	

[43] Economic Operator, the private company that is willing to sell products, services or works.

16.3 Test Scenario

This Test Scenario will check the conformance of an Economic Operator system participating in the e-Tendering e-SENS pilot. The Test Scenario will validate the contents of the submitted subscription request as well as the choreography of the Economic Operator system under test.

16.3.1 Objectives and Success Criteria

This Test Scenario implements a conformance test for the document exchange defined in the e-SENS D5.1 Information Requirements for eTendering following the Profile 46 established in CEN BII. It is not a complete test scenario for the whole pilot, but an initial part to test the electronic document structure and associated business rules for the subscription request document.

The scope of the test is limited to the subscription request and response messages. There are no bindings of these two information requirement models to existing international standard XML languages such as UBL or UN/CEFACT, and this is why CEN BII does not provide any binding for these transactions. The e-SENS project team, though, has been working jointly with the XVergabe initiative from Germany, and as they have a syntax that can support these two information models, this Test Case will use this syntax.

Success criteria:

1. The correct sequence of the messages as defined in the CEN BII Profile 46
2. Validity of the syntax or structure of the documents being exchanged according to XVergabe syntax
3. Validity of the business rules specified in CEN BII Profile 46 and transaction T81 and T82.

16.3.2 Interaction Diagram/Choreography

The business process activity diagram defined in the CEN BII Profile 46 is as follows:

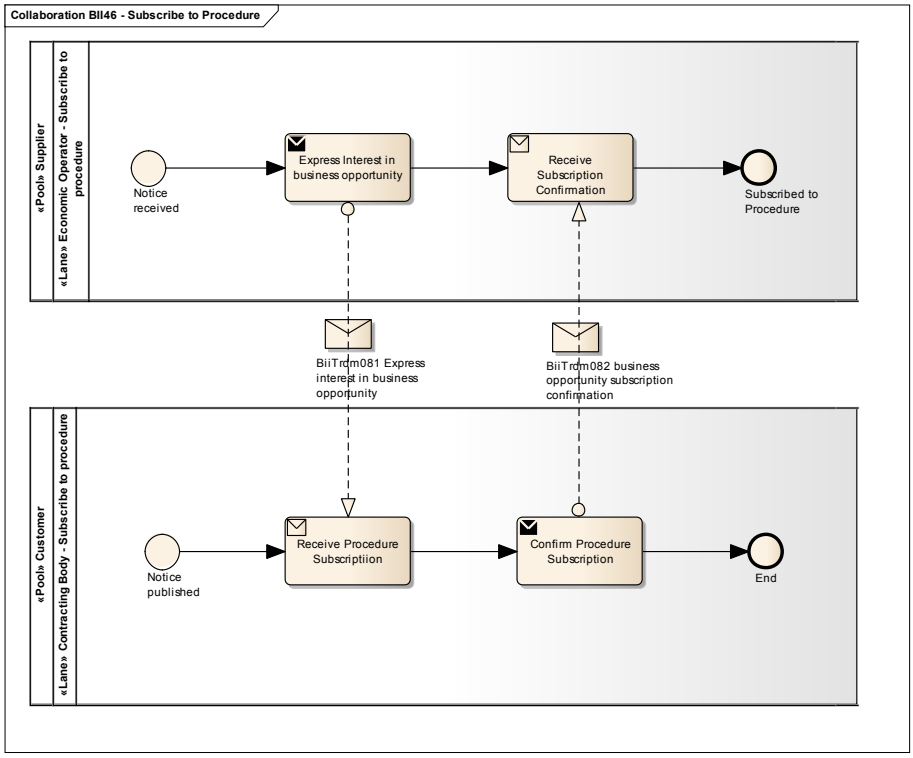

Figure 16-2: CEN BII3 Profile 46 – Subscribe to Procedure

16.3.3 System Under Test (s)

This Test Scenario is used to test the system of the Economic Operator. SUTs are non-production systems as these tests can be run in parallel to the development process.

The GITB simulates the Contracting Authority and the Publisher systems.

16.3.4 Abstract Test Steps

This Test Scenario will check the conformance of an Economic Operator system participating in the e-Tendering e-SENS pilot. The Test Scenario will validate the contents of the submitted subscription request as well as the choreography of the Economic Operator system under test.

- CA sends a notice to the Publisher
- Publisher receives notice and sends acknowledgement to CA
- EO sends a search request to the Publisher
- Publisher searches notices
- Publisher sends a set of notices as a result to the EO
- EO shows his interest in one of the received notices
    - The Economic Operator has interest in one of the received notices and the SUT creates a “subscription request” transaction with the reference number of the notice.
        - **Conformance criteria 1 – The document is well formed**
        - How to test – Check document validity with XSD
        - **Conformance criteria 2 – The document is valid**
        - How to test – The electronic document is valid according to the business rules defined in CEN BII for the subscription of interest transaction.
- CA subscribes interest EO and sends acknowledgement to EO including all documents
- CA sends information updates of the procurement project including documents to all interested EO's whenever there are changes in the procurement process
- EO sends his tender for the procurement project to CA
    - The SUT receives acknowledgment from the test bed and creates a tender document.
        - **Conformance criteria 3 – Correlation.**
        - How to test – The reference number has to be in the list of the business opportunities sent from the test bed.
        - **Conformance criteria 4 – The document is well formed**
        - How to test – Check document validity with XSD.

- **Conformance criteria 5 – Data contents, for instance the submission date and time shall be the one of the transaction, or the hash of the document has to be valid.**
- How to test – Apply schematron and code list validation on specified data elements.
- **Conformance criteria 6 – Security check**
- How to test – Validate electronic signature of the tender document.

16.4 Related Existing Test Artifacts/Tools/Services to Reuse in the Domain

There is no syntax binding from the CEN BII information requirements to the XVergabe syntactical electronic documents. There are also no Business Rules identified for these two information requirement models in the CEN BII Profile 46 yet.

Currently, the CEN BII Profile 46 and related transaction models are being reviewed internally in CEN BII pre-award team.

As per the policy on syntax bindings from CEN BII, the Workshop does not create any other than UBL and UN/CEFACT.

For that reason, the syntax binding to the XVergabe will have to be created within the e-SENS pilot project.

16.4.1 Test Artifacts

Currently the following artifacts already exist:

- XSD Schema for the Subscription Request document from XVergabe
- XSD Schema for the Subscription Response document from XVergabe
- CEN BII3 T81 Expression of interest information requirement model (Draft)
- CEN BII3 T82 Business Opportunity subscription confirmation information requirement model (Draft)

16.4.2 Test Tools and Services

Currently there are no tools or services for these transactions.

16.5 Related Stakeholders

- Industry consortia
  - e-SENS
  - XVergabe
  - CEN WS BII
- Public institutions
  - European Commission
  - Publications Office

17 Connecting Europe Facility (CEF)

17.1 Background and Testing Requirements

Connecting Europe Facility (CEF) is the common financing instrument of trans-European networks for the period 2014-2020. During this period, CEF will finance projects of common interest in three different sectors:

- Transport
- Energy
- Telecommunications

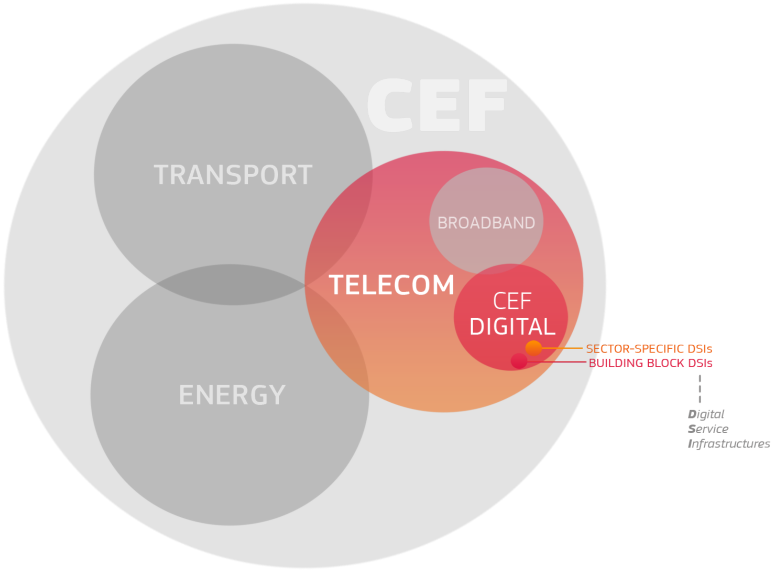

Figure 17-1: CEF Structure

Within the telecommunications area, CEF has a budget to work on Digital Service Infrastructures delivering networked cross-border services for citizens, businesses and public administrations.

The objective of the CEF Programme is to improve the competitiveness of the European economy by promoting the interconnection and interoperability, thus supporting the development of a Digital Single Market.

The aim is to promote these key Digital Service Infrastructures (DSIs) in order to facilitate the cross-border and cross-sector interaction in Europe. One of the building blocks of the CEF programme is the e-Invoicing DSI. This building block will help public administrations implement electronic invoicing in compliance with the e-Invoicing Directive of the European Parliament and the Council.

The European Committee for Standardization (CEN) is defining a new semantic standard for e-Invoicing in public procurement and the binding of the resulting standard to a number of existing syntaxes in a Project Committee known as PC 434.

The e-Invoicing solution of CEF should provide tools for the public administrations to reduce the efforts for complying with the Directive.

This Test Scenario aims at providing a GITB-compliant Test Case to allow the validation of electronic invoices created using different syntaxes against the PC 434 semantic standard for e-Invoicing.

17.2 Verification Scope – What Should Be Tested?

17.2.1 Actors

The following actors participate in the business process:

- **Customer:** The customer is the legal person or organization who is in demand of a product or service. Examples of customer roles: buyer, consignee, debtor and contracting authority.

- **Supplier:** The supplier is the legal person or organization that provides a product or service. Examples of supplier roles: seller, consignor, creditor, and economic operator.

These two parties take the following roles:

- **Creditor:** One to whom a debt is owed. The Party that claims the payment and is responsible for resolving billing issues and arranging settlement. The Party that sends the Invoice. Also known as Invoice Issuer, Accounts Receivable, or Seller.

- **Debtor:** One who owes debt. The Party responsible for making settlement relating to a purchase. The Party that receives the Invoice. Also known as Invoicee, Accounts Payable, Buyer

17.2.2 Business Process

The business process activity diagram defined in the PEPPOL BIS 4a can be used to depict the choreography of the submission of an electronic invoice, although this Test Case does not test the business process but the conformance of the transaction to the CEN PC 434 semantic model.

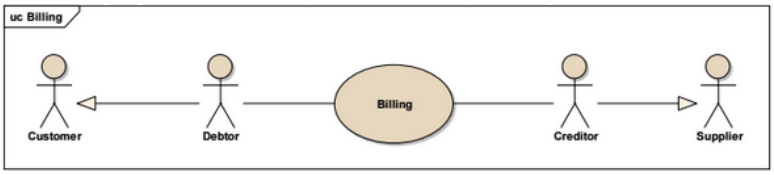

Figure 17-2: PEPPOL BIS4a - Invoice Only

17.2.3 Underlying Standards/Specifications

Directive 2014/55/EU of the European Parliament and of the Council of 16 April 2014 on electronic invoicing in public procurement states that the European "Commission shall request that the relevant European standardisation organisation draft a European standard for the semantic data model of the core elements of an electronic invoice. Based on this standardization request from the European Commission, the CEN Project Committee 434 was created in 2014-05-06. The Work on PC 434 has been divided into several work streams (WS):

- WS1 Definition of scope.
- WS2 Semantic model.
- WS3 External relations
- WS4 List of syntaxes
- WS5 Syntax binding
- WS6 Guidelines at transmission level
- WS7 Extension methodology
- WS8 Test methodology and test results

The PC 434 will finalize by the end of 2016.

The list of syntaxes and the rest of deliverables are not available yet. As long as there is not an official list of syntaxes, PEPPOL BIS for the electronic invoice will be taken as the basis for this Test Case. In order to demonstrate the potential use of additional syntaxes, the CEN BII syntax binding to the UN/CEFACT Cross Industry Invoice will be also considered.

Once the syntaxes are selected and the syntax bindings defined within PC434, they must substitute the PEPPOL BIS and then CEN BII artifacts that will be used as part of this Test Scenario.

Table 17-1: CEF Test Scenario – Relevant eBusiness specifications

	Relevant specifications / standards	**References**
Business Process	Not applicable	
Business Documents	• UBL - PEPPOL BIS • Cross Industry Invoice	• CEN PC 434 o Draft semantic model • PEPPOL o PEPPOL BIS 4a Schematron Validation tools • CEN BII o CEN BII Syntax Binding to UBL o CEN BII Syntax Binding to CII • UBL 2.1 Invoice XSD • Cross Industry Invoice XSD
Transport and Communication (Messaging) Protocols	This test scenario does not test the communications between the parties.	
Profiles	Not applicable	

17.3 Test Scenario

17.3.1 Objectives and Success Criteria

This Test Scenario implements a conformance test for electronic invoices according to the CEN PC 434 semantic model. Its main objective is that regardless of the syntax used to create the invoice, the Test Service shall identify whether the PC 434 semantic data model is correctly implemented in the electronic invoice instance.

The scope of the test is checking both the compliance to the underlying syntax and to the semantic model as defined by the CEN PC 434. In order to test the underlying syntax, the Test Service must identify it through the root namespace, and once the syntax layer is successfully validated, the corresponding Schematron validation artifact must be used to assess whether there are elements in the document not contained within

the PC 434 semantic model, and whether the existing semantic elements in the document instance fulfil the business rules defined by the PC 434.

Success criteria:

4. The electronic invoice is written using one of the syntaxes accepted by the PC 434
5. The structure of the electronic invoice is valid according to that syntax
6. The semantics of the PC 434 are correctly implemented in the electronic invoice.
7. The elements in the electronic invoice not part of the semantics of PC 434 are identified.

This Test Case is a document conformance test to ensure an XML electronic invoice is valid according to the PC 434 semantic data model. This means that the document XML instance belongs to one of the selected syntaxes, that it is valid according to the syntax Schema, and that contains the elements required by the PC 434 semantic data model.

17.3.2 System Under Test (s)

This Test Case is used to test document instances. A document instance can be provided either by the Customer or the Supplier. The System Under Test (SUT) is the one creating the electronic invoice.

SUTs can be production systems and this test case can be used as a the initial step for a certification process of electronic invoices to the CEN PC 434 European Norm.

17.3.3 Abstract Test Steps

The System Under Test (SUT) produces the electronic invoice document. The Test Bed only validates the document instance, not the business process, therefore there is no test on the communication between both actors.

- The operator submits or uploads the electronic invoice to the GITB-Compliant Test Bed
    - The operator wants to know whether the XML document instance is compliant according to the PC 434.
        - **Conformance criteria 1 – The document belongs to an accepted syntax**
        - How to test – Check namespace for the root document being in the list of accepted syntaxes
        - **Conformance criteria 2 – The document structure is valid**
        - How to test – The electronic document is valid according to the XSD structure of the identified syntax.
        - **Conformance criteria 3 – The electronic invoice is conformant to PC 434**
        - How to test – The electronic invoice is valid according to the CEN PC 434 Schematron semantic model and rules.
        - **Conformance criteria 4 – Identification of additional elements**
        - How to test – Use an Schematron file to identify elements of the XML electronic invoice not defined in the CEN PC 434.

17.4 Related Existing Test Artifacts/Tools/Services to Reuse in the Domain

Currently there are no normative artifacts issued by CEN PC434 that can be used to perform these tests. Besides, there is not an official list of accepted syntaxes and versions yet.

PEPPOL BIS should be taken as the basis for this test, therefore, the syntax bindings and artifacts used in PEPPOL to check the electronic invoice document will be used to implement a first release of this Test Case.

Additionally, the CEN BII2 has a syntax binding to the UN/CEFACT CII 3.0 and there are validation artifacts that will also be implemented to perform this Test Case.

The artifacts issued by the PC 434, once they become published, shall substitute these interim artifacts.

17.4.1 Test Artifacts

Currently the following artifacts will be used:

- UBL XSD Schema for the UBL Invoice
- UN/CEFACT XSD Schema for the Cross Industry Invoice
- CEN BII2 T10 Invoice information requirement model (to be substituted by the CEN PC 434 semantic data model)
- PEPPOL BIS 4a Validation Package
- CEN BII2 T10 CII Core Business Rules (to be substituted by the CEN PC 434 Core rules)
- CEN BII2 T10 CII Business Rules (to be substituted by the CEN PC 434 Business Rules)

17.4.2 Test Tools and Services

This Test Case is implemented in a free GITB-Compliant Test Bed service called Validex.net (https://validex.net).

17.5 Related Stakeholders

- Industry consortia
    - OASIS UBL
    - UN/CEFACT CII
    - OpenPEPPOL AISBL
- Public institutions
    - CEF
    - CEN PC 434
    - CEN WS BII
    - European Commission

18 Electronic Invoicing for the National Health Service (NHS)

18.1 Background and Testing Requirements

The National Health Service of UK (NHS) has made the strategic decision to adopt implement electronic invoicing base on the specification of the PEPPOL project. NHS has carried out proof of concept testing and used the GITB Test Bed for Interoperability testing and message conformance testing, respectively. The objective of NHS is to request XML electronic invoices from its suppliers, including manufacturers of medical tools and supplies.

18.2 Verification Scope – What Should Be Tested?

18.2.1 Actors

The following parties and actors assume a role in the invoicing process.

- **Buyer** – National Health Service of UK as receiver of invoices from various suppliers of medical supplies
- **Seller** – Suppliers, such as manufacturers of medical tools and supplies.
- **Access point operator** – A party operating a PEPPOL access point that may be a seller or buyer or a service provider such as a VAN service.

18.2.2 Business Process

The business process for the test is delivery of invoices the UK NHS by using Peppol specifications for transport and document specifications. It is described in the PEPPOL BIS 4A Invoice Only Specification.

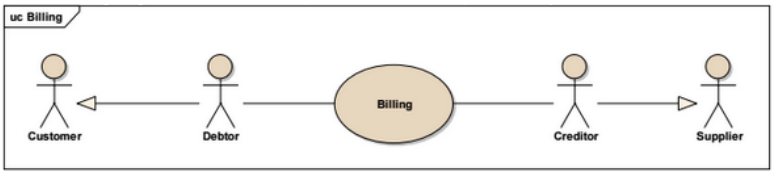

Figure 18-1: PEPPOL BIS4a - Invoice Only

18.2.3 Standards and Specifications

Relevant specifications PEPPOL BIS 4A Invoice Only Specification and in the PEPPOL transport profiles and infrastructure specifications.

Table 18-1: NHS Test Scenario – Relevant eBusiness Specifications

	Relevant specifications / standards	References
Business Process	Business Process specification is defined in the PEPPOL BIS 4A. The BIS 4A is based in CEN BII2 Post Award CWA.	• PEPPOL BIS 4A: http://www.peppol.eu/ressource-library/technical-specifications/post-award/mandatory • CWA 16562
Business Documents	UBL Invoice document customized following the CEN BII transaction data model. Attributes and code list defined using	• UBL Invoice: http://docs.oasis-open.org/ubl/os-UBL-2.1/UBL-2.1.html • CEN BII T10 Trdm

	Genericode by CEN BII. Business rules defined in schematron by CEN BII2 and PEPPOL.	
Transport and Communication (Messaging) Protocols	Messaging protocols for the PEPPOL network are based on OASIS Busdox Technical Specification. The transport protocol is AS2.	• Busdox • PEPPOL SML Service Specification: https://joinup.ec.europa.eu/svn/peppol/PEPPOL_EIA/1-ICT_Architecture/1-ICT-Transport_Infrastructure/13-ICT-Models/ICT-Transport-SML_Service_Specification-101.pdf • PEPPOL SML Service Specification: https://joinup.ec.europa.eu/svn/peppol/PEPPOL_EIA/1-ICT_Architecture/1-ICT-Transport_Infrastructure/13-ICT-Models/ICT-Transport-SMP_Service_Specification-110.pdf • RFC 4130R • AS2 PEPPOL : https://www.ietf.org/rfc/rfc4130.txt, https://joinup.ec.europa.eu/svn/peppol/TransportInfrastructure/ICT-Transport-AS2_Service_Specification-2014-01-15.pdf • Policy for use of Identifiers • Policy for using envelopes (SBDH)
Profiles	PEPPOL BIS 4A defines the profile and provides test files for the electronic invoice.	• PEPPOL Use Case Test Files

18.3 Testing Environment – How Should Be Tested?

18.3.1 Testing Integration in Business Environment

Testing will be used to verify potential senders and authorize them for sending electronic invoices to NHS.

18.3.2 Testing Location

It can be implemented as a web-based remote self-testing tool for interoperability testing. Conformance testing can also be implemented as self-testing tool as well as a real time integration into existing systems.

18.4 Test Scenario

18.4.1 Objectives

The test involves interoperability testing for transfer of XML documents via the PEPPOL infrastructure, followed by a conformance test for an invoice.

NHS followed the following schedule to test transport level interoperability and conformance based on PEPPOL business interoperability and transport specifications:

	Duration	Start Date	End Date
Phase 3 - End-to-end test	**25 days**	**Mon 20/04/15**	**Fri 22/05/15**
Testing using SMP	18 days	Mon 20/04/15	Wed 13/05/15
Testing Message Exchange between Access Point providers	5 days	Mon 20/04/15	Fri 24/04/15
Testing Message Exchange between APs and their clients	5 days	Mon 27/04/15	Fri 01/05/15
Testing workflow	5 days	Mon 27/04/15	Fri 01/05/15
Interoperability testing with all participants (testing 4-corner model)	10 days	Mon 04/05/15	Fri 15/05/15
Full re-run of tests	5 days	Mon 18/05/15	Fri 22/05/15

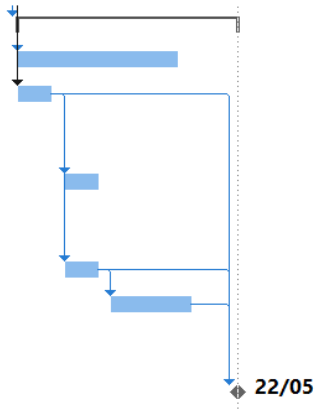

Figure 18-2: Time Schedule

In this timetable, each activity corresponds to a specific testing type and the following table shows the mappings between these testing activities and related testing type.

Table 18-2: Testing Activities and Test Type

Testing Activity	Test Type
Testing using SMP	Conformance Testing
Testing Message Exchange between Access Point providers	Conformance Testing
Testing Message Exchange between APs and their clients	Conformance Testing
Testing workflow	-
Interoperability testing with all participants	Interoperability Testing
Full re-run of tests	-

18.4.2 System under Test (s)

Different test scenarios have been developed for NHS to test the different target SUTs:

- **Sender Access Point.** The test scenario is used to test a senders access point to verify if its interactions with the tranport network conform to the network specifications. An access point is a connection to the Peppol network that allows a party to upload or download messages to the network.
- **Receiver Access Point.** The test scenario is used to test a receivers access point to verify if its interactions with the tranport network conform to the network specifications. An access point is a connection to the Peppol network that allows a party to upload or download messages to the network.
- **An application capable of querying SMLs.** The test verifies if an application correctly queries a Peppol SML. Applications querying the SML are SMP's.
- **An application capable of querying SMPs.** The test verifies if an application correctly queries an Peppol SMP. Applications querying an SMP are Access points (AP).

The following table summarizes the Test Cases that have been developed for NHS for the different target SUTs.

Table 18-3: NHS Test Cases

ID	Name	System Under Test	Test Type	Description
1	PEPPOL-SenderAccessPoint-Order-Validation	Sender Access Point	Conformance Testing	The objective of this Test Scenario is to ensure the Sender Access Point (the System Under Test) can submit a conformant PEPPOL BIS 3A electronic order to a Receiver Access Point (simulated by GITB Engine) over the AS2 protocol. Then submitted document is validated by UBL 2.1 schema and PEPPOL Schematron rules.
2	PEPPOL-SenderAccessPoint-Order-Validex	Sender Access Point	Conformance Testing	The objective of this Test Scenario is to ensure the Sender Access Point (the System Under Test) can submit a conformant PEPPOL BIS 3A electronic Order to a Receiver Access Point (simulated by GITB Engine) over the AS2 protocol. Then submitted document is validated by Validex.
3	PEPPOL-Interoperability-Order	Sender Access Point and Receiver Access Point	Interoperability Testing	The objective of this Test Scenario is to ensure the Sender Access Point (System Under Test) can submit a conformant PEPPOL BIS 3A electronic order to a Receiver Access Point (System Under Test) over the AS2 protocol. Then exchanged document is validated by Validex.
4	PEPPOL-SenderAccessPoint-Invoice-Validation	Sender Access Point	Conformance Testing	The objective of this Test Scenario is to ensure the Sender Access Point (the System Under Test) can submit a conformant PEPPOL BIS 4A electronic invoice to a Receiver Access Point (simulated by GITB Engine) using the AS2 protocol. Then submitted document is validated by UBL 2.1 schema and PEPPOL Schematron rules.
5	PEPPOL-SenderAccessPoint-Invoice-Validex	Sender Access Point	Conformance Testing	The objective of this Test Scenario is to ensure the Sender Access Point (the System Under Test) can submit a conformant PEPPOL BIS 4A electronic invoice to a Receiver Access Point (simulated by GITB Engine) over the AS2 protocol. Then submitted document is validated by Validex tool.
6	PEPPOL-Interoperability-Invoice	Sender Access Point and Receiver Access Point	Interoperability Testing	The objective of this Test Scenario is to ensure the Sender Access Point (System Under Test) can submit a conformant PEPPOL BIS 4A electronic invoice to a Receiver Access Point (System Under Test) over the AS2 protocol. Then exchanged document is validated by Validex tool.
7	PEPPOL-SenderAccessPoint-DespatchAdvice-Validation	Sender Access Point	Conformance Testing	The objective of this Test Scenario is to ensure the Sender Access Point (the System Under Test) can submit a conformant PEPPOL BIS 30A electronic despatch advice to a Receiver Access Point (simulated by GITB Engine) over the AS2 protocol. Then submitted document is validated by UBL 2.1 schema and PEPPOL Schematron rules.

8	PEPPOL-SenderAccessPoint-DespatchAdvice-Validex	Sender Access Point	Conformance Testing	The objective of this Test Scenario is to ensure the Sender Access Point (the System Under Test) can submit a conformant PEPPOL BIS 30A electronic despatch advice to a Receiver Access Point (simulated by GITB Engine) over the AS2 protocol. Then submitted document is validated by Validex tool.
9	PEPPOL-Interoperability-DespatchAdvice	Sender Access Point and Receiver Access Point	Interoperability Testing	The objective of this Test Scenario is to ensure the Sender Access Point (System Under Test) can submit a conformant PEPPOL BIS 30A electronic despatch advice to a Receiver Access Point (System Under Test) over the AS2 protocol. Then exchanged document is validated by Validex tool.
10	SMLClient	An application capable of querying SMLs	Conformance Testing	This Test Scenario implements the lookup interface which enables a Sender Access Point (the System Under Test) to discover participants from Service Metadata Locators (simulated by GITB Engine).
11	SMPClient	An application capable of querying SMPs	Conformance Testing	This Test Scenario implements the lookup interface which enables a Sender Access Point (the System Under Test) to discover services from Service Metadata Publishers (simulated by GITB Engine).

18.4.3 Abstract Test Steps

18.4.3.1 Interoperability Testing

Interoperability testing was carried out as follows.

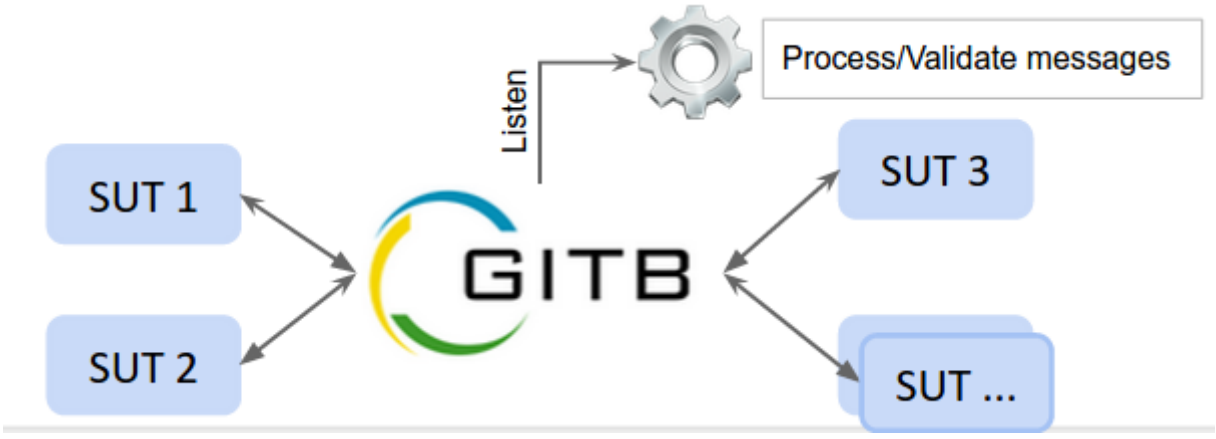

Figure 18-3: Testing Architecture for NHS

Initial efforts of developing GITB Testbed had targeted conformance testing of SUTs first and focused on development of conformance test cases. After NHS's requests, some efforts for developing interoperability testing capabilities have been carried out, as well. These efforts encompass development of **tdl:listen** operation of TDL (Test Description Language), updating messaging architecture for listening message exchanges between SUTs and renewing user interface for managing the interoperation of more than one SUTs.

tdl:listen operation is actually a combination of receive and send operations, respectively, where the testbed receives message from sender SUT actor and sends the received message to receiver SUT actor, therefore "listens" the communication between them. In order to realize this operation, following updates have been performed in GITB Messaging API: development of **IListener** interface to provide methods for listening and transforming messages/configurations, development of **ITransactionListene**r and **IDatagramListener** interfaces to support TCP and UDP based protocols, respectively and implementation **of AbstractTransactionListener** and **AbstractDatagramListener** abstract classes to construct a base implementation for listening and transforming messages/configurations.

As mentioned before, listen operation is a combination of receive and send operations, and so, abstract listeners have to manage two transactions: one for receive and one for send. Furthermore, there may be differences between the structure of the input messages for send operation and the structure of the output messages retrieved from receive operation. Considering that, listen message is a combination of receive and send operations, message received from receive operations must be transformed into the appropriate message format for send operation. Likewise, receive operation configurations must be transformed into appropriate send operation configuration in order to realize the listen operation.

Default implementation of GITB Messaging API provides a number of listeners for the following network protocols: UDP, TCP, HTTP, HTTPS, SOAP and AS2. In order to extend the capabilities of GITB Testbed for listening different types of network protocols, an appropriate listener should be selected as base class and extended according to protocol requirements.

Last but not least, in order to manage the interoperability testing among SUTs, user interface has been enhanced to manage the interoperability sessions. When an interoperability test case is selected to be executed, the SUT operator is provided two options: either creating a new interoperability session or joining an existing one. In either way, testing can not start unless all SUT operators send their configurations regarding their SUTs to test engine. After that, all the operators can track the test execution at the same time.

18.4.3.2 Interoperability and Conformance Test Cases for 3 Document Types

Before the implementation of the interoperability testing there was only one type of test cases, i.e. conformance test cases. Actually, no distinction was made between different types of test cases at all, and test cases were assumed to be conformance test cases. In order to differentiate them from each other, **TestCaseType** enumeration (0 for conformance, 1 for interoperability test cases) as well as type attribute of type TestCaseType in gitb:Metadata element have been introduced in the GITB Core Test Bed Specifications.

Additionally, 6 test cases were developed for 3 different document types (Order, Invoice and Despatch Advice) in order to validate their conformance to PEPPOL BIS (Business Interoperability Specification): Per document type, 1 test case addresses interoperability and 5 test cases address conformance, resulting in a total of 18 test cases. Conformance test cases vary from each other by the validation methods used (PEPPOL BII Rules and Schemas, Validex tool and etc.) and different transport level specifications (inclusion of BusDox (SMLs and SMPs) architecture or not).

An example interoperability test case can be seen below. It should be noted that, there are no actors simulated by the test engine.

```
<?xml version="1.0" encoding="UTF-8"?>
<testcase id="PEPPOL-Interoperability-Invoice" xmlns="http://www.gitb.com/tdl/v1/"
xmlns:gitb="http://www.gitb.com/core/v1/">
  <metadata>
    <gitb:name>PEPPOL-Interoperability-Invoice</gitb:name>
    <gitb:type>INTEROPERABILITY</gitb:type>
    <gitb:version>0.1</gitb:version>
    <gitb:description>The objective of this Test Scenario is to ensure the Sender Access Point (the System Under
      Test) can submit a conformant PEPPOL BIS 4A electronic invoice to a Receiver Access Point using the AS2
      protocol. Then submitted document is validated by Validex.
    </gitb:description>
  </metadata>
  <namespaces>
  </namespaces>
  <imports>
  </imports>
  <actors>
```

```
    <gitb:actor id="SenderAccessPoint" name="SenderAccessPoint" role="SUT" />
    <gitb:actor id="ReceiverAccessPoint" name="ReceiverAccessPoint" role="SUT"/>
    </gitb:actor>
  </actors>
  <variables>
  </variables>
  <steps>


    <btxn from="SenderAccessPoint" to="ReceiverAccessPoint" txnId="t1" handler="PeppolAS2Messaging"/>
    <listen id="as2_output" desc="Sender Access Point sends Invoice document to Receiver Access Point"
from="SenderAccessPoint" to="ReceiverAccessPoint" txnId="t1" >
       <config name="document.identifier">urn:oasis:names:specification:ubl:schema:xsd:Invoice-
2::Invoice##urn:www.cenbii.eu:transaction:biitrns010:ver2.0:extended:urn:www.peppol.eu:bis:peppol4a:ver2.0::2.1</config>
       <config name="process.identifier">urn:www.cenbii.eu:profile:bii04:ver2.0</config>
    </listen>


    <listen desc="Receiver Access Point sends MDN back to Sender Access Point" from="ReceiverAccessPoint"
to="SenderAccessPoint" txnId="t1" />
    <etxn txnId="t1"/>


    <verify handler="ValidexValidator" desc="Validate Invoice document using the Validex validation service">
       <input name="document">$as2_output{business_message}</input>
       <input name="name">"Invoice document"</input>
    </verify>
  </steps>
</testcase>
```

Figure 18-4: Sample Interoperability Test Case

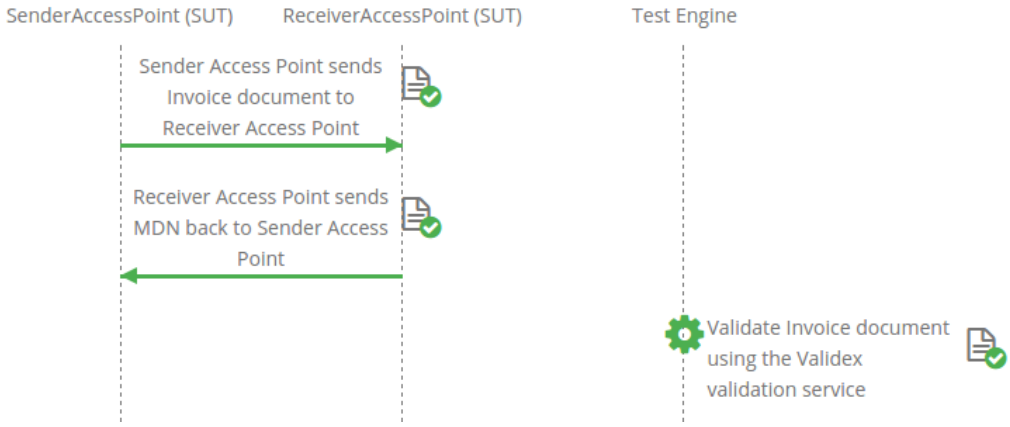

Figure 18-5: Interactions in Interoperability Test Case

An example for conformance test case can be seen below. It should be noted that, there **are** actors simulated by test engine.

```
<?xml version="1.0" encoding="UTF-8"?>
<testcase id="PEPPOL-SenderAccessPoint-Invoice-BusDox-Validex" xmlns="http://www.gitb.com/tdl/v1/"
xmlns:gitb="http://www.gitb.com/core/v1/">
  <metadata>
    <gitb:name>PEPPOL-SenderAccessPoint-Invoice-BusDox-Validex</gitb:name>
    <gitb:type>CONFORMANCE</gitb:type>
    <gitb:version>0.1</gitb:version>
    <gitb:description>The objective of this Test Scenario is to ensure the Sender Access Point (the System Under
       Test) is capable of querying both SML and SMP as well as submitting a conformant PEPPOL BIS 4A electronic
       invoice to a Receiver Access Point using the AS2 protocol. Then submitted document is validated by Validex.
    </gitb:description>
  </metadata>
  <namespaces>
  </namespaces>
  <imports>
    <artifact type="schema" encoding="UTF-8"
name="UBL_Invoice_Schema_File">Peppol_BIS_4A_Invoice/artifacts/UBL/maindoc/UBL-Invoice-2.1.xsd</artifact>
    <artifact type="schema" encoding="UTF-8" name="PEPPOL_BII_CORE_Invoice_Schematron_File"
>Peppol_BIS_4A_Invoice/artifacts/PEPPOL/BII CORE/BIICORE-UBL-T10-V1.0.sch</artifact>
    <artifact type="schema" encoding="UTF-8"
name="PEPPOL_BII_RULES_Invoice_Schematron_File">Peppol_BIS_4A_Invoice/artifacts/PEPPOL/BII
RULES/BIIRULES-UBL-T10.sch</artifact>
    <artifact type="schema" encoding="UTF-8" name="SBDH_Schematron_File"
>Peppol_BIS_4A_Invoice/artifacts/PEPPOL/SBDH.sch</artifact>
    <artifact type="object" encoding="UTF-8"
name="SMP_Metadata_Template">Peppol_BIS_4A_Invoice/artifacts/PEPPOL/peppol-smp-metadata-
template.xml</artifact>
  </imports>
  <actors>
```

```
    <gitb:actor id="SenderAccessPoint" name="SenderAccessPoint" role="SUT" />
    <gitb:actor id="ReceiverAccessPoint" name="ReceiverAccessPoint" role="SIMULATED">
      <gitb:endpoint name="as2">
        <gitb:config name="participant.identifier">0088:12345test</gitb:config>
      </gitb:endpoint>
    </gitb:actor>
    <gitb:actor id="ServiceMetadataLocator" name="ServiceMetadataLocator" role="SIMULATED" />
    <gitb:actor id="ServiceMetadataPublisher" name="ServiceMetadataPublisher" role="SIMULATED" />
  </actors>
  <variables>
    <var name="as2_address" type="string" />


    <!-- Participant Identifier of Sender Access Point (System Under Test). Must be retrieved
        from SUT representative -->
    <var name="sender_participant_identifier" type="string" />
    <!-- Participant Identifier of Receiver Access Point (Simulated) -->
    <var name="receiver_participant_identifier" type="string" />
    <!-- Represents the type of document that the recipient is able to handle -->
    <var name="document_identifier" type="string">
      <value>urn:oasis:names:specification:ubl:schema:xsd:Invoice-
2::Invoice##urn:www.cenbii.eu:transaction:biitrns010:ver2.0:extended:urn:www.peppol.eu:bis:peppol4a:ver2.0::2.1</value>
    </var>
    <!-- Root namespace of the document identifier  -->
    <var name="document_identifier_ns" type="string">
      <value>urn:oasis:names:specification:ubl:schema:xsd:Invoice-2</value>
    </var>
    <!-- The identifier of the process -->
    <var name="process_identifier" type="string">
      <value>urn:www.cenbii.eu:profile:bii04:ver2.0</value>
    </var>
    <!-- XML local element name of the root-element in the business message -->
    <var name="business_message_type" type="string">
      <value>Invoice</value>
    </var>


  </variables>
  <steps>
    <assign to="$as2_address">concat("https://", $SenderAccessPoint{ReceiverAccessPoint}{network.host}, ":",
$SenderAccessPoint{ReceiverAccessPoint}{network.port})</assign>
    <assign to="$receiver_participant_identifier"
source="$SenderAccessPoint{ReceiverAccessPoint}{participant.identifier}" />
    <assign to="$sender_participant_identifier"   source="$SenderAccessPoint{participant.identifier}" />


    <btxn from="SenderAccessPoint" to="ServiceMetadataLocator" txnId="t3" handler="SMLMessaging"/>
    <receive id="sml_output" desc="Locate SMP" from="SenderAccessPoint" to="ServiceMetadataLocator"
txnId="t3">
      <config name="dns.domain">B-351cd3bce374194b60c770852a53d0e6.iso6523-actorid-upis.localhost.</config>
    </receive>
    <send desc="Resolve SMP domain" from="ServiceMetadataLocator" to="SenderAccessPoint" txnId="t3">
      <input name="dns.address" source="$SenderAccessPoint{ServiceMetadataPublisher}{network.host}"/>
    </send>
    <etxn txnId="t3"/>


    <btxn from="SenderAccessPoint" to="ServiceMetadataPublisher" txnId="t2" handler="SMPMessaging"/>
    <receive id="smp_output" desc="Send message to SMP to get Receiver Access Point address"
from="SenderAccessPoint" to="ServiceMetadataPublisher" txnId="t2" />
    <send id="smp" desc="Send SMP Metadata back"  from="ServiceMetadataPublisher" to="SenderAccessPoint"
txnId="t2">
      <input name="smp_metadata" source="$SMP_Metadata_Template"/>
    </send>
    <etxn txnId="t2"/>


    <btxn from="SenderAccessPoint" to="ReceiverAccessPoint" txnId="t1" handler="PeppolAS2Messaging"/>
    <receive id="as2_output" desc="Send message to Receiver Access Point" from="SenderAccessPoint"
to="ReceiverAccessPoint" txnId="t1" >
      <config name="document.identifier">urn:oasis:names:specification:ubl:schema:xsd:Invoice-
2::Invoice##urn:www.cenbii.eu:transaction:biitrns010:ver2.0:extended:urn:www.peppol.eu:bis:peppol4a:ver2.0::2.1</config>
      <config name="process.identifier">urn:www.cenbii.eu:profile:bii04:ver2.0</config>
    </receive>
    <send id="mdn" desc="Send MDN back to Sender Access Point"  from="ReceiverAccessPoint"
to="SenderAccessPoint" txnId="t1">
      <input name="http_headers" source="$as2_output{http_headers}" />
    </send>
    <etxn txnId="t1"/>


    <verify handler="ValidexValidator" desc="Validate Invoice document using the Validex validation service">
      <input name="document">$as2_output{business_message}</input>
      <input name="name">"Invoice document"</input>
    </verify>
  </steps>
</testcase>
```

Figure 18-6: Sample Conformance Test Case

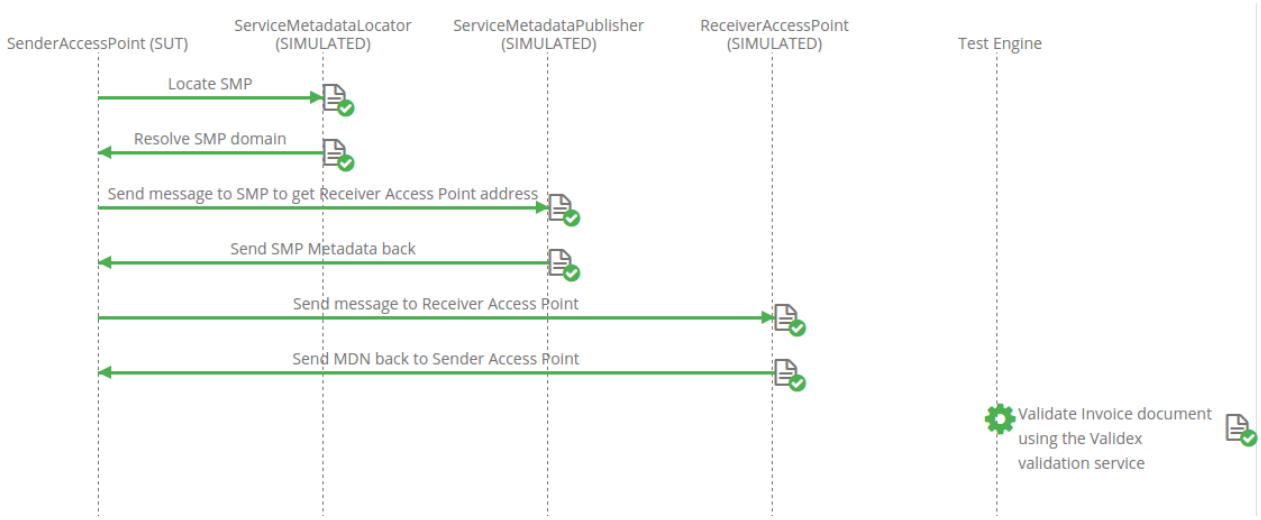

Figure 18-7: Interaction in Conformance Test Case with Actors Simulated by the Test Bed

18.4.3.3 SML and SMP Test Cases

NHS has requested development of test cases that target only the testing of sending SML and SMP requests. SUTs pass the tests if they can successfully send DNS and HTTP queries to SML and SMPs (respectively) simulated by test engine, and receive their responses.

An example for conformance test case that targets SML querying can be seen below:

```
<?xml version="1.0" encoding="UTF-8"?>
<testcase id="SMLClient" xmlns="http://www.gitb.com/tdl/v1/"
xmlns:gitb="http://www.gitb.com/core/v1/">
  <metadata>
    <gitb:name>SML Client</gitb:name>
    <gitb:type>CONFORMANCE</gitb:type>
    <gitb:version>0.1</gitb:version>
    <gitb:description>This test scenario implements the lookup interface which enables senders to discover
      service metadata about specific target participants
    </gitb:description>
  </metadata>
  <namespaces>
  </namespaces>
  <imports>
  </imports>
  <actors>
    <gitb:actor id="SMLClient" name="SystemUnderTest" role="SUT" />
    <gitb:actor id="ServiceMetadataLocator" name="ServiceMetadataLocator" role="SIMULATED" >
      <gitb:endpoint name="http">
        <gitb:config name="participant.identifier">0088:12345test</gitb:config>
      </gitb:endpoint>
    </gitb:actor>
  </actors>
  <variables>


  </variables>
  <steps>
    <btxn from="SMLClient" to="ServiceMetadataLocator" txnId="t1" handler="SMLMessaging"/>
    <receive id="sml_output" desc="Locate SMP" from="SMLClient" to="ServiceMetadataLocator"
txnId="t1">
      <config name="dns.domain">B-351cd3bce374194b60c770852a53d0e6.iso6523-actorid-
upis.localhost.</config>
    </receive>
    <send desc="Resolve SMP domain" from="ServiceMetadataLocator" to="SMLClient" txnId="t1">
      <input name="dns.address" source="$SMLClient{ServiceMetadataLocator}{network.host}"/>
    </send>
    <etxn txnId="t1"/>
  </steps>
</testcase>
```

Figure 18-8: Conformance Test Case Targeting SML Querying

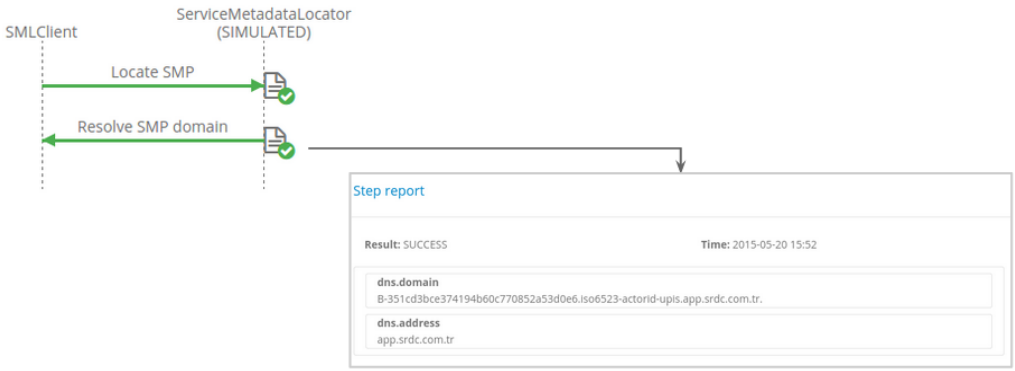

Figure 18-9: Interaction and Sample Test Report for SML Querying

An example for conformance test case that targets SMP querying can be seen below:

```
<?xml version="1.0" encoding="UTF-8"?>
<testcase id="SMPClient" xmlns="http://www.gitb.com/tdl/v1/" xmlns:gitb="http://www.gitb.com/core/v1/">
  <metadata>
    <gitb:name>SMP Client</gitb:name>
    <gitb:type>CONFORMANCE</gitb:type>
    <gitb:version>0.1</gitb:version>
    <gitb:description>This test scenario implements the lookup interface which enables senders to discover
      service metadata about specific target participants
    </gitb:description>
  </metadata>
  <namespaces>
  </namespaces>
  <imports>
    <artifact type="object" encoding="UTF-8"
name="SMP_Metadata_Template">ServiceMetadataPublisher/artifacts/peppol-smp-metadata-template.xml</artifact>
  </imports>
  <actors>
    <gitb:actor id="SMPClient" name="SMP Client" role="SUT" />
    <gitb:actor id="ServiceMetadataPublisher" name="ServiceMetadataPublisher" role="SIMULATED" />
  </actors>
  <variables>
    <!-- Represents the AS2 endpoint address -->
    <var name="as2_address" type="string">
      <value>https://127.0.0.1/as2</value>
    </var>
    <!-- Represents the type of document that the recipient is able to handle -->
    <var name="document_identifier" type="string">
      <value>urn:oasis:names:specification:ubl:schema:xsd:Invoice-
2::Invoice##urn:www.cenbii.eu:transaction:biitrns010:ver2.0:extended:urn:www.peppol.eu:bis:peppol4a:ver2.0::2.1</value>
    </var>
    <!-- The identifier of the process -->
    <var name="process_identifier" type="string">
      <value>urn:www.cenbii.eu:profile:bii04:ver2.0</value>
    </var>
  </variables>
  <steps>
    <btxn from="SMPClient" to="ServiceMetadataPublisher" txnId="t1" handler="SMPMessaging"/>
    <receive id="smp_output" desc="Send message to SMP to get Receiver Access Point address"
from="SMPClient" to="ServiceMetadataPublisher" txnId="t1" />
    <send id="smp" desc="Send SMP Metadata back"  from="ServiceMetadataPublisher" to="SMPClient"
txnId="t1">
      <input name="smp_metadata" source="$SMP_Metadata_Template"/>
    </send>
    <etxn txnId="t1"/>
  </steps>
</testcase>
```

Figure 18-10: Conformance Test Case Targeting SMP Querying


```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<module xmlns="http://www.gitb.com/core/v1/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    id="ValidexValidator" uri="urn:com:gitb:validation:ValidexValidator"
    xsi:type="ValidationModule" isRemote="true"
serviceLocation="http://localhost:9091/service/ValidationService">
 <metadata>
    <name>Validex Validator</name>
    <version>1.0</version>
   <description>Validex wrapper validation service</description>
 </metadata>
 <inputs>
    <param type="string" use="R" name="name" desc="Name of the document to be validated" />
    <param type="object" use="R" name="document" desc="XML document to be validated" />
 </inputs>
 <outputs>
    <param name="string" type="name" desc="Name of the document to be validated"/>
    <param name="string" type="document" desc="XML document to be validated"/>
    <param name="string" type="reportId" desc="Report id given by the Validex service"/>
    <param name="string" type="reportLink" desc="Report link to the Validex reportin interface"/>
 </outputs>
</module>
```

Figure 18-11: Interaction and Sample Test Report for SML Querying

18.5 Existing Test Artifacts/Tools/Services to Reuse in the Domain

Test was carried out by using the following tools

- GITB Test Bed
- PEPPOL messaging network

18.6 Stakeholders

Standard Development Organizations (SDOs), industry consortia, companies, public authorities that may be interested to use the tests:

- Industry consortia
    - National Health Service of UK
    - Peppol project
- Private companies
    - Manufacturers supplying NHS

Part IV. 2: e-Health

19 Clinical Document Architeture (CDA)

19.1 Background and Testing Requirements

The HITCH project (http://www.hitch-project.eu/), the Antilope project (http://www.antilope-project.eu/) and the eHealth Governance Initiative (http://www.ehgi.eu/) recommend the use of integration profilea by the European Union member states in order to promote the interoperability of eHealth applications. Among the recommended profiles, we would like to focus the following 2 profiles:

- XDS.b for sharing document,
- XD-Lab for sharing lab reports.

Austria, France, Luxemburg, Swiss among other countries are publishing specifications on how to share lab reports using these profiles. The proposal is to apply GITB to the purpose of testing the implementation of these 2 profiles in those countries. How could these countries benefit from sharing testing artifact and thus insure better interoperability?

The test case described in that document focus on testing the conformance of CDA documents containing laboratory reports.

19.2 Verification Scope – What Should be Tested?

This Test Case focuses on testing the conformance of CDA Lab reports. CDA Documents are usually designed as Russian dolls as described on the following schema. Looking at the specifications of the "Lab Report" document as specified by various organizations in Europe and elsewhere shows that all of them are referring the same underlying specifications.

1. In France, ASIP santé published with the *CI-SIS : Cadre d'interopérabilité des systèmes d'information de santé* the specifications of a "Volet Compte Rendu d'Examens de Biologie Médicale[44]".
2. In Austria, Elga[45] published the document *HL7 Implementation Guide for CDA® R2: Laborbefund*[46].
3. In Italy, IHE Italy[47] published the document *Rapporto di medicina di laboratorio*[48]
4. In Switzerland, eHealthSuisse [49]published *Format d'échange : Rapports de laboratoire soumis à déclaration en Suisse*[50].

---

[44] http://esante.gouv.fr/services/referentiels/referentiels-d-interoperabilite/cadre-d-interoperabilite-des-systemes-d

[45] http://www.elga.gv.at/index.php?id=28

[46] http://www.elga.gv.at/fileadmin/user_upload/uploads/download_Papers/Harmonisierungsarbeit/140902__upload/HL7_Implementation_Guide_for_CDA_R2_-_Laborbefund.pdf

[47] http://www.hl7italia.it/node/34

[48] http://www.hl7italia.it/sites/default/files/Hl7/docs/public/HL7Italia-IG-CDA2%2020RapportoMedicinaLab-v01.00-SI.pdf

[49] http://www.e-health-suisse.ch/umsetzung/00252/index.html?lang=fr

[50] http://www.e-health-suisse.ch/umsetzung/00252/index.html?lang=fr&download=NHzLpZeg7t,lnp6I0NTU042l2Z6ln1ae2IZn4Z2qZpnO2Yuq2Z6gpJCDdIB5e2ym162epYbg2c_JjKbNoKSn6A--

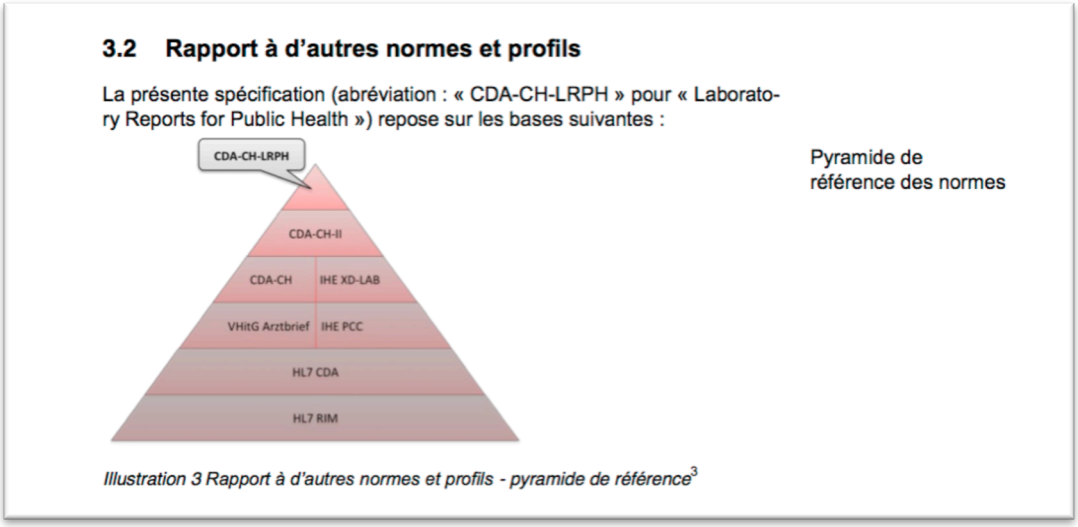

Figure 19-1: CDA-CH Laboratory Reports for Public Health in relation to other norms and profiles.

The specifications provided by these 4 countries rely all on the IHE XD-LAB technical framework[51]. France, Austria and Switzerland provide also a set of testing tools in order to check the conformance of the CDA document that claim to support their specifications.

The purpose of this use case is to optimize testing and test tools development by using the "Russian Doll" architecture of the CDA documents.

The proposed use case described here is designed for the specific Laboratory Report document, but it could also be applied to other type of documents that these countries have specified.

The challenge is to reuse test artifacts for the conformance checking of CDA document in various national/regional project that use common references.

The benefit is clear for all parties. The burden to test the common part can be re-used. Only the tests specific to the requirement of a country specification need to be developed, the rest remains common. Quality of the testing is harmonized and risk of different outcome due to different implementation of the tools are reduced.

19.2.1 Parties/Actors

The following parties take some role in the business process.

- Content Creator : The issuer of the CDA Laboratory Report Document document.
- Content Consumer : the consumer of the CDA Laboratory Report Document. The consumer shall be able to read the document and "digest its content".

19.3 Underlying eBusiness Specifications / Standards

Table 19-1: CDA – Relevant eBusiness Specifications

	Relevant specifications / standards	References
Business Process	IHE XD-LAB	See [52]

[51] http://ihe.net/uploadedFiles/Documents/Laboratory/IHE_LAB_TF_Vol3.pdf

[52] http://ihe.net/uploadedFiles/Documents/Laboratory/IHE_LAB_TF_Vol1.pdf

Business Documents	HL7 CDA IHE XD-LAB	See [53] See [51] and [54]
Transport and Communication (Messaging) Protocols	Transport is out of the scope of this document. See Test Case 2 for the protocol	
Profiles		

19.4 Testing Scenarios

19.4.1 Objectives and Success Criteria

The testing is focused on the conformance of the exchanged laboratory report documents.

The criteria for success:

1. Document is a well-formed XML document
2. Document is valid according the CDA schema (might be extended by the regional specifications)
3. Document meets the requirements specified in the specs (syntax and semantic)

This section will describe what and how we will test based on the target specification. The perspective of this section is from the software architect responsible for utilizing the GITB framework to set up the appropriate Test Services and Test Artifacts to support the Testing Scenarios.

19.4.2 System Under Test (s)

The systems under test in this case are the ***Content Creator***, the system that creates the CDA Laboratory Report document and the ***Content Consumer*** who consumes it.

The Content Creator is tested for its ability to create documents that are conformant to the specifications.

The Content Consumer is tested for its ability to read and correctly display the information provided in the documents created by the Document Creator.

19.4.3 Abstract Test Steps

19.4.3.1 Testing the content creator

The content creator creates a set of documents

Each created document is checked for conformance

A report for conformance is created, the report include the list of requirement that were identified and tested in the documents.

[53] http://www.hl7.org

[54] http://ihe.net/uploadedFiles/Documents/Laboratory/IHE_LAB_TF_Vol1.pdf

19.4.3.2 Testing the content consumer

The content consumer consumes (load) a set of documents.

The content consumer shows evidence that the documents are correctly loaded and that it can display the content of the information contained in the consumed documents.

19.5 Related Existing Test Artifacts/Tools/Services to Reuse in the Domain

A number of test artifacts such as schematron or model can be reused:

- ASIP Santé, Elga and eHealthSuisse provide Schematrons for the validation of their respective specifications.
- IHE and NIST provide schematron for the validation of XD-LAB CDA documents.
- MDHT and IHE provide model based CDA Document validator.

Tooling is also available:

- IHE provides a web based[55] and web service to perform the validation of CDA document either using a schematron[56] or a model[57]
- NIST provide a Web based and Web service schematron validation[58]

19.6 Related Stakeholders

The choice of this Test Case is driven by its potential interest for many organization worldwide. Interested bodies are listed below:

- HL7 : SDO
- IHE : Integrating the Healthcare Enterprise is a not for profit organization for the promotion of the interoperability of solutions in Healthcare
- Public Authorities that customized the profile:
  - ASIP Santé in France
  - ELGA in Austria
  - eHealthSuisse in Switzerland
  - Agence eSanté in Luxemburg
  - NICTIZ in the Netherlands
  - Plate-format eHealth in Belgium
- Companies that implements the profile
  - 30 companies worldwide have shown interest in sharing this type of documents at during the IHE Connectathons in Europe or in North America
    - ALERT Life Sciences Computing
    - Allscripts Healthcare Solutions
    - Atlas
    - Axway
    - CapMed
    - CareEvolution, Inc.
    - eClinicalWorks
    - Eclipsys Corporation

[55] http://gazelle.ihe.net/EVSClient

[56] http://gazelle.ihe.net/SchematronValidator

[57] http://gazelle.ihe.net/CDAGenerator/home.seam

[58] http://cda-validation.nist.gov/cda-validation/

- e-MDs
- Engineering Ingegneria Informatica
- Evolucare Technologie
- Fidelity Information Systems
- Forcare BV
- GE Healthcare
- Get Real Health
- Global Care Quest
- InterSystems Corporation
- Karos Health
- MEDecision
- Medical Informatics Engineering
- NextGen
- No More Clipboard
- Open Health Tools
- SAIC
- SIEMENS Medical Solutions
- Tiani "Spirit" Gmbh - Cisco Systems Inc.
- Topicus Zorg

19.7 Re-usability of Test artifacts/Tools/Services for GITB3

This Test Case gives GITB3 the ability to reuse Test Artifacts, tools and services in cross-organization scenarios.

Scenario 1: ELGA, ASIP and eHealthSuisse to use a common set of tools to check the conformance of the XD-LAB profile implementation in CDA documents target to their respective context.

Scenario 2: Company can test its implementation of the XD-LAB profile and check the conformance to the different extensions made by ELGA, ASIP and eHealthSuisse.

20 IHE – Cross-Enterprise Document Sharing (XDS)

The purpose of this Test Scenario is the sharing of Test Artifacts for testing the interoperability of systems participation to a sharing of document workflow based on XDS.b.

20.1 Background and Testing Requirements

Cross-Enterprise Document Sharing (XDS) provides a standards-based specification for managing the sharing of documents between any healthcare enterprise, ranging from a private physician office to a clinic to an acute care in-patient facility and personal health record systems. Many regional/national projects worldwide[59] are deploying/specifying the sharing of medical document using an infrastructure based on the IHE XDS.b suite of profiles. We are proposing in this scenario to share the tests artifacts that could be common for all these projects, the XDS.b part of the exchange.

Sharing the same set of Tests Artifacts among these project will help them. A set of Test Artifacts allowing the testing of the underlying transport mechanism will be available for them. So that they will only test the parts specific to their projects. SUTs who already got tested for one project know that the underlying transport mechanism has been tested and can focus on the projects specifics.

The benefits are:

- Reuse of test artifacts and test tools,
- Harmonization of the infrastructures deployed by the projects, avoiding “specifics” implementations,
- Reduce the cost of test design and testing.

20.2 Verification Scope – What to Test?

The business process that needs to be tested is described in the IHE Technical Frameworks of the IT-Infrastructure domain and available on the IHE web site.[60]

20.3 Actors

The parties involved are the Document Source, Document Consumer, Document Registry and Document Repository as described in the figure below.

[59] See http://motorcycleguy.blogspot.com/2010/01/where-in-world-is-xds.html

[60] See
http://ihe.net/uploadedFiles/Documents/ITI/IHE_ITI_TF_Vol1.pdf
http://ihe.net/uploadedFiles/Documents/ITI/IHE_ITI_TF_Vol2b.pdf
http://ihe.net/uploadedFiles/Documents/ITI/IHE_ITI_TF_Vol2x.pdf
http://ihe.net/uploadedFiles/Documents/ITI/IHE_ITI_TF_Vol3.pdf

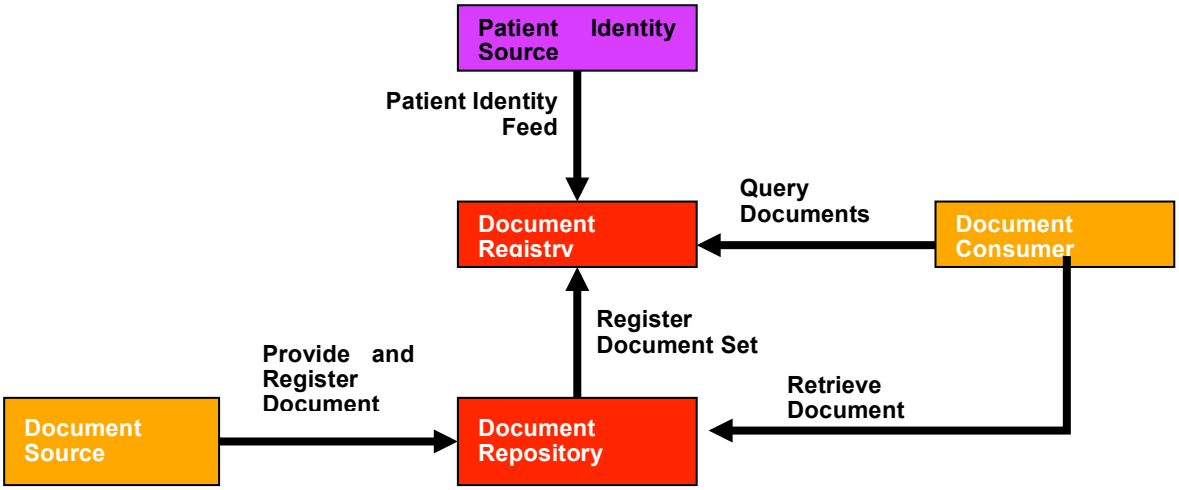

Figure 20-1: Parties involved in IHE XDS

- The **Document Source** Actor is the producer and publisher of documents. It is responsible for sending documents to a Document Repository Actor. It also supplies metadata to the Document Repository Actor for subsequent registration of the documents with the Document Registry Actor.
- The **Document Repository** is responsible for both the persistent storage of these documents as well as for their registration with the appropriate Document Registry. It assigns a uniqueId to documents for subsequent retrieval by a Document Consumer.
- The **Document Registry** Actor maintains metadata about each registered document in a document entry. This includes a link to the Document in the Repository where it is stored. The Document Registry responds to queries from Document Consumer actors about documents meeting specific criteria. It also enforces some healthcare specific technical policies at the time of document registration.
- The **Document Consumer** Actor queries a Document Registry Actor for documents meeting certain criteria, and retrieves selected documents from one or more Document Repository actors.

20.3.1 Interaction Diagram/Choreography

The following diagram shows the interactions between that need to be covered by the tests.

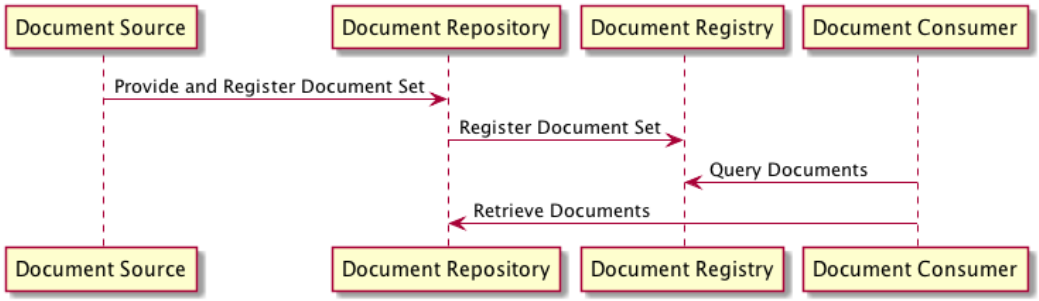

Figure 20-2

20.3.2 Underlying eBusiness Specifications / Standards

The XDS.b profile is relying on the following set of standards and specifications:

- ebXML
- IHE XDS.b
- HL7v3 datatypes
- MTOM
- HTTP
- SOAP

- TLS

20.4 Details/Requirements of Test Scenario

20.4.1 Objectives and Success Criteria

The following test scenarii implement conformance and interoperability tests for the XDS.b profile. Different tests need to be performed depending on which role is played by the system under test.

The conformance to the XDS.b profile specification of the messages exchanged between the SUT and the simulator or the partner will be verified as well as the correct behavior of the actors participating to the test.

If we exclude the Patient Identity feed from the testing scope, in order to test this profile, we have 4 actors to test and 4 transactions to test. Testing is described by considering each of the actors playing the role of the SUT.

An affinity domain needs to be defined in order to perform the testing. The SUT and the Simulator involved in the testing need to share coded values and certificates.

In the preparation of the testing, the actors document registry and document repository need to be feed with data for testing purposes.

20.4.2 System(s) Under Test

Possible SUTs are considered in the following Test Cases. Each scenario describes the test plan for one of them. As described above the Patient Identity Source is not considered here, restraining the SUT to the following list:

- Document Source
- Document Consumer
- Document Repository
- Document Registry

20.4.3 Abstract Test Steps

20.4.3.1 Testing the Document Source

In order to test the Document Source actor we need a simulator playing the role of the Document Repository.

The different test steps required to test the Document Source are presented in the following sequence diagram.

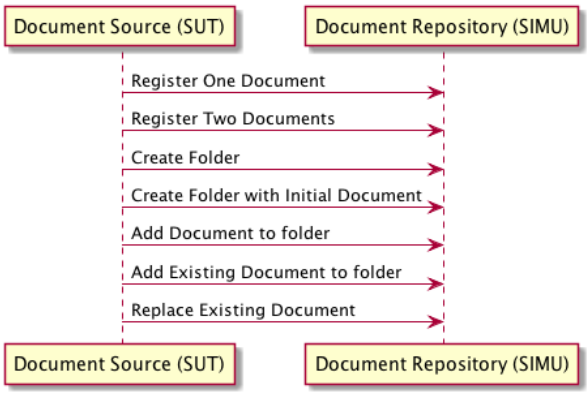

Figure 20-3: Testing the Document Source

20.4.3.2 Testing the Document Consumer

In order to test the Document Consumer actor we need a simulator playing the role of both the Document Registry and the Document Repository actors. The different test steps required to test the Document Consumer are presented in the following sequence diagram.

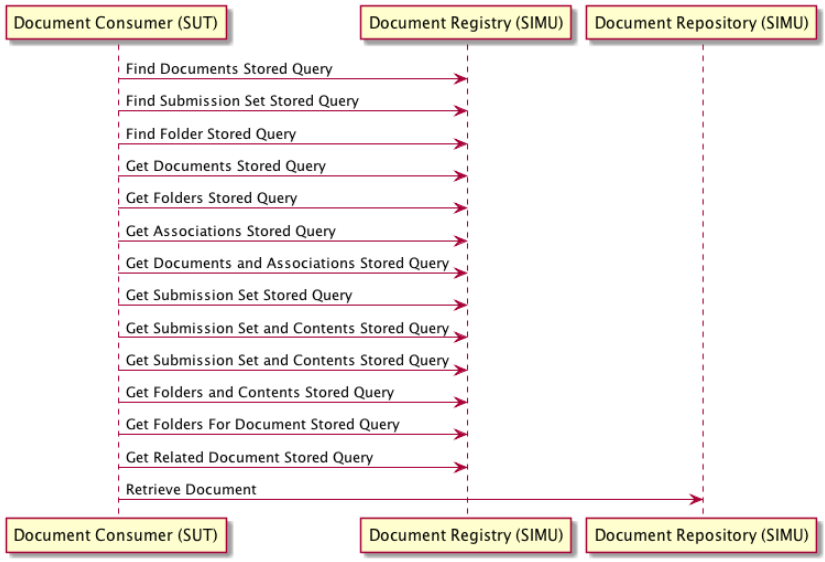

Figure 20-4: Testing the Document Consumer

20.4.3.3 Testing the Document Repository

In order to test the Document Repository actor we need a simulator playing the role of both the Document Consumer and the Document Source actors. The different test steps required to test the Document Repository are presented in the following sequence diagram.

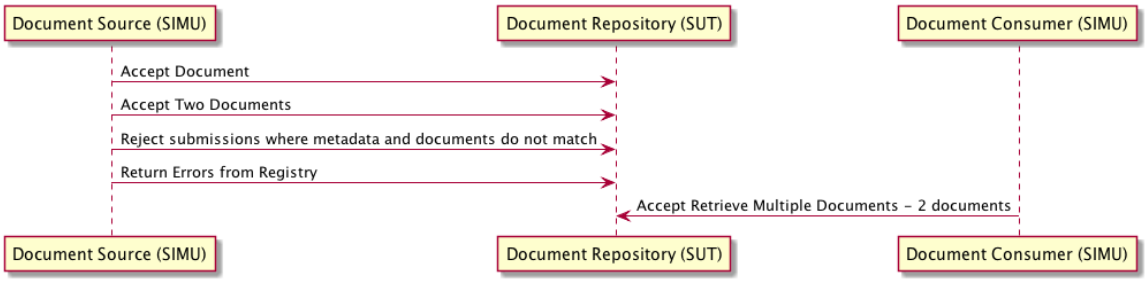

Figure 20-5: Testing the Document Repository

20.4.3.4 Testing the Document Registry

In order to test the Document Registry actor we need a simulator playing the role of both the Document Consumer and the Document Repository actors. The different test steps required to test the Document Registry are presented in the following sequence diagram.

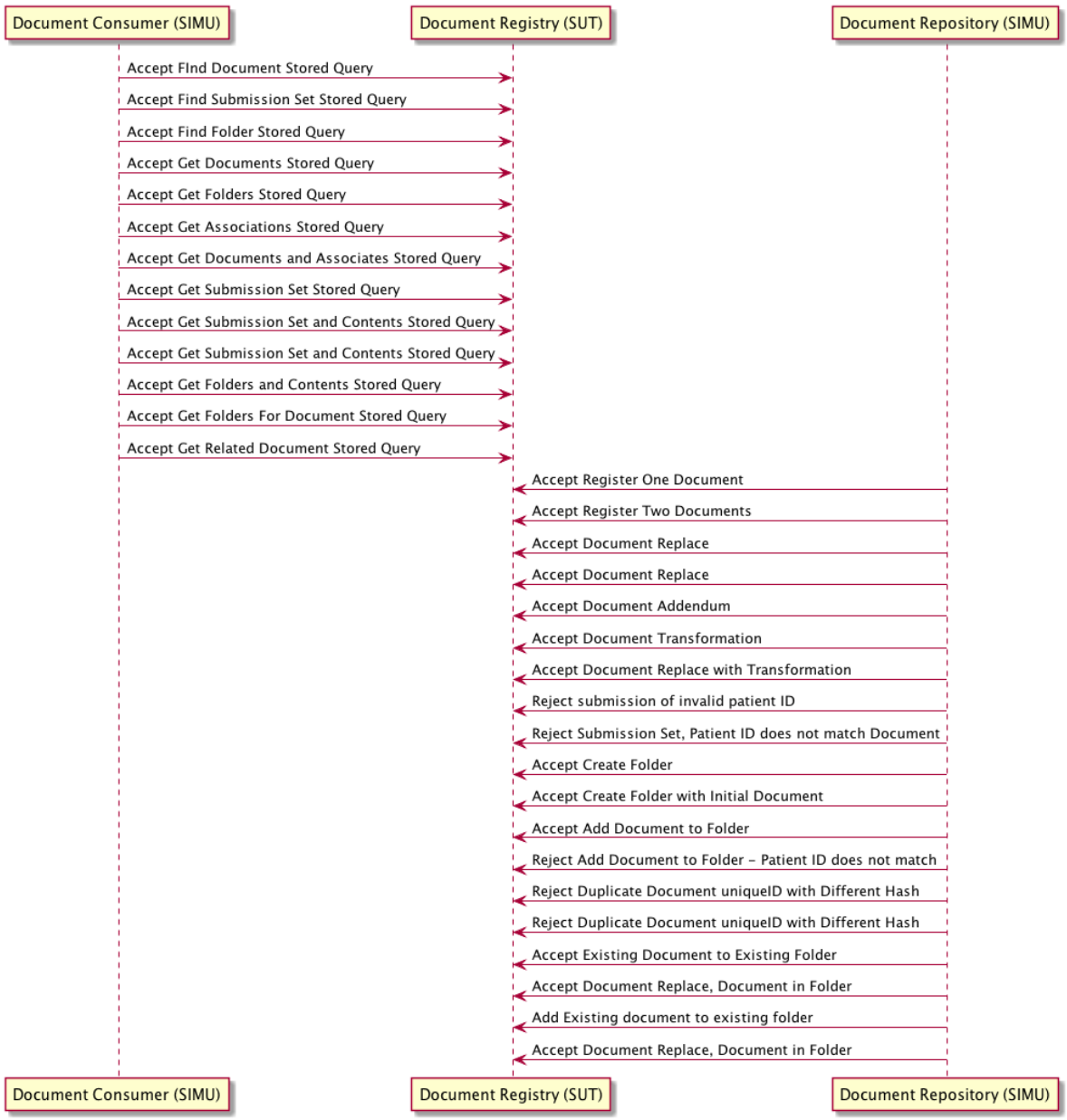

Figure 20-6: Testing the Document Registry

For each test steps, the message send by the SUT will be analyzed and test for conformance with the specifications.

- Verification of the TLS layer
- Verification of the HTTP transport
- Verification of the MTOM layer
- Verification of the SOAP header
- Verification of the Business Message

20.5 Related Existing Test Artifacts/Tools/Services to Reuse in the Domain

The Test Scenarios described in this document are related to the Test Cases used by IHE in both the pre-connectathon and the connectathon testing phases.

Tools are available for simulating the missing actors and checking the conformance of messages to the XDS.b requirements.

One should consider the following existing set of tools:

1. **XDSTools2**[61]for simulation and conformance checking of the XDS Actors.
2. **XDStarClient**[62]for simulation and conformance checking of XDS messages
3. **EVS Client**[63] for the validation of messages
4. **Sharing Value Set Simulator**[64] for the sharing the coded values with the test participants
5. **Gazelle TLS tools**[65] for the needs in term of security testing: Certificate generation, TLS testing

20.6 Related Stakeholders

IHE XDS.b profile users are or will be clearly interested by using these tests. Although IHE is providing already set of tools to perform this testing, the ability to share and re-use test cases might be of interest to organization that extends the XDS.b profile for implementation. Regional projects, national projects (ELGA, ASIP santé (DMP), Agence eSanté, KELA…) might indeed benefit from re-using the test cases in their context.

Over 150 companies worldwide[66] have tested one of the XDS.b profile at one of the IHE connectathon.

20.7 Re-usability of Test Artifacts/Tools/Services for GITB3

The XDS.b profile requires the exchange of messages in a secure TLS connexion. Testing the TLS part of the transaction is not specific to the XDS.b context and the test artifacs/tools/services used to test it could be shared or common to different domain.

The XDS.b profile uses the MTOM, SOAP and HTTP protocol for the transport of the messages. As for the security aspects, those protocols are not specific either and could be considered to be testing using artifacs/tools/services from other domains.

61 http://ihexds.nist.gov/xdstools2/

62 http://gazelle.ihe.net/XDStarClient

63 http://gazelle.ihe.net/EVSClient

64 http://gazelle.ihe.net/SVSSimulator

65 http://gazelle.ihe.net/tls

66 http://connectathon-results.ihe.net

Part V: Manufacturing and Automotive

21 Electronic Invoicing Based on EDIFACT and OFTP2

21.1 Background and Testing Requirements

The smooth running of today's automotive supply chains relies on the seamless electronic exchange of business documents between automotive manufacturers and their suppliers. They make extensive use of EDIFACT messages for electronic communication. European key players in standardization for the automotive industry include Odette (Organisation for Data Exchange by Tele Transmission in Europe) as well as national organizations, such as VDA (Verband der Automobilindustrie) for Germany or GALIA (Groupement pour l'Amélioration des Liaisons dans l'Industrie Automobile) for France. Supported by the European Commission and major organisations of the automotive industry, significant efforts have been made in the last years to improve the integration of automotive companies, particularly SMEs, in the sector's digital supply chains, among them the auto-gration project. These initiatives build on the existing EDI infrastructures and facilitate their integration.

The objective of this use case is to verify the use of GITB for interoperability testing and document conformance testing using Odette OFTP2 messaging protocol and an EDIFACT messages.

GITB demonstrates a test case for electronic invoicing in the automotive industry following document specifications published by the German association of the automotive industry (VDA – Verband der Automobilindustrie) and the using the transfer protocols OFTP2 defined by Odette. The test scenario in the test case demonstrate the capabilities of the GITB testbed to test network connection for both sender and receiver.

21.2 Verification Scope

The test demonstrates the ability of the GITB platform to perform both an interoperability test and a document conformance test on EDIFACT messages and transport protocols used in the automotive industry.

21.2.1 Actors

Two parties interact in the invoicing process:

- **Seller** – The original issuer of a EDIFACT invoice.
- **Buyer** – The original receiver of a EDIFACT invoice. For the purpose of the test, the buyer is simulated by GITB.

21.2.2 Business Documents

ODETTE, which is a pan-European collaboration and services platform working for the entire automotive network, has developed a set of Global Automotive EDIFACT Messages that are used in the Automotive Industry world-wide. Electronic invoicing relies on the ODETTE subset of the UN/CEFACT EDIFACT invoice message, as defined by the ODETTE Global INVOIC - European Profile V3.1.

ODETTE provides an EDI Validation Portal[67][68] based on GEFEG tool for these Global Messages. This internet validation service allows you to perform a compliance check of your messages against the most frequently used Global Automotive EDIFACT specifications.

[67] http://www.gefeg.com/en/gefeg.validation/vp-odette.htm

[68] http://www.gefeg.com/en/standard/automotive/odette.htm

21.2.3 Standards and Specifications

Electronic invoices are based on the ODETTE Global INVOIC - European Profile V3.1.

As transport and communication protocol the OFTP2 protocol is used, as it addresses the electronic data interchange (EDI) requirements of the European automotive industry. OFTP2

- enables secure transfer of business documents over the Internet, ISDN and X.25 networks,
- allows encryption and digitally signing message data, requesting signed receipts and also offers high levels of data compression,
- provides partner authentication mechanism,
- and provides additional session level security over TLS.

A single OFTP2 entity can make and receive calls, exchanging files in both directions.

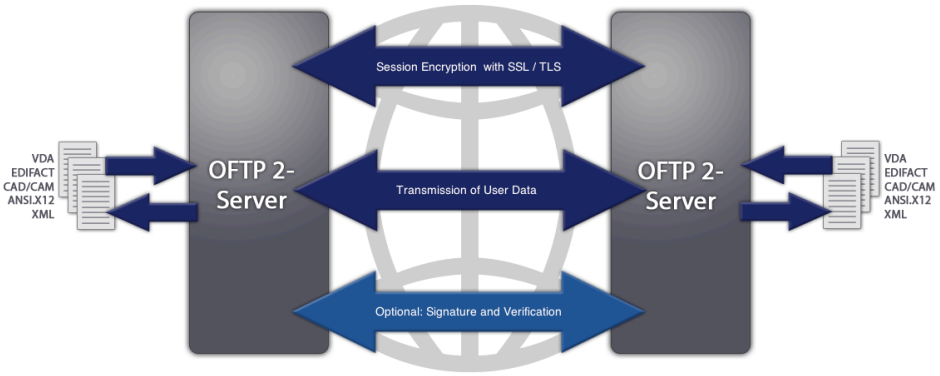

Figure 21-1: OFTP2 Protocol

Table 21-1: Electronic Invoicing Based on EDIFACT and OFTP2 – Relevant eBusiness Specifications

	Relevant specifications / standards	References
Business Process	N/A	
Business Documents	ODETTE Global INVOIC - European Profile V3.1	https://www.odette.org/publications/file/global-invoic-european-profile
Transport and Communication (Messaging) Protocols	ODETTE messaging protocol OFTP2	https://tools.ietf.org/html/rfc5024 https://www.odette.org/publications/file/new-oftp2-implementation-guideline-v2.4
Profiles	None used	

21.3 Test Scenario

21.3.1 Test Objectives / Requirements

This Test Scenario implements a conformance / interoperability test for the exchange of EDIFACT messages delivered over a OFTP2 transport protocol.

Success criteria:

- Successful transfer of message trough transport.
- Testing of document conformance and identification of potential errors.

21.3.2 System under Test (s)

Systems under test are buyer and seller systems using EDIFACT messages via OFTP2 transport network. Both systems are simulated by GITB. This test scenario is representative of real live test scenario where EDIFACT based messages are exchanged between trading parties in the automotive industry using messages specifications and transport protocols designed for that industry.

21.3.3 Abstract Test Steps

For the purpose of the test a suite of 4 test scenarios have been prepared in order to test the compliance of a system under test (SUT) for the automotive use case

1. **EDI Receive Test:** A sender application, simulated by GITB Engine, sends a valid EDI Invoice document over OFTP2 to the SUT.

2. **EDI Receive With Upload Test:** The SUT uploads an EDI Invoice document from his computer and a sender application simulated by GITB Engine sends this document over OFTP2 to the SUT.

3. **EDI Send Test:** The SUT sends an EDI document over OFTP2 to the receiver, simulated by GITB Engine and EDI Validator validates the received document.

4. **EDI Interoperability Test:** A sender SUT sends an EDI Invoice document to a receiver SUT and GITB Engine listens this connection like a proxy. The exchanged document is captured by GITB Engine and validated by EDI Validator.

The test included the following steps.

- Document validation

- Messaging operation validation

21.3.3.1 Document Validation

An EDI validator for EDI INVOIC messages was developed by the GITB project and is included in the PoC GITB test platform.

EDI INVOIC provides the definition of the Invoice message (INVOIC) to be used in Electronic Data Interchange (EDI) between trading partners involved in administration, commerce and transport.

Following image shows a section of a sample message.

```
UNA:::
UNB:::
UNH+0001+INVOIC:D:96A:UN:A14051'
BGM+380::+12345678:9+AP'
DTM+137:19980610:102'
PAI+::31::10'
FTX+AAI+++Free text Free text Free text'
NAD+BY+106321::91+P.O.Box 2021:80803:Munich+Koenig+++++DE'
RFF+VA:DE82248176'
CTA+AD+.ERP'
CTA+PD+:Central Purchase'
NAD+SE+15432211::92+Schoenstrasse7:81543:Munich+Meister+++++DE'
RFF+VA:DE81765430'
CTA+AD+:MrsTreffer'
CUX+2:DEM:4+::+'
PAT+1++:::'
PCD+12:2.00'
MOA+52:2'
LIN+1++7505:IN'
PIA+1+2231:SA+:+:'
IMD+++:::Seal'
QTY+12:100:PCE'
ALI+DE++'
GIN+BN+223CLX11'
MOA+203:100'
PAT+1+++:::'
DTM+171:19980624:102'
PRI+AAA:1::CON:1:PCE'
RFF+AAK:9034711:01'
DTM+171:19980609:102'
TAX+7+VAT+++:::16.00+'
MOA+124:16.00'
```

Documents were submitted to the validation engine which generated validation reports based on the results of EDI Validation

Reports indicate the result of the validation operation after validation step is executed. Reports are provided in two levels.

- Simple level showing the results of the messages validation as whole with Boolean result of success if all tests are passed, but failure if one or more test rule fails.
- Detailed report with segment by segment error indication in validation reports.

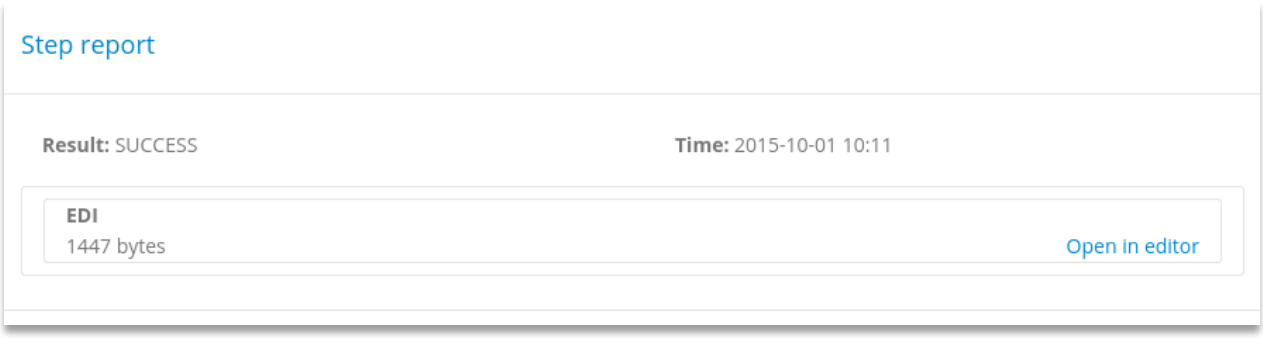

Figure 21-2: Simple report showing document level results

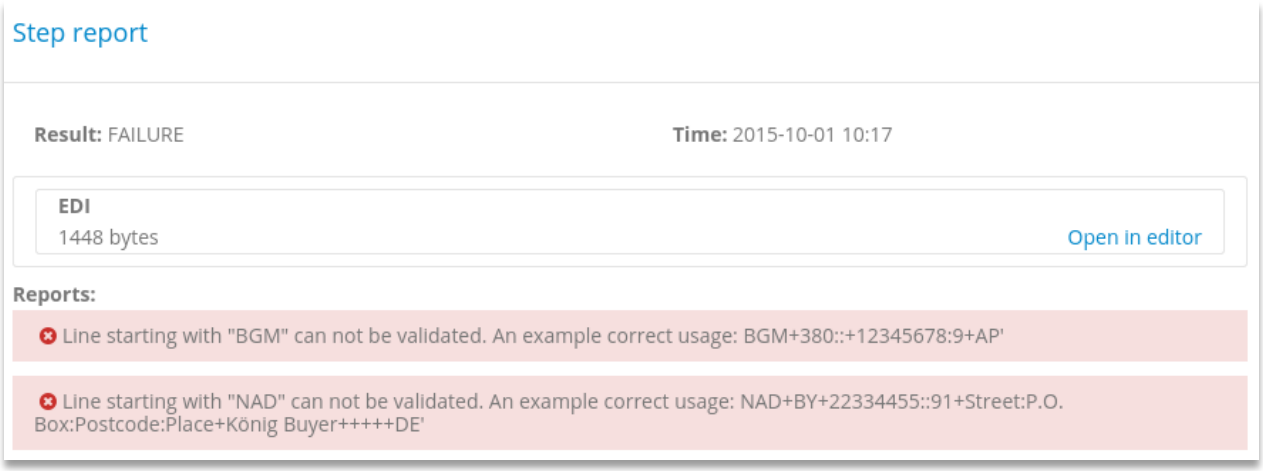

Figure 21-3: Detailed Report Showing Result on Error Level

21.3.3.2 Messaging Operations

An OFTP2 Messaging Adapter was developed by the GITB project and included in the GITB PoC Test Bed platform. The adapter enables the exchange (sending, receiving and listening – as a proxy) of EDI documents over a TCP/IP based network

The message exchange is simulated by the GITB test platform in the following way.

Figure 21-4: Interaction

Test specification is defined using GITB message specification definitions as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<testcase id="EDI-Receiver-Invoice-Upload" xmlns="http://www.gitb.com/tdl/v1/" xmlns:gitb="http://www.gitb.com/core/v1/">
    <metadata>
        <gitb:name>EDI-Receiver-Invoice-Upload</gitb:name>
        <gitb:type>CONFORMANCE</gitb:type>
        <gitb:version>0.1</gitb:version>
        <gitb:description>The objective of this Test Scenario is to ensure the EDI Receiver (the System Under
            Test) can receive an uploaded EDI Invoice document over the OFTP2 protocol.
        </gitb:description>
    </metadata>
    <namespaces>
        <ns prefix="fn">http://www.gitb.com/fn/v1/</ns>
    </namespaces>
    <imports>
    </imports>
    <actors>
        <gitb:actor id="EDISender" name="EDISender" role="SIMULATED"/>
        <gitb:actor id="EDIReceiver" name="EDIReceiver" role="SUT" />
    </actors>
    <variables>
        <!-- Name of the document sent -->
        <var name="file_name" type="string" />
        <var name="file_content" type="binary" />
    </variables>
    <steps>
        <interact desc="GITB Engine needs information" with="EDIReceiver">
            <request desc="Please upload EDI invoice document to be received:"
                     with="EDIReceiver" contentType="BASE64">$file_content</request>
            <request desc="Please enter the file name:" with="EDIReceiver" contentType="STRING">$file_name</request>
        </interact>

        <btxn from="EDISender" to="EDIReceiver" txnId="t1" handler="OFTP2Messaging"/>
        <send id="message" desc="Receive Invoice message from EDI Sender"  from="EDISender" to="EDIReceiver" txnId="t1">
            <input name="file_name" source="$file_name" />
            <input name="file_content" source="$file_content" />
        </send>
        <etxn txnId="t1"/>
    </steps>
</testcase>
```

Figure 21-5: Test Case

21.4 Existing Test Artifacts/Tools/Services to Reuse in the Domain

The most popular Test Resource is the GEFEG FX software.

21.5 Stakeholders

Standard Development Organizations (SDOs), industry consortia, companies, public authorities that may be interested to use the tests:

- Industry consortia
    - ODETTE
    - National automotive associations, such as VDA (Verband der Automobilindustrie) for Germany or GALIA (Groupement pour l'Amélioration des Liaisons dans l'Industrie Automobile)
    - CEN (as organizer of the auto-gration project)
- Private companies
    - Automotive manufacturers and suppliers

22 Cross-Border Transactions

22.1 Background and Testing Requirements

ClearView Trade is a Danish IT solutions provider that specializes in simplifying cross-border transaction processes and to improve trade flow. Their main focus is to support freight forwarders and traders with time and money saving solutions that improve the quality of their customs declarations. They specialize in standards such as World Customs Organisation (WCO) UN/CEFACT, ISO and CEN.

ClairView Trade required a test suite for the testing of UBL Despatch Advice documents with given Schematron rules.

22.2 Verification Scope

22.2.1 Actors

The following actors assume a role in this business process:

- **Seller** – The original issuer of the electronic despatch advice.
- **Buyer** – The original receiver of the electronic despatch advice.

22.2.2 Business Document

The test is concerned with testing an XML document that is being used in a real live manufacturing use case. The document is a UBL Despatch Advice, which is common document in delivery of products and may be applied in multiple business processes.

22.2.3 Underlying Standards and Specifications

Table 22-1: Cross-Border Transactions – Relevant eBusiness Specifications

	Relevant specifications / standards	References
Business Documents	UBL 2.1	http://docs.oasis-open.org/ubl/os-UBL-2.1/UBL-2.1.html UBL-DespatchAdvice-2.1.xsd http://docs.oasis-open.org/ubl/os-UBL-2.1/xsd/maindoc/UBL-DespatchAdvice-2.1.xsd

22.3 Test Scenario

22.3.1 Objectives and Success Criteria

The test is for verifying document conformance against document specifications. Document conformance can be verified either as success or failed, i.e. either the document conforms to specifications or it does not. In the case that it does not, the reasons for failures should be reported.

22.3.2 System under Test (s)

The test is used to test a document conformance against standard specifications. For the purpose of the test the seller and the buyer are simulated by GITB.

22.3.3 Test Steps

The test suite for ClearView Trade comprises only one test case which aims testing of Despatch Advice documents with given Schematron rules. During the test case, SUT operator is requested to upload a

Despatch Advice document which is then, validated by test engine with given Schematron rules. The content of the test case is given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<testcase id="DespatchAdvice-Validation" xmlns="http://www.gitb.com/tdl/v1/"
xmlns:gitb="http://www.gitb.com/core/v1/">
  <metadata>
    <gitb:name>DespatchAdvice-Validation</gitb:name>
    <gitb:type>CONFORMANCE</gitb:type>
    <gitb:version>0.1</gitb:version>
    <gitb:description>The objective of this test scenario is to ensure the Sender can upload DespatchAdvice
    documents in order for Test Engine to validate using Schematron rules.
    </gitb:description>
  </metadata>
  <namespaces>
  </namespaces>
  <imports>
    <artifact type="schema" encoding="UTF-8" name="DespatchAdvice_Schematron_File"
>ClearViewTradeTestSuite/artifacts/DespatchAdviceValidateSvrl.xsl</artifact>
  </imports>
  <actors>
    <gitb:actor id="Sender" name="Sender" role="SUT" />
  </actors>
  <variables>
    <var name="despatch_advice" type="object" />
  </variables>
  <steps>
    <interact desc="GITB Engine needs information" with="Sender">
      <request desc="Please upload your despatch advice document:" with="Sender"
contentType="BASE64">$despatch_advice</request>
    </interact>


    <verify handler="SchematronValidator" desc="Validate Despatch Advice against PEPPOL BII Rules">
      <input name="xmldocument">$despatch_advice</input>
      <input name="schematron" source="$DespatchAdvice_Schematron_File"/>
    </verify>
  </steps>
</testcase>
```

Figure 22-1: Test Case

22.4 Existing Test Artifacts/Tools/Services to Reuse in the Domain

The electronic despatch advice message was tested using the following artifacts:

- UBL XSD Despatch Advice schema:
    - UBL-DespatchAdvice-2.1.xsd
      http://docs.oasis-open.org/ubl/os-UBL-2.1/xsd/maindoc/UBL-DespatchAdvice-2.1.xsd

22.5 Related stakeholders

Standard Development Organizations (SDOs), industry consortia, companies, public authorities that may be interested to use the tests:

- Industry consortia
    - World Customs Organisation (WCO)
    - The European Freight Forwarders Association (EFFA)
    - European E-Invoicing Service Providers Association (EESPA)
- Service providers, such as
    - ClearTrade View Denmark (http://clearviewtrade.com) for on-boarding their customers

23 Test Bed Interoperability with Application for a Truck Manufacturer

23.1 Background and Testing Requirements

If Test Beds and Test services use the GITB specifications, Testing Resources can be more easily shared. To demonstrate the reuse of Testing Resources in a real-world environment, a test case was developed to demonstrate interaction between two independently developed test beds using GITB protocols for communication between the systems.

For this purpose, the Validex document testing tools and the GITB Test Bed were integrated (see 10.2). In conjunction to that the Validex document testing tool was implemented for a Swedish truck manufacturer.

23.2 Verification Scope

The test aims to verify the possibility of making two different test bed solutions work together in an integrated way as a single solution where one system extends the other. The objective is to successfully submit a document from one test bed for conformance testing and deliver the results back.

23.2.1 Parties/Actors

The following parties are involved:

- **GITB** – As interoperability test bed which submits documents to Validex for conformance testing.
- **Validex** – As document conformance test bed that acts as a extension to the GITB test bed.

23.2.2 Standards and specifications

The test bed connections were tested using the following artifacts.

Table 23-1: Relevant eBusiness Specifications

	Relevant specifications / standards	References
Business Documents	The payment initiation message (pain01) based on the ISO 20022 XML messages specifications.	www.iso20022.org/payments_messages.page

23.3 Test Scenario

23.3.1 Objectives

GITB performs an interoperability test for FTP exchanging an ISO 20022 payment initiation message (pain01).

23.3.2 System under Test (s)

GITB test bed.

Validex test bed.

23.3.3 Abstract Test Steps

GITB PoC implementation integrates an online validation tool, called Validex, to demonstrate its capabilities of integrating external systems according to GITB Service Specifications. In order for an external content validation system to be integrated with GITB Testbed, it must implement GITB Content Validation Service Specifications to wrap its functionalities. To realize this integration, a new module, gitb-validator-validex has

been created with **ValidationServiceImpl** class that implements the **ValidationService** web service interface. With this implementation, Validex becomes accessible through GITB Testbed Validation Service. Requests to this service are delivered to Validex and responses are wrapped with internal test reporting format and returned to tester.

The module definition of Validex Validator can be seen below. One difference here from the module definitions provided in gitb-validators module is that, there is an additional **serviceLocation** attribute which denotes the endpoint of wrapper validation service. When the test engine utilizes this validator wrapping the functionalities of Validex, it calls this service which delivers the request to Validex, as mentioned before.

23.3.4 Validex Integration

Validex has been integrated by implementing GITB Content Validation Service Specifications to wrap its functionalities and serve them as a content validation service to other stakeholders

In this way, any external content validation service can be integrated with GITB Testbed

Document Validation is realized by delivering requests to Validex via the REST API provided by GITB Content Validation Services

After validation, Validex results are converted into TR model and presented to user.

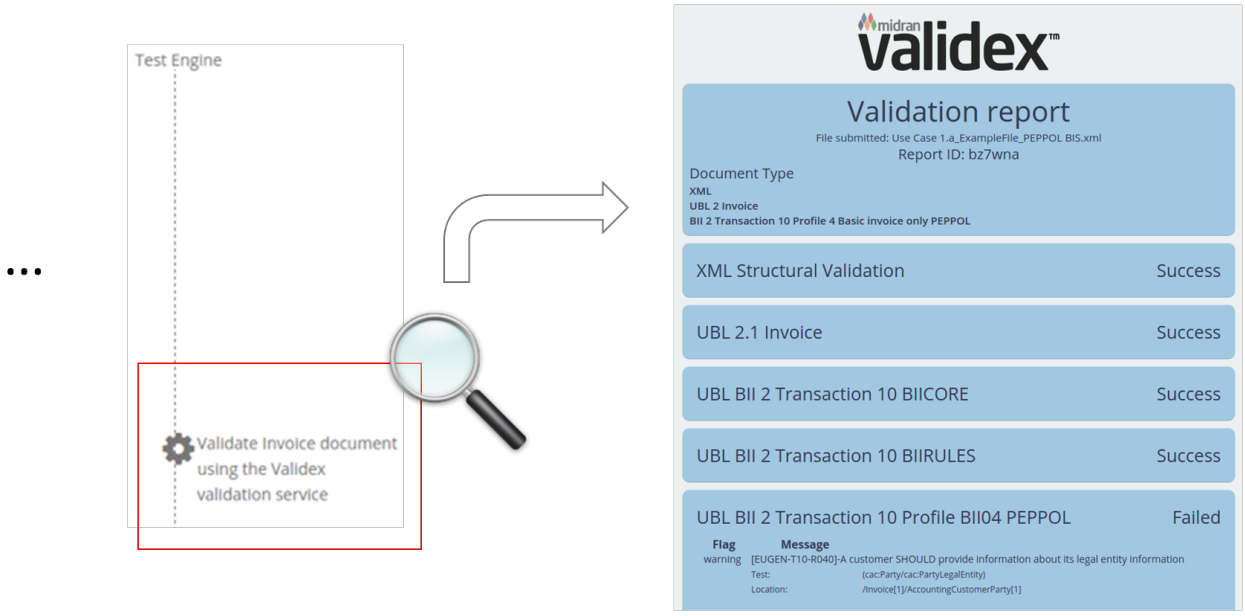

Figure 23-1: Test Report

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Document xmlns="urn:iso:std:iso:20022:tech:xsd:pain.008.001.02"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:iso:std:iso:20022:tech:xsd:pain.008.001.02 pain.008.001.02.xsd">

  <CstmrDrctDbtInitn>
    <GrpHdr>
      <MsgId>1228</MsgId>
      <CreDtTm>2014-07-31T01:46:21</CreDtTm>
      <NbOfTxs>201</NbOfTxs>
      <CtrlSum>157426.20</CtrlSum>
      <InitgPty>
        <Nm>TEST</Nm>
      </InitgPty>
    </GrpHdr>
    <PmtInf>
      <PmtInfId>SDD-FRST-2014-08-07</PmtInfId>
      <PmtMtd>DD</PmtMtd>
      <BtchBookg>false</BtchBookg>
      <NbOfTxs>201</NbOfTxs>
      <CtrlSum>155196.11</CtrlSum>
      <PmtTpInf>
        <SvcLvl>
          <Cd>SEPA</Cd>
        </SvcLvl>
        <LclInstrm>
          <Cd>CORE</Cd>
        </LclInstrm>
        <SeqTp>FRST</SeqTp>
      </PmtTpInf>
      <ReqdColltnDt>2014-08-07</ReqdColltnDt>
      <Cdtr>
        <Nm>TEST</Nm>
      </Cdtr>
      <CdtrAcct>
        <Id>
          <IBAN>FR7555555555555555555010030</IBAN>
        </Id>
      </CdtrAcct>
      <CdtrAgt>
        <FinInstnId>
          <BIC>BREDFRPPXXX</BIC>
        </FinInstnId>
      </CdtrAgt>
      <CdtrSchmeId>
        <Id>
          <PrvtId>
            <Othr>
              <Id>FR89ZZZ544444</Id>
              <SchmeNm>
                <Prtry>SEPA</Prtry>
              </SchmeNm>
            </Othr>
          </PrvtId>
        </Id>
      </CdtrSchmeId>
```

Figure 23-2: A Sample ISO 20022 Pain Message

23.4 Application in a Truck Manufacturer Test Scenario

A major European truck manufacturer has implemented a validation service using the GITB integrated tool Validex (https://scania.validex.net/)

The service performs document conformance testing for ISO20022 payment initiation messages: pain01.

The submit to Validex engine uses API interface that is compliant to GITB test bed which allows the truck manufacturer to extend their validation functions to include the message transport specifications by extending the validation to the GITB platform.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Document xmlns="urn:iso:std:iso:20022:tech:xsd:pain.008.001.02"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:iso:std:iso:20022:tech:xsd:pain.008.001.02 pain.008.001.02.xsd">

  <CstmrDrctDbtInitn>
    <GrpHdr>
      <MsgId>1228</MsgId>
      <CreDtTm>2014-07-31T01:46:21</CreDtTm>
      <NbOfTxs>201</NbOfTxs>
      <CtrlSum>157426.20</CtrlSum>
      <InitgPty>
        <Nm>TEST</Nm>
      </InitgPty>
    </GrpHdr>
    <PmtInf>
      <PmtInfId>SDD-FRST-2014-08-07</PmtInfId>
      <PmtMtd>DD</PmtMtd>
      <BtchBookg>false</BtchBookg>
      <NbOfTxs>201</NbOfTxs>
      <CtrlSum>155196.11</CtrlSum>
      <PmtTpInf>
        <SvcLvl>
          <Cd>SEPA</Cd>
        </SvcLvl>
        <LclInstrm>
          <Cd>CORE</Cd>
        </LclInstrm>
        <SeqTp>FRST</SeqTp>
      </PmtTpInf>
      <ReqdColltnDt>2014-08-07</ReqdColltnDt>
      <Cdtr>
        <Nm>TEST</Nm>
      </Cdtr>
      <CdtrAcct>
        <Id>
          <IBAN>FR7555555555555555555010030</IBAN>
        </Id>
      </CdtrAcct>
      <CdtrAgt>
        <FinInstnId>
          <BIC>BREDFRPPXXX</BIC>
        </FinInstnId>
      </CdtrAgt>
      <CdtrSchmeId>
        <Id>
          <PrvtId>
            <Othr>
              <Id>FR89ZZZ544444</Id>
              <SchmeNm>
                <Prtry>SEPA</Prtry>
              </SchmeNm>
            </Othr>
          </PrvtId>
        </Id>
      </CdtrSchmeId>
```

Figure 23-3: A sample ISO 20022 pain message

23.5 Stakeholders

This test scenario may be of interest for Standard Development Organizations (SDOs), industry consortia, companies, public authorities that want to leverage their existing Testing Resources.

In the specific case, Midran Ltd (as producer of Validex test bed) was involved.

References

[CEN10] *CWA 16093:2010 Feasibility study for a global eBusiness interoperability test bed* http://www.cen.eu/cen/Sectors/Sectors/ISSS/CEN%20Workshop%20Agreements/Pages/downloadArea.aspx

[CEN12] *CWA 16408:2012 Testing Framework for Global eBusiness Interoperability TestBeds (GITB)*

[TAG] OASIS Committee Note Draft 03, "*Test Assertions Guidelines*", June 2011 http://www.oasis-open.org/committees/download.php/42479/testassertionsguidelines-cnd-03-Jun03.pdf

[TAML] OASIS Committee Specification Draft 05, "*Test Assertions, Part 2 Test Assertions Markup Language Version 1.0*", June 2011 http://www.oasis-open.org/committees/download.php/42478/testassertionmarkuplanguage-1.0-csd-05-Jun07.pdf

[WSI10] *WS-I Testing Tools V2 for Basic Profiles 1.2 and 2.0, Web Services Interoperability*, 2010, http://www.ws-i.org/