Document filename: A Guide to the Toolkit Workbench			
Directorate / Programme	Solution Assurance	**Project**	Interoperability Toolkit
Project Manager	Debbie Chin	**Status**	Release
Owner	Damian Murphy	**Version**	V3.5
Author	Damian Murphy and Richard Robinson	**Version issue date**	24.10.2014

A Guide to the Toolkit Workbench

Revision History

Version	Date	Summary of Changes
0.1		First draft for comment
1.0		Release with ITK 1.1
2.0		Uplift for ITK release 2
3.0		Update examples, inclusion of Spine Validate, quickstart guide and corrections
3.1		Update for new TKW Installer and addition of Visual Validator and TLS Mutual Authentication
3.2		Futher updates for TLS Mutual Authentication
3.3		Update to CRL GET functionality
3.4		Rebranding of document for HSCIC
3.5		Update of the minimum Java JRE version from 1.6 to 1.7 Removal of FileCM Document Reference as versioned elsewhere

Reviewers

This document must be reviewed by the following people: author to indicate reviewers

Reviewer name	Title / Responsibility	Date	Version
Jayne Murphy		24/10/2014	3.5

Approved by

This document must be approved by the following people: author to indicate approvers

Name	Signature	Title	Date	Version
Richard Dobson			24/10/2014	3.5

Glossary of Terms

Term / Abbreviation	What it stands for

Related Documents

These documents will provide additional information.

Ref no	Doc Reference Number	Title	Version
1	NPFIT-SHR-QMS-PRP-0015	Glossary of Terms Consolidated.doc	13

Document Control

The controlled copy of this document is maintained in the HSCIC corporate network. Any copies of this document held outside of that area, in whatever format (e.g. paper, email attachment), are considered to have passed out of control and should be checked for currency and validity.

Contents

1. About this Document

1.1 Purpose

This document describes the use and configuration of the TKW “Toolkit Workbench” tool, and associated software. These are distributed to ITK service providers. The Testbench includes logic and supports configurations to validate both ITK and Spine messages.

1.2 Audience

The guide is written for all users and interested parties – managers, analysts, architects, development and test staff. It is also intended for HSCIC staff involved in the definition of ITK service standards, message validation and test support for messaging behaviours. The document has an overview to each section, followed by a detailed description for more technical readers. Note that the guide is intended to cover all aspects of using the Testbench, including how to configure it, at times in significant detail. It is ***NOT*** necessary to read and fully to understand this complete document just to use the Testbench.

See Appendix A for a QuickStart Guide

1.3 Content

The Toolkit Workbench is distributed as a single installation file, and aggregates a variety of test support facilities. This document is organised functionally:

- Introduction
- Installation
- Use
- Configuring properties
- Rules
- Validations

Documentation on the behaviours and validations for particular services are out of scope for the guide, and are covered in the module/domain specifications.

2. Introduction

The Toolkit Workbench itself is a set of service interfaces.

Suppliers of various types are invited to offer both ITK-ready sending and receiving systems. This means that:

- There is no single reference implementation to establish authoritative behaviours against which production implementations can be assessed. This is in contrast to Spine, where because there is only ever one version running at any given time: the production implementation is its own reference.
- ITK service providers may lack counterparties against which to test their systems. Even when a supplier can both send and receive ITK messaging, or is in a partnership which offers both, lack of an independent counterparty leads to increased risk of interoperability failures.

ITK services are called by passing messages. The structure and content of the messages are defined in the ITK domain/module specifications. In some cases, those structures can be very simple – for example, verify NHS Number in SMSP where only the NHS number and date of birth are sent in the request. Others can be very complex, for example the ITK "Correspondence" messages. For complex types, manual verification that a system is sending conformant messages is at best very slow and unreliable. Some machine-processable mechanism is required to perform message validation. Further, "Correspondence" highlights that no single validation approach can demonstrate full conformance with the specifications – so validation requires a range of techniques to be applied.

The Toolkit Workbench operation is not restricted to ITK messaging. It has been designed to be flexible and will validate both ITK and Spine messaging.

The Toolkit Workbench is an easily-deployable system which addresses each of these issues. It is used in a variety of ways to provide test and development support.

Although TKW is used for supplying evidence for accreditation activities, it also can be used as part of development and should be used for transmitting, receiving and validating messages as quickly and as often as possible to minimise rework.

2.1 Validation

As shown in figure 1, the interface specifications will detail machine-processable validation rules to provide conformance checks on messages created by a sending system. The Toolkit Workbench hosts a mechanism for executing validations, on multiple message samples, and producing a single validation report. The report can provide either a set of passes for various individual checks, or detailed descriptions of non-conformance depending on the outcome of the analysis of the message samples.

While the supplier must provide samples of their messages, and run the Testbench, the actual validation process is automatic and is driven by a configuration file provided as part of the interface specification. It therefore requires no configuration by the supplier, beyond telling the Testbench where to find the messages for analysis, and where to write the report.

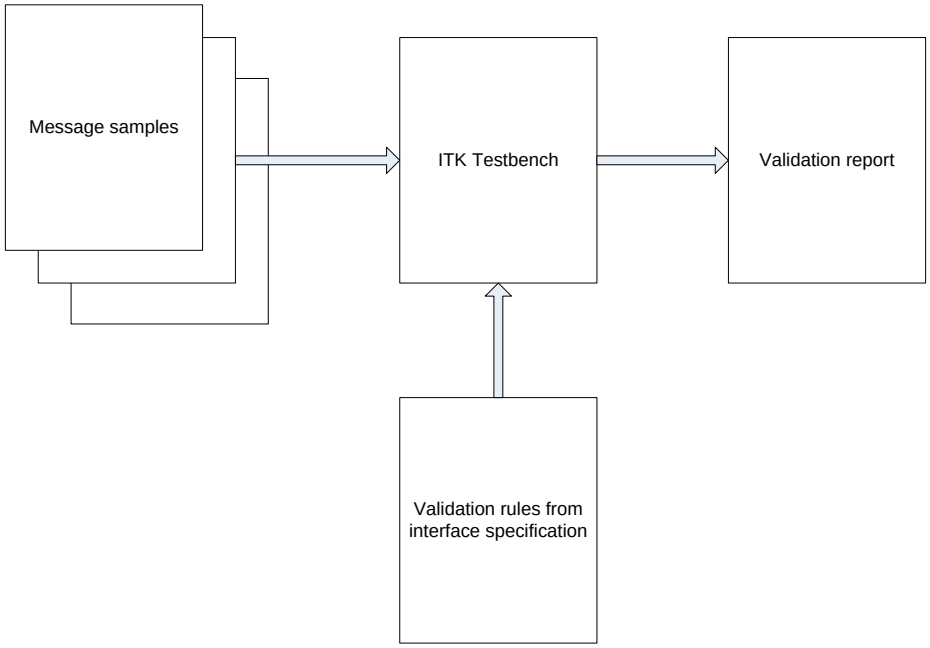


Figure 1 - Message Validation

2.3 Service simulation

During development, a client system needs to be tested against the services it is designed to use. The messages passed to the service, the behaviour of that service and the types of responses returned to the requestor are defined in the interface specification.

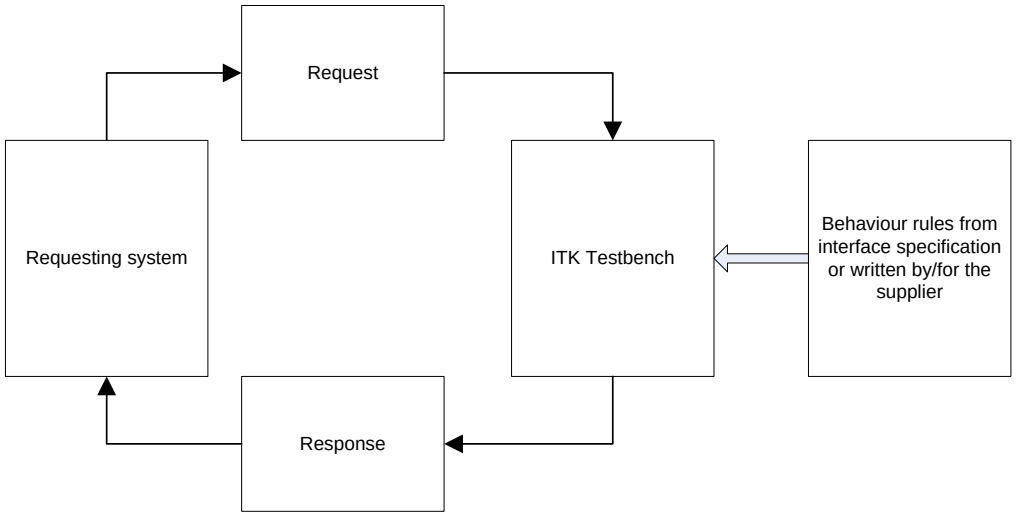


Figure 2 - Simulation

As shown in figure 2, the Testbench provides a simulated ITK “receiver” which can accept a message, and return a response in a way that conforms to the ITK service and infrastructural specifications[1]. The rules to support simulated message behaviour, and associated data such as templates for constructing realistic responses, are part of the interface specification and do not require configuration by the supplier. The Testbench supports all of the infrastructural options from the ITK specification – HTTP/TLS, message signing,

[1] Note that a “receiver” is just that – it may be a simulation of ITK middleware for client testing, or a simulation of a client for testing middleware.

synchronous and asynchronous messaging, and the queue collection interface. These are supported optionally (and can be turned on or off as required by the supplier) and for test purposes the simulation can be run “in clear” with no encryption or signing.

By constructing behavioural rules, and “templates” for responses, ITK interface specification developers are at the same time making the reference implementation of that interface. This approach also saves suppliers’ time by avoiding much duplication of effort – and inconsistent implementation – in each vendor making their own test harness. There is, however, no bar to more advanced suppliers, or those with particular needs, constructing their own test cases – and this document includes information on how to do so.

The approach permits more thorough testing, because the simulation exposes all of the technical interactions between a sender and receiver, and allows testing of error and other conditions that are otherwise very difficult to bring about in a controlled way.

2.3 Message transmission

The counterpart to simulation is the ability to send a message to a service. During service development, a supplier will need to test their system’s reaction to the messages it is designed to receive, under a variety of circumstances.

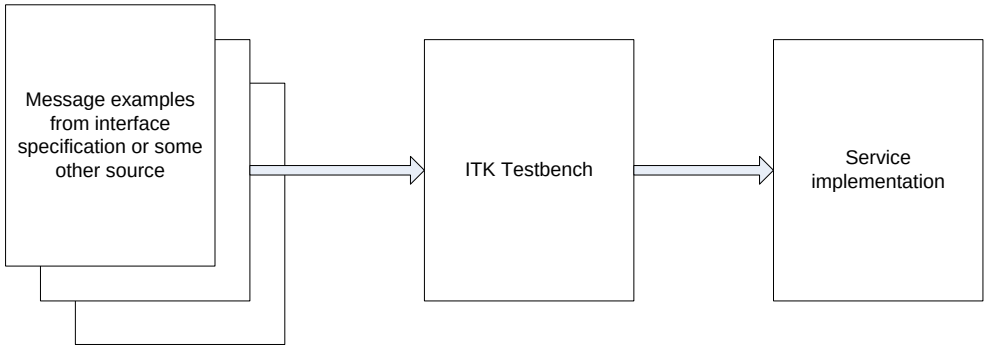


Figure 3 - Transmission

As shown in figure 3, the Testbench will transmit messages to a receiver. The interface specifications contain sample messages which can be used for this purpose, or samples can be sourced from elsewhere[2]. This requires some minimal configuration from the supplier, who is required to tell the Testbench where the messages are to be read, and where they are to be sent. The transmission system supports all of the infrastructural options from the ITK specification – HTTP/TLS, message signing – these can be turned on or off as required by the supplier. For test purposes the transmission can be run “in clear” with no encryption or signing.

[2] Possible sources include sending systems suppliers, additional examples not provided as part of the interface specification, or examples constructed by the service implementation supplier themselves.

3. Installation

3.1 System requirements

The Testbench should run on most modern desktop or laptop hardware. It is not intended as a performance tool so no detailed analysis has been performed on its minimum resource requirements, or its likely performance on any given platform.

Both the Testbench and the support programs (including the installer) are written in Java, using a Java 7 JDK. It has been tested on Java using JRE 1.7.0 and this should be considered a minimum recommended requirement.

3.2 Installation Process

The Testbench and its configurations are distributed as a single jar file, “tkwinstaller.jar”. This contains the binaries, documentation and configuration sets and can be run by double-clicking from the desktop. A Testbench installation contains the binaries/documentation, configurations, contributory and ancillary tool folders. The installer program reports its versions, and asks the user for the installation location, user identification and optionally which elements to install. The installation must be to a location where there are no space characters (or any others illegal in a URI) on the path[3]. On UNIX machines this is typically the case by default. On Windows it requires installing to somewhere other than a location underneath the users’ home directory[4] - and the “Install to” directory will not be accepted if it contains spaces. The user can selectively install any of the “configs” or the “contrib” folders by checking the tree structure in the “Installation Options”. The TKW.jar file and documentation may/may not be installed if, for example, only a new set of configurations are to be installed.

The installer unpacks an embedded zip file to the installation directory. Configurations are stored as templates, which are updated as they are unpacked to “localise” them for the selected directory. Once the installation is completed, the installer will display a message. At this point, configurations will be correct for the selected location. If they are subsequently moved, it is the user’s responsibility to ensure that the locations given in the configuration files are changed to reflect this. The change may be done by re-running the installer, or by some other means.

Note: the schema config library is required in addition to any of the spine validation config folders.

[3] This is to avoid the need to quote or otherwise escape values in the configuration files, in the interests of usability.

[4] For example, the configurations in the “examples” directory assume c:\temp\tk

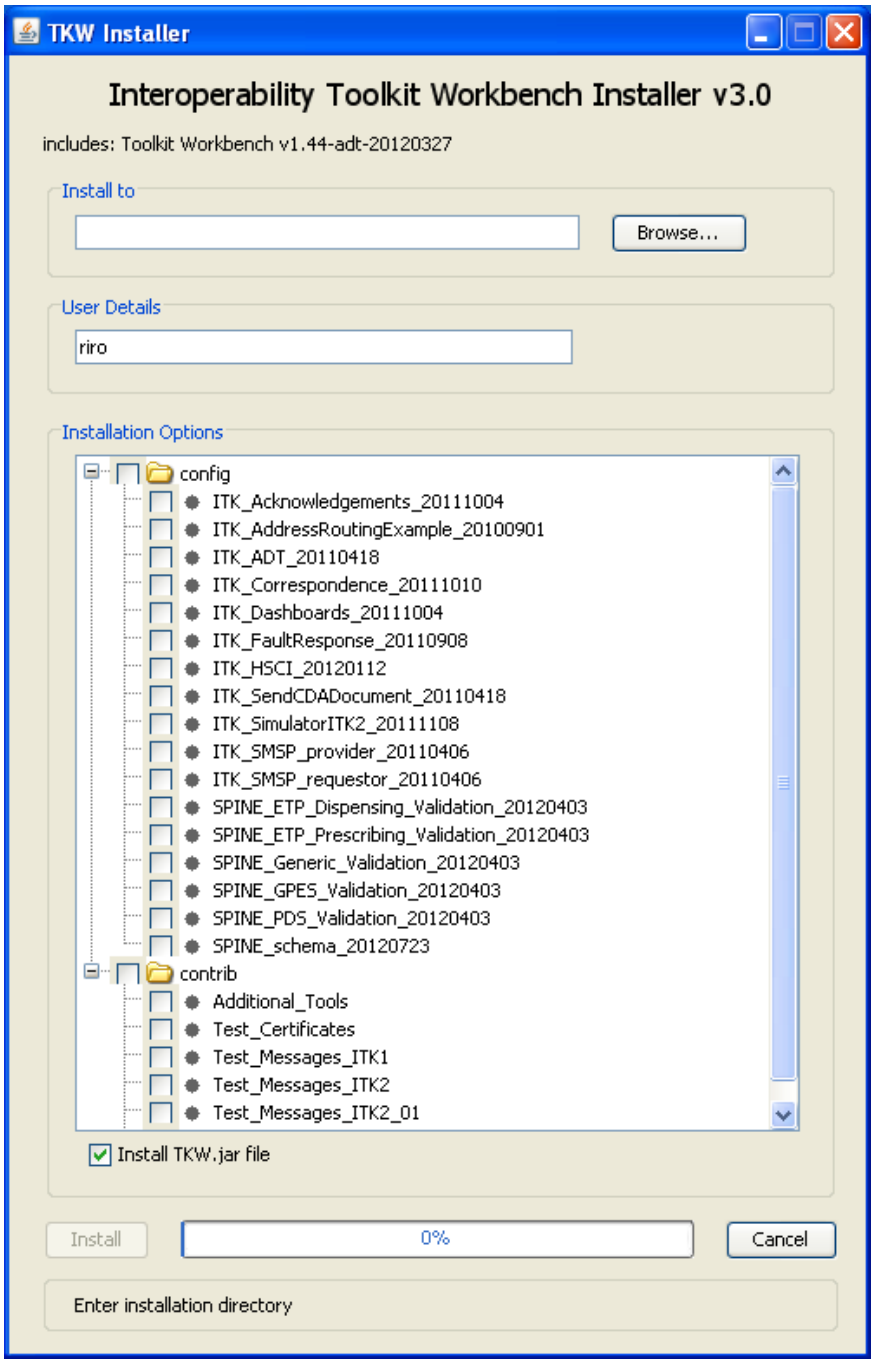


Figure 4 - TKW Installer

3.3 The `contrib` directory

Immediately underneath the installation's "TKW" directory, is one called "`contrib`". This contains various useful things that are not part of the main distribution. The content of this directory is not supported. The "`contrib`" directory contains a number of sub-directories each with their own descriptions and content. The folder contains the following directories:

- Additional_Tools folder containing:

- PHconverter – this is a stand-alone pipe-and-hat to XML converter tool. This is designed to be used to convert PH messaging for schema validation. See the readme.txt within this folder for running instructions. PHconverter uses the same code for pipe-and-hat conversions as TKW does internally.
- VisualValidator – This is a graphical interface for using the TKW Validator functionality. It allows users to drag and drop multiple message files into the application to be validated. The readme.txt file explains how to use the application
- Test_Messages_ITK1 – A directory containing ITK1 sample messages, subdivided by TKW mode type
- Test_Messages_ITK2 – A directory containing ITK2 sample messages, subdivided by domain
- Test_Messages_ITK2_01 – A directory containing ITK2.01 sample messages, subdivided by domain
- Test_Certificates – a directory containing rootCA, subCA and java keystore files used for the testing of TLS and signing.

4. Using the Toolkit Workbench

The Testbench is a command line program. Wrapping a GUI around it was considered, but rejected on the grounds that any complexity is in the configurations, not in calling the simple, two-argument command line invocations – so a GUI adds little or nothing to usability in starting the Testbench. The Testbench is a sophisticated tool and probably unsuitable for users unqualified for anything other than a point-and-click interface.

This section describes running the Testbench using an existing configuration. More advanced topics, such as turning on message signing, authentication and TLS, are covered later, in the sections on how to configure the Testbench.

The Toolkit Workbench is run in one of several “modes”. These support ITK message validation, Spine message validation, ITK transmission and ITK receiver simulation. The mode is selected by an argument on the command line:

```
java –jar TKW.jar –validate {path}tkw.properties

java –jar TKW.jar –transmit {path}tkw.properties

java –jar TKW.jar –simulator {path}tkw.properties

java –jar TKW.jar –spinevalidate {path}tkw.properties[5]
```

These run the Testbench in ITK validation, transmitter, simulator and Spine validation modes respectively. For those unfamiliar with running Java applications, “`java`” calls the Java run time environment (JRE), and “`-jar TKW.jar`” tells the JRE to run the Testbench from the “`TKW.jar`” file. The rest of the command line is for the Testbench itself – first the mode selector, and then the name of the “properties” file that contains the information that the Testbench needs to operate.

[5] See section 8

In these examples, it is assumed that the user is “in” the directory to which the binaries were installed, and therefore that the “TKW.jar” file is in the current directory. As the properties file will likely have been installed to a different location, it needs to be identified precisely, this is done by replacing `{path}` in the examples with the actual path to the place where the properties file was installed. This will depend on where it was installed, and what `{path}` actually looks like will depend on what sort of system is being used. On a Windows machine, where the configurations were installed in D:\TKW\Correspondence the command line for the validator would be:

```
java –jar TKW.jar –validate d:\TKW\Correspondence\tkw.properties
```

The command lines to run the other modes would be modified in a similar way. Note that because the configurations in the properties file specify fully-qualified paths, that file could be put in the same directory as the jar file, in which case no `{path}` is needed. In the interests of cleaner examples, the following sections omit `{path}` when specifying the properties file.

It is not recommended to install multiple copies of the jar file, as this risks breaking configuration control when new versions of the Testbench binaries are released. When installing additional configurations, either update the jar file, or suppress installation of the binary as described in section 3.2.

The properties file contains the information for all the Testbench modes – when it runs it only reads those relevant for the mode in which it has been run. The following sections on running the Testbench describe only those properties needed for basic operation.

Note: mode –spinevalidate is used to run the validator for Spine messages – this introduces the ability to read and check multipart MIME messages and ebXML that are not used in the ITK default SOAP 1.1/HTTPS transport.

NOTE: Visual Validator is a GUI tool for executing TKW validations (including spinevalidate) is located within the Additional tools folder in the contrib directory – see 0

4.1 Identifying your output and other general considerations

After installation, the properties file should be “ready to go”. However some basic changes will help identify any Testbench output better – from validation reports, to log files. Two properties “`tks.configname`” and “`tks.username`” are provided to do this, allowing identification of a site, and a particular configuration. On distribution, the “`tks.configname`” property will be pre-set to the release name of the configuration, but there is nothing special about this and it can be altered as the user requires. The “`tks.username`” property is set to the value entered for the “Username” by the installer program - it is recommended that this be edited, at least before running the validator, because the installer defaults this to whatever is available to Java, which might not be a suitable identifier (it contains no organisation information, for example).

Note that property names given below are case sensitive, and it should be assumed that property values are also case sensitive.

4.2 ITK Validation

The message validator is run using the command line:

```
java –jar TKW.jar –validate tkw.properties
```

It uses three basic properties – all of which are set on installation:

Property	Description
tks.validator.config	The fully-qualified name of the validation rules file.
tks.validator.source	The directory into which messages to be validated, will be placed.
tks.validator.reports	The directory in which the Testbench validator will write its report.

Validation is keyed on *service*. This is because the same message structure may be used for more than one service request, and it is possible for the rules governing the population of that structure to vary depending on the service for which it is used. For example, various services may need a patient to be identified using a common structure – but one may require a "local" identifier, and another an NHS number.

Messages for validation must identify the service they are for – and this can be done in one of two ways:

- SOAP submission. ITK services are SOAP messages which identify the service in the SOAP header. The validator will attempt to read the contents of the "Action" element in the WS-Addressing namespace, from the SOAP header. This requires that the message for validation is a well-formed, intact SOAP request.
- Using "VALIDATE-AS". The interface specifications focus on the message payloads (the content of the transmitted messages' SOAP Body). It will often be easier to obtain samples of payloads for validation. However the payload will typically identify only its type, and not the service for which it is being sent (this service information is in the SOAP and HTTP headers only). As the first line of the file to be validated, enter "VALIDATE-AS:", followed by a space, and then the name of the service. Figure 5 shows an example. "VALIDATE-AS" must not be used to override the action in a SOAP submission, as this will cause failures to be reported due to the validator being instructed to set an incorrect validation root.
- Distribution Envelope submission – If the service is not found either in the SOAP submission or via the Validate-AS keyword, then TKW will attempt to read the service attribute in the header of the DistributionEnvelope.

The Testbench validator will determine which of these approaches – SOAP submission or "VALIDATE-AS:" has been used, and they can be freely mixed in a set of messages to be validated. The validation process checks for the SOAP WS-A Action element first, and then looks for "VALIDATE-AS:" if the Action is not found.

4.2.1 Validating HL7v2 “Pipe and Hat” messages

The Testbench supports validation of ITK message content based on HL7v2. These can be presented either in an XML or an EDIFACT-style “pipe and hat” delimited format. Internally, the validator works on XML and so the Testbench validation mode can be used on HL7v2 ITK messages in this form, directly.

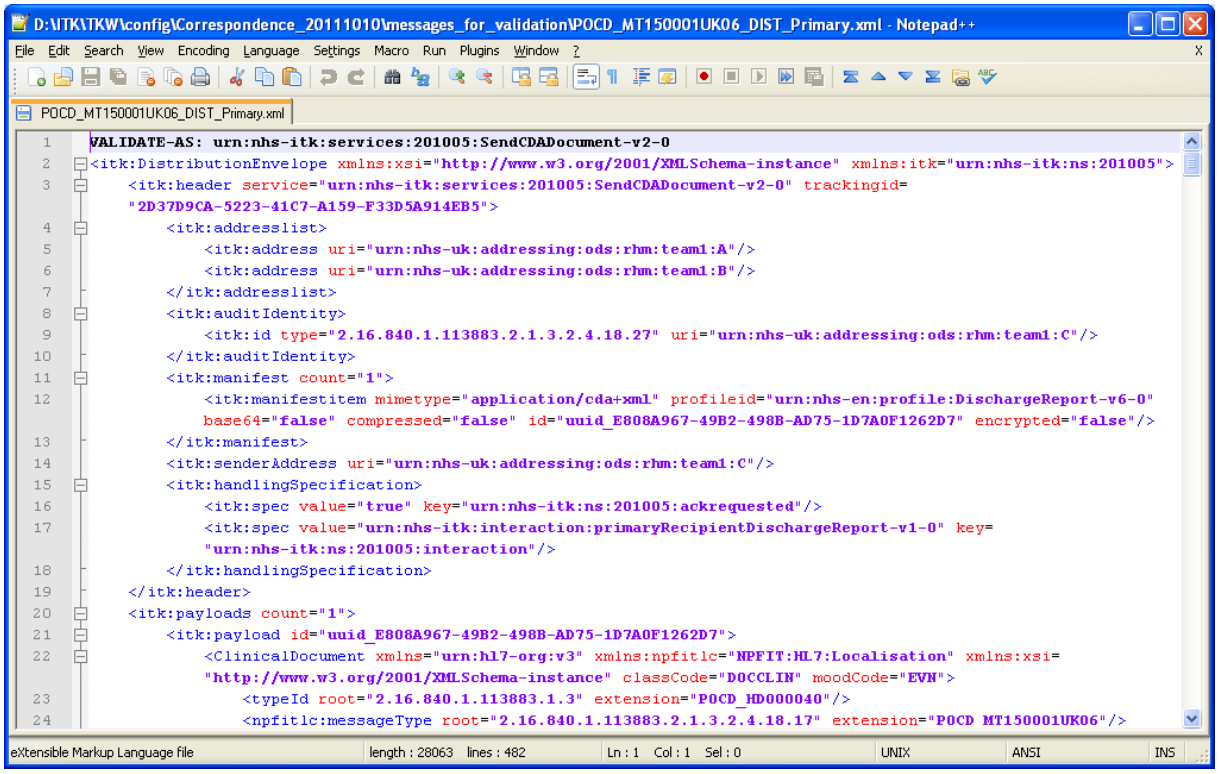


Figure 5 - "VALIDATE-AS:" in a SendCDA Document-v2-0 (Discharge) payload

Copy all the message samples for validation into the “source” directory, and run the validator. A single HTML file, with all the validation reports in it, is written into the “reports” directory, as shown in figure 6.

Note how the output is identified by a time stamp, in the document and in its file name. It is also identified by the values from the “`tks.configname`” and “`tks.username`” properties, at the top of the document.

Typically, the rule sets used for validation will be provided as part of the interface specification, and suppliers are not expected to develop their own. As such the details of the validation configuration files are not discussed in this section, but later in this document, where detailed configuration is covered.

TKW validator tool is not an absolute arbiter for validation and a failure can be ignored if a valid explanation is provided. A pass/fail result should be regarded as indicative as it is nearly impossible to include logic to handle all permissible variations of validation.

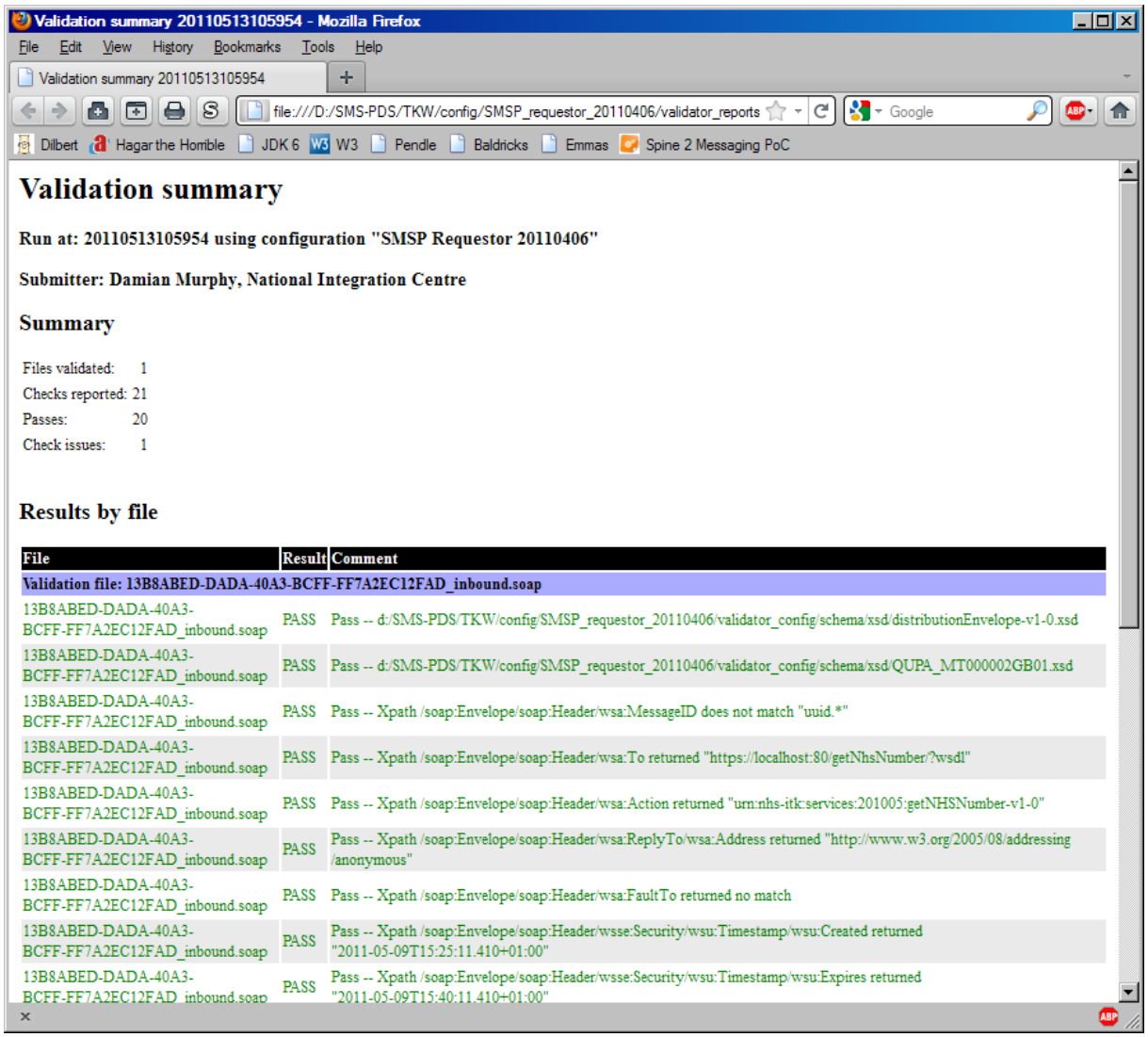


Figure 6 - Validation report

4.3 Spine Validation

Spine validation is executed and works in the same manner as ITK message validation using the command line, however the -spinevalidate argument is used:

```
java –jar TKW.jar –spinevalidate tkw.properties
```

TKW Spine validation is able to validate Full HTTP wrapped messages, ebXML multipart, SOAP HL7 and bare HL7.

The rulesets for the Spine Validation are transferred/ported from the existing OMVT2 tool and will act in the same manner as the existing OMVT2 tool

4.4 Visual Validator

Visual Validator is a graphical interface for the execution of TKW validations. It will selectively validate ITK or Spine messages.

To Install: Extract the tool from the zip file, navigate to contrib\Additional_Tools\ and execute setup.exe

To run: Navigate to contrib\ Additional_Tools\ and execute VisualValidator.application

1. Choose tkw.properties file

Click on Select and navigate to the appropriate *.properties file for the TKW configuration which is to be used

-"Clear down validation folder before validation" checkbox will delete all existing validation reports from the validator_reports folder

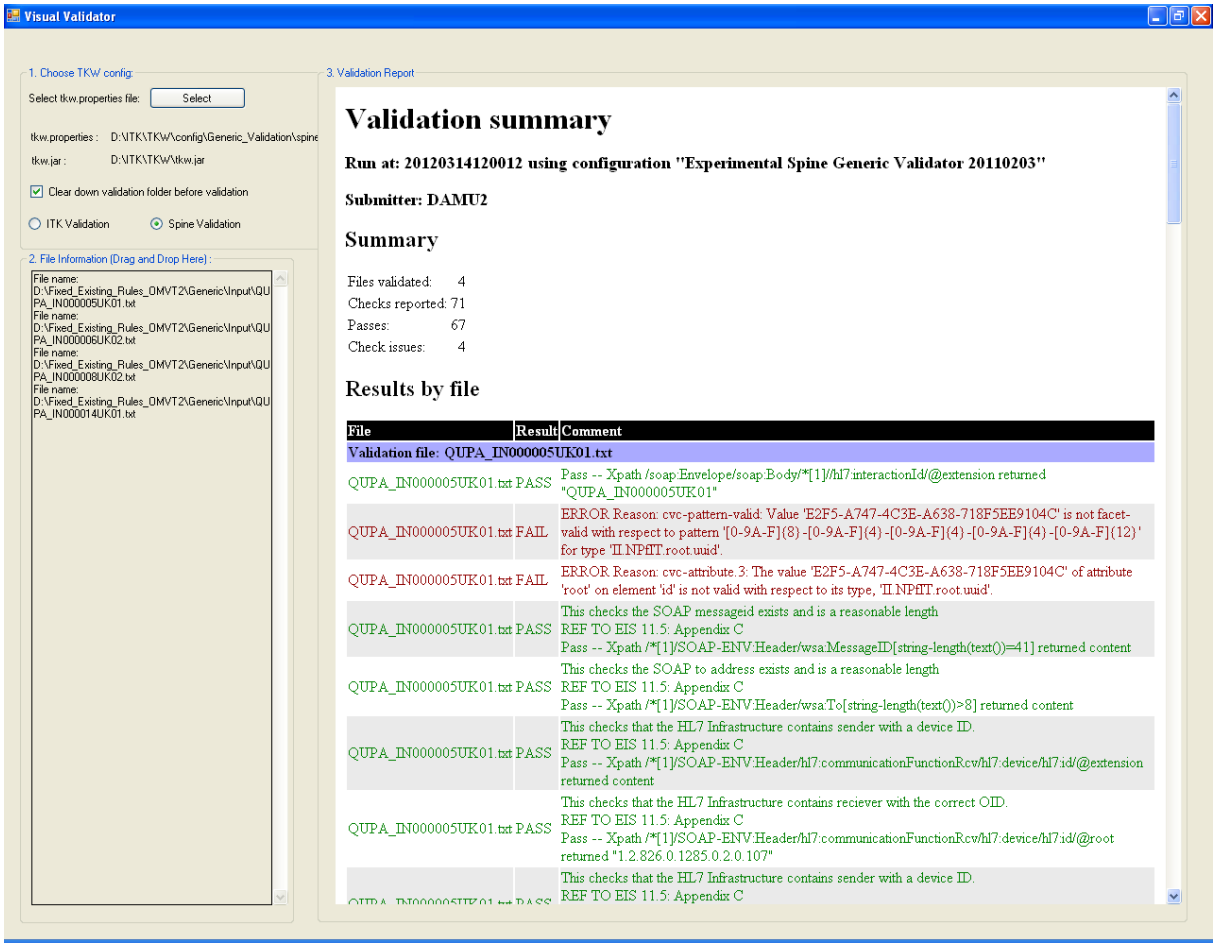
-The validation type radio button allows a choice of TKW or Spine validations

2. File Information (Drag and Drop here)

-Highlight and drag and drop into this area all the messages which require validation

3. Validation Report

Validation report appears in this panel. The report is also stored in the TKW Validation folder



4.5 Transmitting ITK Messages

The Testbench will transmit messages, and batches of messages, using the command line:

```
java –jar TKW.jar –transmit tkw.properties
```

Transmission is controlled by the following basic properties:

Property	Description
tks.transmitter.source	The directory into which messages to be sent, will be placed.
tks.transmitter.send.url	The URL to which messages are to be sent.
tks.sender.destination	The directory to which messages are written as-sent (with HTTP headers) are copied.
tks.transmitter.send.chunksize	If present and set non-zero, an HTTP 1.1 “chunked” request is sent with the given chunk size.

Messages to be sent are SOAP envelopes, placed into the directory indicated by the “`tks.transmitter.source`” property. Figure 7 shows the top of a SOAP envelope ready for transmission.

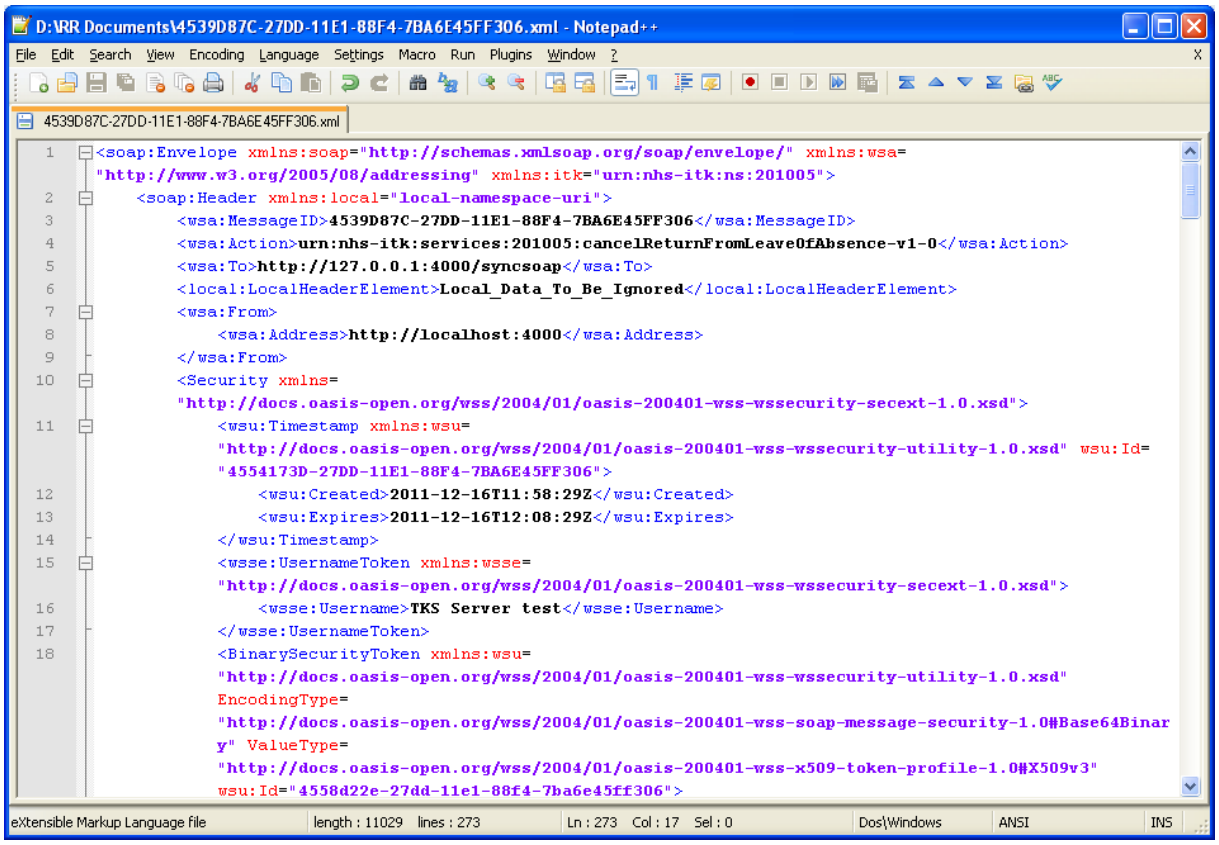


Figure 7 - Message ready for transmission

One call to the Testbench transmitter will send all messages in the "source" directory, to the URL given in the "`tks.transmitter.send.url`" property. Note that this is not a load-generating tool, messages in the source directory are sent one at a time.

The transmitter sends only, and will receive only synchronous responses. It does not provide a mechanism for receiving asynchronous responses for messages which it sends. However, ITK interactions are all "one way"[6], so an instance of the Testbench run in simulator mode can accept an asynchronous response.

4.6 Simulating an ITK receiver

The simulator mode is an ITK test harness that allows a message sender to talk to a counterparty that implements the ITK specifications. Responses can be synchronous or asynchronous, and error conditions can be forced by the configuration. The simulator mode is run using the command line:

```
java -jar TKW.jar -simulator tkw.properties
```

The simulator is controlled by many properties, most of which are covered later under "advanced configuration". For basic operation, the following are important:

[6] Asynchronous request/response interaction patterns are sets of two or more one-way, synchronous calls, orchestrated at a level above the calls themselves.

Property	Description
tks.rules.configuration.file	This is the fully-qualified file name of the rules file, that controls the simulator's response to messages sent to it.
tks.savedmessages	The directory in which the Testbench will save copies of messages it receives.
tks.Toolkit.listenaddr	Address of the interface on which to listen (use "0.0.0.0" to listen promiscuously)
tks.Toolkit.listenport	Port on which to listen.

In the simulation, the Testbench receives a message and looks up the service in its rules (loaded from the rules configuration file). If it finds an entry for the service, the rules provide instructions on how to handle the message – returning a particular response, or handing the request off for other processing. The details of how the behaviour is configured are discussed later in this document.

In the stock configuration, the Testbench rules are not inherently "synchronous" or "asynchronous"[7]. Instead their behaviour depends on the URL they're sent to – and the nature of the handler that services that URL. Synchronous, asynchronous, pipe-and-hat all have different handles called via an associated URL and are configured in the tkw.properties file. Suppliers should note that this is an artefact of the way that the simulator works – ITK itself will not give a synchronous service, asynchronous behaviour in this way, or vice versa. Services **MUST** be implemented, and used, according to the messaging patterns stated in the ITK interface specification.

4.6.1 Queue Collection

The stock simulator properties implement the QueueCollection services, as described in the ITK interface specification. "QueueCollection" delivers an asynchronous message to a "subscriber name" – a queue maintained by the ITK software from which the actual recipient can collect its messages, without having to implement and secure a socket service. Subscriber names, and the types of queue used, are given via the

[7] That is, the Testbench rules and responses can be applied equally to synchronous or asynchronous requests – or to the same request sent either way. Real ITK services are not like this, and are conformant only against the messaging pattern given in the interface specification.

"tks.queues.configfile" property. This contains the full-qualified name of a file which lists subscriber name, queue type[8], and a timeout value in seconds.

In the ITK specifications, the way in which asynchronous messages may be addressed to a queue is not defined. This is intentional as an optimal design is still a matter for discussion, but broadly directed by implementation. The Testbench supports two mechanisms, both of which are based on the content of the wsa:ReplyTo and, if present, the wsa:FaultTo elements of an asynchronous request.

Explicit Subscriber Naming

An asynchronous request specifies a return address in the wsa:ReplyTo header element. This address is a WS-Addressing endpoint reference which is a URI. The URI may be an HTTP URL, or it may be given by a subscriber name (set up using the "tks.queues.configfile" property), identified by having "urn:nhs-itk:subscriber:" prepended:

```
<wsa:ReplyTo>
   <wsa:Address>urn:nhs-itk:subscriber:example-
    subscriber</wsa:Address>
</wsa:ReplyTo>
```

It is an error to specify a subscriber name in this way, where the queue does not exist in this instance of the Testbench.

Address/Action Mapping

An address/action map intercepts a message, with a particular SOAP action (or "ALL"), and directs it to a subscriber name, rather than attempting to delivery it directly. To use this, the property "tks.delivery.routingfile" must be set to the fully-qualified name of a file containing the addresses and services for which queue collection is to be used. This file maps URLs and SOAP actions, to "subscriber names":

https://my-endpoint.com/messaging queryresponse my_subscriber

This will cause any message deliveries for the SOAP action "queryresponse", for the URL "https://my-endpoint.com/messaging" to be delivered to "my_subscriber". The real "my-endpoint.com" can then collect its "queryresponse" messages by sending queue collection requests specifying the "my_subscriber" name.

Each line of the routing file contains a single mapping of this sort – the file can be used to specify as many mappings as needed. Routing information which is not used because either the URL or the SOAP action (or both) is unused in a given test has no effect.

4.6.2 Message Signing

The ITK infrastructure specifications mandate that "original" messages be signed, and described what should be signed (the timestamp) and how. The specifications also provide an example of what a signed request looks like. An "original" message is one which needs a

[8] In the current version, because the "GetMessagesWithConfirmation" interface is not implemented, only the "org.warlock.tk.internalservices.queue.SimpleQueue" type can be used.

new connection – so it might be a request from one system to another, or an asynchronous response. Synchronous responses, acknowledgments and SOAP faults are not signed.

The Testbench supports message signing both for transmitter, and for asynchronous responses from simulator modes. This is controlled by values in the properties file, and in the distributed file, message signing is turned off.

The Testbench implements ITK message signing, using the standard capabilities of the Java 6 API. To activate message signing requires changing properties to tell the Testbench to load its internal signing service, and where to find its key store.

To load the internal signing service, find the "`tks.servicenames`" property[9]. Make sure that it includes "Signer", somewhere in the set of names following the property name. For example (the "\" is where the line wraps, in the file this is all on one line):

```
tks.servicenames Toolkit SoapHeader RulesEngine Sender Signer \
DeliveryResolver Processor QueueManager
```

The rest of the properties are information about the Java keystore file:

[9] Testbench internal services are described later. For now, all that needs to be known is that the Testbench is constructed from a number of "internal services" that can be loaded independently to provide functions. The discussion here shows how to get Testbench to load the "Signer" service when it boots.

Property	Description
tks.signer.keystore	This is the fully-qualified file name of the keystore file.
tks.signer.keyalias	The “alias” used to identify the key to be used for signing the message. This should be the same as the username that will be used in the messages’ “Username” header, because it will be the X500 Principal used in the signature, which is what will be checked by the receiver as part of validation.
tks.signer.storepassword	The password put on the keystore when it was created.
tks.signer.keypassword	The password put on the key when it was created.
tks.asynchronousreply.wrapper	This is the fully-qualified name of a template used to make the SOAP header of an asynchronous response. It is indirectly used to control signing, because for an asynchronous response the message will only be signed if a timestamp is present.
tks.signer.showreference	Set this to “Yes” to have the signing and validation code write the “URI reference” data to the Testbench console when a message is signed, or when a signature is verified. It is this string (including whitespace and line-breaks) that is used to calculate the digest, which is included in the signature, and encrypted to form the signature value. Inconsistencies in the reference between the signing and verification operations are a common cause of signature verification failures.
tks.signer.digestalgorithm	TKW supports SHA-1 (default) and SHA-2 for 256 and 515 block sizes for the DigestMethod. Use SHA-1 OR SHA-256 OR SHA-512. Please note: the SignatureMethod is RSA SHA-1

The keystore is managed using the Java `keytool` which is not described in this document. Information on `keytool` can be found in the JDK documentation[10].

The Testbench will sign a message when all of the following are true:

- The internal “Signer” service is running
- It is “originating” a message (i.e. either transmitting, or sending an asynchronous response)
- A Timestamp element is present in the header, and has been set to the current time by substituting a tag[11] "`__TIMESTAMP__`"

As the Testbench is intended to allow testing before message signing might be ready in an ITK system (for example, when the target environment assumes TLS), the Testbench will function in the same way irrespective of whether message signing is turned on or not. The Testbench validation mode does not inspect the SOAP header, when provided, for any information other than the WS-Addressing “Action” element – so the presence or absence of a signature does not affect validation results.

Message signing is an implementation of the XML Digital Signature[12] specification, which uses XPointer references[13]. The Testbench implements the references uniquely, by generating DCE UUID values, for example:

```
<wsu:Timestamp xmlns="" wsu:Id="4612BCAA-06BE-11DF-BB82-
1BFA5B76D855">

…

<Reference URI="#4612BCAA-06BE-11DF-BB82-1BFA5B76D855">
```

Suppliers should note that this is an implementation choice, in the interests of a clean, unique reference for the signature. There is no particular recommendation that it should be followed.

4.6.3 Checking Message Signatures

The ITK infrastructure specification requires that a signed message should be verified on receipt. When message signing is turned on (see section 4.4.2 above), the Testbench will validate a signature both cryptographically, and by checking that the declared WS-Security “Username” element matches the X500 Prinicpal in the key information. If either fails, an HTTP 500 response is returned[14].

---

[10] The documentation is included with the JDK download at http://java.sun.com/javase/downloads/index.jsp, or online from http://java.sun.com/javase/6/docs/technotes/tools/solaris/keytool.html and http://java.sun.com/javase/6/docs/technotes/tools/windows/keytool.html

[11] See the section on Templates in this document.

[12] See http://www.w3.org/TR/xmldsig-core/

[13] See http://www.w3.org/TR/2003/REC-xptr-framework-20030325/

[14] Currently this is returned with an “Invalid signature” string. Future versions will return a SOAP fault or other response consistent with the finalised ITK specifications.

In order to avoid this signature verification dictating the pace of development and testing, it is possible to prevent it. When message signing is turned off, no signature checking is done. Alternatively, the property "`tks.signer.alwaysacceptsignature`" can be included in the properties file, set to "Yes". When this is done the Testbench will accept any messages sent to it, regardless of the validity of their signatures. If "`tks.signer.alwaysacceptsignature`" is set, "`tks.signer.showreference`" will have no effect as the URI de-referencer will not be called.

4.6.4 TLS

TKW supports both Server Authenticated TLS and Client Authenticated TLS (Mutual Authentication) message transmission, using the stock Java Secure Sockets Extension (JSSE) that is part of the Java runtime environment. It is recommended that the Testbench be used in clear for most test purposes, as doing so allows for easier "snooping" of network traffic for debugging purposes.

TKW test bench has the following properties for the use of TLS:

Property	Description
tks.receivetls	Instruct simulator mode to use TLS (Yes/No)
tks.sendtls	Instruct transmitter mode to use TLS (Yes/No)
tks.tls.truststore *	The fully-qualified name of the Java keystore (JKS) file containing certificate authorities.
tks.tls.trustpassword *	Password for the certificate authorities file.
tks.tls.keystore	The fully-qualified name of the Java keystore file containing certificates. Both certificates and authorities can be held in the same file, but both sets of properties must still be set.
tks.tls.keystorepassword	Password for the certificates file.

* See TLS Mutual Authentication guidance for use of the truststore below

Additionally the following properties are solely for TLS Mutual Authentication:

Property	Description
tks.tls.servermutualauthentication	Yes/no parameter for use as a mutually authenticated server
tks.tls.ma.filterclientsubjectdn	Optional (This is currently used for 111) - if present contains a value for a substring check against the peer subject DN from incoming Client system certificates.
tks.tls.clientmutualauthentication	Yes/no parameter for use as a mutually authenticated client.

Configuring the properties for Mutual Authentication:

The parameters must be set as indicated above, however for TLS mutual authentication, the cacerts truststore is used which is part of the JRE runtime environment. To configure this trusted certificate store the following must be performed:

- Copy all the test0x.jks keystores from contrib\Test_Certificates\Test0x into the certs folder in the config which is to be tested.
- Remove the tks.tls.truststore and tks.tls.trustpassword values from the properties configuration file (a # may be inserted at the start of the individual lines in order to “hash” them out)
- The transmitter url properties should be changed to a secure HTTP connection url (https) and it is recommended that the transmit and listen ports should be changed to the default secure port 443.
- Using Java “keytool” utility, add any trusted certificates to the cacerts Trusted Certs store. If the trusted certificate has inheritance from a root CA then all certificates must be added. Using a command prompt execute the following “keytool” import:
- keytool -import -trustcacerts -file /path/to/ca/ca.crt -alias CA_ALIAS -keystore $JAVA_HOME/jre/lib/security/cacerts
- Where “ca.crt” is the trusted certificate to be added (Here, tlssubca.crt and TKWCA.crt should be imported), “CA_ALIAS” is an identifiable alias for the certificate and “$JAVA_HOME” is the directory where the JRE is held locally. A password will be prompted - The default keystore password for the cacerts file is "changeit". While the documentation recommends that system administrators change the access rights and the password for the cacerts file, this password will probably work on developer or testing machines.

- The import can be verified using:
- keytool -list -v -keystore $JAVA_HOME/jre/lib/security/cacerts
- A detailed description using on "`keytool`" can be found in the JDK documentation[15].

Test Certificates

Certificates and keystores are provided for test purposes in the contrib/Test_Certificates directory. The directory contains certificates and keystores for several test cases, both for signing and TLS Mutual Authentication.

The TKWCA.crt root certificate authenticates 2 sub CA certificates (one each for Signing and TLS MA). The TKWCA and tlssubca certificates should be imported into the cacert trust store (as described above).

The following certificates are provided for testing and can be used to provide accreditation evidence:

- test01.jks contains a valid certificate which can be used for "good day" testing purposes
- test02.jks contains a valid certificate but can be used to test a "bad subject DN". On server instances the peer subject DN from the received certificate is checked against the property tks.tls.ma.filterclientsubjectdn. When TKS is operating as a server it will reject any messages encrypted with a non matching subjectDN.

  The string "Test Endpoint" will match on the subjectDN of all the certificates provided.

  In order to test the failure of this, Host suppliers should alter their subjectDN filter so that it no longer matches this string, Client suppliers should alter the this value in their TKW configuration.
- test03.jks contains an expired certificate
- test04.jks contains a revoked certificate. The certificate revocation list file is within the Test04 folder (test04revoked.crl). Within the Test Certificates pack,there is also a blank.crl which can be used for positive testing.

Passwords for the keystore certificates can be found in passwords.txt file in the contrib\Test_Certificates directory

Certificate Revocation List (CRL) Server Functionality

TKW can be configured to act as a CRL server, responding with TLS MA, signing or general CRLs.

The CRL server can be used by systems to request a revoked certificate list, which will allow it to demonstrate that the system will not respond to messages encrypted with certificates which have been revoked.

[15] However, a quick and more accessible introduction can be found at http://www.sslshopper.com/article-most-common-java-keytool-keystore-commands.html

(Note: When configured as a mutually authenticating host, TKW does not have the functionality to use this CRL server in order to identify and block incoming messages encrypted with revoked certificates).

To configure TKW as a CRL server, add the following entries to the tkw.properties file:

Property	Description
tks.tls.crl	The fully-qualified name of the TLS Certificate Revocation List
tks.signing.crl	The fully-qualified name of the Signing Certificate Revocation List
tks.crl	The fully-qualified name of the General CA Certificate Revocation List

The following port name must be added to the handler namelist:

HTTPGetCRL

e.g. tks.Toolkit.namelist qrequest qreqitk HTTPGetCRL

And the definition of the handler is also added to the handler definitions:

tks.Toolkit.HTTPGetCRL.class org.warlock.tk.handlers.HTTPGetCRLHandler

tks.Toolkit.HTTPGetCRL.path

The list is requested with an HTTP header including as a minimum:

- GET with the context path of:
- /gettlscrl for TLS mutual authentication
- /getsigningcrl for message signing
- /getcrl for General CA CRL
- content-length: 0
- connection: close

4.7 Versioning

TKW is an evolving tool with updates and bug-fixes added regularly. It is important that users maintain their version and update when newer versions are available. Users can obtain their tool's version number with the following command:

```
D:\ITK\TKW>java -jar TKW.jar -version
```

The latest version of the TKW can be downloaded from the HSCIC TRUD site or at Messaging Test Tools site. Users who have signed up to TRUD will receive notification when major updates have been made.

4.8 Issue Reporting

Issues can be reported to the TKW development team at itkaccreditation@hscic.gov.uk

When reporting an issue please clearly state:

- Date
- TKW Version Number
- Full Issue description, including all steps taken to the recreate the issue and if it is repeatable
- Inclusion of messages involved and logs from TKW, if applicable

5. ITK and Spine Message Validation Rules[16]

The ITK and Spine service specifications define messages passed with service requests. Those messages are described in terms of structure, and rules governing content. For XML messages, schema defines structure and can enforce some content rules.

The extent to which schema can help ensure that a message is "good" is limited both by the capabilities of schema, and by the processes that produce the ITK service specifications. In many cases, schema alone is unable to enforce a rule. In others, the rule is dependent on other content, or on the service for which the message is sent – in neither of these cases can schema help. But there are other techniques which can be applied to validation. Xpath[17] is a way of describing the location of a piece of data in an XML document and extracting it. Some rules can be written to inspect data extracted in this way – testing its value, presence or absence. XSLT[18] permits the production of sophisticated checks on XML documents that can include external reference data when needed.

[16] Unless stated otherwise within the text this section refers to both ITK and Spine validations

[17] See http://www.w3.org/TR/2007/REC-xpath20-20070123

[18] See http://www.w3.org/TR/2007/REC-xslt20-20070123

As described in section 0 above, the Testbench can validate messages, and can be presented with them either as payloads (plus a directive to say for which service the payload is intended), or as complete messages (that is, enclosed in a SOAP envelope). In each of these cases, the XML document that the validator has to work with starts in a different place. Many message types have “optional” content – and validating the content of an “optional” part of a message will fail, artificially, if that part is validly absent.

The Testbench’ validator can handle multiple validation techniques, and “conditional” validation. What it does is governed by the contents of a “validator configuration” file – as described in section 0, this is located by the `tks.validator.config` property.

A validator configuration file consists of a *set of service validation rules*. A service validation rule looks like this:

```
# Comments
VALIDATE servicename
# More comments
CHECK test_type resource
IF test_type resource
THEN
    CHECK test_type resource
ELSE
    CHECK test_type resource
ENDIF
…
```

5.1 Validator configuration syntax

A validator configuration file can contain as many service validation rules as required. Looking at the structure more closely, lines beginning with “#” are comments and ignored by the validator. Blank lines, or lines containing only whitespace, are also ignored. The configuration file is case sensitive and consists of a series of lines of text, the beginning of each line being one of VALIDATE, CHECK, IF, THEN, ELSE or ENDIF.

5.1.1 VALIDATE

Lines beginning “VALIDATE” introduce a new service validation rule. The name of the service to which the rules apply follow “VALIDATE” after a space or a tab. All parts of a script, following a VALIDATE line is part of the service validation rule, until either another VALIDATE is found, or the end of the file is reached. ITK validation rulesets require fully qualified service names and Spine validation rulesets require Spine Interaction ID only.

5.1.2 CHECK

Lines beginning CHECK are validations. A CHECK specifies the type of check, and one or more “resources” used for that validation. What the resources are depends on the validation type – they may be file names, or XPath expressions, Java class names or calls to SUBSET routines. The resources are separated by whitespace – and the number of resources also

depends on the type of validation. When a check is executed, its results are added to the validation outputs and form part of a report such as that shown in Figure 6. Each of the lines on that example, under “Results by file”, is the output of a CHECK line from the validator configuration.

ANNOTATION

A line beginning “ANNOTATION”, following a CHECK[19] will be output in the report for that check. An annotation MUST be either a line of plain text, or a valid fragment of HTML such that when it is included in the HTML validation report, that report remains viewable in a browser. The purpose of ANNOTATION is to include static text such as references to requirements.

5.1.3 IF

Lines beginning IF are conditions. A condition works in exactly the same way as a check, except that instead of its output being added to the service validation results, its pass/fail state is used to determine which validations are done next. When the IF “passes”, the validator executes the block of checks under “THEN”. Otherwise it executes the block of checks under ELSE. Conditions can be nested as required.

5.1.4 THEN

THEN is found only on a line by itself. Lines beginning THEN start a block of rules that are executed when a condition passes. Apart from comments, THEN must appear immediately after an IF. The block ends at either ELSE or ENDIF.

5.1.5 ELSE

ELSE is found only on a line by itself. It indicates the end of a block of rules under THEN, and the start of a block executed if the parent IF condition “fails”. The ELSE block ends at an ENDIF.

5.1.6 ENDIF

ENDIF is found only on a line by itself. ENDIF ends an IF. Service validation rule processing carries on unconditionally after an ENDIF.

5.1.7 SUBSET

Lines beginning “SUBSET” introduce a referenced subset within the validation ruleset. The name of the subset follows “SUBSET” after a space or a tab. All parts of a script, following a SUBSET line is part of the subset, until either another “SUBSET” or “VALIDATE” is found, or the end of the file is reached. A “SUBSET” is called from a CHECK validation using “CHECK sub <subsetname>”

5.2 Check types

The following check types are supported:

[19] Whilst it is legal to put an annotation after other validation script lines, only the output from CHECK is actually displayed, so any annotations will only appear if they follow CHECKs.

Type	Resources	Description
Schema	Fully-qualified name of the XML schema to be used for payload validation. Optional Xpath expression locating validation root	Payload will be schema validated using the XML schema file with the given fully-qualified name. Validation will start at the point indicated by the validation root Xpath expression, if present. Passes if no errors are detected.
ConformanceSchema	Fully-qualified name of the XML schema to be used for payload validation. Optional Xpath expression locating validation root	Payload will be schema validated using the XML schema file with the given fully-qualified name. Validation will start at the point indicated by the validation root Xpath expression, if present. Passes if no errors are detected.TKW will create a copy of the transformed CDA which the conformance schema is validated on in the validator_reports folder. This has the format "supportingData_"+message name+"-"+timestamp
Xpathequals	XPath expression Comparison string	Evaluates the XPath expression against the submitted message, and returns pass or fail depending on a case-sensitive match between what the XPath expression resolves, and the comparison string.
xpathnotequals	XPath expression Comparison string	As for "xpathequals", but the check passes if the string extracted by the XPath expression does not match the comparison string.
xpathcontains	XPath expression	Returns a pass if the value returned

	Comparison string	by the XPath expression contains the comparison string in a case-sensitive search.
xpathnotcontains	XPath expression Comparison string	Returns a pass if the value returned by the XPath expression does not contain the comparison string in a case-sensitive search.
Xpathexists	XPath expression	Returns a pass if the XPath expression resolves a non-null value, and not an empty string.
Xpathnotexists	XPath expression	As for "xpathexists", but returns a pass if the XPath expression returns null or an empty string.

Xpathmatches	XPath expression Regular expression	Returns a pass if the value extracted by the XPath expression matches the regular expression otherwise the check will return a fail.
Xpathnotmatches	XPath expression Regular expression	As for "xpathmatches", but a pass on "no match".
Xpathin	XPath expression Comparison string	Returns a pass if the value returned by the XPath expression does not contain one of the list of comparison strings in the search. . These are separated by space and encapsulated in quotes
Xslt	Fully-qualified name of transform file. Comparison string	Executes the XSLT-1 or XSLT-2 transform given in the file. The test passes if the output does NOT contain the comparison string).
Signature	None	Verifies a signature. Reports error if verification fails or if the signature is not present.

5.2.1 Notes on Schema checks

ITK Messages submitted as SOAP envelopes, and as payloads using the VALIDATE-AS method are processed in different ways. With a SOAP envelope, the validator must parse the message first, and extract the payload – which it does as the first element child node of the SOAP Body. This is a DOM Node and is checked using a DOM validator. The DOM validator reports schema non-conformance but is not able to provide information on the location of the problem in the input file[20]. Messages submitted using VALIDATE-AS where the check specifies a validation root, also use a DOM validator.

ITK Messages submitted as payloads using VALIDATE-AS, where the check specifies no validation root, are processed as files, with the first line removed. Schema validation failures in these cases are reported with line and column numbers from the input file. Reported line numbers should have 1 subtracted to allow for removal of the VALIDATE-AS line. Note that

[20] This is because it reports the content of an org.xml.sax.SAXParseException which does not report location when validating a document based on an in-memory DOM.

these values are given with reference to the submitted file containing the message – so, because it is perfectly legal XML to have the complete message on a single line (i.e. without line breaks), how useful this is depends on the layout of the submission. It is acceptable to "pretty-print" payloads prior to validation, as this will assist locating any errors reported by the validator.

For Spine validations VALIDATE-AS is not required as the HL7 payload contains the service name as the message root element. Spine messages can be validated as full HTTP POST, ebXML multipart MIME, SOAP/HL7 or bare HL7 messages. SOAP/HL7 messages are checked using a DOM validator as it will preserve the relative Xpaths used in the validator config but still allow use of the namespaces declared at a SOAP level.

ITK Distribution Envelope

The ITK distribution envelope has a schema where the "payload" element has content of type "xs:any". So for a message using the distribution envelope, the schema can only validate the envelope itself, and not the payload. To validate the payload it must be identified as the validation root using an XPath expression, and the appropriate schema selected to validate from the root element of that payload. For example, in the ADT "leaveOfAbsence" message:

```
CHECK schema distributionEnvelope-v2-0.xsd

CHECK schema ITKEvents_ADT_A21-2010-05.xsd

     //itk:DistributionEnvelope/itk:payloads/itk:payload[1]/*[1]
```

The first applies the distribution envelope schema to the complete request. That validates the distribution envelope, but it cannot validate the HL7v2 ADT payload. The second extracts the payload by specifying its location in the overall message, with an XPath expression. The ADT message schema is then applied to the extracted XML.

5.2.2 Notes on XPath checks

XPath expressions are compiled by the validator ahead of time, and the compiled expression is evaluated against the complete[21] submitted message. Because of this, and because it is not possible to know in advance what set of namespace prefixes will be used across all suppliers, the compiler provides its own namespace context. This namespace context supplies a set of prefixes for the namespaces in use in ITK messaging – and these prefixes MUST be used when writing XPath expressions for the validator[22]:

[21] The XPath validator does not strip any SOAP header from the submitted message as the schema validator does. XPath expression authors have to bear this in mind when validating payloads.

[22] This does not apply to XSL transforms, which are standalone XML documents and can provide whatever namespace bindings the developer considers appropriate.

Prefix[23]	Namespace URI[24]
ds	http://www.w3.org/2000/09/xmldsig#
dsig	http://www.w3.org/2000/09/xmldsig#
hl7	urn:hl7-org:v3
hl7v2	urn:hl7-org:v2xml
itk	urn:nhs-itk:ns:201005
SOAP	http://schemas.xmlsoap.org/soap/envelope
soap	http://schemas.xmlsoap.org/soap/envelope
test	urn:warlock-org:test
wsa[25]	http://www.w3.org/2005/08/addressing for ITK messaging http://schemas.xmlsoap.org/ws/2004/08/addressing for Spine messaging
wss	http://docs.oasis-open.org/wss/oasis-wss-wssecurity-secext-1.1.xsd

[23] Caveat lector: These prefixes are either all lower case, or all upper case. If any appear with an upper case initial character, then unless the entire prefix is upper case (as in “SOAP”), then this is an artefact of an over-enthusiastic word processor.

[24] Note: ITK pilot namespaces are no longer supported.

[25] ITK and Spine messaging use different namespace URIs for wsa addressing

wsse	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd
wsu	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd
xlink	http://www.w3.org/1999/xlink
soap12	http://www.w3.org/2003/05/soap-envelope
ihexdsb	urn:ihe:iti:xds-b:2007
eblcm	urn:oasis:names:tc:ebxml-regrep:xsd:lcm:3.0
ebrim	urn:oasis:names:tc:ebxml-regrep:xsd:rim:3.0
xop	http://www.w3.org/2004/08/xop/include
eb	http://www.oasis-open.org/committees/ebxml-msg/schema/msg-header-2_0.xsd
uim	http://spine.nhs.uk/spine-servicev1.0/uim
spml	urn:oasis:names:tc:SPML:2:0
spmlsearch	urn:oasis:names:tc:SPML:2:0:search
nasp	http://spine.nhs.uk/spine-servicev1.0/

npfitlc	NPFIT:HL7:Localisation

The XPath engine and its pass/fail mechanism cannot distinguish between “not there” and completely empty content[26]. So it is important to write XPath expressions for “xpathexists” and “xpathnotexists” checks to resolve content that is expected clearly to return a value, or to return no value. This applies in particular to elements which have no non-attribute children.

Given:

…

```
<hl7:id root=”foo” extension=”bar”/>
```

…

The following check:

```
CHECK xpathexists //hl7:id[@root='foo']
```

returns an empty string, and hence fail because the expression actually returns:

```
string(//hl7:id[@root='foo'])
```

The output from the string() function is empty in this case, because the <hl7:id> element has no non-attribute content. This is probably not what the check (or, worse, the condition) is intended to look for. So re-write the expression to return explicit content:

CHECK xpathexists //hl7:id[@root=’foo’]/@extension

This will return “bar” in the example above, which will pass the “exists” test as required. It will fail under any conditions where the “extension” attribute is either absent, empty, or the `<hl7:id root=”foo” … />` element is not found.

5.2.3 Notes on XSLT checks

XSLT transforms are located by fully-qualified path, and both versions 1 and 2 are supported. The transform is executed on the complete submitted message – so in order to handle payload validations, the XSLT transform author must consider the possibility that a payload is presented in SOAP Envelope and Body wrappings. This can be done either by ensuring that expressions in the transform use the descendant-or-self axis, or that the transform is called only for payload validation when the message is submitted using the VALIDATE-AS mechanism.

[26] This is because a successfully-executing XPath expression that does not match returns an empty string which is exactly the same as when the XPath expression does match, on empty content. The author would have written it differently…

Transforms may use any external resources they require. The transform author is responsible for the location of the resource, and the impact on that, of a user installing the Testbench configuration set, to locations other than a developer-specified directory. Locating an external resource through means other than files accessible from the validator's process is not supported. This is because there is no guarantee that, for any given installation, a network resource will be accessible. The results of trying such a reference are therefore undefined.

Transform authors using external resources should consider version control of those resources, and how changes to the data they contain, are managed. This is particularly the case when an external resource has a maintenance cycle different from that of the ITK specifications.

The validator judges an XSLT check's pass/fail state by interrogating its output as a text stream[27]. If the comparison string is found by a case-sensitive search in the output, the check is considered to have failed. It is therefore recommended that transform authors use obvious values such as "ERROR" in their tests. The transform output is written to the HTML validation report – so text output (as opposed to HTML or XML) from transforms is recommended.

The validator can support schematron[28] checks by compiling[29] them to XSLT. Again, schematron assertion authors must be aware that in order to support validations of payloads presented both in SOAP envelopes, or bare using the VALIDATE-AS mechanism, the assertions should use the descendant-or-self axis.

5.2.4 Note on signature check

This will report on various sorts of signature and certificate failures for the default ITK WS-Security headers under SOAP 1.1. It is strongly recommended that it is used inside a condition that checks for the presence of a signature value in order to avoid spurious failures when the signature is not present for legitimate reasons. Lack of a signature value should be checked explicitly:

```
CHECK xpathexists
    /soap:Envelope/soap:Header/wsse:Security/ds:Signature/ds:Signatu
    reValue

IF xpathexists
    /soap:Envelope/soap:Header/wsse:Security/ds:Signature/ds:Signatu
    reValue

THEN

    CHECK signature

ENDIF
```

[27] This is why the whole-string-match behaviour of the Java regular expression support is unsuitable for interrogating XSL transform outputs. There is too much risk of multiple matches requiring over-complex regular expressions.

[28] See http://www.schematron.com, and ISO/IEC 19757-3:2006 at http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html

[29] The ISO schematron compilers are available from http://www.schematron.com/

5.3 Service validation rule walk-through

Figure 8 shows a real service validation rule for the SendCDADocument-v2-0 message (Discharge). This service shows each of the techniques described above (except for “annoation” and the “signature” check).

Note that the “real” configuration file is commented – the comments and rules relating to other CDA document types have been removed here purely for space reasons.

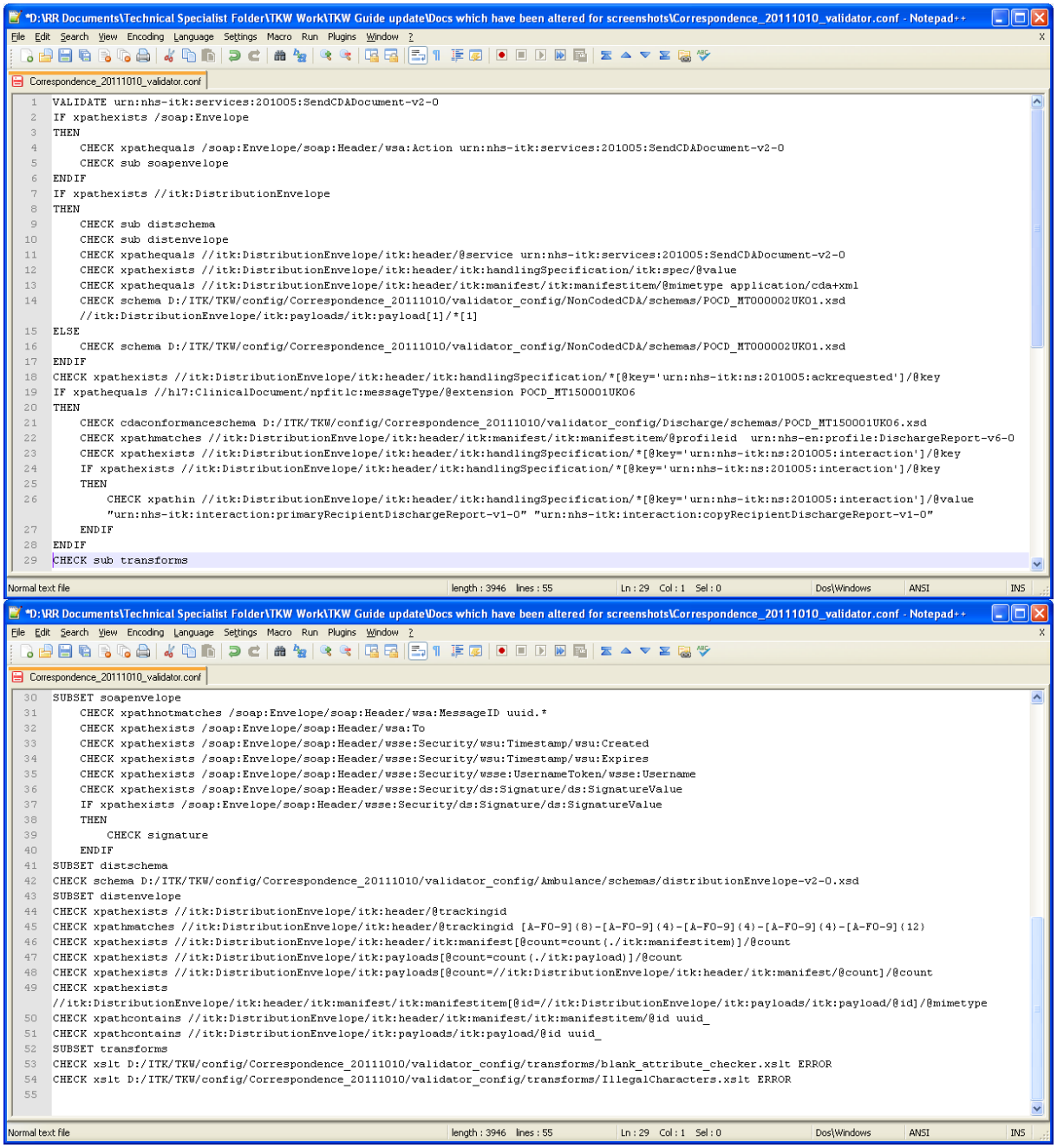


Figure 8 - Service validation rule - walk-through

Line 1: `VALIDATE urn:nhs-itk:services:201005:SendCDADocument-v2-0` This indicates the start of a new service validation rule, and says that the service this rule is for is “urn:nhs-itk:services:201005:SendCDADocument-v2-0”. Submitted messages with this either

as their WS-Addressing action, service attribute in the DistributionEnvelope or in their VALIDATE-AS line, will use this set of checks.

Line 2: `IF xpathexists /soap:Envelope`

Evaluate the XPath expression /soap:Envelope – if anything is returned, processing continues at line 4. Otherwise, it continues at line 7. The service validation rule uses this condition to include a set of checks that interrogate the SOAP header, if an envelope is present. Where a bare payload is submitted, these checks will not be performed.

Line 4: `CHECK xpathequals /soap:Envelope/soap:Header/wsa:Action urn:nhs-itk:services:201005:SendCDADocument-v2-0`

Evaluate the Xpath /soap:Envelope/soap:Header/wsa:Action against the string “urn:nhs-itk:services:201005:SendCDADocument-v2-0”. The result is output to the validation report.

Line 5: `CHECK sub soapenvelope`

Call to a SUBSET “soapenvelope” executes the code within this subset. This is for generic SOAP envelope checking, used with all ITK validations.

Line 30: `SUBSET soapenvelope`
Receives the call from line 5.

Lines 31-36:
Elements: Message ID, WS Addressing, Timestamp, Username and Signature are interrogated using “CHECK” for existence or matches. The checks are executed one at a time in the order that they are presented.

Lines 37-40:
If the condition in line 37 “passes”, this block is executed, which checks the signature as described previously. After line 40, a new SUBSET is declared so the program returns to line 6.

Lines 7-28:
The ruleset continues checking, when present, different aspects of the DistributionEnvelope, using SUBSETs as before.

Line 14: `CHECK schema D:/ITK/TKW/config/Correspondence_20111010/validator_config/NonCodedCDA/schemas/POCD_MT000002UK01.xsd //itk:DistributionEnvelope/itk:payloads/itk:payload[1]/*[1]`

Execute schema validation, using the XML schema file found at D:/ITK/TKW/config/Correspondence_20111010/validator_config/NonCodedCDA/schemas/POCD_MT000002UK01.xsd (on a Windows machine). A subsequent Xpath parameter is passed to locate the validation root. Validation failures are reported, but not fatal. So if there are any schema validation failures, or if (for example) the schema cannot be found at the given location, the validator will record the fact and carry on.

Lines 29 and 52-54:

These checks are executed last, because they are the last pair of checks in the service validation rule. Both run XSLT transforms on the complete submitted XML. The check on line 53 will pass if the text output from the transform in the file c:/temp/tk/validator/transforms/blank_attribute_checker.xslt does NOT contain the string “ERROR”, and fail if it does. Likewise the check on line 54 will pass if the output from its transform does not contain the string “ERROR”, and will fail if it does.

5.4 Validation reports

Figure 9 shows the result of running the service validation rule dissected in section 5.3, on a message sample presented as a SOAP envelope. Figure 10 shows the same service validation being applied to some sample messages, this time presented as a payload only using the VALIDATE-AS mechanism.

Note how the schema validation and the payload checks are executed in both cases, but that the SOAP header checks have been run only for the example where the SOAP header is present.

5.5 HL7v2 Validations

The Testbench validator incorporates a variety of check techniques, all of which are based on the payload being XML. The Testbench supports the HL7v2 XML namespace and as such will validate HL7v2 messages in an XML form[30].

ITK services that use HL7v2 message structures can be presented either as XML, or in the EDIFACT-style “pipe-and-hat” delimited form. The Testbench validator is unable to read the delimited form directly. However the distribution contains a directory, “`./contrib/PHconverter`” which provides a simple “pipe-and-hat” to XML converter. This is a “semantically neutral” converter that translates format only, and makes no modifications to the content.

The converter is a Java program that is run:

```
java –jar PHXMLconverter.jar messgaetype phfile xmlfile
```

where:

- *messagetype* is the HL7v2 structure
- *phfile* is the name of the file containing the delimited-form message
- *xmlfile* is the name of the file to write, containing the XML translation

For example:

```
java –jar PHXMLconverter.jar A05 A31.txt A31.xml
```

[30] See “XML Encoding Rules for HL7 v2 Messages”, ANSI/HL7 XML-2003, June 4th 2003

The output “A31.xml” file can then be validated, using the “VALIDATE-AS” mechanism to specify which service rule set should be used.

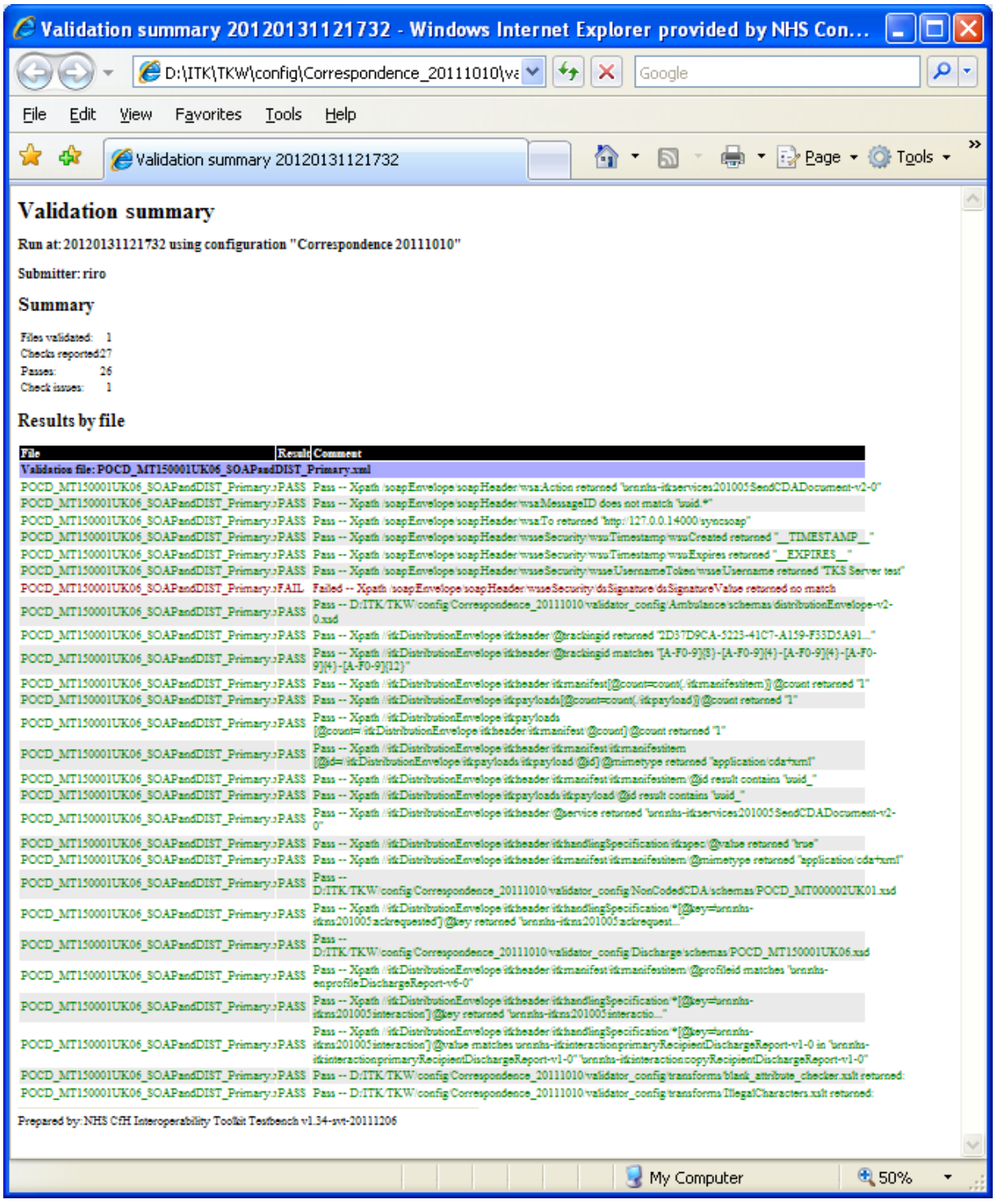


Figure 9 - The service validation rule on a SOAP-wrapped message

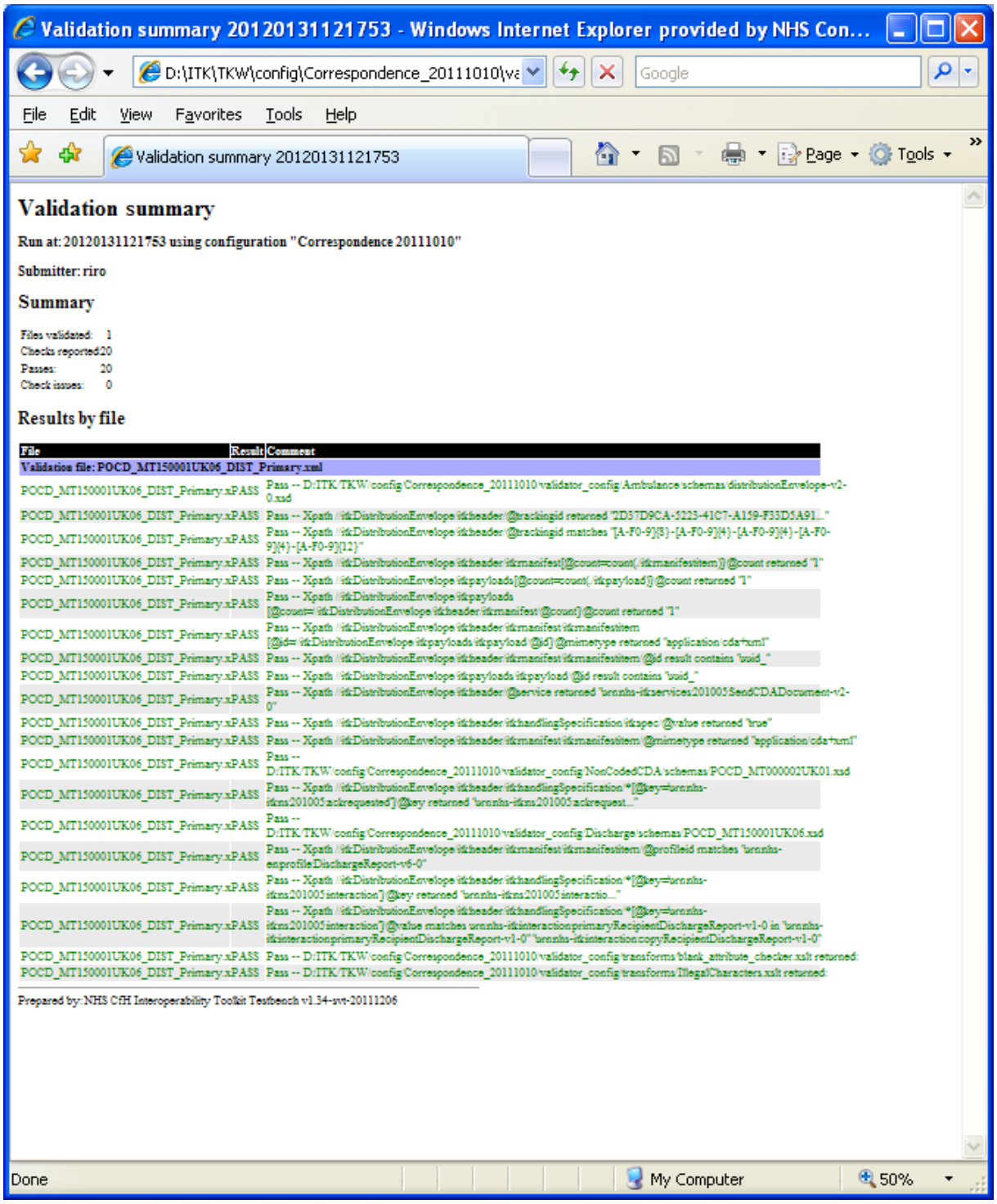


Figure 10 - The service validation on a payload. All green like this is a good sign.

6. Simulation Rules

The ITK infrastructure and service specifications describe messages sent to ITK service requests, and the responses which those requests trigger. The Testbench simulation mode allows sending applications to perform tests under controlled circumstances. This version supports the “default” ITK transport of SOAP 1.1/HTTP with- or without- TLS, and the Queue Collection interface over that transport.

The simulator is “programmed” with a simple set of rules which have the following components.

6.1 Response templates

A response template is just a file that is used to make a simulated response. In some cases, the file may be returned unchanged. But the Testbench simulator also allows a template to contain “substitution tags” which will be replaced before the response is sent back to the requestor. Response templates may be written by hand, may be generated by some automated process (for example, as part of the construction of sample content, during the development of service specifications). Or a template may be made from captured real responses from an existing system.

The simplest type of response template has no substitution tags. An example is the “simple acknowledgment”:

```
<itk:SimpleMessageResponse
    xmlns:itk="http://www.nhs.cfh.org/interoperability.toolkit/Toolk
    itUtilities/1.0">OK</itk:SimpleMessageResponse>
```

Note that the response template is a message payload. It does not contain any of the SOAP envelope or header parts found in a full message. The simulator handles that part of constructing the response. It does so because there are some things that are only known at run time – i.e. when the request message is actually handled. These include whether the response is returned synchronously or asynchronously, and whether message signing is turned on in the simulation. As these affect what the SOAP header looks like, they are left to the simulator to take care of. Making response templates as payloads only is simpler, and more flexible.

A more sophisticated template can be seen in the SOAP fault, which is based on the example given in the ITK infrastructure specification:

```
<soap:Fault
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

<soap:faultcode>Client</soap:faultcode>

<soap:faultstring>A client related error has occurred, see
    detail element for further information</soap:faultstring>

<soap:detail>

<toolkit:ToolkitErrorInfo
    xmlns:toolkit="http://www.nhs.cfh.org/interoperability.to
    olkit/ToolkitUtilities/1.0">
```

```
<toolkit:ErrorID>uuid:__ERRORID__</toolkit:ErrorID>
<toolkit:ErrorCode>__ERRORCODE__</toolkit:ErrorCode>
<toolkit:ErrorText>__ERRORTEXT__</toolkit:ErrorText>
<toolkit:ErrorDiagnosticText>__ERRORDETAIL__</toolkit:ErrorDi
    agnosticText>
</toolkit:ToolkitErrorInfo>
</soap:detail>
</soap:Fault>
```

Note how in this example (of a synchronous SOAP fault) there are some substitution tags: "__ERRORCODE__", "__ERRORTEXT__" and "__ERRORID__". The simulator replaces these with something else, before returning the response to the requestor. How the substitution tags are defined, is the topic of the next section. Any intended substitution tags in a response template, without a definition in the simulator's set of known substitutions, are treated as normal text and are not changed when the response is made.

The response templates each have a global name. When the simulator determines it is to return a particular response, it refers to it by that name.

The rules file entry for the response allows the user to specify the HTTP code returned synchronously, when a response of that type is used. It also allows the user to specify the SOAP action to be used, when a response is returned. This is applied for both synchronous and asynchronous responses. The simulator will apply the action specified for the response if it is present, otherwise it will reflect the request action[31].

A special case exists to support "passing through" the rules engine. An HTTP response code of zero, and the special file name "NONE", will cause the rules engine to pass the request on for internal processing. This is used, for example, in the Testbench' implementation of the ITK "QueueCollection" service – errors can be modelled if required, using the rules engine, but if no errors are to be modelled, the request can be passed through and any messages queued for the named subscriber, returned to them.

6.2 Substitution tags

In the SOAP fault example, "__ERRORID__" and the others are used to tell the simulator where to put content which is (at least potentially) unique to a given instance of a response. That is, a response is the result of substituting the tags in a response template.

Substitution tags are defined "globally" – that is for any given instance of the simulator, the set of substitution tags is available for use by any response. A response uses a tag simply by including it in that part of the template where the substitution is to occur. Substitution tags that are defined, but not used for a particular response template are ignored when making responses based on that template (which is how the "simple acknowledgment", with no substitution tags, works).

A tag is composed of two parts, with an optional third part that provides extra data:

[31] Responses generated by a "processor" should populate the ServiceResponse.action attribute to set this.

6.2.1 Tag names

A tag name is both the name of the tag, and the substitution point in a template. So in the example above, ”__ERRORCODE__”, “__ERRORTEXT__” and “__ERRORID__” are tag names.

When a response is made from a template, each tag in the template is evaluated once, and all instances of the tag name in that template are replaced with the same value. So for example a tag name __MESSAGE_ID__ might appear more than once in a response template. When it is used the first time, it might be evaluated as the UUID 53B6D610-2D02-11DF-9A94-2F45E926EB69 and all occurrences of __MESSAGE_ID__ in the template will be replaced with this value for that response. The next time it is called, it will be evaluated as a different UUID, and all occurrences of __MESSAGE_ID__ in the template will be replaced by that different value, for the next response.

6.2.2 Substitution types, and additional data

When a substitution tag is evaluated, there are a number of ways in which the replacement value can be calculated. Which is used for a particular tag is controlled by the substitution type. The example above already shows one type, the UUID. Others supported by the Testbench are:

Substitution type	Additional data	Description of substituted content
UUID	-	Generates an upper-case, serialised UUID
HL7datetime	-	Generates an HL7 format date time string of the form "yyyyMMddHHmmss", of the time when the substitution occurs.
ISO8601datetime	-	Generates an ISO 8601 format date time string of the form "yyyy-MM-dd'T'HH:mm:ss", of the time when the substitution occurs.
XPath (or Xpath)	XPath expression	Evaluates the XPath expression given, on the inbound message[32].
Literal	String value	The substituted content is always the given string.
Property	Property name	The substituted content is the value associated with the given System property name.
Class	Class name	The "class name", is the name of a user-supplied Java class that implements the `org.warlock.tk.internalservices.rules.SubstitutionValue` interface. The substitution content is the output of the getValue() method of an instance of that class.

Some examples:

UUIDs and Timetamps

UUIDs and timestamps are generated internally and require no additional data:

[32] XPath expressions for this substitution type must use the same standard namespace prefixes as the XPath expressions used in validation. This set of standard bindings is given in section 5.2.2 of this document.

```
__MESSAGEID__ UUID
__ERRORID__  UUID
__CREATION_DATE__ HL7datetime
```

XPath, System properties and Literals

Substitutions based on XPath expressions, system properties and literals take additional data:

```
__REFTOMESSAGEID__ Xpath /soap:Envelope/soap:Header/wsa:MessageID
```

Substitutes the value of the inbound message id (note that this will include the “uuid:” part of the string.

```
__NACS_CODE__ Property nacs.code
```

Substitutes the value of the Java system property with the name “nacs.code”.

```
__DOCTOR_SMITH__ Literal Dr. A Smith
```

Substitutes the value “Dr. A Smith”.

6.3 Expressions

When the simulator receives a message, it needs to interrogate that message to determine how to respond to the requestor. The interrogation process is controlled by the rule sets. The rule sets work by applying expressions in a structured way until a decision is reached.

Expressions are very similar to the validation checks that are described in section 5 of this document. They apply an assertion to the inbound message, which returns true or false.

Expressions are named, and are global. That is, like the substitutions, one set of expressions are defined, and an expression can be used by any rule that requires it. This allows, for example a single expression to be written that returns true for a particular NHS number (or pseudo NHS number) in an HL7v3 payload:

```
examplerule XpathEquals
    //hl7:id[@root='2.16.840.1.113883.2.1.4.1']/@extension
    9999999999 XpathEquals
      //hl7:id[@root='2.16.840.1.113883.2.1.4.1']/@extension
      9999999999
```

This returns true if a message sent to the simulator contains an HL7v3 id, with the pseudo NHS number “9999999999”. It does not distinguish between payload types, or service requests, so it can be used in handling a wide range of cases.

6.3.1 Expression types

Expressions can be of various types – the example above shows the “XPathEquals” type. Supported expression types are:

Type	Parameters	Description
XpathEquals	XPath expression Comparison string	Executes the given XPath expression on the inbound message, and returns the result of a case-sensitive match with the comparison string.
XpathNotEquals	XPath expression Comparsion string	Executes the given XPath expression on the inbound message, and returns the logical negation of the result of a case-sensitive match with the comparison string.
Xslt	Fully-qualified file name of an XSLT transform Comparison string	Executes the transform at the given file name, on the inbound message. Returns true if the text output of the transform does NOT contain the comparison string[33].
Contains	A comparison string	Returns true if the inbound message, processed as a string, contains the comparison string.
NotContains	A comparison string	Returns true if the inbound message, processed as a string, does NOT contain the comparison string.
Always		Always returns true.

[33] The comparison is done this way round, partly for consistency with the XSLT-based validation checks, but also (like the validation checks) to make looking for strings such as “ERROR” easier.

Never		Always returns false.

Expressions using XPath obey the same rules, for the same reasons, as XPath expressions used for substitutions and for validations. That is, they MUST use the prefixes documented under the namespace bindings in section 0 of this document.

XSLT-based expressions can use XSLT-1 or XSLT-2.

6.4 Rules

A rule is an expression, followed by a description of the action that the simulator should take depending on whether the expression returned true or false. Rules are grouped into rule sets, with each rule set bound to a service.

So, when the simulator receives a message, it looks up the rule set[34] for that message, and starts processing it. The rule set is a sequence of rules, which are executed in the order they are given in the rules configuration file.

A rule has three parts:

- Expression. This is the name of the expression to evaluate on the inbound message.

- True response. The name of the template to use for a response, where the expression evaluates to true. Alternatively, the reserved response name “next” explicitly instructs the rules engine to carry on to the next rule in the set.

- False response. The name of the template to use for a response, where the expression evaluates to false. This can be the reserved response name “next”, to explicitly instruct the rules engine to carry on to the next rule in the set.

Rules processing continues until a response template is identified. The simulator will return an error if processing “runs off the end” of the rule set without identifying a response.

The simulator will also return an error where it is sent a message, for a service for which it has no configuration.

6.5 Simulator rules syntax

A simulator rules file contains declarations for the response templates, substitutions, expressions and rules. It can contain as many of each as are required. All references to expressions and response templates[35] must be satisfied, or an error will be reported when

[34] See section 0

[35] As stated in the description of the response template, in section 0, the substitution tags drive the substitution process that turns a template into a response. So the simulator cannot tell that a piece of text inside a template

the simulator boots. It is not an error for a substitution, expression or response template to be defined, but not used.

The rules file is a text file. Comments are supported on complete lines by making the first character a hash ("#") symbol. Empty lines are ignored. Simulator elements MUST be defined in the order:

- Response templates
- Substitutions
- Expressions
- Rules

The rules come last because they reference both the expressions and the response templates.

6.5.1 Responses

Response templates are defined between the lines:

```
BEGIN RESPONSES
```

and

```
END RESPONSES
```

Each response template is defined on a single line. It is a whitespace-delimited line:

```
BEGIN RESPONSES
ack_response c:/temp/tk/tk_test_response.xml 200 response
error_response c:/temp/tk/tk_test_error.xml 500
do_process NONE 0
END RESPONSES
```

This defines three response templates:

- "`ack_response`", using the template found in the file "`c:/temp/tk/tk_test_response.xml`", returns an HTTP response code of 200 synchronously. When a response is returned asynchronously, it uses the SOAP action "`response`".
- "`error_response`", using the template found in the file "`c:/temp/tk/tk_test_error.xml`", returns an HTTP response code of 500. This has no asynchronous SOAP action, because the HTTP response code 500 stops the simulator sending an asynchronous response, for requests received over an asynchronous channel.
- "`do_process`" is a special case. It has no template file. The zero in the HTTP response code field instructs the simulator that, when this response template is indicated by a rule,

---

file is intended to be a substitution, if there is no definition for that substitution in the simulator rules configuration.

the request should be handed off to the Testbench' internal processor. Configuring the internal processor is discussed in the next section of this document.

The fields are:

Field	Optional	Description
Response name	No	The name by which rules that need it, will reference this response.
Template file	No (but can be NONE)	The fully-qualified name of a file containing a template from which to built responses of this type.
HTTP response code	No	The HTTP response code to return to a requestor, when this response is indicated. If the HTTP response is zero, the simulator will pass the request on for internal processing. If the HTTP response is 500, no asynchronous response will be sent. For a "202" response, the synchronous response is HTTP-only.
Asynchronous SOAP action	Yes	Where a request is received over an asynchronous channel, and the response is to be returned asynchronously, this is the SOAP action under which the response is sent.

6.5.2 Substitutions

Substitutions are defined between the lines:

```
BEGIN SUBSTITUTIONS
```

and

```
END SUBSTITUTIONS
```

Each substitution is defined on a single, whitespace-delimited line.

```
BEGIN SUBSTITUTIONS
__MESSAGEID__ UUID
__REFTOMSGID__ Xpath
    /soap:Envelope/soap:Header/wsa:MessageID
__FAULT_TO__ Literal http://localhost/faultreport
END SUBSTITUTIONS
```

This defines three substitutions:

All instances of "`__MESSAGEID__`" in a response template, will be replaced by a UUID.
All instances of "`__REFTOMSGID__`" in a response template, will be replaced by the result of evaluating the XPath expression "`/soap:Envelope/soap:Header/wsa:MessageID`" on the inbound request.
All instances of "`__FAULT_TO__`" in a response template will be replaced by the fixed string "`http://localhost/faultreport`".

The fields are:

Field	Optional	Description
Tag name	No	Tag string as used in the response templates.
Substitution type	No	Substitution type as described in section 6.2.2 of this document.
Additional data	Yes	Supporting data, dependent on substitution type. Ignored for those substitutions that do not use it.

Response templates describe response payloads only. So whilst substitutions can be provided that are based (as seen in the example above) on data in the request's SOAP header, these will only be used for substitutions inside the payloads themselves. The mechanism the simulator uses for constructing and addressing response wrappers does use the same substation mechanism, but it does so on templates specified in a different part of the configuration. These details are discussed in section 0, on response wrapping and routing.

6.5.3 Expressions

Expressions are defined between the lines:

```
BEGIN EXPRESSIONS
```

and

```
END EXPRESSIONS
```

Each expression is defined on a single, whitespace-delimited line:

```
BEGIN EXPRESSIONS

testrule XpathEquals
    //hl7:id[@root='2.16.840.1.113883.2.1.4.1']/@extension
    9999999999

passthrough Always

END EXPRESSIONS
```

This defines two expressions:

> "`testrule`" returns true if the XPath expression "`//hl7:id[@root='2.16.840.1.113883.2.1.4.1']/@extension`" evaluated against the inbound request, equals "`9999999999`".
> "`passthrough`" always returns true.

The fields are:

Field	Optional	Description
Expression name	No	The name by which this expression will be referenced in the rule sets.
Expression type	No	The type of expression, as described in section 0 of this document.
Additional data	Yes	Additional information dependent on the type of expression.

6.5.4 Rules

Rule sets are defined independently for each service. In each case the rules are defined between the lines:

`BEGIN RULE`

servicename

and

`END RULE`

The "`servicename`" declares the service name to which this rule set applies. Rules are written on single, whitespace-delimited lines. They identify an expression, and the actions to take on a "true" or "false" result of evaluating the expression. The padding words "if", "then" and "else" are used to make the rule more readable.

Rules for multiple services can be defined in the same file:

```
BEGIN RULE
urn:nhs-itk:services:201005:SendCDADocument-v2-0
if testrule then error_response else ack_response
END RULE
BEGIN RULE
GetMessages
if passthrough then do_process
END RULE
```

This defines rules for two services," `urn:nhs-itk:services:201005:SendCDADocument-v2-0`" and "`GetMessages`". In each case there is a single rule, which has the structure:

if expression_name then true_response else false_response

Where:

- "`expression_name`" is the name of one of the expressions defined in the rules file. It is an error to reference an expression name that has not been defined.
- "`true_response`" is the name of a response template to use if the expression evaluates to true. Alternatively the key word "`next`" can be used to instruct the rules engine to proceed to the next rule for this service, if there is one.
- "`false_response`" is the name of a response template to use if the expression evaluates to false. Alternatively the key word "`next`" can be used to instruct the rules engine to proceed to the next rule for this service, if there is one.

Looking at the rule for "GetMessages", the simulator implements this ITK service request in this way so that rules, and simulated responses, can be applied before the actual queue collection operation is executed. Let us look at how to do that. Assume that we have defined an additional expression, called "qc_test": one which will return an error when a collection is requested for the subscriber name "fatal_subscriber":

```
qc_test XpathEquals //quc:SubscriberName fatal_subscriber
```

When the simulator receives a queue collection request for this subscriber, we want it to return the SOAP fault response we have already defined, called "error_response" in the example above. If the queue collection request is made for any other subscriber, we want the simulator to process it in the same way as it does now. So, add an extra rule that says when the "qc_test" expression is true, return "error_response", otherwise go on to the existing rule[36]:

```
BEGIN RULE
GetMessages
if qc_test then error_response else next
if passthrough then do_process
END RULE
```

6.6 Response addressing and wrappers

Response templates are payloads only. When the simulator constructs a response from one of the templates, it still needs to "wrap" the resulting message payload in a form that is protocol-conformant before it is returned to the requestor.

The "wrappers" used for synchronous and asynchronous responses are referenced from the Testbench properties file with the "tks.synchronousreply.wrapper" and "tks.asynchronousreply.wrapper" properties. These give the fully-qualified file names of template files for these wrappers. The wrapper templates are processed in the same way as response templates, by substituting tags in the templates. Unlike a response template, the simulator itself provides the substitutions.

The property "tks.soap.ack.template" contains the fully-qualified name of a template file used for synchronous acknowledgments to asynchronous requests.

[36] The observant will have noticed that the single rule: if qc_test then error_response else do_process would achieve the same thing, but it would not have illustrated chaining rules in the same way.

The Testbench implements WS-Addressing and ITK fault addressing for asynchronous responses. If a request has a FaultTo address, and the response SOAP action is "`http://schemas.xmlsoap.org/ws/2004/08/addressing/fault`", then the response will be sent to the given FaultTo address. Otherwise (and in all cases where the response SOAP action does not detail a fault), the response will be sent to the ReplyTo address. Responses where the rules indicate that the request gets an HTTP 500 result, do not generate any asynchronous response, as a fault is returned synchronously to the requestor.

6.6.1 Synchronous reply wrapper template

The stock synchronous reply template is:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:wsa="http://www.w3.org/2005/08/addressing"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
    wssecurity-utility-1.0.xsd" xmlns:wsse="http://docs.oasis-
    open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
<soap:Header>
<wsa:MessageID>uuid:__MESSAGEID__</wsa:MessageID>
<wsa:Action>__ACTION__</wsa:Action>
</soap:Header>
<soap:Body>
__PAYLOAD_BODY__
</soap:Body>
</soap:Envelope>
```

The definitions of the substitutions are:

Tag	Substituted by
__MESSAGE_ID__	New UUID
__ACTION__	Response action of incoming message action
__PAYLOAD_BODY__	Response payload from substituted template

At present, the stock synchronous response wrapper does not carry a wsu:Timestamp element, however the simulator will also carry out the following substitution if the tag is present in any alternative wrapper template:

Tag	Substituted by
__TIMESTAMP__	Current date/time in ISO 8601 format.
__EXPIRES__	Current date/time PLUS value of "`tks.Toolkit.default.asyncttl`" property (in seconds), in ISO 8601 format.

These are ignored in the stock synchronous wrapper, as the substitution tags are not present.

6.6.2 Asynchronous reply wrapper template

The stock asynchronous reply wrapper is:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:wsa="http://www.w3.org/2005/08/addressing"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
    wssecurity-utility-1.0.xsd" xmlns:wsse="http://docs.oasis-
    open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
<soap:Header>
<wsa:MessageID>uuid:__MESSAGEID__</wsa:MessageID>
```

```
<wsa:Action>__ACTION__</wsa:Action>
<wsa:To>__TO_ADDRESS__</wsa:To>
<wsse:Security>
<wsu:Timestamp>
<wsu:Created>__TIMESTAMP__</wsu:Created>
<wsu:Expires>__EXPIRES__</wsu:Expires>
</wsu:Timestamp>
<wsse:UsernameToken>
<wsse:Username>TKS</wsse:Username>
</wsse:UsernameToken>
</wsse:Security>
<wsa:RelatesTo>__ORIGINAL_MESSAGEID__</wsa:RelatesTo>
</soap:Header>
<soap:Body>
__PAYLOAD_BODY__
</soap:Body>
</soap:Envelope>
```

The substitutions in the asynchronous case are the same as for the synchronous reply wrapper, with the inclusion of the "`__TIMESTAMP__`" and "`__EXPIRES__`" tags, but with one important change.

In a *synchronous reply*, the response is returned along the same connection as that on which the request was received. The request comes "from" a sender, and because the messaging pattern is synchronous, any reply must go back to the sender.

In an asynchronous reply, the ITK use of WS-Addressing provides two possible return addresses: "FaultTo" and "ReplyTo". In the ITK interface, "From" is ignored, and "ReplyTo" MUST be present in a request for which an asynchronous response is expected. The rules governing the precedence and use of these addresses under normal reply, and fault scenarios are shown in figure 11. An asynchronous fault is signalled with the SOAP action "`http://schemas.xmlsoap.org/ws/2004/08/addressing/fault`".

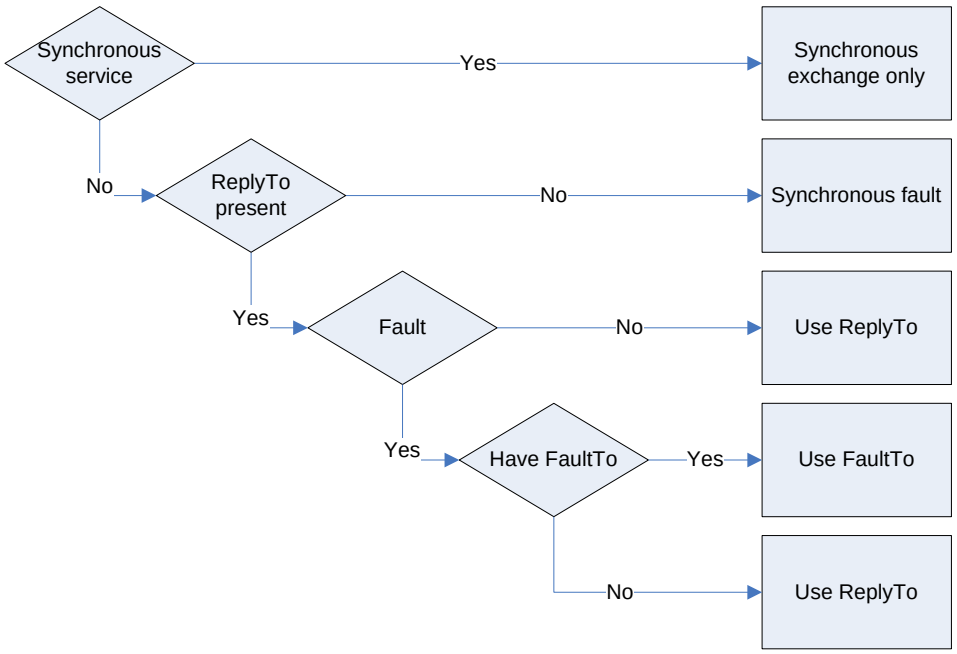


Figure 11 Asynchronous addressing

These rules are implemented internally by the simulator, and provide the substitution value for the "__TO_ADDRESS__" tag in the stock asynchronous wrapper template. The value determined for "__TO_ADDRESS__" is used by the Testbench' sending mechanism, to transmit the asynchronous reply.

6.7 Simulating HL7v2 Delimited-form Services

The Testbench simulator incorporates the logic of the "PHXMLconverter" described in section 5.5. As such, it can be used with delimited-form HL7v2 payloads – it converts the delimited form first to XML, applies the rules and, if needed, converts the response to a delimited form before returning it. The same rule sets are used for both XML and delimited HL7v2 requests.

This does depend on the specialised "PH" service handlers being used, and identified by URL context paths. Interested users should inspect the existing Testbench configurations for the ITK ADT service sets, to see how it is set up.

7. TKW Testbench URLs and Processing

A request sent to the Testbench is distinguished by its action, and by the context path of the URL to which it is sent. The context path is the part of the URL that comes after the hostname, and optional port:

```
https://itk.example.nhs.net/this/is/the/context/path
```

Inspection of the HTTP header that results from a request to this URL, will show (in part):

POST /this/is/the/context/path HTTP/1.1

Host: itk.example.nhs.net

Normally, the context path is specified by the WSDL file that defines the service. However the point in the WSDL which defines this (`//soap:address/@location`) also specifies the hostname, which is inapplicable to a service offered by many sites.

The Testbench simulator is not constrained by the WSDL-provided context path, although it can use it. What the simulator needs is a mapping between a context path, and the component within the Testbench that will handle requests made using that path. Because the simulator gets the request' SOAP action, it can use the same handler for many types of service. The stock properties file does just this:

```
# Port names for the request handlers (DO NOT REMOVE the BADURI)
#
tks.Toolkit.namelist BADURI syncsoap asyncsoap qrequest qreqitk
      dischargesummary
#
# And then the definitions needed by each of these (DO NOT REMOVE the
      BADURI)
tks.Toolkit.BADURI.class    org.warlock.tk.handlers.BadUriHandler
tks.Toolkit.BADURI.path     /
tks.Toolkit.dischargesummary.class
         org.warlock.tk.handlers.SynchronousSoapRequestHandler
tks.Toolkit.dischargesummary.path
      /interoperability.toolkit/ToolkitUtilities/SendDischargeSummary
tks.Toolkit.syncsoap.class
         org.warlock.tk.handlers.SynchronousSoapRequestHandler
tks.Toolkit.syncsoap.path   /syncsoap
tks.Toolkit.asyncsoap.class
         org.warlock.tk.handlers.AsynchronousSoapRequestHandler
tks.Toolkit.asyncsoap.path /asyncsoap
tks.Toolkit.qrequest.class
         org.warlock.tk.handlers.SynchronousSoapRequestHandler
tks.Toolkit.qrequest.path    /ToolkitUtilities/GetMessages
```

```
tks.Toolkit.qreqitk.class
        org.warlock.tk.handlers.SynchronousSoapRequestHandler
tks.Toolkit.qreqitk.path        /interoperability.toolkit/ToolkitUtilities/GetMessages-
    v1-0
```

Note that each property is defined on a single line. In this example some of the lines are wrapped – and this is indicated by "\" at the wrap point, and the continuation being indented two spaces.

Users should ignore the "BADURI" (which is needed by the simulator's HTTP server), and the "Diagnostic" entries (which is not yet implemented).

The "`tks.Toolkit.namelist`" property lists the contexts that the Testbench simulator will support. For each entry in that list (in this case "`BADURI diagnostic syncsoap asyncsoap qrequest`") two properties are defined:

- "`tks.Toolkit.{name}.class`" is the name of the Java class that will handle requests on that context path
- "`tks.Toolkit.{name}.path`" is the actual context path for that name

The Toolkit simulator has two[37] classes that handle requests using the rules engine discussed in section 6:

"`org.warlock.tk.handlers.SynchronousSoapRequestHandler`" handles synchronous requests, and "`org.warlock.tk.handlers.AsynchronousSoapRequestHandler`" handles asynchronous requests (and is actually a subclass of the synchronous handler, since their behaviour when receiving a message is the same). The same handler class can be used for multiple contexts: for example the use of the synchronous rule processor "`org.warlock.tk.handlers.SynchronousSoapRequestHandler`" for both the "`/syncsoap`" generic and the "`/ToolkitUtilities/GetMessages`" queue collection contexts. Note also how the post-pilot service name for the same "GetMessages" interface,is added simply by including:

```
tks.Toolkit.qreqitk.class
org.warlock.tk.handlers.SynchronousSoapRequestHandler
```

[37] These are supplemented by a further two classes, that provide the same functionality but support HL7v2 delimited-form requests by converting between delimited and XML forms either side of implementing the rules- and process functions.

tks.Toolkit.qreqitk.path
/interoperability.toolkit/ToolkitUtilities/GetMessages-v1-0

and "qreqitk" to the list in the "tks.Toolkit.namelist" property.

The Testbench can be used to implement other functionality by providing handler classes, and plugging them in by this mechanism, for example:

```
tks.Toolkit.namelist BADURI myhandler
#
# And then the definitions needed by each of these (DO NOT
    REMOVE the BADURI)
tks.Toolkit.BADURI.class
    org.warlock.tk.handlers.BadUriHandler
tks.Toolkit.BADURI.path /
tks.Toolkit.myhandler.class com.example.handlers.MyHandler
tks.Toolkit.myhandler.path /my/context/path
```

However, implementing that sort of functionality is outside the scope of the current document.

The reason for using the "/syncsoap" and "/asyncsoap" generic context paths is that it makes adding support for new services very easy. The properties file does not need to be changed, and support only needs to be added to the rules engine and validator configurations, which is where the majority of the effort would be focussed in any case. But it remains true that, strictly, to host the "SendDischargeSummary" on the "/syncsoap" context path is to break conformance with the WSDL. This can be fixed by adding a suitable name to the "tks.Toolkit.namelist", and by providing class and path names for it. For example:

```
tks.Toolkit.namelist BADURI qrequest sdc
tks.Toolkit.BADURI.class org.warlock.tk.handlers.BadUriHandler
tks.Toolkit.BADURI.path /
tks.Toolkit.sdc.class \
  org.warlock.tk.handlers.SynchronousSoapRequestHandler
tks.Toolkit.sdc.path  \
     /interoperability.toolkit/ToolkitUtilities/SendDischargeSummary
```

7.1 How is the rule set “do_process” done?

Looking back at the discussion of the rules configuration, we saw that the ITK queue collection interface was implemented by passing the requests through the rules engine first, and then to a handler that actually did the message retrieval for the requesting subscriber. We have not yet covered how that works.

The “response template” for the “`GetMessages`” service is:

```
do_process   NONE 0
```

It has no template file, and an HTTP response code of zero. The zero response code tells the simulator rules engine (actually the “`org.warlock.tk.handlers.SynchronousSoapRequestHandler`” class) not to return a response there and then, but to hand the request to another class, for processing.

Which other class? In the properties file, the property “`tks.processors.configurationfile`” gives the fully-qualified file name of a configuration file that maps SOAP actions to handling classes:

```
GetMessages   \
org.warlock.tk.internalservices.queue.QueueRequestHandler
GetMessagesWithConf  \
org.warlock.tk.internalservices.queue.QueueRequestHandler
```

Again, this is a whitespace-delimited file, and wrap points are indicated by “\”. Each line configures one SOAP action, mapping it on to the class which will handle it. Implementation of such classes is outside the scope of the current document.

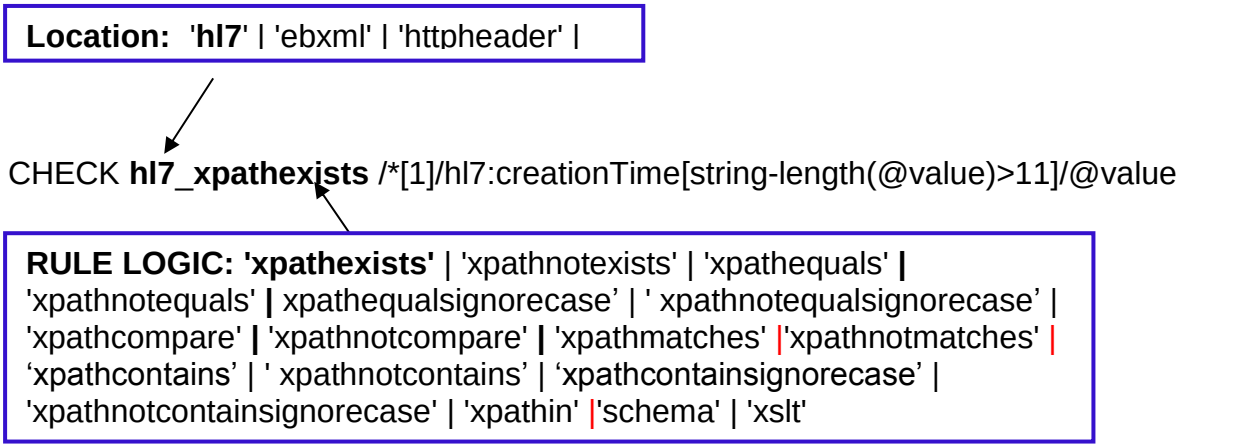
8. Spine message validation

The usage and outputs of the Spine message validation function is mostly very similar to that described for ITK services. Differences lie in the way messages are processed, and how service types are identified. As “VALIDATE-AS” still works, these differences are internal.

8.1 Validation syntax used in –spinevalidate

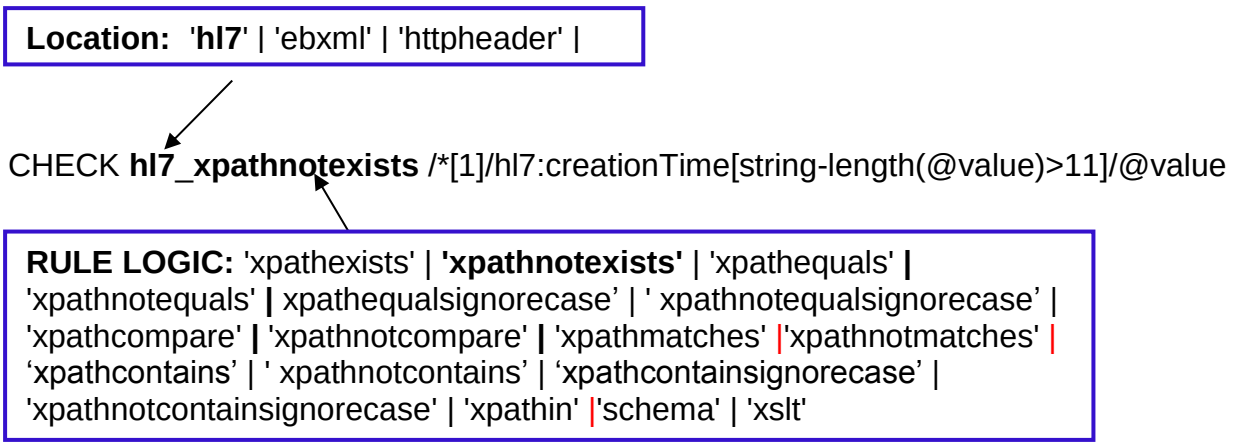
8.1.1 Rule Type – xpathexists

Check used to test if a value is present.



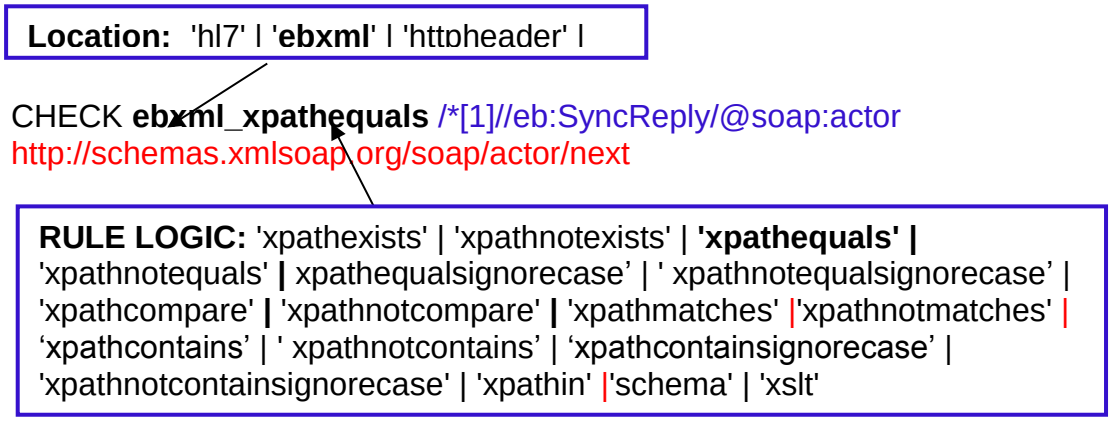
8.1.2 Rule Type – xpathnotexists

Check used to test if a value is not present.



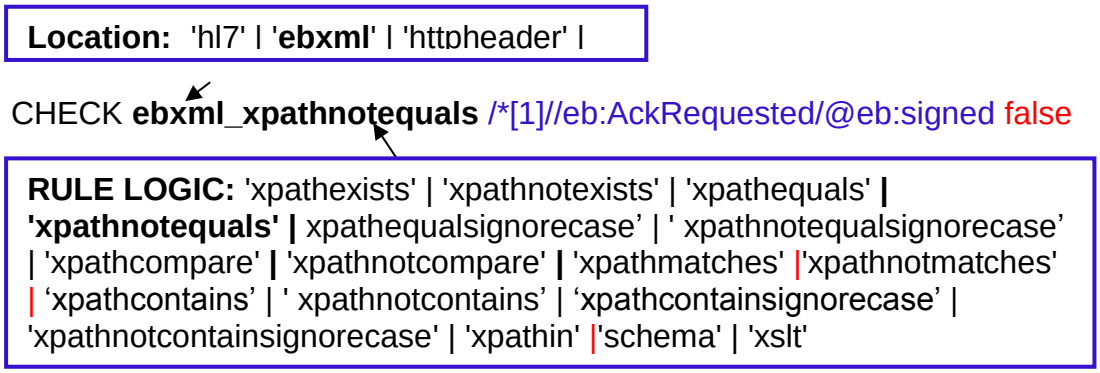
8.1.3 Rule Type – xpathequals

Check used to test if the value the xpath expression evaluates to equals the given fixed value, this check is case sensitive.



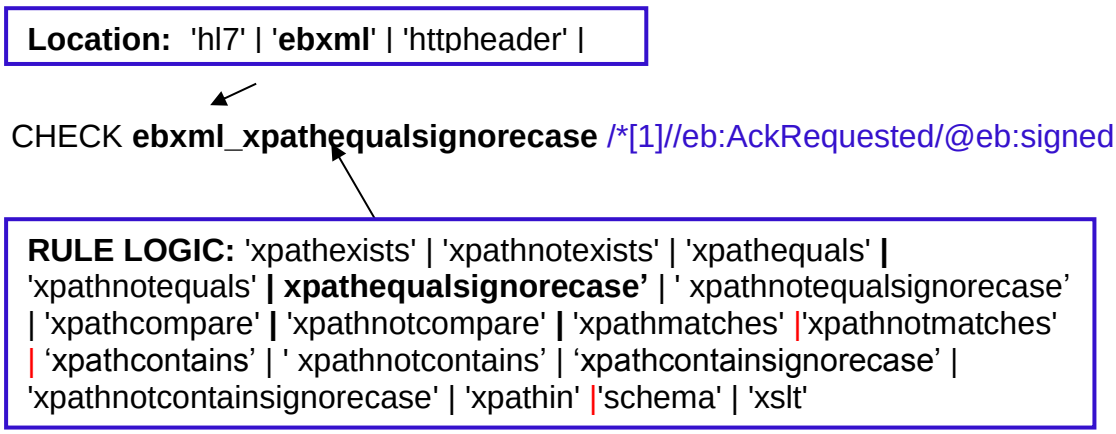
8.1.4 Rule Type – xpathnotequals

Check used to test if the value the xpath expression evaluates to does not equal the given fixed value, this check is case sensitive.



8.1.5 Rule Type – xpathequalsignorecase

Check used to test if the value the xpath expression evaluates to equals the given fixed value, ignoring the case of the value. I.e In the below example if the value found is 'FALSE' or 'false' then it will return true.



8.1.6 Rule Type – xpathnotequalsignorecase

Check used to test if the value the xpath expression evaluates to, does not equal the given fixed value, ignoring the case of the value. I.e In the below example if the value found is 'FALSE' or 'false' then it will return false.

Location: 'hl7' | '**ebxml**' | 'httpheader' |

CHECK **ebxml_xpathnotequalsignorecase** /*[1]//eb:AckRequested/@eb:signed false

RULE LOGIC: 'xpathexists' | 'xpathnotexists' | 'xpathequals' **|** 'xpathnotequals' **|** xpathequalsignorecase' | ' **xpathnotequalsignorecase'** | 'xpathcompare' **|** 'xpathnotcompare' **|** 'xpathmatches' |'xpathnotmatches' | 'xpathcontains' | ' xpathnotcontains' | 'xpathcontainsignorecase' | 'xpathnotcontainsignorecase' | 'xpathin'

8.1.7 Rule Type – xpathcompare

Check used to test if the value the xpath expression evaluates to is equal to the second xpath expression evaluation value, this check is case sensitive.

Location: **'hl7'** | 'ebxml' | 'httpheader' |

CHECK **hl7_xpathcompare**
/*[1]/hl7:ControlActEvent/hl7:subject/hl7:PatientPrescriptionReleaseRequest/hl7:author/hl7:AgentPerson/hl7:telecom/@use
/*[1]/ControlActEvent/subject/PatientPrescriptionReleaseRequest/author/AgentPerson/representedOrganization/telecom/@use

RULE LOGIC: 'xpathexists' | 'xpathnotexists' | 'xpathequals' **|** 'xpathnotequals' **|** xpathequalsignorecase' | ' xpathnotequalsignorecase' | **'xpathcompare' |** 'xpathnotcompare' **|** 'xpathmatches' |'xpathnotmatches' | 'xpathcontains' | ' xpathnotcontains' | 'xpathcontainsignorecase' | 'xpathnotcontainsignorecase' | 'xpathin' |'schema' | 'xslt'

8.1.8 Rule Test Type – xpathnotcompare

Check used to test if the value the xpath expression evaluates to is equal to the second xpath expression evaluation value, this check is case sensitive.

Location: **'hl7'** | 'ebxml' | 'httpheader' |

CHECK **hl7_xpathnotcompare**
/*[1]/hl7:ControlActEvent/hl7:subject/hl7:PatientPrescriptionReleaseRequest/hl7:author/hl7:AgentPerson/hl7:telecom/@value
/*[1]/ControlActEvent/subject/PatientPrescriptionReleaseRequest/author/AgentPerson/representedOrganization/telecom/@value

RULE LOGIC: 'xpathexists' | 'xpathnotexists' | 'xpathequals' **|** 'xpathnotequals' **|** xpathequalsignorecase' | ' xpathnotequalsignorecase' | 'xpathcompare' **|** **'xpathnotcompare' |** 'xpathmatches' |'xpathnotmatches' | 'xpathcontains' | ' xpathnotcontains' | 'xpathcontainsignorecase' | 'xpathnotcontainsignorecase' | 'xpathin' |'schema' | 'xslt'

8.1.9 Rule Test Type – xpathmatches

Check used to test if the value the xpath expression evaluates to matches the regular expression value.

Location: '**hl7**' | 'ebxml' | 'httpheader' |

CHECK **hl7_xpathmatches**
/*[1]/hl7:ControlActEvent/hl7:subject//hl7:subject/hl7:patientRole/hl7:patientPerson/hl7:multipleBirthOrderNumber[@updateMode='added' or @updateMode='altered']/@value ^[1-9]$

RULE LOGIC: 'xpathexists' | 'xpathnotexists' | 'xpathequals' **|** 'xpathnotequals' **|** xpathequalsignorecase' | ' xpathnotequalsignorecase' | 'xpathcompare' **|** 'xpathnotcompare' **| 'xpathmatches'** |'xpathnotmatches' | 'xpathcontains' | ' xpathnotcontains' | 'xpathcontainsignorecase' | 'xpathnotcontainsignorecase' | 'xpathin' |'schema' | 'xslt'

8.1.10 Rule Test Type – xpathnotmatches

Check used to test if the value the xpath expression evaluates to does not match the regular expression value.

Location: '**hl7**' | 'ebxml' | 'httpheader' |

CHECK **hl7_xpathmatches**
/*[1]/hl7:ControlActEvent/hl7:subject//hl7:subject/hl7:patientRole/hl7:patientPerson/hl7:multipleBirthOrderNumber[@updateMode='added' or @updateMode='altered']/@value ^[1-9]$

RULE LOGIC: 'xpathexists' | 'xpathnotexists' | 'xpathequals' **|** 'xpathnotequals' **|** xpathequalsignorecase' | ' xpathnotequalsignorecase' | 'xpathcompare' **|** 'xpathnotcompare' **|** 'xpathmatches' |**'xpathnotmatches'** | 'xpathcontains' | ' xpathnotcontains' | 'xpathcontainsignorecase' | 'xpathnotcontainsignorecase' | 'xpathin' |'schema' | 'xslt'

8.1.11 Rule Test Type – xpathcontains

Returns a pass if the value returned by the XPath expression contains the comparison string in a case-sensitive search.

Location: '**hl7**' | 'ebxml' | 'httpheader' |

CHECK **hl7_xpathcontains**
/*[1]/hl7:ControlActEvent/hl7:subject//hl7:subject/hl7:patientRole/hl7:patientPerson/hl7:multipleBirthOrderNumber[@updateMode='added' or @updateMode='altered']/@value "1"

RULE LOGIC: 'xpathexists' | 'xpathnotexists' | 'xpathequals' **|** 'xpathnotequals' **|** xpathequalsignorecase' | ' xpathnotequalsignorecase' | 'xpathcompare' **|** 'xpathnotcompare' **|** 'xpathmatches' |'xpathnotmatches' | '**xpathcontains'** | ' xpathnotcontains' | 'xpathcontainsignorecase' | 'xpathnotcontainsignorecase' | 'xpathin' |'schema' | 'xslt'

8.1.12 Rule Test Type – xpathnotcontains

Returns a pass if the value returned by the XPath expression does not contain the comparison string in a case-sensitive search.

Location: '**hl7**' | 'ebxml' | 'httpheader' |

CHECK **hl7_xpathnotcontains**
/*[1]/hl7:ControlActEvent/hl7:subject//hl7:subject/hl7:patientRole/hl7:patientPerson/hl7:multipleBirthOrderNumber[@updateMode='added' or @updateMode='altered']/@value "1"

RULE LOGIC: 'xpathexists' | 'xpathnotexists' | 'xpathequals' **|** 'xpathnotequals' **|** xpathequalsignorecase' | ' xpathnotequalsignorecase' | 'xpathcompare' **|** 'xpathnotcompare' **|** 'xpathmatches' |'xpathnotmatches' | 'xpathcontains' | ' **xpathnotcontains'** | 'xpathcontainsignorecase' | 'xpathnotcontainsignorecase' | 'xpathin' |'schema' | 'xslt'

8.1.13 Rule Test Type – xpathcontainsignorecase

Returns a pass if the value returned by the XPath expression contains the comparison string in a case-insensitive search.

Location: '**hl7**' | 'ebxml' | 'httpheader' |

CHECK **hl7_xpathcontainsignorecase**
/*[1]/hl7:ControlActEvent/hl7:subject//hl7:subject/hl7:patientRole/hl7:patientPerson/hl7:multipleBirthOrderNumber[@updateMode='added' or @updateMode='altered']/@value "1"

RULE LOGIC: 'xpathexists' | 'xpathnotexists' | 'xpathequals' **|** 'xpathnotequals' **|** xpathequalsignorecase' | ' xpathnotequalsignorecase' | 'xpathcompare' **|** 'xpathnotcompare' **|** 'xpathmatches' |'xpathnotmatches' | 'xpathcontains' | ' xpathnotcontains' | '**xpathcontainsignorecase'** | 'xpathnotcontainsignorecase' | 'xpathin' |'schema' | 'xslt'

8.1.14 Rule Test Type – xpathnotcontainsignorecase

Returns a pass if the value returned by the XPath expression does not contain the comparison string in a case-insensitive search.

Location: '**hl7**' | 'ebxml' | 'httpheader' |

CHECK **hl7_xpathnotcontainsignorecase**
/*[1]/hl7:ControlActEvent/hl7:subject//hl7:subject/hl7:patientRole/hl7:patientPerson/hl7:multipleBirthOrderNumber[@updateMode='added' or @updateMode='altered']/@value “1”

RULE LOGIC: 'xpathexists' | 'xpathnotexists' | 'xpathequals' **|** 'xpathnotequals' **|** xpathequalsignorecase’ | ' xpathnotequalsignorecase’ | 'xpathcompare' **|** 'xpathnotcompare' **|** 'xpathmatches' |'xpathnotmatches' | ‘xpathcontains’ | ' xpathnotcontains’ | ‘xpathcontainsignorecase’ | **'xpathnotcontainsignorecase'** | 'xpathin' |'schema' | 'xslt'

8.1.15 Rule Test Type – xpathin

Returns a pass if the value returned by the XPath expression does not contain one of the list of comparison strings in the search. These are separated by space and encapsulated in quotes

Location: '**hl7**' | 'ebxml' | 'httpheader' |

CHECK **hl7_xpathin**
/*[1]/hl7:ControlActEvent/hl7:subject//hl7:subject/hl7:patientRole/hl7:patientPerson/hl7:multipleBirthOrderNumber[@updateMode='added' or @updateMode='altered']/@value “1” “2” “3”

RULE LOGIC: 'xpathexists' | 'xpathnotexists' | 'xpathequals' **|** 'xpathnotequals' **|** xpathequalsignorecase’ | ' xpathnotequalsignorecase’ | 'xpathcompare' **|** 'xpathnotcompare' **|** 'xpathmatches' |'xpathnotmatches' | ‘xpathcontains’ | ' xpathnotcontains’ | ‘xpathcontainsignorecase’ | 'xpathnotcontainsignorecase' | **'xpathin'** |'schema' | 'xslt'

8.1.16 Rule Type – schema

Check used to test the message against a schema, in this case a tightened schema is used to check the message structure and that the correct data types and vocabularies are used.

Location: '**hl7**' | 'ebxml' | 'httpheader' |

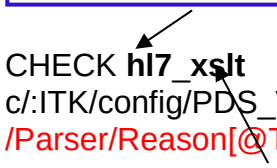
CHECK **hl7_schema**
c:/ITK/config/schema/omvt/TightenedSchemas/PORX_IN020102UK31.xsd

RULE LOGIC: 'xpathexists' | 'xpathnotexists' | 'xpathequals' **|** 'xpathnotequals' **|** xpathequalsignorecase’ | ' xpathnotequalsignorecase’ | 'xpathcompare' **|** 'xpathnotcompare' **|** 'xpathmatches' |'xpathnotmatches' | ‘xpathcontains’ | ' xpathnotcontains’ | ‘xpathcontainsignorecase’ | 'xpathnotcontainsignorecase' | 'xpathin' |'schema' | 'xslt'

8.1.17 Rule Type – xslt

Check used to test the message against an xslt, in this case the file ActRefChecker.xslt checks the message and checks the xml output for attribute type with the value Error.

Location: '**hl7**' | 'ebxml' | 'httpheader' |



CHECK **hl7_xslt**
c/:ITK/config/PDS_Validation/validator_config/transforms/ActRefChecker.xslt
/Parser/Reason[@Type='Error']

RULE LOGIC: 'xpathexists' | 'xpathnotexists' | 'xpathequals' **|** 'xpathnotequals' **|** xpathequalsignorecase' | ' xpathnotequalsignorecase' | 'xpathcompare' **|** 'xpathnotcompare' **|** 'xpathmatches' |'xpathnotmatches' | 'xpathcontains' | ' xpathnotcontains' | 'xpathcontainsignorecase' | 'xpathnotcontainsignorecase' | 'xpathin' |'schema' | 'xslt'

Appendix A

TKW Quickstart Guide:

Installation of the TKW:

1. Download the latest version of the TKWInstaller.jar from the appropriate store.
2. Double click[38] on the installer and follow prompts to finish[39].

Open the TKW properties file TKW_HOME\config\ Correspondence_20111010\tkw.properties[40] file - it is referred to throughout the guide

Validation Mode:

1. Place messages to be validated in the folder: TKW_HOME\config\ Correspondence_20111010\messages_for_validation
2. Using command prompt, execute[41]:
   ```
   java -jar TKW.jar -validate TKW_HOME\config\ Correspondence_20111010\tkw.properties
   or
   java –jar TKW.jar –spinevalidate TKW_HOME\config\
   Correspondence_20111010\spine_pds.properties[42]
   ```
3. The validation report will be generated in TKW_HOME\config\ Correspondence_20111010\validator_reports

The validation folder locations defined in the TKW properties file are identified by '*tks.validator.source*' and '*tks.validator.reports*' respectively

Simulator Mode:

1. Verify that the '*tkw.properties'* file has the value:
   tks.Toolkit.listenport 4000[43]
2. Start the simulator using:
   ```
   java –jar TKW.jar –simulator TKW_HOME\config\ Correspondence_20111010\tkw.properties
   ```

All messages sent to the TKW in simulator mode will be copied to TKW_HOME\config\ Correspondence_20111010\simulator_saved_messages. The simulator folder location defined in the TKW properties file is identified by '*tks.savedmessages'*

This will display:

38 If the installer does not work when double clicked, execute the installer through command prompt.

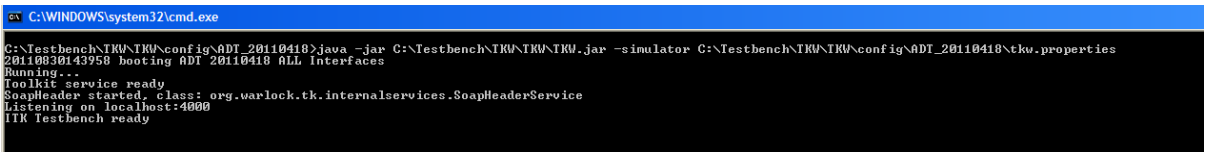
39The root installation folder of the TKW is referred to as TKW_HOME throughout this guide.

40 Correspondence_20111010 is being used here as an example but could be any domain

41 Navigate to the TKW.jar location in the command line program

42 pds is being used here as an example but could be any spine domain name e.g. spine_etp_prescriber, spine_etp_dispenser, spine_pds and "spine_generic".

43 Port 4000 has been used here as an example but could be any available port

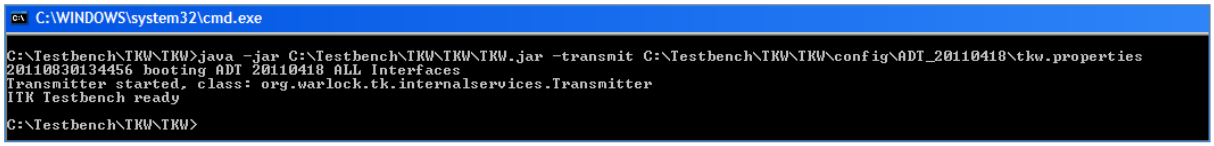


TKW is now ready to receive messages on port 4000 from a system under test or another instance of TKW

Transmitter Mode (Synchronous Response):

To send messages synchronously:

1. Verify that the '*tkw.properties'* file has the value:
   tks.transmitter.send.url http://127.0.0.1:4000/syncsoap[44]
2. Place the messages to be transmitted in:
   TKW_HOME\config\Correspondence_20111010\transmitter_source
3. Start the transmitter using the command:
   `java -jar TKW.jar -transmit TKW_HOME\config\ Correspondence_20111010\tkw.properties`
4. The following confirms that the messages were sent sucessfully:



5. The transmitted messages are stored in:*TKW_HOME\config\ Correspondence_20111010\transmitter_sent_messages*

6. The files are received with the names as below:
   <Sender IP Address>_sent_<timestamp>.log,
   <Sender IP Address>_sent_<timestamp>.log.signature

7. The *.log* file contains the SOAP request sent by the transmitter and the HTTP response received from the simulator.

The transmission folder locations defined in the TKW properties file are identified by '*tks.transmitter.source'* and *'tks.sender.destination'*

[44] Note that this setting (port 4000 along with the setting for simulator above) will allow 2 instances of TKW to interact